FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

An Android GUI Crawler for Test Case Generation and Runtime Analysis

Miguel Bruno Rego Freitas

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

FEUP Supervisor: Ricardo Morla

Vodafone Supervisor: Pedro Nogueira

July 28, 2016

Abstract

In the new era of mobile devices, thousands of applications are developed daily either by companies or self-taught developers. One of their main concerns is, with no doubt, the *User Interface* (UI). The UI is defined as the space where interaction between humans and applications occur. Due to that, applications should be the most user-friendly and easy to use as possible, involving other fields of study such as ergonomics, psychology and design. In this way, before each new mobile applications' release, several manual tests must be conducted to verify if the application isn't prone to errors and robust enough to be available for users. As this is a routinely and error-prone task, that spends a huge amount of time, there is a great and rightful interest to automate it. Related to that, there is also an interest to increase the bug report process' speed and, above all, to assure the product's quality. Currently, there are several tools and frameworks to automate these tests in mobile platforms. However, it is almost always required some device's interaction or having some knowledge about the concerned application, calling for an automated way to reduce the tester's work.

In response to that necessity, this dissertation introduces a new approach in mobile software test automation area. It is proposed a crawler-based and black-boxed solution to run across the Android application's *Graphical User Interface* (GUI) automatically, simulating user's inputs. The main goal of this tool is to try to achieve all possible application's patterns and, based on them, generate test cases for each one. Through time, the data model's representation will be refined and improved and, when all patterns were discovered, tests in natural programming language will be generated automatically. This format is used to be understandable for any person independently of their technical skills. At the same time, some runtime evidences are captured such as: traffic network, video and logs. These facts can be useful for further analysis regarding security, control flow and bug reporting issues. In this study were also included all background knowledge for a better understanding of the final solution, as well as the literature review of this area, software testing. In the end, was still made a tool's evaluation and some conclusions are taken, without forgetting the future work that can be made in order to improve the tool.

Resumo

Na nova era dos dispositivos móveis, milhares de aplicações móveis são desenvolvidas por dia tanto por empresas como por programadores autodidatas. Uma das suas principais preocupações é, sem dúvida, a *Interface do Utilizador* (UI). A UI é definida com o espaço onde a interação entre os humanos e as aplicações ocorrem. Devido a isso, as aplicações devem ser o mais amigáveis e fáceis de usar possíveis, envolvendo outras áreas de estudo tais como a ergonomia, psicologia e *design*. Neste sentido, antes do lançamento de novas aplicações móveis, são necessários conduzir vários testes manuais para verificar se a aplicação é ou não susceptível a erros e robusta o suficiente para ser disponibilizada para os utilizadores. Como esta tarefa é rotineira e susceptível a erros, gastando enormes quantidades de tempo, há um grande e legítimo interesse em automatizá-la. Em relação a isso, há também interesse em aumentar a rapidez do processo de relatório de erros e, acima de tudo, assegurar a qualidade do produto. Atualmente, existem várias ferramentas e *frameworks* que automatizam os testes em plataformas móveis, contudo é quase sempre necessário alguma interação com o dispositivo ou ter algum conhecimento acerca da aplicação em questão, necessitando de uma forma de reduzir o trabalho de um *tester*.

Em resposta a esta necessidade, esta dissertação introduz uma nova abordagem na área de automação de testes de software para plataformas móveis. É proposta como solução um *crawler* do tipo caixa preta para percorrer a *Interface Gráfica do Utilizador* (GUI) de uma aplicação Android automaticamente, simulando as acções de um utilizador. O principal objectivo desta ferramenta é tentar encontrar todos os caminhos possíveis da aplicação e, baseando-se neles, gerar casos de teste para cada um. Ao longo do tempo, o modelo de representação de dados será refinado e melhorado, e, quando todos os caminhos forem encontrados, serão gerados automaticamente testes em linguagem natural. Este formato é usado para ser percebido por qualquer pessoa independente das suas capacidades técnicas. Ao mesmo tempo, são capturadas algumas evidências durante a interação com a aplicação tais como: o tráfego de rede, video e *logs*. Estas evidências podem ser úteis para análise futura tendo em conta questões de segurança, controlo de fluxos e reporte de erros. Neste estudo, é também incluído tanto o conhecimento prévio necessário para melhor compreensão da solução final, como a revisão da literatura para esta área de estudo, testes de software. No final, é ainda feita uma avaliação da ferramenta, algumas conclusões serão tiradas, sem esquecer o trabalho futuro que pode ser feito para melhorar a ferramenta.

Acknowledgements

First and foremost, I want to thank to my supervisors, Professor Ricardo Morla, at FEUP, and Project Manager Pedro Nogueira, at Vodafone, for their guidance and expertise, and for taking the time to keep me focused on the task at hand. I am most grateful for the role they performed in this project.

To my parents, José Freitas and Maria Rego, and my brother, Tiago Freitas, I want to thank their support, caring and patience that helped me along the path I have walked so far. I am fully aware of their sacrifices and it will not go without saying that they made all the difference.

I would also like to express my appreciation to Engineers Rui Maia and Marco Rodrigues, at Vodafone, for their help, motivation, advice, concern and programming knowledge given to me and to this dissertation, in order to make this project as better as possible.

I want to thank to all Customer Operations - Online & Mobile team members, at Vodafone, Pedro Tavares, Rui Eusébio, Susana Banha, Cláudia Simões, Ricardo Santos, Ricardo Silva, Rui Coimbra, Rita Madureira, Sofia Teixeira, Sara Pinto and Mário Correia, for their willingness to provide me a good place to work and learn on this project, for proposing the project itself, for their good reception and integration in the team, and for all funny moments lived during this journey.

To all my friends, Patrícia Magalhães, Hugo Fonseca, Tiago Costa, Juliana Araújo, Ana Andrade, Adriana Simões, André Ferreira, Marisa Agra, Daniela Macedo, António Tomé, Rubén Brito, André Granja, Filipe Morais, Márcia Rodrigues, Guilherme de Sá Pires, Jorge Corujas and Filipe Pena, I want to thank their availability, their caring and their ability to always find the right words or gestures to lift my spirit and to motivate me to do this dissertation.

To all my colleagues, Isabel Fragoso, António Pintor, Ricardo Sousa, André Coelho, Bruno Jorge, Rui Gomes, Wilson Silva, José Valente, Matteo Benčić, I want to thank for all help given to me and for all commitment given to the projects that we worked on.

To all the professors that teach me all things that I know today, thank you for molding me into the student and person that I am today.

Finally, I thank to all the people that have contribute, directly or indirectly, so I could achieve this stage in my life. To all, thank you for taking a part in this journey.

