FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Android Testing

Ana Rita Silva Ferreira

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ana Cristina Ramada Paiva

July 22, 2017

Android Testing

Ana Rita Silva Ferreira

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Nuno Honório Rodrigues Flores
External Examiner: Alberto Silva
Supervisor: Ana Cristina Ramada Paiva

July 22, 2017

Abstract

Nowadays, mobile applications are essential in people's daily lives, especially in mobile applications for Android platforms. This is due to the fact that this system holds a large majority of the mobile applications market that are available through the Google Play Store and the large number of devices that use Android as an operating system. With this in mind, as well as the influence that some of these applications have had on people's lives, it becomes increasingly clear how important it is to ensure the quality of the applications available to users, including the official platform, the Google Play Store.

One way to increase the quality of these applications is through testing. However, often companies, due to lack of time and resources, do not give due attention to this component of the development and maintenance of their products / services. This factor led to the emergence of several tools that automate and facilitate the testing of applications. However, the existing approaches are still not satisfactory.

In this sense, this project intends to continue the development of a new approach initiated in a previous work (the pattern-based mobile application testing tool - iMPAcT tool), which tests whether good programming practices in Android are fulfilled by programmers and suppliers.

The iMPAct tool tests Android applications in an iterative process that combines reverse engineering, pattern matching and testing. The purpose is to test recurring behavior that is defined in a catalog (UI patterns). For each behavior there is an associated test strategy (UI Test Pattern) that verifies whether or not the behavior was well implemented.

The objective of this research is to extend iMPAcT with more behavior to be tested by adding new test strategies (UI Test Patterns).

In short, the development and improvement of iMPAcT Tool will allow Android developers and entities involved in the development process of these applications to increase the quality of their products and services and improve the quality of the applications that arrive to us every day through the smartphone or tablet and which increasingly influence our daily lives.

Resumo

Nos dias de hoje, as aplicações móveis são essenciais no dia-a-dia das pessoas, especialmente no que toca às aplicações móveis para plataformas Android. Isto deve-se ao facto deste sistema deter uma larga maioria do mercado de aplicações móveis que são disponibilizadas através da Google Play Store e ao largo número de dispositivos que utilizem o Android como sistema operativo. Tendo isto em vista, bem como a influência que algumas destas aplicações têm tido na vida das pessoas, torna-se cada vez mais evidente a importância de garantir a qualidade das aplicações disponibilizadas aos utilizadores, nomeadamente na plataforma oficial, a Google Play Store.

Uma forma de aumentar a qualidade destas aplicações é através do teste. Contudo, muitas vezes as empresas, por falta de tempo e recursos, não dão a devida atenção a esta componente do desenvolvimento e manutenção dos seus produtos/serviços. Este fator levou ao surgimento de diversas ferramentas que automatizam e facilitam o teste de aplicações. No entanto, as abordagens existentes ainda não são satisfatórias.

Neste sentido, este projeto pretende continuação o desenvolvimento de uma nova abordagem iniciada num trabalho anterior (a ferramenta de teste de aplicações móveis baseada em padrões - *iMPAcT tool*), que testa se as boas práticas de programação em Android são cumpridas por parte dos programadores e empresas fornecedoras.

A ferramenta iMPAct *tool* testa as aplicações Android num processo iterativo que combina *reverse engineering*, *pattern matching* e teste. O objetivo é testar comportamento recorrente que está definido num catálogo (*UI patterns*). Para cada comportamento existe uma estratégia de teste associada (*UI Test Pattern*) que verifica se o comportamento foi ou não bem implementado.

O objetivo deste trabalho de investigação é estender a iMPAcT com mais comportamento a testar adicionando novas estratégias de teste (*UI Test Patterns*).

Em suma, o desenvolvimento e melhoria da *iMPAcT Tool* permitirá aos programadores de Android e entidades envolvidas no processo de desenvolvimento destas aplicações aumentar a qualidade dos seus produtos e serviços, e melhorar a qualidade das aplicações que nos chegam todos os dias através do smartphone ou tablet e que influenciam cada vez mais o nosso dia-a-dia.

Acknowledgements

At the end of this stage that is finishing my academic course, there is many people to thank and remember, that were or have been a big part in my life and stood by me along the way.

But first, because this particularly journey of writing this Dissertation could not be possible without her, I would like to give a special thanks to my supervisor, Prof. Ana Paiva, for all the support, advice, help and guidance throughout this all process.

I would also like to thank all my "crazy" friends, for making this path a little bit more lighter to take and I dare say, a lot more fun.

A very special thanks to Jorge, for always be by my side for the last five years, for the love and understanding and, most of all, for believing that I could do it and succeed, even when I did not think I could do it.

And last but not least, to my parents, for always supporting me in my choices no matter what they were, as long as it made me happy. Thank you for giving me the great life base support and love, that has made me what I am today.

Ana Rita Ferreira

"You sort of start thinking anything's possible
if you've got enough nerve."

J.K. Rowling, *Harry Potter and the Order of the Phoenix*

Contents

List of Figures

LIST OF FIGURES

List of Tables

Abbreviations

API	Application Programming Interface
APK	Android Application Package
APP	Application
AUT	Application Under Test
DSL	Domain Specific Language
GUI	Graphical User Interface
HTML	HyperText Markup Language
IDE	Integrated Development Environment
MBT	Model Based Testing
OS	Operating System
PBGT	Pattern-Based GUI Testing
PDF	Portable Document Format
SDK	Software Development Kit
UI	User Interface
UITP	User Interface Test Pattern

Chapter 1

Introduction

In today's world, mobile applications and the Internet in general, have become almost essential in our daily lives.

As of 2016, the global mobile internet users have exceeded half the population of the whole world. And the time spent on the internet by the users of smartphones, tablets, computers or wearables, on a daily average, has reached 185 minutes for Millennials, 110 for Generation X and 43 amongst the Boomers [Staa].

The number of applications available in the online stores keeps increasing exponentially reaching more than 2.8 million available apps in Google Play Store alone, on March 2017, making it the largest app store followed by Apple's App Store with 2.2 million apps available in the same period.

Much of this growth is due to the fact that mobile applications present themselves as relatively easier to develop than computer applications, not to notice the fact that they are also developed at a lower price, which makes it possible for the industry to grow at these rates and produce every day more.

1.1 Problem

In order to guaranteer that their applications are successful in a world of millions now, developers and companies in the field need to assure that their products are in its correct functioning and behaviour.

With the constant changes and new concepts of applications development becoming frequent, testing this applications becomes one big challenge.

As referred by the World Quality Report 2014-15 [Capgemini *et al.*, 2014] the number of organizations that are testing mobile applications has grown from 31% in 2012, to 55% in 2013 and almost 87% in 2014. However, the biggest challenge is still the lack of the correct methods and processes to do it, mixing with the short time and lack of in-house testing environments.

Regarding all this, we can easily come to the conclusion that it is very important to automate mobile applications testing.

Along with that, the increase that developers and companies are giving to testing, now more than ever, is because the quality assurance of a product or service is, most of the time, the key to success. Considering the rate growth of applications available to the common user, the growing need the users have for these applications and their demand for a well developed and easy to use product makes the suppliers aware of these issues and eager to have access to tools and frameworks that will offer a simple, fast and economic way of guaranteeing that their product is the best possible and the one picked by the users in a market of millions now that is becoming more and more competitive.

1.2 Motivation and Goals

Even though there is already out there a few solutions that help developers and testers in this task, with some of them being official Google Testing Frameworks, none of them, as of the date of this dissertation, is capable of doing the whole process in an automatic manner, both generating and applying the tests, which leaves a gap in the testing world.

This is where the iMPAcT Tool (Mobile Pattern Testing) enters. A previous research work that has been developed that aims to verify if the guidelines - or best practices - of Android programming are followed. This tool automates the whole process of testing by using a combination of Reverse Engineering and UI (User Interface) Patterns. It presents a catalogue of recurring behaviours (Patterns) that matches with the ones in Application Under Testing (AUT), and tests them.

So for this dissertation, the work consisted in improving this tool, by improving its catalogue and enlarge its number of patterns to match in order to enhance the testing range of the applications and improve its results.

1.3 Methodology

For this dissertation, the methodology used was based on a research, in a first instance, both on the topics covered, like mobile testing and patterns, but also on the concept of the tool explored, the iMPAcT tool. Then, the details on the implementation of this approach, what was done and achieved, and finally, a set of experiments to validate the work performed.

Research

The first step in developing this dissertation, like any work of research, was to make research on the state of the art in the topics involved with the theme. There is a review, on Chapter 2, of the topics covered, like the existent approaches on mobile testing, model based testing and mobile reverse engineering. It is also covered topics like patterns and automated testing based on patterns. Finally, there is also coverage on the best practices for Android development.

Most of the information covered in this point is taken and then analyzed from several scientific papers published in conference proceedings, but also from official documentation and websites on the topics (in the case of the Android best practices).

Concept of the iMPAcT Tool

The next step is to analyse the tool that is the basis of this Dissertation and its concept. What does the tool do, what are its functionalities, where it comes from and where it intents to go, what can be done from this point. It is also mentioned the contributions of this tool to the field, what it brings that is new to the world. All this is reviewed in Chapter 3.

Implementation

After analysing the concept and functionalities of the tool used, in this phase it is reviewed the implementation of the work achieved in this dissertation.

The goal was to implement new patterns to the catalogue of UI and Test Patterns already existent in the iMPAcT tool. In order to choose, which patterns would be added to the tool and that would test if the best practices in Android development are being used, a few selection criteria were chosen for the new patterns:

- Not yet implemented;
- Be a native characteristic of Android applications and/or system;
- Be possible to implement, *i.e.* passable of being fully tested by the tool, not in need of external input;
- Be an embracing pattern amongst Android applications, *i.e.* be present in a varied number of applications and not in a very few number of them;
- Contribute to enhance and improve the quality and coverage of the iMPAcT tool's testing results.

A description of the new patterns in the iMPAcT tool, with what is expected of them according to the Android Best Practices Guidelines, their behaviour and details, as well as the formal definition of both the UI and Test Patterns for each one of the patterns added, can be found in Chapter 4.

Validation

In order to validate the work developed throughout this Dissertation, several experiments were conducted and presented in Chapter 5. The goal was to understand how the work achieved improved the iMPAcT tool and in which ways can it contribute to the field.

The methodology of this process of validation went through several stages, like defining the research questions that should be answered by the end of the experiments, the technical specifications of the device for the test environment, the criteria to select the applications to be tested and the final list of applications under testing, as well as the performance of the said tests and a sample of the results for each new patterns and the quality of the tests performed.

The discussion and conclusions of such experiments can later be found in Chapter 6.

1.4 Structure of the Dissertation

Besides the introduction, this report has five more chapters.

On chapter 2, the state of the art is described and the related works are presented. On chapter 3, the tool that is used in this work and the methodologies, technologies and solution is presented. On chapter 4, the implementation details, such as the descriptions of the new patterns added to the catalogue and the test strategies are described. On chapter 5, there are presented the results of the experiments and tests done in the work developed through this dissertation.

On chapter 6, a summary for the topics covered in this dissertation, a wrap up and the conclusions for the work done are taken.

Chapter 2

Testing Android Applications

In this chapter, there is a literary review of the state of the art in Testing in Android Applications.

First, a review of the evolution of mobile applications in the world in the last years is presented on Section 2.1.

Next, there is a technological review of the current approaches existing in Mobile Testing on Section 2.2, in which are presented some of the existing approaches for testing applications consisting in a few tools and frameworks, both official frameworks and non-official, that have changed the way we look into testing and quality assurance of mobile and particularly, Android applications.

Once these approaches are presented, we step into some of the field's research methods and review them, such as Model Based Testing on Section 2.3, going to Mobile Reverse Engineering on Section 2.4.

Following that, a review on Pattern-Based Testing on Section 2.5, where there is an analysis on Patterns, both UI and Test Patterns. It is also mentioned the PBGT - Pattern Based GUI Testing Project, a project for web that has been developed in the last few years and that plays as a base for the approach of the solution used in this Dissertation and explored in the next Chapter 3, finishing with Automated Pattern-Based Testing with a review on the basis of the iMPAcT tool.

Finally, a review on the Android Best Practices Guidelines, on Section 2.6, which guide this work and the way the iMPAcT tool should test the patterns present in applications and the conclusions that can be achieved from the state of the art presented.

2.1 Evolution of Mobile Applications

> "In 2016, the global mobile internet user penetration has exceeded half the world's population, while the average daily time spent accessing online content from a mobile device, such as a smartphone, a tablet computer or wearable, has reached 185 minutes

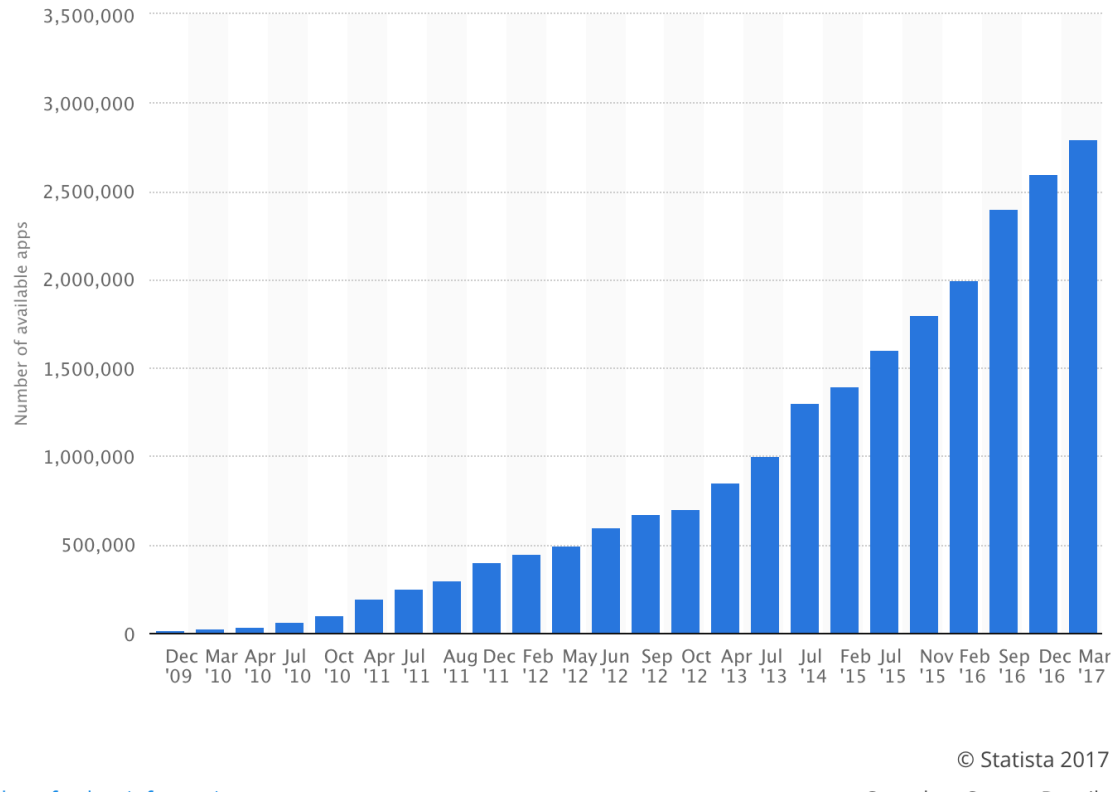

Figure 2.1: Number of available applications in Google Play Store from December 2009 to March 2017

daily among Millennials, 110 minutes for Generation X and 43 daily minutes for Boomers."[Staa]

If we alienate that fact that mobile applications are easier to create than computer applications with the fact that developing them comes at a lower price to developers and companies, it is easy to see why the market keeps growing at an exponentially rate.