Gratefully,

Miguel Bruno Rego Freitas

"You can mass-produce hardware;
you cannot mass-produce software -
you cannot mass-produce the human mind."
Michio Kaku

Contents

List of Figures

List of Tables

Abbreviations and Symbols

AAPT *Android Asset Packaging Tool*

AC *Activity Coverage*

ADB *Android Debug Bridge*

API *Application Program Interface*

APK *Android Package*

ART *Android Runtime*

AVD *Android Virtual Device*

BDD *Behavior-Driven Development*

BFS *Breadth-First Search*

DDMS *Dalvik Debug Monitor Server*

DEX *Dalvik Executable*

DLL *Dynamic-Link Library*

DSL *Domain-Specific Language*

GROPG *Graphical On-Phone Debugger*

GUI *Graphical User Interface*

HAL *Hardware Abstraction Layer*

HTML *HyperText Markup Language*

HTTP *Hypertext Transfer Protocol*

ID *Identification*

IDE *Integrated Development Environment*

IMSI *International Mobile Subscriber Identity*

IoT *Internet of Things*

IP *Internet Protocol*

JAR *Java ARchive*

JD *Java Decompiler*

JDWP *Java Debug Wire Protocol*

JSON *JavaScript Object Notation*

MSISDN *Mobile Station International Subscriber Directory Number*

MTTC *Mean Time per Test Case*

NAFA *Number of Activities Found Automatically*

NAFM *Number of Activities Found Manually*

NDK *Native Development Kit*

OS *Operative System*

PID *Process Identification*

PIN *Personal Identification Number*

QA *Quality Assurance*

RAD *Rapid Application Development*

REST *Representational State Transfer Protocol*

SDK *Software Development Kit*

SGL *Scalable Graphics Library*

SIM *Subscriber Identity Module*

SOAP *Simple Object Access Protocol*

SQL *Structured Query Language*

SSL *Secure Sockets Layer*

SUT *System Under Test*

TAD *Test-After Development*

TCP *Transmission Control Protocol*

TDD *Test-Driven Development*

TID *Task Identification*

TNA *Total Number of Activities*

UDP *User Datagram Protocol*

UI *User Interface*

USB *Universal Serial Bus*

VM *Virtual Machine*

XML *Extensible Markup Language*

Chapter 1

Introduction

In the age of *Internet of Things* (IoT), more than 2 billion people[1] use smartphones to communicate with each other, to play games, to monitor their lives or to take some advantage of all the potential of having a small computer in their pocket. In response to the users' needs to fulfill this potential, many applications are developed with many different purposes by software companies - in order to improve their clients' customer services - and by many self-taught developers. Before these applications become available in their respective mobile markets, it is viewed as a good practice to submit them under thorough and extensive testing stage, with the goal of verifying if the application works as required or not. This generic phase includes several different kinds of testing such as integration, system, usability and acceptance testing. Generally, this testing phase, depending of the development methodology used, as reported in [4], precedes the deployment into production and all these mentioned tests are included in a certain software *Quality Assurance* (QA) process.

During the testing phase, one of the main concerns of any company building a mobile application is its UI, since it'll be the context in which the interaction between user and application happens and it correlates positively with the customer's usage or payment: the more engaging a UI is, the more amount of time a given customer spends in the mobile application, and the highest the likelihood that a given customer ends up paying for its usage. As reported in [5, 6, 7], before developing a product, companies must be concerned about the UI, demonstrating that software engineering isn't just about engineering itself, but it also involves ergonomics, psychology and design, known as User Experience (UX).

1.1 Motivation

Before each release of new mobile applications it is necessary to test them as much as possible, tracking unreported bugs, verifying if the UI is designed exactly as planned and if the application does what it is supposed to do. Generally, this process is done manually by testers, and using a

[1] Statista. *Number of smartphone users* worldwide from 2014 to 2019 (in millions).* January, 2016. [Accessed: Jan. 29 2016]. URL: http://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

different device at once, requiring high amounts of time for each test and due to that it becomes a tedious task and highly prone to mistakes. Because, in a corporate perspective, time means money for every company, there has been a great interest to automate this task and speed up the bug reporting process. Currently, there are already in the market some tools and frameworks to automate it, however, it is always necessary to have some technical and programming skills, usually related with the native environment or programming language of the mobile application and/or its visual components. Another limitation of the automated testing tools is that they only perform the tests that were previously programmed, which leaves open the possibility that some of the application's patterns and states are left untested. In addition to these facts, some of the existent automated tools do not capture device's and application's evidences during test execution, which would definitely be useful for developers to track and fix errors. Furthermore, the manual specification, implementation and maintenance of each test case requires a great amount of knowledge and time.

1.2 Goals

This work will focus on Android applications because this mobile platform currently has more than 1.4 billion of active users around the world[2] and leads the market share, with wide difference of their main opponent iOS (*Apple*, approximately 15% of market), with 82.8% of sales in the second quarter of 2015[3].

This comprehensive study aims to implement an Android GUI automated testing black-box tool that, with no knowledge of the application's source code, generates and simulates user interactions and triggers transition events, based on the provided dumps by *UIAutomator* tool.

The main goal of this work is try to discover all of the application's patterns automatically, translating them into a tree data model, in order to generate natural language tests, like *Cucumber* [8], to be used by other automated tools as *Jenkins* [4]. At the same time, some evidences should be captured such as screenshots, video records, logs and network traffic, in order to facilitate the finding of errors, bugs and security issues for reporting. The tool should allows the connection of several devices at the same time and provides two connection modes (*Universal Serial Bus* (USB) and Wi-Fi), taking advantage of *Android Debug Bridge* (ADB) capabilities. Furthermore, the tool's modularity should allows the use of each module, crawler, test case generation, network traffic capture or logs capture, separately.

1.3 Structure of Document

This section exposes the structure and organization of this report, with an overview of each chapter.

[2] John Callaham. *Google says there are now 1.4 billion active Android devices worldwide*. AndroidCentral. September, 2015. [Accessed: Jan. 29 2016]. URL: http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide.

[3] International Data Corporation (IDC). *Smartphone OS Market Share, 2015 Q2*. IDC. August, 2015. [Accessed: Jan. 29, 2016]. URL: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[4] *Jenkins*. April, 2016. [Accessed: Apr. 04 2016]. URL: https://jenkins.io/.

In chapter 2, it is made an overview about the software development models and the Android's World, providing a previous knowledge to understand the following chapters.

In chapter 3, it is described the literature review as well as the description of different types of tests, analysis, testing methods and testing automation tools.

The chapter 4 includes all integral specification about the solution and its components, detailing each decision taken to build it.

In chapter 5, the tool is evaluated and it is described the dataset collected, the methodology and the obtained results.

Finally, in chapter 6, it is presented the conclusions that can be taken from this study, explaining its contributions and making some suggestions for future improvements.

Chapter 2

Background

This chapter presents contextual information that is important for a better understanding of the scope of this dissertation. It is presented a description about the most common software engineering processes used nowadays by companies. Furthermore, it is presented the most widely used mobile *Operative System* (OS) in the world, Android, with a description of its architecture and components. At the end, it is also performed a conclusion about them all.

2.1 Software Engineering

Since 1968, as reported in [9], software engineering has been arising ever since, becoming more concentrated in the design of better processes, methodologies and tools to develop innovative and complex systems or technologies. In order to simplify and help to manage software projects, it was created several models to represent the abstraction of some objects and activities involved in software development [10]. These models are also used to improve efficiency and to reduce the time and costs of projects. Essentially, they are divided in three different types, as reported in [3]: sequential, that includes Waterfall and *Rapid Application Development* (RAD) models; evolutionary, that includes the Incremental model, and a mixture of both, that includes the Spiral model. There are another models that can be included in each type, but only the mentioned types will be approached in this dissertation and will be described in the next subsections.

2.1.1 Waterfall Model

As the name says, the Waterfall model is a sequential process, conducting software projects with one phase at a time. This model was initially proposed by Winston Royce in [1], but it isn't the only one. In this paper are presented some models more interactive than Waterfall, although they are characterized as risky and prone to failures. Focusing on Waterfall model, in each software release, the projects needs to go sequentially through every single one of the phases. This model has, in total, seven phases, as it can be seen in figure 2.1, but they are generally grouped in five, resulting in:

- Requirements Phase: functional and non-functional requirements are specified, defining what the system should do and describing how the system should behave, respectively;
- Design Phase: the proposed solution is specified;
- Development Phase: the solution is implemented into a working product;
- Verification Phase: tests and integration of the solution is made;
- Maintenance Phase: it is given the support as necessary to maintain the solution operational.