On November 2015, there were 1.8 million apps available on Google Play Store. That is two hundred thousand more than in July of the same year. And in June 2016, there were already 2.2 million apps available to Android users. This means that in a space of a year, the number of apps in this store increased to almost the double. More so, in March 2017, there were already 2.8 million applications available to Android users on the Google Play Store.

A timeline of the growth of the number of applications available on Google Play Store from December 2009 to March 2017 [1] is displayed in Figure 2.1.

Google Play Store is the biggest online app store in the world followed by Apple's App Store,

[1] Statista;http://www.statista.com/graphic/5/266210/number-of-available-applications-in-the-google-play-store.jpg; 2017

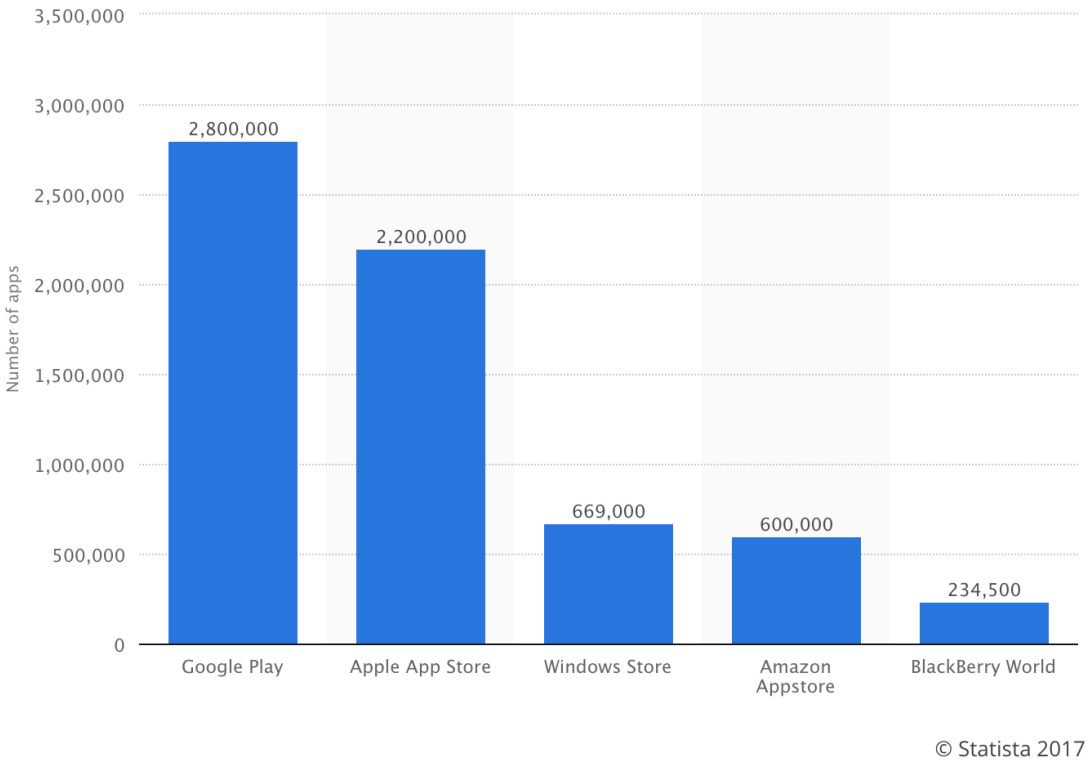

Figure 2.2: Number of available apps in Online Stores on March 2017

that on March 2017 [2], had 2.8 million applications available for its customers as depicted in Figure 2.2, compared to the 2.2 million apps on the Apple App Store.

The difference between these big stores is increasing, as the number of available applications on the Apple App Store on March 2017 is the one that the Google Play Store had previously achieved almost one year before, on June 2016 [Stab]. This might be explained by the differences in the prices applied by both technological giants, with Google's Android OS being the less expensive of the two and therefore, the most bought and used by the majority of devices. Also, with Android having the big majority of applications free of charge for a basic use, it is a huge contribute to the increasing difference and fast grow on the Android side.

As stated in [MP15c] and according to the World Quality Report 2014-15, the number of organizations performing mobile testing has grown from 31% in 2012 to 55% in 2013 and almost 87% in 2014. But, it is also mentioned that the biggest challenge for mobile testing is the lack of the right methods and processes for testing as much as the insufficient time to do so and the absence of in-house mobile test environments. These are some of the reasons why it is so important to automate mobile testing.

Furthermore, considering the increase in the number of companies and freelancer developers caring about mobile testing to ensure the quality of the applications launched in the official stores,

[2] Statista;https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/;2017

which more than ever, is important and directly connected with the success of such applications, in the last few years have emerged some frameworks and tools, both official and non-official, that have helped the testing process and that will be reviewed in this next section 2.2.

2.2 Mobile Testing - Current Approaches

In the automation of tests, the current approaches aim their focus on either the test cases generation or its execution. The community has had a special focus on automating the generation of test cases because there are already some official frameworks and tools by Android that help in the execution of such tests. Those frameworks are better explained in the next subsections [MP15a].

More so, after the official frameworks, it is also reviewed some non-official frameworks for Android testing. These frameworks exist as a way to fill the gap that existed at the beginning of Android development, back at 2008. Even though Android had launched an official framework by that time, its basic core forced the community of Android developers to develop tools that would be satisfactory in its purposes and necessities.

The understanding of these existent frameworks helped the development of the tool presented in Chpater 3, that was continued in this work and explored in Chapter 4.

2.2.1 Official Frameworks

Official Frameworks offered by Android exist since its launch in 2008. However, the first frameworks did not perform nor offered the right features as desired by developers.

At the end of the year of 2012, in November, the first version of UIAutomator [uia] was released by Google, being the first sign of time and effort invested by Google in testing applications. A new version of UIAutomator was later released in 2015, the UIAutomator 2.0, along with the Android 5.1.

UI Automator

> "The UI Automator testing framework provides a set of APIs to build UI tests that perform interactions on user apps and system apps." [3]

It allows to perform operations, like opening the Settings menu or the application launcher in a test device. This framework is suited for writing black box-style automated tests, which means that the test code does not rely on internal implementation details of the target application.

Some of the features of this framework are:

- **UI Automator Viewer** - A viewer to inspect layout hierarchy and that provides a GUI to scan and analyze the UI components currently displayed on the screen. It can be used to view the properties of UI components that are visible on the current screen;

[3] https://developer.android.com/topic/libraries/testing-support-library/index.html#UIAutomator; 2017

- **UI Automator APIs** - APIs that support cross-app UI testing;
- An API to **retrieve state** information and perform operations on the target device - provides a class to access and perform operations on the device on which the target application is running.

Espresso

Released in 2014, Espresso 2.0 [esp] was, for many, the official version of the Robotium framework, which will be explained later on in this section.

> "The Espresso testing framework provides a set of APIs to build UI tests to test user flows within an app." [4]

Such APIs let the developer write automated concise and reliable UI tests. The framework is also well-suited for writing white box-style automated tests, which means it uses implementation details from the AUT.

Some of its key features are:

- **View matching** - flexible APIs for view and adapter matching in target applications;
- **Action APIs** - extensive set of APIs in order to automate UI interactions;
- **UI thread synchronization** - in order to improve test reliability.

Monkey and monkeyrunner

In Android SDK there are two tools that do the testing on a functional level[5].

The **Monkey**, which is a command-line tool that sends streams of keystrokes, touches and gestures to a device. It can be used to stress-test the application and report on errors found. It can run multiple times and repeat a stream of events.

The **monkeyrunner** is an API and also execution environment for programs in Python. It helps in tasks like installing and uninstalling packages, connecting to a device, comparing two images, taking screenshots and running a test package against an application. Using this API allows to write a large number of complex tests.

2.2.2 Non-Official Frameworks

As previously stated, the non-official frameworks were born out of necessity of the community of having better testing frameworks than the very simple one that existed at the beginning of Android development and that did not correspond to the needs and requirements of developers in testing their applications.

These next frameworks are the most popular amongst the community in testing Android applications.

[4] https://developer.android.com/topic/libraries/testing-support-library/index.html#Espresso;2017
[5] https://developer.android.com/training/testing/start/index.html; 2017

Robotium

Robotium [rob] is the most popular testing framework amongst the community, spreading itself quickly and that appeared in 2010. As a test automation framework, it has a simple API that works in any Android device, regardless of its version.

As any framework, it presents both advantages and disadvantages. The advantages of using the Robotium framework are:

- Works with any Android version;
- Testes both native and hybrid Android applications;
- Does not require access to the AUT's source code;
- Able to access and modify the device's sensors and services's state.

On the other end, it has some disadvantages such as:

- Is not able to read the content of the screen;
- It only accesses and modifies those services for which the AUT has permissions;
- The AUT's main activity name needs to be hard coded.

Appium

As for Appium [appb], it is an open-source testing framework, that appeared in 2013. It presents several advantages, like:

- Does not require access to the AUT source code;
- Supports both Android and iOS applications;
- Tests native, hybrid and mobile web applications;
- Able to access and modify the services and sensors's status;
- Supports different frameworks and languages;
- AUT's package name and main activity can be inserted by input and does not have to be hard coded in the test code.

Among the disadvantages are:

- Not stable, sometimes creating unexpected erros;
- Does not return the screen content;
- It is not possible to locate an item by its *id*.

Calabash

The Android and iOS test automation framework, Calabash [cal], joined the community in 2012 and brought to it advantages such as:

- Reports on the performance of the device;
- Syntax based on natural language easy to understand;
- Possible to access and modify the status of services and sensors.

As for the disadvantages:

- It needs to have access to the AUT's source code;
- It is not capable of reading the content of the screen.

Selendroid

Like Appium, Selendroid [sel] was released in 2013 and it is a test automation framework, similar to Selenium[6], but for Android applications.

The main advantages of this framework are:

- Supports both native and hybrid applications;
- Has an inspector tool to analyze the contents of the screen;
- Does not need access to the source code of the AUT;
- Simpler than Appium when it comes to the localization of the screen elements.

But it also has its disadvantages, like:

- It does not have access or is able to modify the status of the services and sensors;
- The inspector tool does not offer an API.

When comparing all these frameworks, it becomes clear that there is a lot in common between them. There is also a few characteristics that make them distinguish themselves from the others. But there are a few main problems amongst all of them:

1. It is necessary to know which AUT is to be tested beforehand;
2. It is not possible to analyze the screen to know automatically which events should be fired.

[6] Selenium - http://www.seleniumhq.org/ ; 2017

2.3 Model Based Testing

There is a technique that enables the automation of test generation, which is the Model Based Testing - MBT [MP15c, MPM13b, PFTV05, KMPK06, MP13]. In this technique, the test cases are generated from a model/specification. More precisely, a model of the application's behaviour, that is an input whereas the output is a test suite.

However, the MBT has two main issues:

- The need of an input model of the application - that requires manual construction, which is an error prone process and very time consuming;
- The combinatorial explosion of test cases.

To tackle the first problem, we can use reverse engineering in the application under testing (AUT), which will be addressed in the next section, and/or by increasing the level of abstraction of the model. More so, the PBGT project, which will be reviewed later in this chapter, diminishes the effort required to build this model [MP15b]. As for the second problem, focusing on testing recurring behaviours - behaviour patterns - will help in the addressing of the problem.

2.4 Mobile Reverse Engineering

For a very long time, Software Reverse Engineering has been a field of research. There has been many approaches in extracting models from desktop and web applications.

However, for this Dissertation, the focus is on the research on reverse engineering for mobile applications. In depth, Android reverse engineering, since the solution for this Dissertation focus on Android and also because most approaches have been focusing on Android particularly.

The main goal is to obtain the source code of the application from the APK (Android Application Package) - the Android executable file - through reverse engineering, which is used in the solution of this Dissertation and will be better explained in the next chapter, and as long as the application is not obfuscated, Java disassembling or decompiling techniques may be used [MP15c]. As such, reverse engineering approaches may focus on:

- **Static** reverse engineering - how to automatically analyze the source code;
- **Dynamic** reverse engineering - automatically analyze the application at run time;
- **Hybrid** reverse engineering - analyzing both the source code and the application at run time.

If we consider the nature of mobile applications, since they are event-based, dynamic and hybrid approaches are the most common in this case, although some authors were able to identify possible security vulnerabilities by using a static approach (Batyuk *et al.*, 2011) [BHC+11].