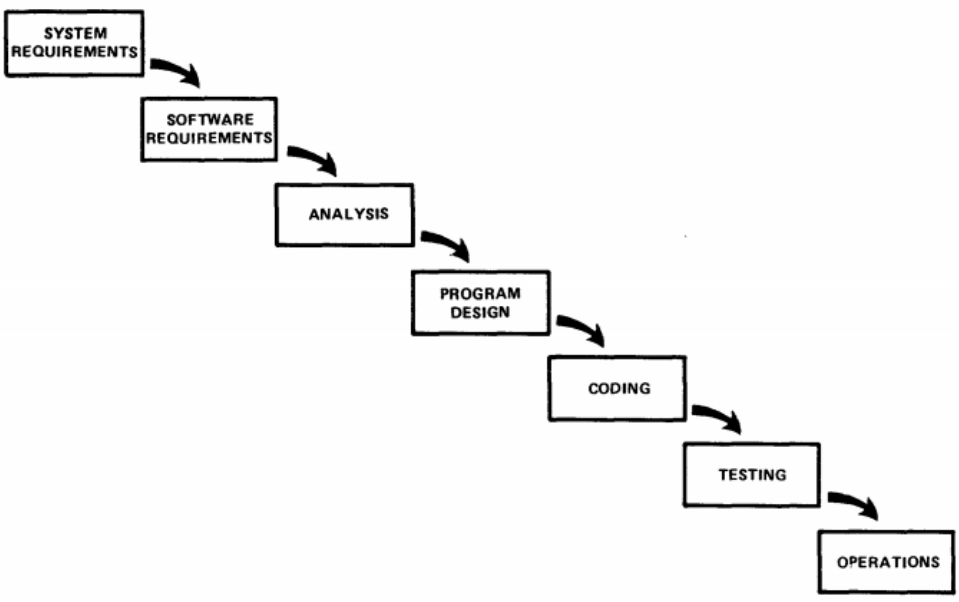


Figure 2.1: The Waterfall model [1].

Although this model seems to fill all the steps of software engineering, it is very static and rarely these steps are fulfilled, because there is almost always an issue, either in design or the requirements before or during the implementation phase. Furthermore, in big projects, the gap between implementation and design is huge, requiring recall and redefinition of the requirements. This problem can generally be corrected by using an iterative Waterfall model or by using some other software model.

2.1.2 RAD Model

The RAD model is a sequential software development process, initially supplied by James Martin in the 1990s [11], to distinguish a model from the traditional used, Waterfall. RAD was designed for short development cycles, usually between 60 and 90 days, and enables the building of a *"fully functional system"* in this period [12]. RAD model takes advantage of teams with diversity of skills and roles, to allow its implementation, as can be seen in figure 2.2. This model is composed, as reported in [2], by the following five phases:

- Business modeling: in this phase all the concerns are taken around the information that drives the business model and its flow, from its generation, through processing, to its destination;
- Data modeling: The flow of information is translated into relations between objects with several attributes;
- Process modeling: in this phase are described and detailed all the processes that will deal with the objects created in the data modeling;
- Application generation: in this phase, this model try to reuse, as far as possible, existent program components and automated tools to ease up the application construction;
- Testing & turnover: RAD reuses other software parts, which are usually already tested, hence reducing the allocated testing time. However, the integration of all the software parts with the new components must be tested themselves, in order to guarantee robustness in the final software product.

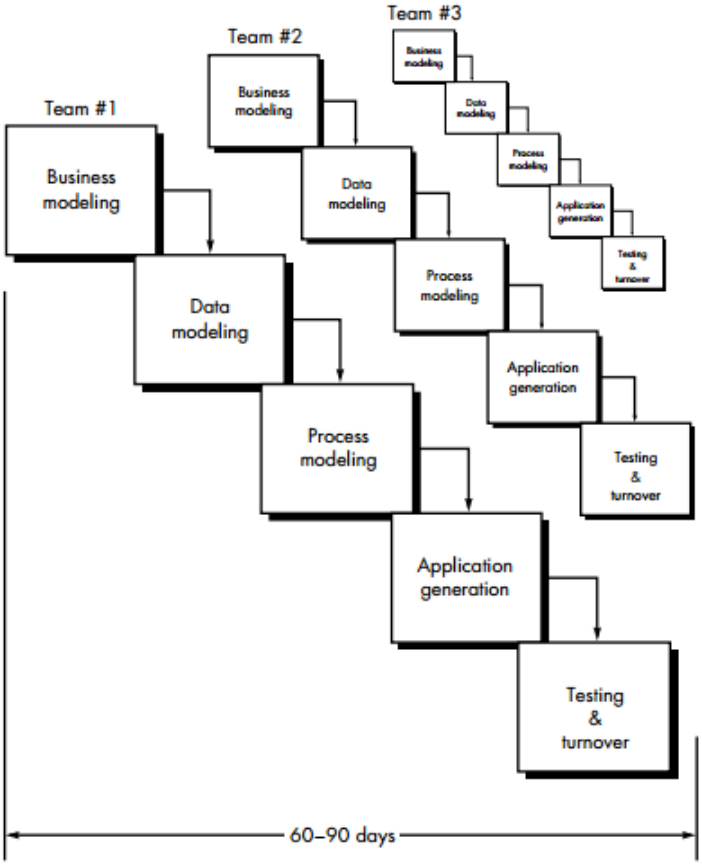


Figure 2.2: The RAD model [2].

This model seems to resolve the problems of the Waterfall model, but it has some *quid pro quos*, as reported in [13]. RAD requires, first of all, a satisfactory number of people to create RAD teams with the right size. Also, it requires the full commitment of the developers and customers to fulfill the project planning. Another of the drawbacks is that RAD requires modularity and if that can't be achieved, it can produce block states and dependencies. Finally, due to short development cycles, sometimes at the end of the project, some extra project requirements can appear, forcing a new development cycle with substantial changes, consuming time and money.

2.1.3 Spiral Model

The Spiral model was initially proposed by Barry W. Boehm at 1988 [14] and includes the best of two types of models, characterized by its sequentiality and evolution, as presented in [3]. This model enables the transformation of a prototype, achieved in the early iterations, into a functional and complete system, that is improved through new iterations. Generally this model is divided into several activities, that are presented in 2.3.

This process is started by the customer's requirements establishment, followed by the planning to define the timeline, required resources and other project related issues. Then, before the design of the product (engineering task), is made a risk analysis both in the technical and management perspectives. Finally, it is made the construction of the product and its testing, providing to the customer all the documentation and training support, before having customer's feedback, as reported in [2], and its development can involves several cycles around the spiral.

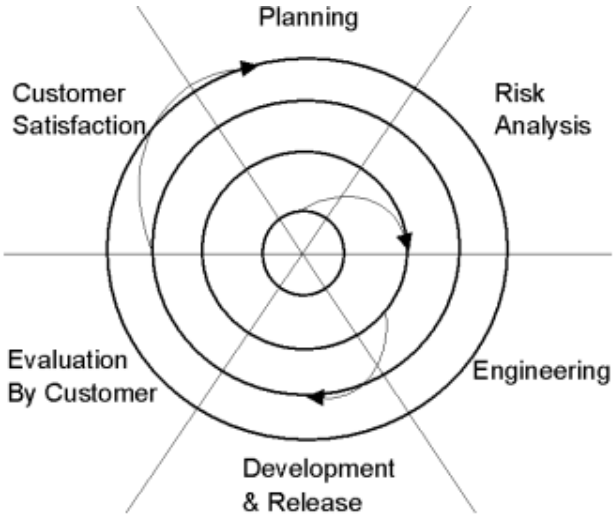


Figure 2.3: The Spiral model [3].

This model is usually applied to the development of a large-scale systems and software. However, it has some issues around its practical application in a real-world environment. To apply

correctly this model, it's necessary to practice its implementation for several in years and it's required some expertness to make a successful risk assessment. One of its main difficulties is to convince the customers of its controllability and efficiency, as described in [2].

2.1.4 Incremental Model

The Incremental model was initially introduced by Victor R. Basili in 1975 [15]. This model combines several linear sequential models with iterations between them. Each linear sequence represents an increment in the software product, and with new increments, its capabilities and features are improved. Every delivery considers the customer's feedback and attempts to include the proposed changes in subsequent increments. The project will have several increments until the product is completed, as reported in [2, 3, 16].

Such as it happens with the Waterfall model, the Incremental model includes some similar and correspondent phases, apart of the maintenance phase, that was already described in subsection 2.1.2.

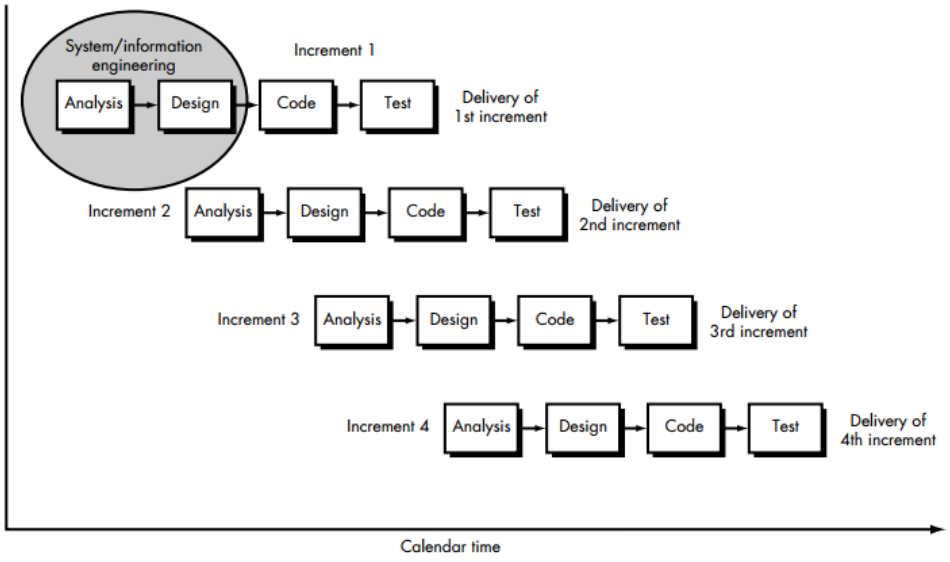


Figure 2.4: The Incremental model [3].

However, this model has some advantages and limitations, as it happens with any other model. Regarding the advantages, this model is often useful when there isn't sufficient human resources to do the implementation, because early increments can be performed with few people, and some issues can be implemented in the next increments, as reported in [2]. Despite that, if some fundamental requirements of the customers are considered in the last increments, it can increase the risk of the project delivery, as described in [3]. Currently, there is an extension of incremental model very used in software companies known as Agile.

Concluding, there is no right model to conduct in the right way the software engineering process, because each case has its own peculiarities. Despite that fact, they all have, at least,

one thing in common: the testing phase. With every model, software testing must be performed until the release of the software product. In this way, software testing will be the focus in this dissertation.

2.2 Android

As said in chapter 1, Android is the most used mobile OS in the World, as the figure 2.5 shows. In second quarter of 2015 Android had a market share about 82%, crushing its directly competitor, *Apple*'s iOS, with only about 15% of market share. On other hand, others competitors have insignificant shares comparing with Android, like Blackberry and Windows Phone.

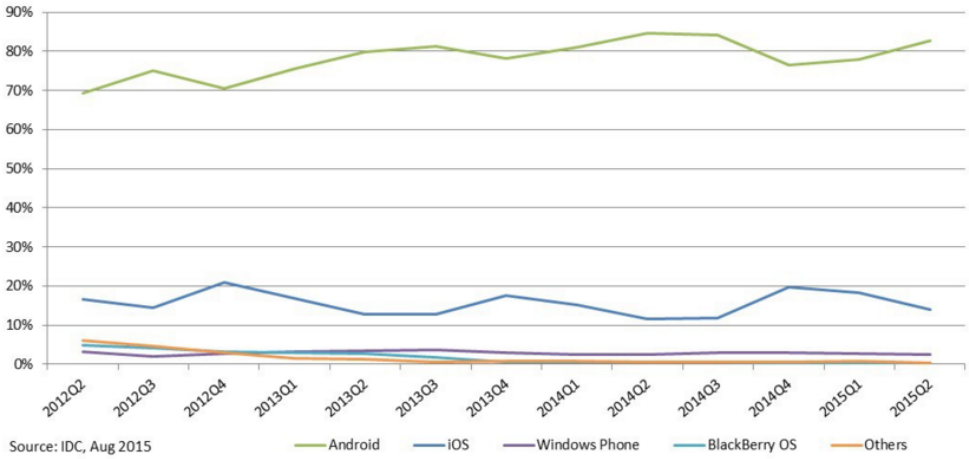


Figure 2.5: Evolution of the mobile devices market share[1].

Android was created by *Open Handset Alliance* and led by *Google* at 5 November of 2007 sharing a common goal to deliver to its users innovation, giving them a better experience and opening on mobile devices. The first Android phone released to the market was HTC Dream (also known as T-Mobile G1) in 2008, with 1.0 version [2].

Currently, there is a huge number of versions that are already in use over the world and have its distributions presented in figure 2.6. The present version is 6.0, called Marshmallow, and is scheduled a new Android version release till the summer of 2016 (Android N - 7.0 version). Each Android version released has the peculiarity in its name of always having an alphabetic confectionery-themed code name. The main reason of this variety of versions available on the market is because there is a huge community of developers that constantly improve and modify the OS, and create new applications.

[1] International Data Corporation (IDC). *Smartphone OS Market Share, 2015 Q2*. August, 2015. [Accessed: Jan. 29, 2016]. URL: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[2] Dan Morrill. *Announcing the Android 1.0 SDK, release 1.* Android Developers Blog. September, 2008. [Accessed: Feb. 07 2016]. URL: http://android-developers.blogspot.pt/2008/09/announcing-android-10-sdk-release-1.html.

Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	2.7%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	2.5%
4.1.x	Jelly Bean	16	8.8%
4.2.x		17	11.7%
4.3		18	3.4%
4.4	KitKat	19	35.5%
5.0	Lollipop	21	17.0%
5.1		22	17.1%
6.0	Marshmallow	23	1.2%

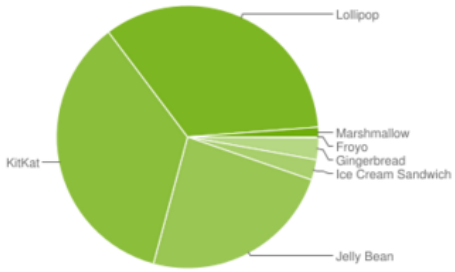


Figure 2.6: Android versions distribution [3].

Android OS is also known by its Easter eggs, that are hidden features on the OS such as developer options, and Android version animations.

Other aspect that is also important to approach is that Android doesn't have a fixed UI, i.e., for each type of screen resolution an application has a different layout's disposition. Currently, there are a several distinct number of mobile devices, with different screen sizes and pixel density, as can be seen in figure 2.7, and is necessary takes it into consideration when an Android application is developed.