But the purpose of using reverse engineering in mobile applications is to obtain a model and/or test it. Some authors have already shown works regarding this, like Yang *et al.* in 2013 [YPX13] that use an hybrid approach to obtain a model by identifying events to be fired in a first static step

to then fire those events in a second dynamic phase; or Amalfitano *et al.* in 2012 [AFT+12], in which they generate test cases following a dynamic approach.

2.5 Pattern-Based Testing

2.5.1 Patterns

In the last few years, there have been studies on the usefulness of using patterns for testing mobile applications.

The author Erik Nilson [Nil09], in 2009, came to the conclusion that UI design patterns might be useful in resolving recurring problems in Android applications development. If there is a behaviour associated with the patterns, then it is possible to identify the pattern by identifying automatically the behaviour.

Shirazi *et al.* [SSHS+13], in 2013, performed a study on the layout of applications. One of their goals was to verify if the same layouts presented any patterns. In their conclusions, there was 75.8% of unique combinations of elements that seemed to appear only once in the application, though the study only took in consideration a static analysis of the layout and the elements of the application might represent the same behaviour even if presented in a different combination. Which means, different combinations of elements may represent the same behaviour, therefore, the same pattern.

However, even being useful in testing mobile applications, some authors have different approaches, focusing on different types of patterns. For example, Batyuk *et al.* [BHC+11] consider a pattern as a set of malicious access, while Amalfitano *et al.* [AAF+14] define three different types of patterns: event patterns, model patterns and GUI patterns; Costa *et al.* [CPN14] consider UI Test Patterns.

The approaches made by Costa *et al.* and Amalfitano *et al.* are similar to the ones in the solution presented in this Dissertation in the next chapter, in the way that Amalfitano *et al.*'s patterns are similar to the UI Patterns considered here and Costa *et al.*'s are similar to the Test Patterns that will be used.

To assure that the reuse of patterns and the definition and addition of new patterns is an easy process, a formal definition of these patterns is very important. It is important to understand that the goal is not to test everything, but instead, to test recurrent behaviour. So the model is composed by the UI Patterns and the Test Patterns. The formal definition presented next is both applied to UI Patterns an Test Patterns.

Therefore, we can represent a pattern by a tuple **< Goal, V, A, C, P >**, in which:

- **Goal** is the ID of the pattern;
- **V** is a set of pairs {variable, value} that relates input data with involved variables;
- **A** is the sequence of actions to run;
- **C** is the set of checks to perform;

- **P** is the precondition (a boolean expression) that defines the conditions in which the pattern is applied.

In a formal definition:

$$G[configuration] : P -> A[V] -> C \tag{2.1}$$

,i.e., for each goal's configuration **G[configuration]**, if precondition **P** is true, a sequence of actions **A** is executed with the input values **V**. Finally, a set of checks **C** is performed [MP15c, MP15a, MP15b].

UI Patterns

More particularly, in UI Patterns:

- **P** defines when to verify if the pattern exists;
- **A** defines which actions should be executed to verify the presence of the pattern;
- **C** validates the presence of the pattern.

Test Patterns

And in Test patterns:

- **P** defines when the test is applied - includes validation of the corresponding UI Pattern;
- **A** defines the action of the test in itself;
- **C** indicates whether the test passed or failed.

2.5.2 PBGT - Pattern Based GUI Testing Project

As previously stated, Pattern-Based GUI Testing Project (PBGT) [MP14b] is a project that looks to diminish the effort that is required in building a model for MBT.

It does this by presenting a modeling framework with an easy use [MP13] and also by providing a Domain Specific Language (DSL) - PARADIGM [MP14a], which increases the abstraction level of the model by describing test goals instead of system functionality. PARADIGM is built on top of User Interface Test Patterns (UITPs) which provide the test strategies for the UI Patterns themselves.

In the context of this project, there was conducted an experiment on mobile applications to realize if this approach could also be applied to those applications. It was proved with the success of the experiment, the necessity to develop appropriate test strategies for mobile applications.

This project is also based on the assumption that GUIs that are based on the same UI Patterns should share the same UI test strategy as realized before. Basically, in this project was developed

the concept of UI Test Pattern - the association of a set of test strategies to the corresponding UI Pattern.

Furthermore, in the context of this project, a tool was developed on top of the Eclipse Modelling Framework [7] and has the following components:

- A modelling environment to build and configure GUI models;
- A Domain Specific Language (PARADIGM) which builds the test models based on UITPs;
- A test case generator based on the PARADIGM models;
- A reverse engineering process that generates and extracts automatically the model of a web application.

As stated before, and even though this project was originally goaled to test web applications, with the experiments on mobile applications, there was the conclusion that the same approach could be applied to those. However, in the case of mobile applications, there are adaptations needed to be developed like specific test strategies, as stated before.

This is because there are different development concepts in the mobile world, like activities, interaction gestures and limited memory [MP15b]. These differences will give space to a new tool, the iMPAcT tool, which is the main focus of this Dissertation and will be later described in Chapter 3.

2.5.3 Automated Pattern-Based Testing

From this point of research, it was important to develop another work of research - [MPF14] - that puts all this together and that would, eventually, lead to projects like the tool from the solution of this Dissertation - the iMPAcT tool.

The purpose of the research was to construct a catalogue of recurring behavioural patterns and develop a tool that would explore a mobile application in an automatic way while identifying and testing such patterns during its exploration.

In this approach, the main steps are:

1. Define a catalogue of mobile GUI patterns and the corresponding test strategies;
2. Apply an hybrid reverse engineering approach in order to explore the mobile applications automatically;
3. Identify patterns on the fly and apply the corresponding and predefined test;
4. Store the information that regards all the behaviour explored;
5. Produce a test and matched patterns report.

[7] https://www.eclipse.org/modeling/emf/; 2017

To validate this approach, there was a case study that was conducted in order to verify three main aspects:

- Induce errors on an application to verify if the tool detects them;
- Analyse the percentage of code explored and tested;
- Compare results with those of other approaches.

This case study and some of its results as well as the specifics of the iMPAcT tool will be looked deeper in the next chapter, Chapter 3.

2.6 Android Best Practices - Guidelines

The guidelines for the Best Practices in Android Development are a series of documents in the web provided by Google in which the developers can and should follow in order to create the very best apps in Android provided to the user. These guidelines are to be found in the documentation of Android development provided to the developers. This documentation can be found at [And].

These guidelines are in the basis of this research work, as they tell how the patterns and other specifications of Android applications should be and look in the final application. They are used in the tool later described in chapters 3 and 4, as also guidelines in which the tool tests applications.

The documentation is divided in three big modules: Design, Develop and Distribute. The documentation most used in this work can be found in the Develop module and also partially in the Distribute module. Even though the Design module is not directly used, most of the Android components depend on all three modules in one way or another, which means that the design documentation also affects those from develop and distribute.

On the Develop module, the first part is focused on training, in which the guidelines are at, and aims to give the Android developers the basics and the best practices they should follow while developing their applications. Inside this, these series of documents are also divided in smaller inner modules, where the modules for best practices are included. These modules are:

- Best Practices for Interaction and Engagement
- Best Practices for User Interface
- Best Practices for Background Jobs
- Best Practices for Performance
- Best Practices for Security and Privacy
- Best Practices Permissions and Identities

From these modules, we will focus on the ones later used directly in this dissertation: Interaction & Engagement, User Interface and Background Jobs.

On the Best Practices for **Interaction and Engagement**, it is possible to find the guidelines in *Design Effective Navigation*, in which it is possible to find the document "Providing Ancestral and Temporal Navigation" [Anc] where it gives a brief overview on these types of navigation. These are later better explained in *Implementing Effective Navigation*, where the documents "Providing Up Navigation" [UpN] and "Providing Proper Back Navigation" [BkN] can be found.

As for the Best Practices for **User Interface**, in *Adding the App Bar*, there is documents on "Setting up the App Bar" [Appa] and "Adding an Up Action" [UpA]. And in the Best Practices for **Background Jobs**, the documents present in *Running in a Background Service* [Bac] all were fundamental in the research and development of this dissertation's work.

More over, on the Distribute module, on the *Core App Quality*, there is also a list on which details developers should check to make sure their applications are developed according to the Android's standards, which can be found on [Qua]. As for the influence that Design also has on these applications, which has increased its importance in the last few years, as one of the pillars of user interface and user experience, Android has developed what they call the **Material Design** [Mat], a series of guidelines on applications design in which the Android applications become more similar and thus, more familiar and friendly to the user.

2.7 Conclusions

With the technologies and new approaches that keep appearing, there is a lot of new information and new researches being born in the field of testing applications, either for desktop, web, or more importantly in this case, mobile applications. But with the exponentially growing market for these applications, the results and researches done and the tools and different approaches that come from that are just not fast enough, nor cost and time effective for the developers, testers and companies in the business.

There is a growing need from the parties involved, for tools and systems that may assist them in better performing and becoming more competitive in what is already, a market of millions. That may help them prosper in a world of automation and constant change, both technological and social. And as far as we know, there is not a tool in the market that tests the best practices in mobile applications, except for the iMPAcT tool, presented in Chapter 3.

Furthermore, it is becoming obvious how automation and easiness in testing needs to be developed and soon. So in the next chapter, it will be presented one solution for this problem. A solution that might help developers and testers a great deal by easing the task of testing, but also open new doors in research for more improvement and new approaches and tools - the iMPAcT tool. The goal of this research work is to help develop this tool, in order to test new patterns and if the best practices are being applied.

Chapter 3

The iMPAcT Tool

The tool here presented aims to automate the process of testing recurring behaviours that are present in Android applications. Its main concern is to assure that the guidelines - or best practices - in Android programming are followed.

In this chapter it is described the tool's approach in Section 3.1, how it is divided and how its configurator looks like, the phases it goes through, how it identifies the events, how it is implemented and how its catalogue is built. It will also be referred what were the results achieved in a previous case study in Section 3.2, in which ways the iMPAcT tool contributes to the community in Section 3.3 and what is the work expected to be achieved with this dissertation in Section 3.4, which will later be described in Chapter 4 with more detail.

3.1 Approach

It is presented in Figure 3.1 [MPF14] the testing approach that is supported by the iMPAcT tool which has four main principles:

1. The goal is to test UI Patterns;
2. The whole process is completely automatic;
3. It is an iterative process combining automatic exploration, reverse engineering and testing;
4. The reverse engineering process is completely dynamic [MP16].

In the end, the tool shows two different types of results:

- Matched patterns - UI patterns present in the application;
- Failures detected - patterns that are not correctly implemented [MP15a].

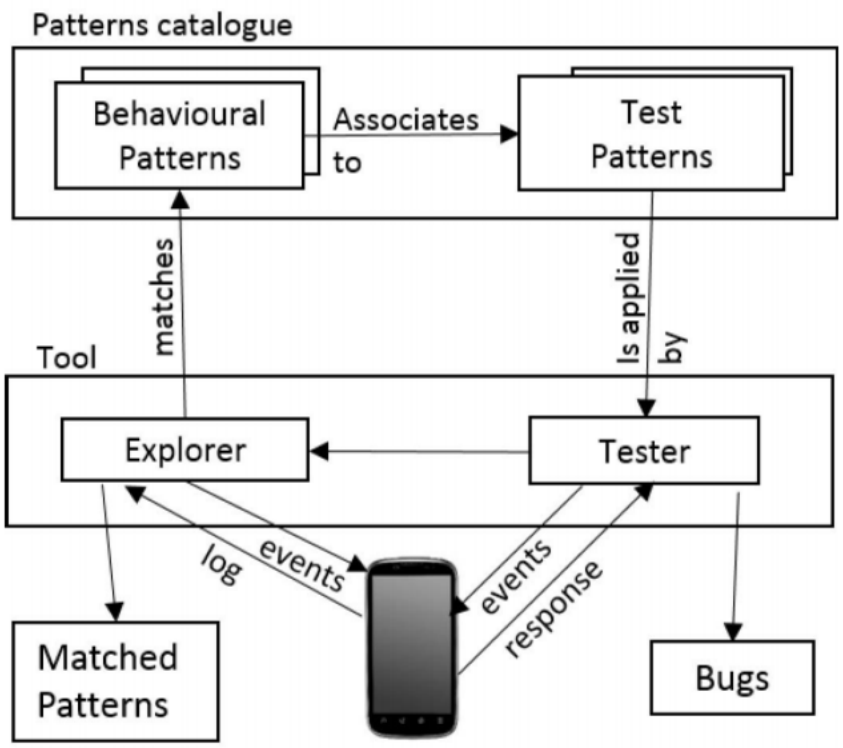

Figure 3.1: Block Diagram of the Architecture of the Approach

3.1.1 Test Configuration

The iMPAcT tool is divided in two parts:

- A **Configurator** - handles the configuration of the test process. It is where the user is able to provide input information about the application about to go under testing.
- A **Tester** - Implements the approach itself, which will be explained in the next subsection 3.1.2.

Both these parts were developed in Java, the first one (*Configurator*) being a thir-party application, while the second part (*Tester*) is an Android test project.

So the first step in using the iMPAcT tool, is giving the tool the necessary information it needs to perform the test. That information is inputed by the user of the tool through an interface as shown in Figure 3.2.

In this window, the user has to input and choose the configurations for the test to be performed such as:

- The AUT - either by the APK of the application or its package name;
- Choose the type of exploration - later reviewed in this chapter, in subsection 3.1.3;

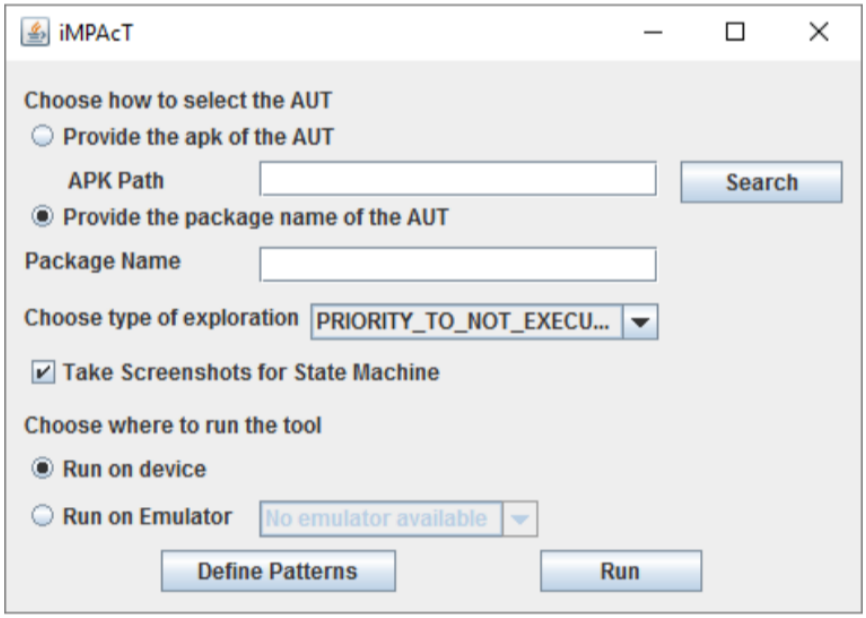

Figure 3.2: iMPAcT tool Configurator - Main window

- Where to run the test - on a device or an emulator;
- Define the Patterns - choose the patterns to be tested from the catalogue available (showed in another window).

3.1.2 Phases

The iterative process of this approach is divided in three phases:

1. **Exploration**
2. **Pattern Matching**
3. **Tester**

In the first phase - **Exploration** - there is an analysis of the current state of the application. It identifies the different elements present, e.g. buttons or text boxes, in the current screen and also determines which events are enabled, like click, edit or check. At the moment, the decision of which event is fired is random. After selecting the event, the same is then fired. A summary of this phase can be seen in Figure 3.3 [MP15a].

In the **Pattern Matching** phase, after the event from the previous phase has been identified and fired, the tool will try to identify which UI Patterns are present on the screen at the moment. All the patterns in the catalogue are analyzed to check if they are present in the current state of the application. The matching itself, more precisely, consists in checking if the precondition of the UI Pattern holds, executing the actions necessary and verifying if all checks are met [MP15c]. If this happens, then the UI Pattern is considered found.

Code 4. Exploration

```
function fire_event
  call get_current_screen
  for each node in screen
    call node.get_possible_events
  call decide_event
  call execute_event
}
```

Figure 3.3: Exploration Phase

Finally, in phase three - **Tester** - once the UI Pattern is matched from phase two, it is tested. So this phase applies, to the UI Pattern found, the correspondent Test Pattern from the catalogue that the tool provides. This process is similar to the one in phase two. It verifies is the preconditions are met, executes the actions and verifies the checks. If everything goes through, then it means that the test passes. If not, the test will fail and it will be reported in a log file.

It should be noted that if there is no pattern found in phase two, this phase is skipped altogether.

A sample of how this two phases work can be summarized in Figure 3.4 [MP15a].

Also, a summary of the approach's cycle can be found in Figure 3.5 [MP15a].

3.1.3 Identification of Events

As previously stated, the decision of which event to execute in the exploration phase is random. However, the set of events that are available for execution depends on a variable defined by the user - the exploration mode [MP16].

The modes available are three:

1. **Execute Once**: Are only considered the events that have not yet been executed;

2. **Prioritize Not Executed**: Choice made among events that have not yet been fired. Note that it may be possible to run an event a second time if all other events have already been fired and the event is necessary to give access to a screen with not executed events;

3. **Prioritize List Items and Not Executed**: Similar to the previous mode but with two differences: 1) events executed on elements from a list have priority; 2) "click on the App button" event is only fired when there is no more events to do so. This mode, basically, tries to fire all events of each screen before moving on.

3.1.4 Stop Condition

When there is not an available event, the *back button* is actioned to reach the previous screen. If this takes to the home screen of the device, the exploration stops. This stop condition should be refined in the future.

Code 5. Apply pattern: try_to_find_UI_pattern() and apply_test_pattern()

```
if pattern precondition holds then
    call execute pattern actions
    call verify pattern checks
    if checks are met then
      return true
    else
      return false
    endif
else
   return false
endif
```

Figure 3.4: Pattern Matching and Tester Phases

3.1.5 Patterns Catalogue

The definition of the patterns catalogue is one of the most important aspects in this approach. This catalogue is what ultimately defines what is to be tested and how. It consists on a set of UI Patterns and the corresponding test patterns - the test strategies [MP15c].

This catalogue is defined only once and may be reused for any Android mobile application.

A Pattern in here is considered a pair of both UI and Test patterns and it is defined as a set of tuples **< G, V, A, C, P >**, in which [MP16]:

- **G** is the ID (goal) of the pattern;
- **V** is a set of pairs [variable, value] that relates input data with the variables involved;
- **A** is the sequence of actions to execute;
- **C** is the set of checks to perform;
- **P** is the precondition that defines in which conditions the pattern should be applied.

To apply a pattern, the tester or developer will need to configure the value for each V variable. There is the possibility of using the same pattern several times with different configurations.

Therefore, applying a pattern consists in: if a precondition P is true, a sequence of actions A is executed with the values V. After this, a set of checks C is performed.

While in Pattern Matching, the preconditions result will indicate if the tool should try to identify the pattern and the checks result will indicate if the pattern is present or not. Then in the Tester - when applying the Test Pattern - the preconditions tell if the test strategy that corresponds to the pattern should be applied and the checks indicate whether or not the pattern is implemented in a correct way.

The patterns that are currently present in the catalogue are based on the guidelines that Android provides on how to design applications and how to test.

Code 1. An iteration of the approach's cycle

```
set exploring to true
while exploring=true
   call fire_event
   call try_to_find_UI_patterns
   if UI pattern is found then
     call apply_test_pattern
    endif
   read exploring
endwhile
```

Figure 3.5: Iteration of the Approach's cycle

In order to understand better these patterns and the differences between the UI Pattern and the Test Pattern, two examples of patterns identified and tested by the tool will be presented: the **Side Drawer Pattern** and the **Orientation Pattern**.

The **Side Drawer Pattern** is a form of navigation through different screens and hierarchy provided by the Android OS. It is a transient menu that opens when the user swipes the screen from left to centre or clicks on the icon in the left of the application's *Action Bar*. In Figure 3.6 [MP15c] there is an example of this pattern. The main challenge in the implementation of this pattern is the correct identification of the side drawer element.

The UI Pattern is [MP15c, MP15a]:

Goal:"Side Drawer exists"

V: {}

A: []

C: {"side drawer exists and is hidden"}

P: {true}

The corresponding Test Pattern is:

Goal: "Side Drawer occupies full height"

V: {}

A: [open side drawer]

C: {"covers the screen in full height"}

P: {"UIP present && side drawer available && TP not applied to current activity"}

As for the **Orientation Pattern**, in Android devices there are two possible orientations of the screen: portrait (a) and landscape (b) as seen in Figure 3.7 [MP15c].

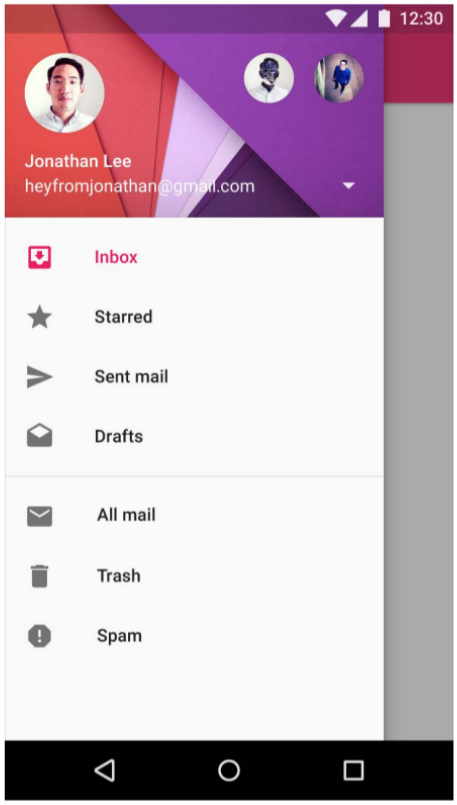

Figure 3.6: Example of the Side Drawer Pattern

When rotating the device, the layout of the application is updated. Nevertheless, according to the guidelines from Android for testing, developers should be aware of two aspects that they should test. Those are that the custom UI code can and should ensure that the main elements of the layout are present and that no user input data is lost. This is also visible in Figure 3.7. As for the main challenges in the implementation of this pattern, it is how to match the elements from the pre-rotation screen to those on the post-rotation screen.

In this case, the UI Pattern is:

Goal: "Rotation is possible"

V: {}

A: []

C: {"it is possible to rotate screen"}

P:{"true"}

The respective Test Patterns - there is two possibilities in this case - are:

Goal: "Data unchanged when screen rotates"

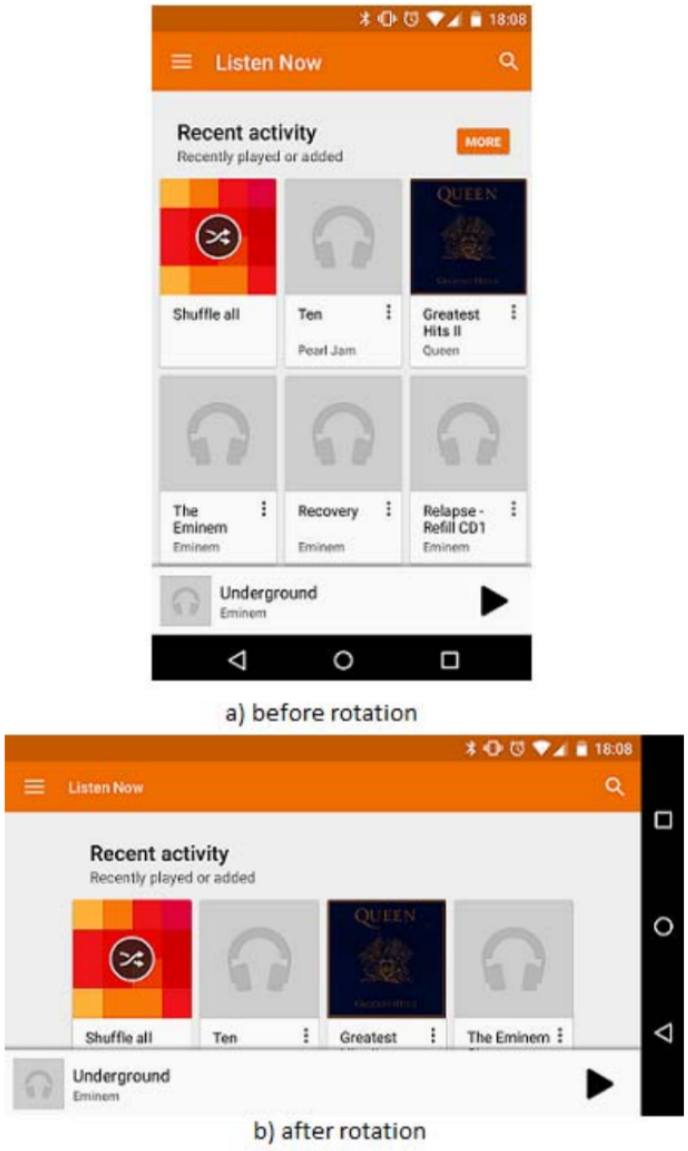

Figure 3.7: Example of Orientation Pattern. a)portrait and b) landscape

V: {}

A: [read screen, rotate screen, read screen]

C: {"user entered data was not lost"}

P: {"UIP is present && user data was entered && TP not applied to current activity"}

and

Goal: "UI main components are still present"

V:{}

A: [read screen, rotate screen, read screen]

C: {"main components still present"}

P: { "UIP is present && TP not applied to current activity"}

One of the main challenges that both examples share is on how to tell apart the activities which means that a special care is needed regarding this aspect.

3.1.6 Technologies

The iMPAcT tool uses, essentially, two technologies/tools: Java and Android SDK.

Java is the main language in which the tool is developed and it is also the base language for Android development itself.

Android SDK - which stands for Software Development Kit - is the framework provided by Android in which compiles several development tools. It includes a debugger, libraries, documentation, tutorials and example codes.

Since the tool was already in development and uses these technologies, they were also the ones used on this Dissertation.

3.2 Case Study

Regarding this new tool, and to validate its approach, a case study was conducted [MP16].

As stated before, in the iMPAcT tool it is possible to select the exploration mode as well as defining the order in which the patterns are identified and tested. So it is easy to realize that it is possible that the results might change according to this settings. Moreover, depending on this setting, the failures detected may be different as well as the number of events executed.

The goal to this experiment was to compare results obtained by the three different exploration modes and two testing orders. At the end, there was a total of six possible combinations for each application tested.

Analysing the results of the test on Tomdroid application, which can be found in [MP16], it was possible to conclude that there is a significant difference between the exploration times of mode 1 compared to modes 2 and 3. This can be explained by the fact that exploration mode 1 executed a much lower number of events. It was also possible to notice that as modes 2 and 3 have the possibility of a more deep exploration, this leads to more failures being found. An image of the results obtained can be found in Figure 3.8.

As for the testing on the Book Catalogue application, which is a bigger application than the previous one, as mode 3 detects more events and executes a higher percentage of those events, it gives us a more insightful exploration and also detects more failures. On the negative side, this mode takes much longer to perform its exploration when compared to the other two modes. An image with these results should be found in Figure 3.9.

Table 1. Results obtained when applying different execution modes to Tomdroid

Testing Order	Orientation failures	Execution Time	Events identified	% of events	% events over execution mode	% events over exploration mode	% events over all executions
exploration mode 1: execute only once							
SD-O	1	2m 4s 410ms	76	52.7	33.6	33.6	24
O-SD	2.75	2m 8s 402ms	51	67.6	54	36.2	26
exploration mode 2: priority to not executed							
SD-O	1.75	5m 45s 466ms	106	85.6	63	63	63
O-SD	2.75	5m 50s 584ms	89	76.3	75	63	63
exploration mode 3: priority to not executed and list items							
SD-O	2	5m 39s 951ms	89	82.2	74.5	65.6	62.5
O-SD	2.5	5m 47s 346ms	101	76.8	61.1	61.1	59.15

Figure 3.8: Results of the case study on Tomdroid application

In the end we can conclude that modes 2 and 3 are better suited for better results (more coverage of the applications), yet, mode 3 is better for bigger applications. When we have smaller applications the differences between modes 2 and 3 is not significant. But the main conclusion is that the exploration mode affects the overall time and number of events tested. The number of failures found is not necessarily affected.