[3] Android Developers. *Dashboards*. Google. February, 2016. [Accessed: Feb. 07, 2016]. URL: https://developer.android.com/about/dashboards/index.html.

	ldpi	mdpi	tvdpi	hdpi	xhdpi	xxhdpi	Total
Small	2.4%						2.4%
Normal		5.1%	0.1%	41.5%	22.9%	14.8%	84.4%
Large	0.3%	5.0%	2.3%	0.6%	0.5%		8.7%
Xlarge		3.5%		0.3%	0.7%		4.5%
Total	2.7%	13.6%	2.4%	42.4%	24.1%	14.8%	

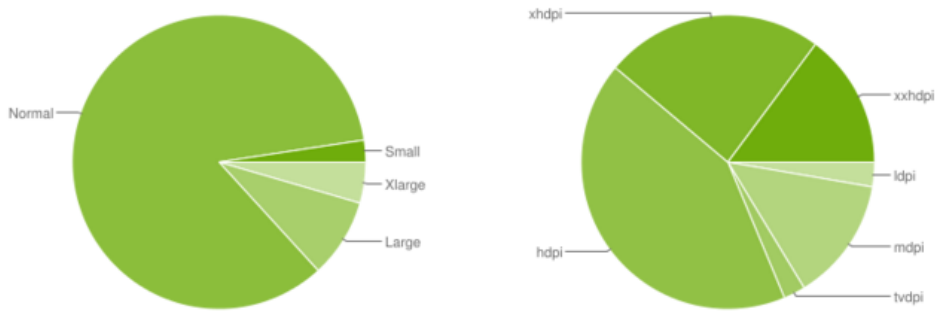


Figure 2.7: Android screen sizes and densities distribution[3].

After this kind of analysis about Android devices, it is necessary to understand the architecture of the OS, that will be approached in the next subsection.

2.2.1 Architecture

Android is a mobile OS based on Linux and a Java virtual machine called *Android Runtime* (ART). This OS has a well-defined architecture [4] represented in figure 2.8, being constituted, from the bottom to the top, by the following parts:

- Linux kernel: that is the lowest layer and the basis of Android. It is responsible for the memory's management, processes communications, security, network, files, peripheral and sensor drivers issues;
- *Hardware Abstraction Layer* (HAL): it is a standard interface that allows the Android system to call into the device driver layer without concerns about the lower-level implementations of drivers and hardware. This is specially important for embedded systems present in Android devices such as audio, camera, Bluetooth, external storage, sensors and graphics;
- Native libraries: these libraries, written and compiled in C/C++, deal with processor native instructions. It is also possible to create new native libraries using the *Native Development Kit* (NDK). These libraries are composed by:
  - the Surface Manager that composes the UI on the screen;

[4] Android. *The Android Source Code*. Google. [Accessed: Feb. 11 2016]. URL: https://source.android.com/source/index.html.

  - the *Scalable Graphics Library* (SGL) that is a 2D graphics engine;
  - the OpenGL ES (3D library);
  - the Media Framework responsible for recording and playback audio, video and image formats;
  - the FreeType responsible for rendering bitmap and vectors fonts;
  - the SSL responsible for Internet security;
  - SQLite to provide a relational database engine for the applications;
  - the WebKit to provide embedded browser and web views;
  - the Libc to provide implementation of C native libraries.

- Android runtime: it is composed by an optimized Java virtual machine (ART) and a Java core library;
- Android Framework: it is a top-level Java library for creation of Android applications, composed by content providers and managers (activity, location, package, notification, resource, telephony and window);
- Applications: the OS has several default application that are built-in in Android devices such as a browser, camera, alarm and contacts.

In addition to the architecture, Android has also a feature called Widget [5], corresponding to a small rectangular portion that can be displayed in the Home application with different purposes.

After this detailed description, it is important to approach how the applications are built from Java source code point-of-view. That is important to understand the reverse engineering process that will be described in the subsection 3.4.1.

[5] Android Developers. *Widgets*. Google. [Accessed: Feb. 11 2016]. URL: https://developer.android.com/design/patterns/widgets.html.

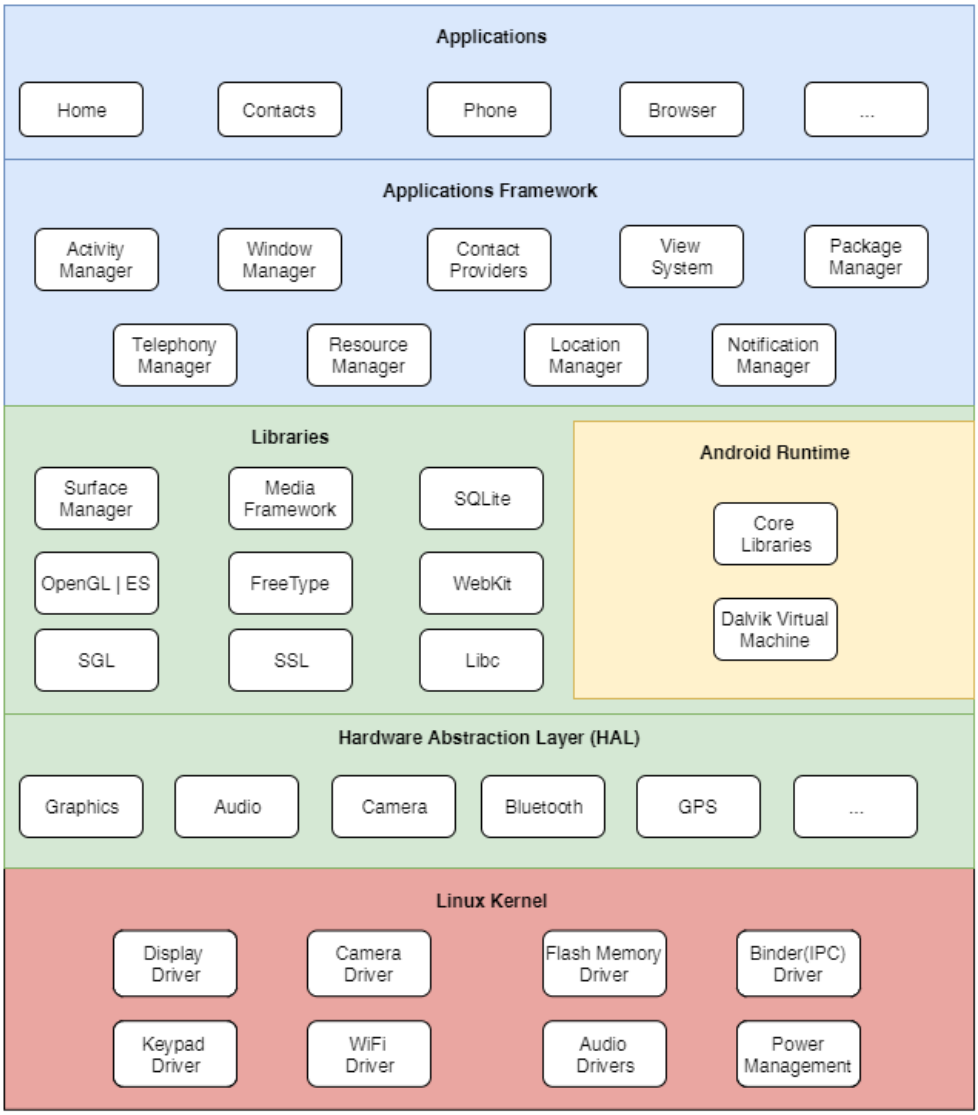


Figure 2.8: Android OS architecture.

When the applications are compiled from the Java source code, it firstly uses *javac* tool to translate java files into class files, then using *Dalvik Executable* (DEX), the class files are converted into DEX files and finally with the help of *aapk* tool, the DEX files, *Extensible Markup Language* (XML) resources (containing the manifest file) and assets are joint into an *Android Package* (APK) file [6] and represented in figure 2.9.

[6] Android Studio. *Configure Your Build*. Google. [Accessed: Feb. 11 2016]. URL: https://developer.android.com/studio/build/index.html.

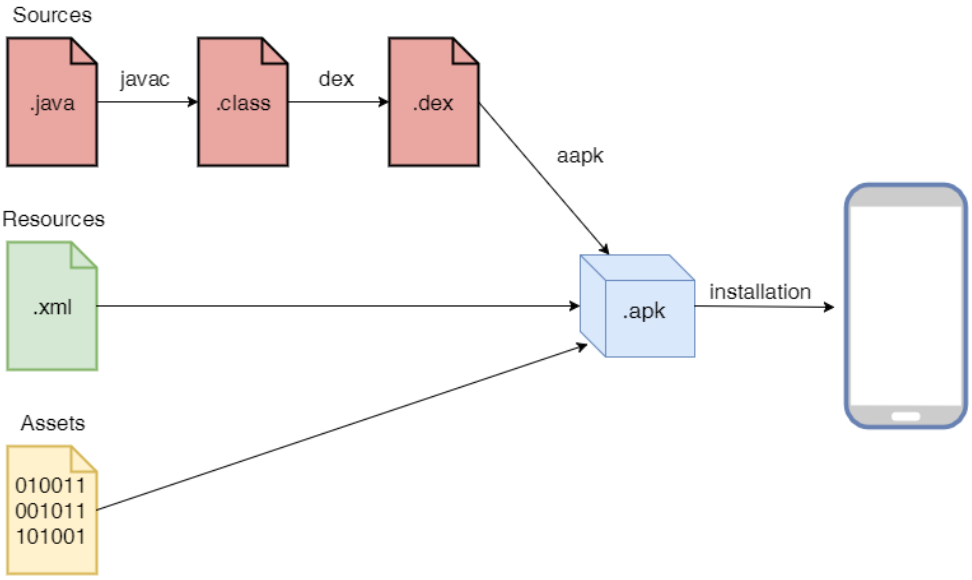


Figure 2.9: Android application build process.

Beyond this, an Android application has several components that will be approached in the next subsection.

2.2.2 Applications' Components

Android applications are characterized by not having a single entry point like *main()* in Java programs, and they are composed by several independents components [7], such as:

- Activities: generally composed by an independent portion of UI, composed hierarchically by views that can be invoked by other views, which users can interact performing actions. This component is characterized by its lifecycle callbacks, as presented in appendix A, such as:
  - onCreate() - is the callback where the activity is firstly created and where occurs the UI inflation, the data binding or recovery from previous callback;
  - onStart() - is the callback where the layout becomes visible to the user. This callback can be reached after onCreate() or onRestart();
  - onResume() - is the callback where the activity starts to interact with user from the use of layout changes/animations;
  - onPause() - is the callback where a new activity is called and the previous is sent to background. In this callback should be saved quickly the persistent data, memory issues has to be considered;

[7] Android Developers. *Application Fundamentals*. Google. [Accessed: Feb. 11 2016]. URL: https://developer.android.com/guide/components/fundamentals.html.

  - onStop() - is the callback where the activity is no longer visible to the user, but is not called a new activity. In this callback the process of application could be killed because of memory issues;
  - onRestart() - is the callback where an activity is already created, not visible, and is called again;
  - onDestroy() - is the callback where the activity is destroyed by closing the application or calling finish() method in the source code;

- Services: are executed in background, not providing a specific UI and generally are used to establish connection with an *Application Program Interface* (API), *Representational State Transfer Protocol* (REST) or *Simple Object Access Protocol* (SOAP)-based;

- Broadcast receivers: this component allows to the developed application to receive and react to notifications generated by another applications or systems;

- Content providers: they allow the providing of data from the developed application to other applications, and it is composed by an UI.

To invoke the first three presented components it is used an invocation mechanism called Intent with a specific name/action. In other hand, to invoke content providers it is necessary its declaration in the manifest file with a well defined authority and name for its data collection.

The manifest file is required for all Android applications and are described in it all important information about the application such as the package name, permissions, minimum and target *Software Development Kit* (SDK) versions, application's activities and components, and instrumentation classes.

There is another Android device's features such as sensors, Bluetooth, camera and others, but due to the fact that this work won't require to include them, it will only focused the input events.

Many Android devices don't have a built-in keyboard and the most of all user's inputs available are touches and screen gestures. There are a huge amount of events that can be click, long click, key insertion, scroll down, scroll up, swipe right, swipe left and many others. It is also important to refer that Android supports multitouch, and this information will be very useful for solution's implementation.

After this overview analysis about Android, it is important take some conclusions about this chapter, that will be presented in the next section.

2.3 Conclusions

With this background study, now it is possible have more knowledge about the different types of software development models generally used in this scientific area and how Android OS works.

Regarding the first section, can be concluded firstly that there isn't a well defined software development model process. As George Box says[8], "Essentially, all models are wrong, but some are useful". All of them have their advantages and drawbacks and every project is a isolated case that has it peculiarities. However, in each one of the studied models, a testing phase is included, justifying its need.

Regarding the Android OS, it has many advantages, however it has also its drawbacks. Focusing its strengths, Android is the most used mobile OS in the world, having a largest number and diversity of applications available in the official market. It is frequently maintained and updated, possessing an attractive and user-friendly UI and having multi-tasks and an open source system, that can be considered as a drawback as well. On the other hand, its main limitations rely on: the inconsistency between devices, due to the great amount of versions, screen resolution and pixels density of the screen; the version updates force the users to root the device or to buy a new one; and the non validation of applications in the official market may compromise the users' data security.

After this preliminary research done and analyzed, in the next chapter will be presented the literature review.

[8]George E. P. Box and Norman R. Draper. *Empirical Model-building and Response Surface*. John Wiley & Sons, Inc. 1986. [Accessed: Feb. 12, 2016]. URL: http://dl.acm.org/citation.cfm?id=17317

Chapter 3

State of Art

This chapter describes the literature review made about this dissertation's theme, including an overview of the software testing processes, techniques and mobile GUI test automation tools for Android mobile platform. Finally, it will be described some relative work done in this field of study.

3.1 Software Testing Overview

As it happens with hardware, software can be very powerful, unpredictable and risky, requiring some type of verification before its execution and release to the market. Software testing is an important component of software QA process. In this way, software needs to be tested several times to verify if it does what is expected to do. With every new code developed, new errors' possibilities are introduced, increasing the risk of its failure. Because of it, there is commonly a testing phase, after the software product's development and before its deployment into production phases, with the goal of reducing the risk of customer deal with software's bugs and errors [17].

> *"The best software is that which has been tested by thousands of users under thousands of different conditions, over years. It is then known as "stable." This does NOT mean that the software is now flawless, free of bugs. It generally means that there are plenty of bugs in it, but the bugs are well-identified and fairly well understood"* - Bruce Sterling in [18].

Currently, compilers provide some help in this task, generating warnings and errors about the code, but they aren't enough to software testing. To assure the robustness and strength of an application, must be conducted different types of analysis and several tests, that will be described in the following sections. This sections' division is based in different references with different types of content and due to that couldn't be intended as ideal by others researchers.