3.3 Contributions to the Community

There are several advantages that the iMPAcT tool brings to the community of developers, testers and researchers of the testing in mobile applications field in general.

Among this advantages, there are characteristics [MP15a] like:

- **Access to Source Code** - the source code is never accessed nor is any code instrumentation required since the reverse engineering process is fully dynamic;
- **Manual Effort** - None is needed since the whole process is automatic;
- **Reuse** - As UI Patterns are present in several applications, the ones defined in the catalogue may be reused without any modifications needed;
- **Maintenance and Evolution** - The tool's code structure eases the process of adding new interactions, therefore, it is prone to evolution.
- **Usability** - The user does not need to have any information or knowledge about the application, nor the patterns that it contains, in order to use the tool;
- **Innovation** - Up until now, there is no knowledge of a tool that tests mobile applications that mixes reverse engineering techniques and UI Patterns.

Table 2. Results obtained when applying the different exploration modes and pattern orders to Book Catalogue

Testing Order	Orientation failures	Execution Time	Events identified	% of events	% events over execution mode	% events over exploration mode	% events over all executions
exploration mode 1: execute only once							
SD-O	5.75	4m 27s 362ms	141	45.6	25	25	5.4
O-SD	9	5m 15s 798ms	141	50.5	23.3	23.3	6.5
exploration mode 2: priority to not executed							
SD-O	13.25	13m 24s 297ms	372	54	34.3	34.3	19.5
O-SD	12.75	8m 32s 318ms	230	52.7	34.4	20.8	12.1
exploration mode 3: priority to not executed and list items							
SD-O	23.75	34m 1s 170ms	653	79.1	52.2	52.2	52.2
O-SD	33.25	32m 13s 346ms	418	83.6	76.1	48.7	48.7

Figure 3.9: Results of the case study on Book Catalogue application

3.4 Work from here

So, for this Dissertation, the main work consisted in, from this state of the tool:

- **Increase the testing behaviour** - Add new patterns to the catalogue in order to increase the range of the testing on the applications and improve the final results. The number of patterns presented in catalogue was one of the biggest limitations that the tool had and that needed to be fulfilled in order to get better testing.

- **Improve Test Results Visualization** - At this point, the results of the testing were presented in a log file that showed the patterns and failures found. The goal was to change this presentation to that of a proper report (an HTML or PDF file) with this information and other information that may be considered relevant for the developers/testers.

The first step is to parametrize the tool's configuration and the behaviour to be tested. After this, new patterns are added to the tool. With this step concluded, a case study is made in order to test the new changes and comparing the new results with the previous ones.

3.5 Summary and Conclusions

Summarizing, the iMPAct tool has revealed to be in a good path with a new and different approach in the testing of mobile applications. Therefore, the continuing of this work, thus the improvement of the tool, may lead to a very good and simple way of helping developers/testers/companies in the testing and quality assurance of their applications which are brought to the market and also open new trails and challenges in the future of mobile applications, and more specifically in this case, Android applications testing research.

Chapter 4

Implementation

In this chapter, it is presented an overview on the work developed in this dissertation. The patterns that are new to the catalogue are reviewed in Section 4.1. In Section 4.2, a summary and some conclusions of the work achieved are made.

The results from a series of tests to the work developed and described in this chapter will be looked through in Chapter 5.

4.1 Patterns Catalogue

In this section, the new patterns on the catalogue added to the iMPAcT tool are described and reviewed, as well as described the formal tests performed in order to check if these same patterns are correctly implemented by the AUT and that they follow the Android guidelines of the Best Practices in Android Developing.

The pattern added that are described next are the Background Pattern, the Action Bar Pattern, the Back Pattern and the Up Pattern.

4.1.1 Background Pattern

In Android applications, most of the work done by the app is performed on the foreground. As some of the tasks that the several applications running on a device tend to use a lot of resources and, thus, affect the performance, the Android framework has provided several classes that allow some operations to run on a background service. Besides this factor, and on the line of providing the user the best experience possible, it is also possible for the user, and should be expected, that when using an application and hitting the home button of the device, the user is taken to the home screen of the device. But this action does not fully exit the application that was being used. It is expected that the application continues to run on background and to keep the state exactly the same as the one last seen by the user. This means, that if the user wished to open said application again, he should click the app item on the device or click on the recent apps menu, look for that

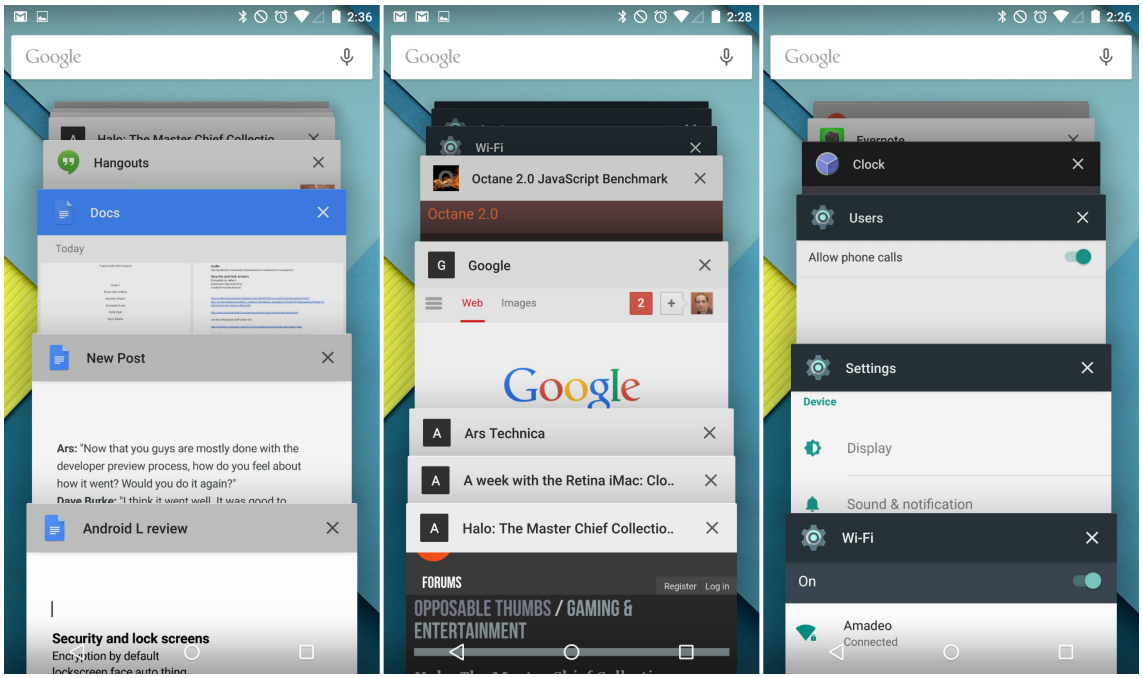

Figure 4.1: Recent Apps working in background on Android

app and click to open it. By opening the app, the screen presented to the user, should be the same, and in the same state that it was, just before the user hit the home button and left the app.

The goal of testing this pattern is to check if the states, before the application goes to background and after it is brought back to foreground, are the same. So in this test, the strategy is to send the AUT to the background by clicking on the home button. Then it is performed a click on the recent apps menu and the AUT is clicked to open again. Then, a comparison between the screen in the beginning of the test and the current screen is performed. A sample of how the recent apps menu, which hold all applications running in background, is shown in Figure 4.1

In short, this test aims to:

- Check if the AUT goes to background when the home button is pressed and the process is not killed or the app does not crash.
- Check that when bringing from background to foreground, the app is in the same state as it was previous to being sent to background.

Considering that to test this pattern the only condition is that the AUT is open, there is not the necessity of formally defining an UI Pattern, since it exists by nature.

Only the Test Pattern is formally defined as follows:

Goal: "App goes to background and keeps the same state from before background event"

V: {}

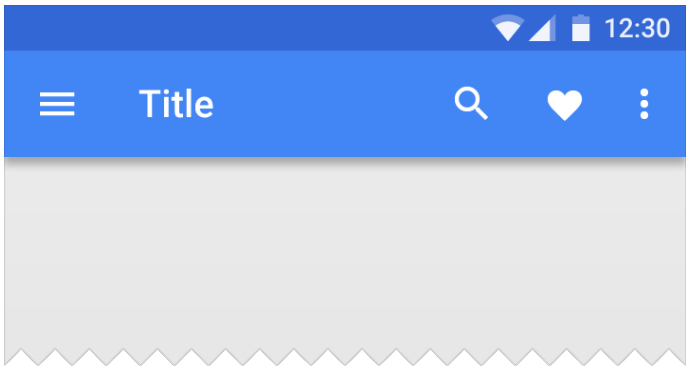

Figure 4.2: Action Bar on Android Apps

A: ["observation", app goes to background, "observation", send app to foreground, "observation"]

C: {"Verify the app state is equal to the previous one"}

P: {"AUT is open && TP not applied to current activity"}

In the end, if the test fails, the error is displayed in the final report.

4.1.2 Action Bar Pattern

One of the most important design elements in an application is its App Bar, or also commonly known as Action Bar. This is due to the fact that it gives the user a visual structure and elements that are familiar to him/her. The use of the Action Bar, according to the Android guidelines, provides consistency, making it similar to other Android applications, which allows users to understand the application quickly and easily and in order to enhance to user's experience.

The main functions of the Action Bar are the following:

- A space dedicated to give identity to the app and localize the user inside it;
- Give access to the user to important actions in a predictable way, i.e. search or new;
- Support to navigation and changes in the view within the app;

In its most simple mode, the Action Bar presents the activity title on the left side and a floating menu on the right side. This floating menu should link to the most important actions in the context of the app. If these actions can not be all displayed in the Action Bar, an *overflow* menu is added to fit the actions necessary. Besides these two main components, the Action Bar should also include on the left side of the app (also on the left of the title), an Up caret when not in the main screen of the application (later review in Up Pattern 4.1.3), or the button for the side drawer when one exists. In Figure 4.2 there is an example of the Action Bar with its components described above.

The goal of this test is to:

- Check if the app bar (Action Bar) is present in the AUT screen as described above;
- Check if the existing Action Bar components follows the rules stated (Title and floating menu).

Like in the Background Pattern, considering that the Action bar should exist in the application, there is no need to check for the UI Pattern.

The corresponding Test Pattern is:

Goal: "Action Bar exists in the app screen"

V: {}

A: ["observation"]

C: {"is Action Bar present"}

P: {"AUT is open && TP not applied to current activity"}

4.1.3 Up Pattern

In line with the previous pattern, the Up Pattern and the Up Action is a very specific part of the Action Bar and also one of the fundamental ways of navigation in Android applications, along with the back pattern (in Section 4.1.4).

According to the guidelines of good practices in Android developing, every screen in an application which are not the home screen of the app, should offer the user a way to navigate to the screen that is its logical parent, in the hierarchy of the application, by pressing the up button present in the action bar. This gives the user an easy way to follow his/her path backwards on the application right until the app's main screen.

The main goals in testing the Up pattern are to:

- Verify if, in every screen different from the home screen of the app, the action bar is present and that, except in the existence of a Side Drawer, there is an Up button;
- Check if the up button, when clicked, sends the app to the current screen's logical parent in the hierarchy.

A sample of the way the up button is presented in applications is shown in Figure 4.3.

The UI Pattern of the test is:

Goal:"Exists Up Pattern"

V: {}

A: []

C: {"Has Action Bar && Does not Exist Side Drawer"}

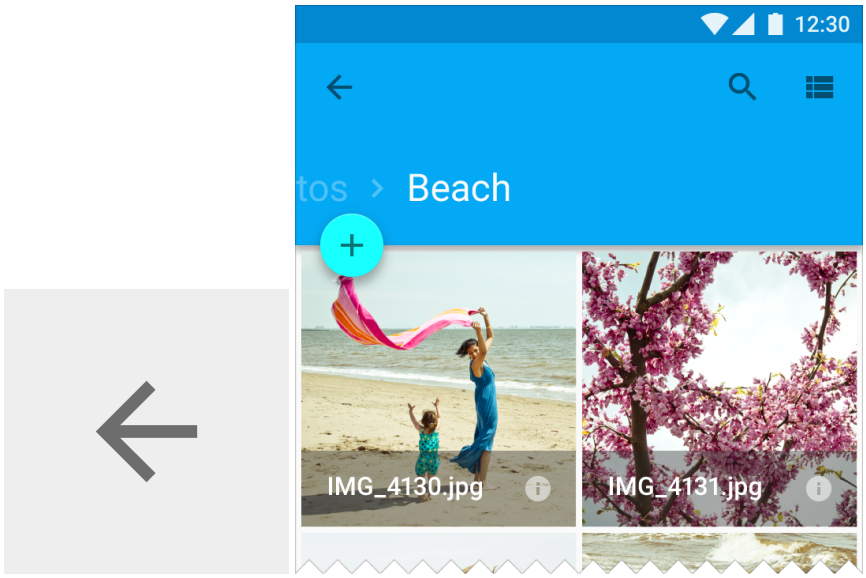

Figure 4.3: Up Button on the left of Android Action Bar

P: {"Not home screen" }

The corresponding Test Pattern is:

Goal: "Exists Up button and goes to parent screen"

V: {}

A: ["observation", click on the up button, "observation"]

C: {"App goes to the logical parent screen"}

P: {"UIP present && TP not applied to current activity"}

The up button is one way of navigating backwards in an application, in this case, in the hierarchy of the application and not necessarily the immediate previous screen. For this case, in contrast, there is the back button, described in the following section.

4.1.4 Back Pattern

In all Android devices a back button is provided to correctly use the back navigation. This type of navigation is the way users have to go backwards on the history of screens previously visited by them, regardless of other state. This type of navigation is also known as temporal navigation and should be ensured to respect the Android conventions.

As stated in Android best practices, the application itself should not add a back button to the UI as this button is already existent in all Android devices and, therefore, the only one that should be used for this kind of action and navigation.

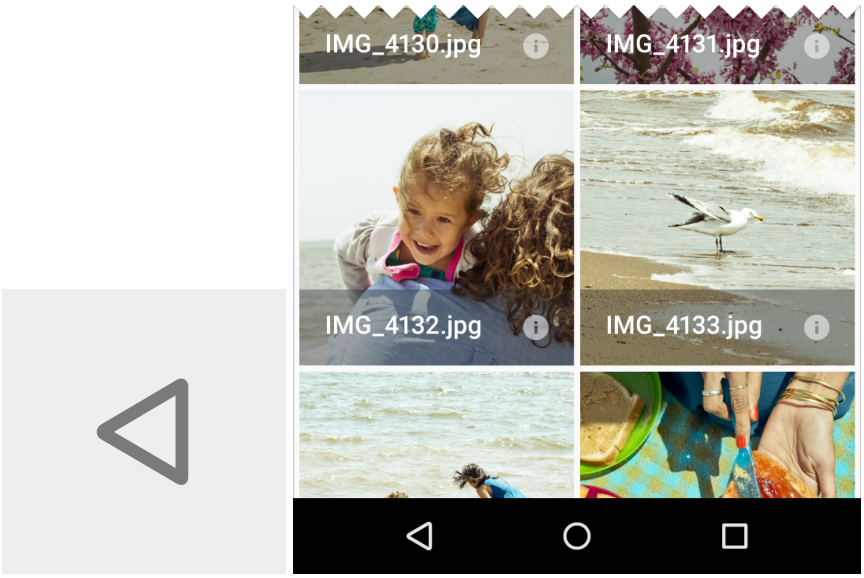

Figure 4.4: Back Button on Android devices

In most situations, the system will handle this type of navigation by using the *back stack*. However, it is possible to override this and manually specify what the back behaviour should be, in order to give the user the best experience possible. This is possible in situations like:

- Certain cases when navigating between fragments;
- When the users enters through a notification, navigation drawer or app widget;
- Navigating in pages through a *WebView*.

Even so, once this behaviour is no longer in use, the app should use the system default *back stack* once again.

In Figure 4.4 there is an example of the back button provided in the Android devices.

So in short, the Back Pattern tests if:

- The AUT uses the default back button navigation of the system and not a personalized back navigation;
- Checks if the AUT is not on the initial screen (screen 0, or else the exploration would finish immediately and there would be no point in executing the test);
- Checking that by clicking on the back button, the AUT changes to the previous visited screen.

As stated before, the goal is to test if the application uses correctly the back button from the device. Due to this, there is also no need to verify if the UI Pattern exists, since it is an integrated part of the device and thus, it always exists.

The corresponding Test Pattern is:

Goal: "Back goes to previous screen"

V: {}

A: ["observation", execute back, "observation"]

C: {"Changed screen to the one immediately visited before "}

P: {"Not the first screen visited && TP not applied to current activity"}

Up Pattern vs. Back Pattern

It is important to understand the differences between the up button and the back button. Since Android 3.0 version that the back button is a part of the Android system itself and therefore, present in all devices. So this made possible to create two different types of navigation on the applications (ancestral and temporal), making more clear for the user what behaviour they should expect when going back on their actions.

The up button is an ancestral type of navigation, which means that it navigates according to the hierarchy of the application and it is only for the app itself. This means that when achieving the screen that is on top of the hierarchy (the app's main screen), there should not be an up button present, since there is no more screens above on the hierarchy.

As for the back button of the device, a temporal type of navigation, its behaviour is to navigate on a chronological way through the history of screens visited by the user in the device.

It is common for the up button and the back button to have the same final result, if the previous screen is also the superior hierarchical of the current screen. However, the up button guarantees that the user stays inside the application and the back button, on the other hand, allows the user to leave the application to the device's home screen or even another application previously visited.

To better understand these differences, a pratical example is visible in Figure 4.5. In this figure, there is an example of navigation in an Android application. From the screen *Book List*, the user clicks on one of the list items, which takes him/her to the *Book 1 details* screen. From this screen, the user has several options. If he decides to press the *Back Button* on the device menu, it will take him/her to the previous screen visited, in this case, the *Book List* Screen. If he/she decides to press the *Up Button*, it will take the user to the parent screen. In this case, also the *Book List* screen, the same as the back navigation. Or he/she can continue navigating to related screens, like the *Book 2 details* screen. The differences start here. On hitting the *back button*, the user will go back to the *Book 1 details* screen. If he/she decides to instead hit the *up button*, then it will go to its parent - the *Book list* screen. On the other hand, going to the parent screen does not need necessarily to be a screen previously visited before. Proof of this, is that it is possible to go to *Movie 1 details* screen from the *Book 2 details* screen, and then, by pressing the *up button*, go to the *Movie 1 details* parent, the *Movie list* screen, which was not previously visited by the user.

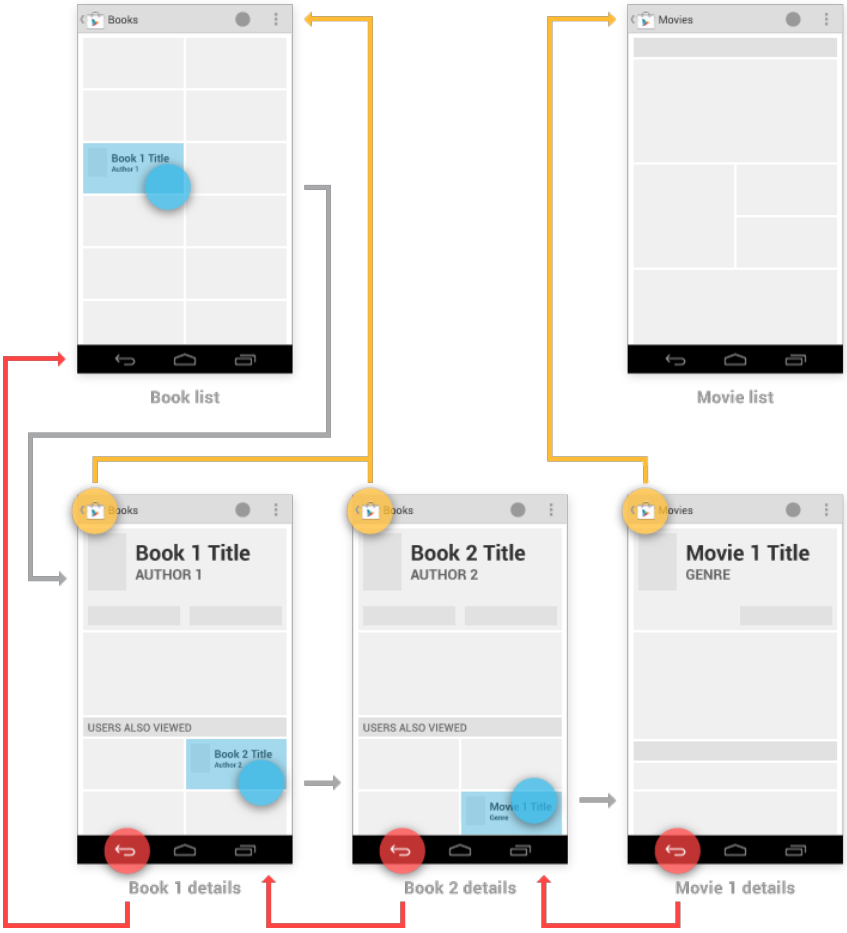

Figure 4.5: Up button behaviour vs. Back button behaviour

4.2 Summary and Conclusions

On a review on the work developed during this dissertation, it was possible to add four new patterns to the iMPAcT tool.

As mentioned in Chapter 1, the patterns added were chosen according to the following requirements:

- Not be already implemented;
- Be a native characteristic of Android applications and/or system;
- Be possible to implement, *i.e.* passable of being fully tested by the tool, not in need of external input;
- Be an embracing pattern amongst Android applications, *i.e.* be present in a varied number of applications and not in a very few number of them;
- Contribute to enhance and improve the quality and coverage of the iMPAcT tool's testing results.

So, in the end, the choice fell on the **Background Pattern**, the **Action Bar Pattern**, the **Up Pattern** and the **Back Pattern**.

We believe that these aforementioned patterns, once implemented in the iMPAcT tool, bring a good contribution to increase the behaviour that the tool can test and that they test patterns present in a huge majority of the existent Android applications, specially for its characteristics, for which some of them are deeply involved with the Android system itself and therefore, very important that these applications follow the guidelines and behave as expected so not to interfere both with the system and with the user experience.

In short, applications working in background should keep their state, in order to, when brought to foreground, the user knows exactly where in the app he/she is and what it is doing. The Action Bar should exist at all times and have the same characteristics for all applications as to facilitate the user experience and learning of the application. The up button should be present in the Action Bar whenever the user is not on the *home screen* of the app, so he/she can have an easy way of going back/up in the app's hierarchy. The back navigation should always send to the previous screen and use the system stack.

In order to validate the implementation of these patterns, a set of experiments was conceived and performed. The results to these experiments can be found in the next chapter, Chapter 5.

Chapter 5

Experiments

This chapter aims to show the results of the experiments made on the new patterns added to the tool in order to validate the new developments and changes to the iMPAcT tool.

It will be presented in Section 5.1 the research questions. On Section 5.2, it is presented the technical specifications of the device and the environment of the experiments made with the tool.

As for the results and quality of those experiments, Sections 5.4 and 5.5 follow in this chapter, with the requirements in choosing the applications that would go under testing, the final list of applications to test, as well as the results of these experiments.

Finally, research questions are answered and there are drawn conclusions, which will also be reviewed, among other conclusions, in Chapter 6.

5.1 Research Questions

In this section, it is presented the research questions raised within the development of this work. Their goal is that, by the end of these experiments, their answers provide a clear vision and conclusion as to whether the results achieved with this work are satisfactory or not and if the contribute to the improvement of the iMPAcT tool is reached. The answer to these following questions (RQ) will be held in the next Chapter 6.

RQ1 - Is the iMPAcT tool able to detect failures in Android applications based on the new patterns of the tool's catalogue?

RQ2 - Are the existing failures in the applications properly identified by the iMPAcT tool?

RQ3 - Are the failures detected by the iMPAcT tool actual failures of the applications?

The answers to these questions can be found in Chapter 6, according to the results of the experiments held that are described in the next sections of this chapter.

On a side note, it is important to underline that the main goal of the iMPAcT tool is to detect failures in applications, but the non detection of failures does not mean that the AUT is free of having them.

5.2 Technical Specification

For the experiments conducted in this Dissertation and presented and analysed in this chapter, the iMPAcT tool was run on an Android device - Wiko U Feel Prime -, with the following technical specifications:

- Operating System: Android 6.0.1
- CPU: Octa-Core 1.4 GHz, Cortex-A53
- Chipset: Qualcomm Snapdragon 430 MSM8937
- GPU: Adreno 505
- RAM: 4GB

The characteristics of the *smartphone* used to test are the only ones relevant since the tool runs solely on the Android device. The specifications of the computer used to connect the *smartphone* are not relevant in this case.

5.3 Test Methodology

In order to assess the most reliable results possible, it is important to define the methodology to use in these experiments. To achieve this, it was defined a plan of testing and a selection of applications to go under testing.

So, the steps followed during the experiments were the following:

1. Select the applications to be tested by the iMPAcT Tool.
2. Run the tool for each of the patterns to be tested.
3. Collect the results of the test, *i.e.* failures found by the tool.
4. Perform a manual inspection on the applications tested to assess the existent failures.
5. Compare both results (from the tool inspection and the manual inspection).
6. Draw conclusions on the results taken from the whole process.

Taking the first step in the experiment, and considering the massive amount of applications available in the *Play Store*, as stated previously in Chapter 2, Section 2.1, it was imperative to make a selection that was both consistent and widely ranged.

The first step of the selection was to select five categories of applications available in the Store: **Books & Reference**, **Food & Drink**, **Sports**, **News & Magazines** and **Health & Fitness**. These categories conferred a range of different features and aspects amongst the applications tested in the end. From these five categories of applications, there were selected three applications for each category, making a total of fifteen applications.

The selection of the applications was executed in April 2017 and later reviewed and updated in June 2017, to assure the novelty and up-to-date state of the applications under testing.

The requirements each application had to meet in order to be selected were:

- Run on a Wiko Ufeel Prime with Android 6.0.1;
- Be free;
- Be a native Android application;
- Not depend on other applications;
- Not require login to interact;
- Not require access to the device's camera, contacts or documents;
- Have more than five hundred thousand downloads;
- Possible to be read by *uiautomatorviewer*;
- Has at least one of the following characteristics:
  - Works in background;
  - Has present the Action Bar;
  - Has present the Up Button;
  - Responds to the device's Back Button.

The final selection of applications to be tested can be found in Table 5.1.

To have reliable results, due to the fact that the events fired during the exploration are random and, consequently, the order of the events fired might change the outcome and results of the test experiment, each result for the experiments conducted in this chapter is the average of the three results, since that, for each application, the iMPAcT tool was run three times. This makes it possible to diminish the randomness of the tool's exploration.

It is also important to notice that, in order to reach the best test coverage on each application, *i.e.* test the biggest number of screens possible, the exploration algorithm chosen was *Priority to Not Executed*.