3.2 Test Types

In this section it will be described two different kind of tests, the functional tests, that are related tangibly with features, and the non-functional tests, that are related with the operation of the system. These tests are included in the testing phase of every software engineering model, as described in section 2.1.

3.2.1 Functional Tests

According [17], this type of tests is part of software QA process and is used to verify if the product does what is supposed to do, by testing its features. Functional tests, generally, are characterized as a black box testing method (3.3.2), but can be performed also as white box testing method (3.3.1), as reported in [19]. They are used to guarantee that all the product's requirements and features, previously defined, are accomplished. There is a huge amount of tests that fit in this tests' family, as reported in [17, 19, 20] , such as:

- Alpha testing: these tests are performed with a product's prototype version, and at this point, can suffice to have some issues and bugs regarding the system and interface;
- Beta testing: these tests are performed with most of the system's and interface's issues resolved, bearing in mind performance and cosmetic concerns.
- Unit testing: this type of tests is the first one usually performed and commonly this work is done by the developers, in single units. For a number of specified inputs it's obtained the outputs and these are compared with the expected values, determining if there are errors or not;
- Integration testing: in contrast to unit testing, these tests are performed to verify the group's assembly of units, focusing in the interaction between them;
- System testing: this kind of tests is performed on the final product, verifying if the product gives the right outputs, given a specified set of inputs, and it handles the exceptions in the right way;
- Regression testing: the changes made in a new release of software are simply verified, validating if they haven't compromised the previous developed code's functionalities;
- User Acceptance testing: generally this is the last phase of tests before the product's release, and they are performed to guarantee the minimum standards of product's quality being accepted by the customer. This usually is focused on what the end-user can see and do, but it can includes some documentation and training material.

After functional tests were properly distinguished and described, now the non-functional test will be presented in the following subsection.

3.2.2 Non-Functional Tests

Unlike the functional tests aforementioned, the non-functional tests [17] verify the system's tolerance and robustness, testing it with non-validated cases and inputs. Furthermore, some kinds of these tests verify its performance, reliability and scalability. As reported in [20], there are many types of these tests that can be divided in:

- Performance testing: to verify if the software meets the performance requirements such as memory consumptions and time responses;
- Security testing: to assure that the software is protected against external threats, verifying its confidentiality and integrity and also the system's availability, as specified in [21];
- Usability testing: to evaluate how easily the user can learn and use the software. It is generally related to the visual interface and the understandability of the error messages;
- Stress testing: to determine the system's limits and to test its defense mechanisms, also known as load tests;
- Recovery testing: to verify how the system recovers, after a system's crash or an unstable situation;
- Reliability testing: to measure if the software assures the data's delivery to the right destination;
- Smoke testing: a simple test that verifies if the system is running or not.

With this detailed analysis of almost all test types in software engineering, can be concluded that due to the enormous quantity and diversity of tests, currently there is a urgent need to automate them. To perform each one, there are some different methodologies that can be used, and them will be described in the next section.

3.3 Testing Methodologies

Due to different type of persons involved in the test of the software product (testers, developers, product owner, customer...), different methodologies can be approached to test it, such as white, black and gray methods, as can be seen in figure 3.1, and they will be described in the following subsections.

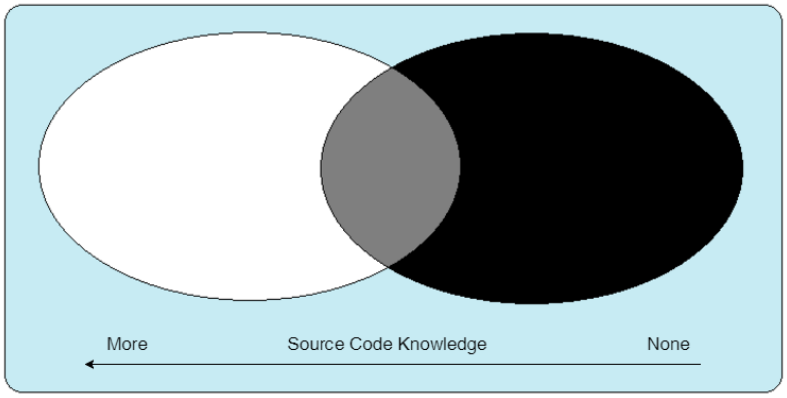


Figure 3.1: Testing Methodologies.

3.3.1 White Box

One of most common testing strategies in software development is white box testing. Also known as glass-box and logic-driven, this method is characterized by the transparency of the internal structure of programs to the tester [17]. Generally, this kind of tests are performed by developers during the coding process, although there are some limitations. One of them is the huge amount of possibilities that the program can produce, making even impossible this testing task. The other is that the code can have untraceable algorithm errors (ascending-order sorting routine instead of descending, per example), missing paths or data-sensitivity errors, as reported in [22].

3.3.2 Black Box

As the name says, black box testing treats the program as a black-box, which means that there isn't any knowledge about the internal structure of the code, bearing in mind exclusively the input against expected output, as described in [17]. Some commons examples of black box tests are user acceptance testing and system testing, that were described in subsection 3.2.1, verifying the implementation of the requirements set by the client [22]. This method, as gray box (3.3.3), becomes a challenge to every tester regarding the functional and non-functional tests. Generally, these tests are performed manually, requiring a huge amount of time from the testers schedule and hence becoming a boring task, costing money that companies could be allocating into more interesting and intellectually intensive activities.

3.3.3 Gray Box

This type of test methods unites both white and black box methods and takes advantage of reverse engineering, as described in [23]. The cross-linking data between source code analysis and the expected outputs, given a specific input, providing a great resource to make a deeper and more complete software testing.

All these three types of methodologies are only based on the access to the source code or not. However, testing an Android application requires different types of analysis based on the running state of it or not. This topic will be approached in the following section.

3.4 Analysis Types

There are two different approaches that can be performed to analyze and evaluate the Android software product's quality, such as static and dynamic analysis. They will be described in the following two subsections.

3.4.1 Static Analysis

As reported in [24, 25], static analysis involves access to the source code, in a non-running state of the program, and is used to find defects in it, as compilers do, but with code inspections and reviews improvements, trying to rectify confusing or misleading code, even if it doesn't result in misbehavior. Furthermore, this kind of analysis can be used to improve code understanding, that nowadays is very important for every developer that needs to deal with someone else's code. This analysis is extremely effective, both in terms of costs as in defect spotting, and could be made by a people's group, also known as pair programming, or by automated tools, as reported in [25].

One of those tools is *Soot*, as reported in [26, 27, 28]. This tool is an optimization and language manipulation framework that can be used to improve Java programs and applications. Soot takes advantage of accepting input files as Java source code, Android or Java byte-code (Baf), Java simple (Jimple) or Jasmin - a low-level intermediate representation and that could be converted to Jimple, a simplified version of Java source code, containing only three components per statement, or into Java/Android byte-code or Jasmin. Soot provides a call-graph construction, dependency analysis, intra and inter-procedural data-flow analysis and defects' analysis in use of chains.

Another tool is *FindBugs* [1], also reported in [24], an open source tool sponsored by *Google* that makes static analysis of *Java* byte-code and source code, finding defects and reporting warnings to the user, classifying bugs in false positive, trivial and serious. This tool is already very known in the market, having available plugins for the main *Java Integrated Development Environment* (IDE)s such as NetBeans, Eclipse and IntelliJ.

Once this dissertation will focus in Android applications, it is important to consider, also, some tools provided by Google, like Lint, and some reverse engineering tools that allow source code analysis of APK files, such as *APKTool*, *Smali*, *DEX2Jar*, *Java Decompiler (JD)-GUI*, *ded* and *Dare*. Some of these tools are used, generally, to find vulnerabilities and malicious applications, as reported in [29, 30, 31].