Table 5.1: Final Selection of Applications to be Tested

Application	Version
Books & Reference	
Wattpad	6.47.1
Sky Map	1.9.2
Wikipedia	2.6.198
Food & Drink	
myTaste	2.3.5
Kitchen Stories	6.4.1A
Receitas de Culinária	8.63
Sports	
Placard	1.1.1
Onefootball	9.6.0
MSN Desporto	1.2.0
News & Magazines	
BBC News	4.3.0.21
BuzzFeed	5.9.1
CNN	5.2
Health & Fitness	
Desafio Fitness 30 dias	1.0.33
Lifesum	5.1.1
Calm	3.4.1

5.4 Detecting Failures

In this section, the goal is to make an experiment that answers the **RQ1** (*Is the iMPAcT tool able to detect failures in Android applications based on the new patterns of the tool's catalogue?*). In order to do this, a simple experiment was conducted, testing five applications, one for each of the categories and the taken from the final selection of applications from Table 5.1.

All four patterns developed in this work were considered, so for all five applications chosen, all four patterns were tested, one at a time.

So in the end of this experiment, it should be possible to understand if the tool is able to both identify the patterns present in the applications and detect failures when existent.

The results for this first experiment can be depicted in Table 5.2. In this table, the results are shown as: DT - Detected Failure; NF - Not Failure (Does not exist). As all the new patterns should be present in all Android applications, all patterns apply to all applications tested.

Analyzing the reports from each of the applications tested, it is possible to conclude that:

1. **Sky Map**: The Sky Map application is a good example of applications that do not follow most of the best practices. One of the major problems found was that, even though an Action Bar exists and it is recognized by the iMPAcT tool, most of the times it is hidden from the user (most importantly, it is hidden on the very first screen, obliging the user to touch the screen to be able to see it). This goes against the best practices since it is not intrinsic to

Table 5.2: Results of Experiment 1

Application	Background	Action Bar	Up	Back
Sky Map	DF	DF	DF	DF
Kitchen Stories	NF	DF	NF	NF
Placard	NF	DF	DF	NF
BuzzFeed	NF	NF	DF	DF
Calm	NF	DF	DF	NF

the user to find it. This also works for the Up button and also for the back button, since the device menu is also hidden, which forces the user to touch the screen in order to see the back button, only being able to press it after this action.

As for the background, the tool detected that before going to background, when the action bar and other tools of the application were available, disappeared once the application was brought to foreground again, as depicted in Figure 5.1.

2. **Kitchen Stories**: Action Bar is not correctly implemented. In a few screens it is missing or not correctly identified, as in some screens the elements usually present in an action bar are present in the top of the screen, but no bar appears. In the main screen instead of an action bar with a search icon, there is a search bar, which does not comply with the best practices for Android applications.

3. **Placard**: As for Placard application, one of its major problems it is not presenting an Action Bar throughout the entire application, as well as an up button in most of the screens, showing only a little bar on top of the screen with the app name and a refresh button, as seen in Figure 5.2. In one of the screens, there is an arrow similar to the up button that mimics its action, but besides not being present in all the screens (except for the home screen), it is not present in an action bar.

   The absence of an up button makes the user only be able to use the back button to navigate backwards, losing the possibility of navigating back in the hierarchy.

4. **BuzzFeed**: Up button is sometimes missing in a few screens and Back button sometimes does not behave as expected.

5. **Calm**: The Calm application is missing in some screens the Action Bar, as well as the up button. On a side note, sometimes, the buttons are not clearly visible to the user.

Considering these results, the next Section 5.5, presents another experiment, that aims to answer the other two research questions presented in Section 5.1.

5.5 Quality of Results

In this section, after knowing that the iMPAcT tool is able to detect failures, like shown in the previous Section 5.4, the goal is to understand if these detected failures are actual failures

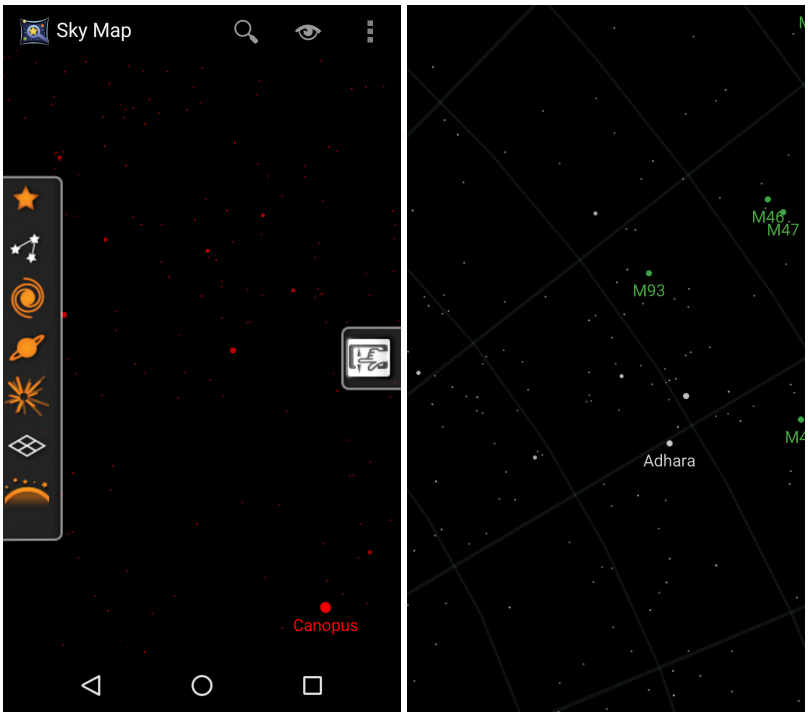

Figure 5.1: Different Screens on the Sky Map Background Test. On the left, screen before background. On the right, screen back to foreground

(**RQ3** - *Are the failures detected by the iMPAcT tool actual failures of the applications?*) and if the existing failures in the application are properly identified by the tool (**RQ2** - *Are the existing failures in the applications properly identified by the iMPAcT tool?*).

In this second experiment, as aforementioned in the test methodology in Section 5.3, besides the test performed by tool, it is also performed a manual inspection, in order to analyze if the results given by the tool are correct and, therefore, reliable.

Taking this into account, the experiment conducted and analysed in this section is, in short, an analysis of the true and false positives and true and false negatives. This means that:

- A true positive means that a failure detected by the iMPAcT tool is an actual failure - the tool works correctly;
- A false positive means that a failure detected is not actually a failure of the application;
- A false negative is when the iMPAcT does not detect what it is a failure;
- Finally, a true negative means that the tool does not detect a failure when it does not exist one - the tool works correctly.

The results for each experiment of each application present in the final list (Table 5.1, can be found in Appendix A. In this appendix, there is a table for each application and pattern with the count of true/false positives/negatives taken from the experiment. An example of those tables can be found in Table 5.3.

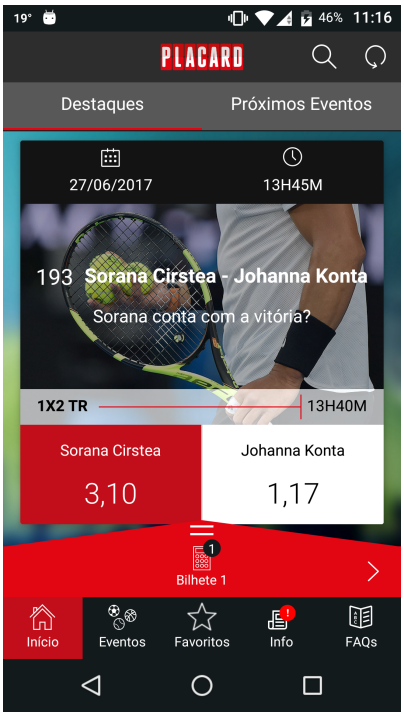

Figure 5.2: Placard screen - The Action Bar is not correctly implemented according to the Best Practices

Table 5.3: Definition of the counting table of positives and negatives

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	True Positive	False Negative
	Is Not Failure	False Positive	True Negative

Once again, in order to minimize the randomness of the exploration, each application was run three times, and the results presented are the average of all three explorations.

On a side note, since the goal is to find failures, the count of true negatives is not taken into account in these experiments.

The final results on the new patterns added (*Background*, *Action Bar*, *Up* and *Back*), can be analized in Tables 5.4, 5.5, 5.6 and 5.7.

Table 5.4: Final Results for the Background Pattern

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	1	2
	Is Not Failure	0	-

Table 5.5: Final Results for the Action Bar Pattern

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	52	0
	Is Not Failure	3	-

Table 5.6: Final Results for the Up Pattern

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	34	8
	Is Not Failure	2	-

Table 5.7: Final Results for the Back Pattern

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	7	0
	Is Not Failure	2	-

5.6 Conclusions

In this chapter, it was presented two different experiments. The first one, a more simple one, aimed to verify if the iMPAcT tool was able to detect failures for the new patterns added, therefore, to answer the first research question, presented at the beginning of the chapter. As for the other two questions, in order to answer them, a more complete set of tests was conducted, and its results are expressed in this chapter in Section 5.5 and also, a more detailed version of the results, can be found in Appendix A.

The answers to the questions raised, a brief discussion and overview on the experiments can be found in the next chapter, Chapter 6, along with the conclusions for the work performed in this Dissertation and the results achieved.

Chapter 6

Discussion and Conclusions

In the last chapter (Chapter 5), there were research questions raised, which answers aim at validate the work developed in this Dissertation.

In this final chapter, those same questions are answered and discussed in Section 6.1.

Finally, as a way to summarize all items previously exposed, the conclusions wrap up the Dissertation in Section 6.2.

6.1 Discussion

6.1.1 RQ1 - Is the iMPAcT tool able to detect failures in Android applications based on the new patterns of the tool's catalogue?

The goal to this question was to make an experiment that was able to answer it and show that the iMPAcT tool is capable of testing the new patterns added.

By the experiment conducted in Chapter 5, Section 5.4, we can easily conclude that the new patterns were added successfully and that the iMPAcT tool is, indeed, capable of identifying and detect failures for all four new patterns (*Background*, *Action Bar*, *Up* and *Back*).

6.1.2 RQ2 - Are the existing failures in the applications properly identified by the iMPAcT tool?

This question can be answered by the experiment conducted in the Section 5.5.

By analysing the results of this experiment, it is possible to say that the iMPAcT is able to identify a good percentage of errors existent in the applications. However, this depends on the pattern and the percentage of identification may vary from pattern to pattern.

Looking more closely to the results in Appendix A, we can say that:

- Most errors not detected in the Up Pattern are due to the tool not being able to identify an Action Bar;

- The error in the Background Pattern not detected is due to the work of a timer that starts over when coming to the foreground, which the iMPAcT tool is not able to identify as it is just a normal component of the screen like any other.

6.1.3 RQ3 - Are the failures detected by the iMPAcT tool actual failures of the applications?

The experiment from Section 5.5 can also answer this question. We can say that the results are positive, and that the iMPAcT tool is a reliable one. Most of the failures detected by the tool that are not actually failures, happen because of other characteristics of the application and its implementation and navigation form, not directly related to patterns under testing by the iMPAcT, which makes it, most of the time, difficult to avoid.

6.2 Conclusions

In the work performed in this Dissertation, it has become obvious, that the field of testing and the market for mobile applications, and more specifically, the Android applications, have become huge, becoming a market of millions in the last couple of years and that keeps growing at an exponentially rate.

So it is very difficult to grasp all that it is already out there about this issue. It is needed a constant research, since we are talking about such a metamorphosing market and field.

Almost every day now, there are new information, new approaches, new tools, new devices, new development ways, a bit of a new everything.

Yet, the solution presented that was continued in this Dissertation, tries to fulfill a need that has not yet been completely eased. The need to automate testing in its full range, diminishing time and cost for companies and developers when assuring the quality of their applications.

As expected by the end of this Dissertation, the iMPAcT tool has improved its quality of testing Android applications, by having new patterns in its catalogue. This increases the behaviour the tool is able to test and also the test coverage of Android applications.

The work achieved might even be possible to grow more, to be continued and improved with even more patterns and better visualization. It may even give space to new approaches that might help in guaranteeing that the applications that reach us on a daily basis are what we expect them to be, the best app possible. Simple, quick and not failing.

References

[AAF+14] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Ugo Gentile, Gianluca Mele, Roberto Nardone, Valeria Vittorini, and Stefano Marrone. Improving code coverage in android apps testing by exploiting patterns and automatic test case generation. In *Proceedings of the 2014 International Workshop on Long-term Industrial Collaboration on Software Engineering*, WISE '14, pages 29–34, New York, NY, USA, 2014. ACM.

[AC13] Andrea Avancini and Mariano Ceccato. Security testing of the communication among android applications. In *Proceedings of the 8th International Workshop on Automation of Software Test*, AST '13, pages 57–63, Piscataway, NJ, USA, 2013. IEEE Press.

[AFT+12] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[AMM15] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 83–93, New York, NY, USA, 2015. ACM.

[AN13] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 641–660, New York, NY, USA, 2013. ACM.

[Anc] Google android developers documentation - providing ancestral and temporal navigation. https://developer.android.com/training/design-navigation/ancestral-temporal.html. Last Accessed: 2017-06-19.

[And] Google android developers documentation. https://developer.android.com/index.html. Last Accessed: 2017-06-19.

[ANHY12] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

[Ant] Android Developers android design in action: Navigation anti-patterns. https://www.youtube.com/watch?v=Sww4omntVjs. Last Accessed: 2017-06-17.

[Appa] Google android developers documentation - setting up the app bar. https://developer.android.com/training/appbar/setting-up.html. Last Accessed: 2017-06-19.

[appb] JS Foundation appium - automation for tests. http://appium.io/. Last Accessed: 2017-06-22.

[Bac] Google android developers documentation - running in a background service. https://developer.android.com/training/run-background-service/index.html. Last Accessed: 2017-06-19.

[BB16] Young-Min Baek and Doo-Hwan Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 238–249, New York, NY, USA, 2016. ACM.

[BHC+11] Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, MALWARE '11, pages 66–72, Washington, DC, USA, 2011. IEEE Computer Society.

[BkN] Google android developers documentation - providing proper back navigation. https://developer.android.com/training/implementing-navigation/temporal.html. Last Accessed: 2017-06-19.

[cal] Xamarin calabash - automated acceptance testing for mobile apps. http://calaba.sh/. Last Accessed: 2017-06-22.

[CNS13] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 623–640, New York, NY, USA, 2013. ACM.

[Cop17] Riccardo Coppola. Fragility and evolution of android test suites. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 405–408, Piscataway, NJ, USA, 2017. IEEE Press.

[CPN14] P. Costa, A.C.R. Paiva, and M. Nabuco. Pattern based gui testing for mobile applications. pages 66–74, 2014. cited By 6.

[DMAO15] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt. Towards mutation analysis of android apps. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10, April 2015.

[esp] Google Android testing apps on android - espresso. https://developer.android.com/training/testing/espresso/index.html. Last Accessed: 2017-06-22.

[HN11a] Cuixiong Hu and Iulian 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.

[HN11b] Cuixiong Hu and Iulian Neamtiu. A gui bug finding framework for android applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1490–1491, New York, NY, USA, 2011. ACM.

[Imp15] Gennaro Imparato. A combined technique of gui ripping and input perturbation testing for android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 760–762, Piscataway, NJ, USA, 2015. IEEE Press.

[KMPK06] Antti Kervinen, Mika Maunumaa, Tuula Pääkkönen, and Mika Katara. Model-based testing through a GUI. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3997 LNCS, pages 16–31, 2006.

[LVWBC+15] Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys 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.

[Mat] Google material design documentation. https://developer.android.com/develop/quality-guidelines/core-app-quality.html. Last Accessed: 2017-06-19.

[MDFE12] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito. Software testing of mobile applications: Challenges and future research directions. In *Proceedings of the 7th International Workshop on Automation of Software Test*, AST '12, pages 29–35, Piscataway, NJ, USA, 2012. IEEE Press.

[MEK+12] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. A whitebox approach for automated security testing of android applications on the cloud. In *Proceedings of the 7th International Workshop on Automation of Software Test*, AST '12, pages 22–28, Piscataway, NJ, USA, 2012. IEEE Press.

[MLVP17] Kevin Moran, Mario Linares-Vásquez, and Denys Poshyvanyk. Automated gui testing of android apps: From research to practice. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 505–506, Piscataway, NJ, USA, 2017. IEEE Press.

[MP13] T. Monteiro and A.C.R. Paiva. Pattern based gui testing modeling environment. pages 140–143, 2013. cited By 4.

[MP14a] R. M. L. M. Moreira and A. C. R. Paiva. A gui modeling dsl for pattern-based gui testing paradigm. In *2014 9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 1–10, April 2014.

[MP14b] Rodrigo M.L.M. Moreira and Ana C.R. Paiva. Pbgt tool: An integrated modeling and testing environment for pattern-based gui testing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 863–866, New York, NY, USA, 2014. ACM.

[MP15a] I.C. Morgado and A.C.R. Paiva. The impact tool: Testing ui patterns on mobile applications. pages 876–881, 2015. cited By 1.

[MP15b] I.C. Morgado and A.C.R. Paiva. Test patterns for android mobile applications. volume 08-12-July-2015, 2015. cited By 0.

[MP15c] I.C. Morgado and A.C.R. Paiva. Testing approach for mobile applications through reverse engineering of ui patterns. pages 42–49, 2015. cited By 0.

[MP16] Inês Coimbra Morgado and Ana C.R. Paiva. Impact of execution modes on finding android failures. *Procedia Computer Science*, 83:284 – 291, 2016. The 7th International Conference on Ambient Systems, Networks and Technologies (ANT 2016) / The 6th International Conference on Sustainable Energy Information Technology (SEIT-2016) / Affiliated Workshops.

[MPF12] Inês Coimbra Morgado, Ana CR Paiva, and João Pascoal Faria. Dynamic reverse engineering of graphical user interfaces. 2012.

[MPF14] I.C. Morgado, A.C.R. Paiva, and J.P. Faria. Automated pattern-based testing of mobile applications. pages 294–299, 2014. cited By 3.

[MPM13a] R.M.L.M. Moreira, A.C.R. Paiva, and A. Memon. A pattern-based approach for gui modeling and testing. pages 288–297, 2013. cited By 13.

[MPM13b] Rodrigo M L M Moreira, Ana C R Paiva, and Atif Memon. A pattern-based approach for GUI modeling and testing. In *2013 IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013*, pages 288–297, 2013.

[Nil09] Erik G. Nilsson. Design patterns for user interface for mobile applications. *Adv. Eng. Softw.*, 40(12):1318–1328, December 2009.

[PFTV05] Ana C R Paiva, Jo??o C P Faria, Nikolai Tillmann, and Raul A M Vidal. A model-to-implementation mapping tool for automated model-based GUI testing. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3785 LNCS, pages 450–464, 2005.

[PPG17] Fernando Paulovsky, Esteban Pavese, and Diego Garbervetsky. High-coverage testing of navigation models in android applications. In *Proceedings of the 12th International Workshop on Automation of Software Testing*, AST '17, pages 52–58, Piscataway, NJ, USA, 2017. IEEE Press.

[Qua] Google android developers documentation - core app quality. https://developer.android.com/develop/quality-guidelines/core-app-quality.html. Last Accessed: 2017-06-19.

[rob] Robotium robotium. https://robotium.com/. Last Accessed: 2017-06-22.

[sel] Selendroid selenium for android. http://selendroid.io/. Last Accessed: 2017-06-22.

[SI16] Ibrahim Anka Salihu and Rosziati Ibrahim. Systematic exploration of android apps' events for automated testing. In *Proceedings of the 14th International Conference on Advances in Mobile Computing and Multi Media*, MoMM '16, pages 50–54, New York, NY, USA, 2016. ACM.

[SMT16] Jose Lorenzo San Miguel and Shingo Takada. Gui and usage model-based test case generation for android applications with change analysis. In *Proceedings of the 1st International Workshop on Mobile Development*, Mobile! 2016, pages 43–44, New York, NY, USA, 2016. ACM.

[SSHS+13] Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjörg Schmauder. Insights into layout patterns of mobile user interfaces by an automatic analysis of android apps. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 275–284, New York, NY, USA, 2013. ACM.

[Staa] Statista number of apps available in leading app stores as of march 2017. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. Last Accessed: 2017-06-21.

[Stab] Statista number of available applications in the google play store from december 2009 to march 2017. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. Last Accessed: 2017-06-22.

[Su16] Ting Su. Fsmdroid: Guided gui testing of android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 689–691, New York, NY, USA, 2016. ACM.

[uia] Google Android testing apps on android - ui automator. https://developer.android.com/training/testing/ui-automator.html. Last Accessed: 2017-06-22.

[UL07] Mark Utting and Bruno Legeard. Practical model-based testing: a tools approach. *Book*, page 433, 2007.

[UpA] Google android developers documentation - adding an up action. https://developer.android.com/training/appbar/up-action.html. Last Accessed: 2017-06-19.

[UpN] Google android developers documentation - providing up navigation. https://developer.android.com/training/implementing-navigation/ancestral.html. Last Accessed: 2017-06-19.

[YPX13] Wei Yang, Mukul R. Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 250–265, Berlin, Heidelberg, 2013. Springer-Verlag.

[YT08] Siena Yu and Shingo Takada. External event-based test cases for mobile application. In *Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering*, C3S2E '15, pages 148–149, New York, NY, USA, 2008. ACM.

[YT16] Siena Yu and Shingo Takada. Mobile application test case generation focusing on external events. In *Proceedings of the 1st International Workshop on Mobile Development*, Mobile! 2016, pages 41–42, New York, NY, USA, 2016. ACM.

[ZLZ+16] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. Automated test input generation for android: Are we really there yet in an industrial case? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 987–992, New York, NY, USA, 2016. ACM.

[ZYZS15] H. Zhu, X. Ye, X. Zhang, and K. Shen. A context-aware approach for dynamic gui testing of android applications. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 248–253, July 2015.

Appendix A

Results of the Experiments

A.1 Quality of Results

Books & References

Wattpad

Table A.1: Results for the Background Pattern for the Wattpad application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.2: Results for the Action Bar Pattern for the Wattpad application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.3: Results for the Up Pattern for the Wattpad application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	4	0
	Is Not Failure	0	-

Table A.4: Results for the Back Pattern for the Wattpad application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	1	-

SkyMap

The inspection for this application is quite quick, since this is a very simple application.

Table A.5: Results for the Background Pattern for the SkyMap application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	1	0
	Is Not Failure	0	-

Table A.6: Results for the Action Bar Pattern for the SkyMap application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	1	0
	Is Not Failure	0	-

Most of the times the Action Bar is hidden.

Table A.7: Results for the Up Pattern for the SkyMap application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	8	1
	Is Not Failure	0	-

Table A.8: Results for the Back Pattern for the SkyMap application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	1	0
	Is Not Failure	0	-

Wikipedia

Table A.9: Results for the Background Pattern for the Wikipedia application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.10: Results for the Action Bar Pattern for the Wikipedia application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.11: Results for the Up Pattern for the Wikipedia application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	2	0
	Is Not Failure	0	-

Table A.12: Results for the Back Pattern for the Wikipedia application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	2	0
	Is Not Failure	0	-

Food & Drink

myTaste

Table A.13: Results for the Background Pattern for the myTaste application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.14: Results for the Action Bar Pattern for the myTaste application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.15: Results for the Up Pattern for the myTaste application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	1	1
	Is Not Failure	0	-

The tool does not detect the up button in one or two screens because it considers it is the main screen (due to inferior tabs that should lead to a new screen), which is not.

Table A.16: Results for the Back Pattern for the myTaste application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	1	0
	Is Not Failure	1	-

This false positive maybe due to a bug in the tool. The tool triggered the back event, but this did not occur.

Kitchen Stories

Table A.17: Results for the Background Pattern for the Kitchen Stories application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	1
	Is Not Failure	0	-

There was one minor error in the background behaviour that was not detected by the tool. Besides that, no errors were found.

Table A.18: Results for the Action Bar Pattern for the Kitchen Stories application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	22	0
	Is Not Failure	2	-

The false positives may be due to the fact that the Action bar is bigger than usual, and has the tabs embedded in it, which does not follow the guidelines.

Table A.19: Results for the Up Pattern for the Kitchen Stories application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.20: Results for the Back Pattern for the Kitchen Stories application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Receitas de Culinária

Table A.21: Results for the Background Pattern for the Receitas de Culinária application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.22: Results for the Action Bar Pattern for the Receitas de Culinária application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	2	0
	Is Not Failure	0	-

Table A.23: Results for the Up Pattern for the Receitas de Culinária application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	1
	Is Not Failure	0	-

The non detection of the failure is due to the fact that the action bar is not well implemented.

Table A.24: Results for the Back Pattern for the Receitas de Culinária application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Sports

Placard

Table A.25: Results for the Background Pattern for the Placard application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.26: Results for the Action Bar Pattern for the Placard application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	19	0
	Is Not Failure	0	-

Table A.27: Results for the Up Pattern for the Placard application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	9	1
	Is Not Failure	0	-

In one of the screens, there is an up button, but as it is not inserted in a correctly implemented action bar, the tool could not detect it properly.

Table A.28: Results for the Back Pattern for the Placard application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Onefootball

Table A.29: Results for the Background Pattern for the Onefootball application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.30: Results for the Action Bar Pattern for the Onefootball application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.31: Results for the Up Pattern for the Onefootball application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	4	2
	Is Not Failure	0	-

The non detection of the failure is due to other errors in the navigation of the application, not necessarily connected to the tool missing the failure, nor the patterns here tested.

Table A.32: Results for the Back Pattern for the Onefootball application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

MSN Desporto

Table A.33: Results for the Background Pattern for the MSN Desporto application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.34: Results for the Action Bar Pattern for the MSN Desporto application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.35: Results for the Up Pattern for the MSN Desporto application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	2	0
	Is Not Failure	0	-

Table A.36: Results for the Back Pattern for the MSN Desporto application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

News & Magazines

BBC News

Table A.37: Results for the Background Pattern for the BBC News application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.38: Results for the Action Bar Pattern for the BBC News application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.39: Results for the Up Pattern for the BBC News application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.40: Results for the Back Pattern for the BBC News application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	1	0
	Is Not Failure	0	-

BuzzFeed

Table A.41: Results for the Background Pattern for the BuzzFeed application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.42: Results for the Action Bar Pattern for the BuzzFeed application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.43: Results for the Up Pattern for the BuzzFeed application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	3	1
	Is Not Failure	1	-

Table A.44: Results for the Back Pattern for the BuzzFeed application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	1	0
	Is Not Failure	0	-

CNN

Table A.45: Results for the Background Pattern for the CNN application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.46: Results for the Action Bar Pattern for the CNN application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.47: Results for the Up Pattern for the CNN application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.48: Results for the Back Pattern for the CNN application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Health & Fitness

Desafio Fitness 30 dias

Table A.49: Results for the Background Pattern for the Desafio Fitness 30 dias application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.50: Results for the Action Bar Pattern for the Desafio Fitness 30 dias application

		iMPAcT Tool	
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	1	-

The false positive is due to the fact that the application has an opening screen with the logo, which in the new versions of Android applications should not exist.

Table A.51: Results for the Up Pattern for the Desafio Fitness 30 dias application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.52: Results for the Back Pattern for the Desafio Fitness 30 dias application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	1	0
	Is Not Failure	0	-

Lifesum

Table A.53: Results for the Background Pattern for the Lifesum application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.54: Results for the Action Bar Pattern for the Lifesum application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	4	0
	Is Not Failure	0	-

Table A.55: Results for the Up Pattern for the Lifesum application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Table A.56: Results for the Back Pattern for the Lifesum application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-

Calm

Table A.57: Results for the Background Pattern for the Calm application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	1
	Is Not Failure	0	-

In one of the screens, there is a timer that, when sent to background and then brought back to foreground, starts all over again.

Table A.58: Results for the Action Bar Pattern for the Calm application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	4	0
	Is Not Failure	0	-

Table A.59: Results for the Up Pattern for the Calm application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	1	1
	Is Not Failure	1	-

Table A.60: Results for the Back Pattern for the Calm application

	iMPAcT Tool		
		Detects Failure	Does not Detect Failure
Manual Inspection	Is failure	0	0
	Is Not Failure	0	-