[1] David Hovemeyer. *FindBugs™ - Find Bugs in Java Programs*. The University of Maryland, June, 2015. [Accessed: Feb. 07 2016]. URL: http://findbugs.sourceforge.net/index.html.

Lint [2] is a tool provided by Android SDK that makes a static analysis of source code in order to find potential bugs and conducts code optimizations. The main goal of this tool is to increase code's robustness, improving an Android application in terms of performance, security, accessibility, usability and internationalization. *Lint* analyze the Java source code of an Android application and characterize the bugs found using three distinct metrics: category, that defined the issues' severity nature, defining the problem's impact in the whole application; and priority that identify what issues should be resolved firstly. Furthermore, this tool also generate bug reports and can be customized.

APKTool [3], *Smali*, *Dex2Jar* and *JD-GUI* are used together, having each one a specific role and function in this reverse engineering process. The first is used to compile/decompile APK files, resources and XML; the second one is used to assembler/disassembler *Dalvik Executable* (DEX) files; the third one to convert DEX files into *Java ARchive* (JAR) files; and the last one to decompile class files into Java source files, as described in [31].

The *ded* decompiler [29, 30] was created to help the finding of malicious applications by a given APK file, extracting its source code. Currently, this tool has been replaced by Dare [32], that is more precise and powerful in the decompiling process.

This type of analysis relies in white or gray box testing methods, that were already explained in subsection 3.3.1 and 3.3.3, and it main limitation is that, currently, many developers use obfuscation of their code [33], making this kind of analysis very hard and sometimes even impossible.

3.4.2 Dynamic Analysis

As described in [17], dynamic analysis unlike static analysis, looks for the code when it is executed in the appropriate environment, making an analysis of the quantities of the variables used, looking at the memory and processor consumptions or overall performance. Dynamic analysis is usually done by developers, running the debugging mode of the program to check if variables have the supposed values and take care of another issues.

Android SDK already provides a set of tools that make possible to perform this type of analysis in several different ways, such as *ADB*, *LogCat*, *Dalvik Debug Monitor Server (DDMS)*, *HierarchyViewer*.

ADB [4] is a debug bridge to establish connection between desktop and the plugged Android device. It acts like a client-server application including the following three components:

- A client running on desktop, that can be invoked from a shell by issuing an *ADB* command.
- A server running as a background process on desktop, managing communication between client and daemon.

[2] Android Studio. *Improve Your Code with Lint*. Google. [Accessed: Apr. 27 2016]. URL: https://developer.android.com/studio/write/lint.html.

[3] Connor Tumbleson. *A tool for reverse engineering Android apk files*. APKTool, 2016. [Accessed: Apr. 27 2016]. URL: http://ibotpeaches.github.io/Apktool/.

[4] Android Studio. *Android Debug Bridge*. Google. [Accessed: Feb. 24 2016]. URL: https://developer.android.com/studio/command-line/adb.html.

- A daemon running as a background process on the device.

When *ADB* client is initiated, it checks if the server process is running, and if not, client starts it. When the server is started, it binds the *Transmission Control Protocol* (TCP) port 5037 to communicate with all clients. Then, the server scans a pair of ports, an even-numbered port for console and an odd-numbered port for *ADB*. These ports range between 5555 and 5585 of each connected devices/emulators, starting the correspondent daemon. In this way, it is possible to conclude that the limit of connected devices/emulators that *ADB* supports at the same time is 15. The server manages connections with all devices/emulators, and because of that, it is possible have multiple *ADB* clients controlling any device or emulator.

LogCat [5] is the Android logging system that collects and display the system and applications debug outputs. To use this tool, is required an *ADB* connection to the device. It is possible to see in the *LogCat* the *JavaSE* system's outputs as System.out or System.err, but it isn't recommended. *Logcat*'s outputs generally are composed by time, priority, *Process Identification* (PID), *Task Identification* (TID), tag and message, but can be displayed in other formats. Furthermore, the output messages can have different types of severity, such as verbose, debug, info, warning, error, fatal or silent. However, this tool sometimes isn't precise as a developer desires, and only gives a specific way for tracking states or outputs of an Android application or system.

DDMS [6] is an integrated debugging tool in the Android Device Monitor, that provides a huge amount of informations about the device and its applications, such as:

- Threads states, heap usage and memory allocation tracking;
- TraceViewer: provides a timeline panel of methods and threads called, and a profile panel presenting time statistics about them;
- Running applications on the device;
- Screen Capture;
- Spoofing of incoming calls, text messages and location data;
- Port-forwarding and network services.

Firstly, when *DDMS* starts, it connects to *ADB* and creates a *Virtual Machine* (VM) to system's monitoring. Once every application has its own process and VM, it is possible to connect *DDMS* to the debug port of each VM's, via an *ADB* daemon. Typically, *DDMS* listens to a range of ports from 8600 to 8700, and is possible to forward all devices' traffic to these ports, using the *Java Debug Wire Protocol* (JDWP). This is a protocol that establish communication between the *ADB* daemon and the Dalvik VM.

HierarchyViewer[7] provides a hierarchy view about the UI of an Android application, profiling

[5] Android Studio. *logcat Command-line Tool*. Google. [Accessed: Feb. 24 2016]. URL: https://developer.android.com/studio/command-line/logcat.html.

[6] Android Studio. *Using DDMS*. Google. [Accessed: Feb. 24 2016]. URL: https://developer.android.com/studio/profile/ddms.html.

[7] Android Studio. *Hierarchy Viewer Walkthrough*. Google. [Accessed: Feb. 24 2016]. URL: https://developer.android.com/studio/profile/hierarchy-viewer-walkthru.html.

each view. For each application's activity, it renders each element, based on a tree model, been possible to observe the detailed information about it, such as class type, resource *Identification* (ID), position on the screen and other properties. Furthermore, with this tool is also possible have time statistics about the layout rendering, making easier the GUI debugging.

Another tool developed to perform dynamic analysis is *Graphical On-Phone Debugger (GROPG)* [34]. This tool was developed at University of Texas at Arlington and allows the debugging, on the device, of applications in real-time, providing a traditional debugging environment and containing common actions such breakpoints, step out/into/over and thread/variable values analysis.

Furthermore, dynamic analysis is also used in terms of security. Generally, it is used to prevent security threats, as described in [35]. It is commonly used to find input or output validation error (*Structured Query Language* (SQL) injection and cross-site scripting, per example), server configuration mistakes and to verify if the connection with the server is secure, ensuring, above all, confidentiality, integrity and authenticity, as defined in [21]. It can prevents different types of attacks, such as man-in-the-middle and sniffing, that currently have been discussed as a big problem in mobile application systems. As reported in [36], dynamic analysis is immune to obfuscation and is often used to identify malicious applications and threats. Its main difficult remains in the protection after its execution. The best way to do it is to take advantage of virtualization, that reproduces a controlled environment, allowing the prevention of information's escape and the infection of other machines and devices.

There are some tools available to perform security dynamic analysis such as *Wireshark*, a well-known network protocol sniffer, that is an open sourced and platform-independent tool capable to decode more than 400 protocols, as reported in [37]. This tool contains a non-GUI version called *TShark*, that acts like *tcpdump* with some options to filter and analyze network traffic. *Tcpdump* [8] is a command-line tool used to capture network traffic, having different configurable parameters. There are another tools such as *IBM Rational AppScan* [38] and *HPE Fortify* [39], but because they are limited capability and commercial tools, they will not be described.

This type of analysis is characterized by the use of both testing methods, black and gray box, that were described in subsection 3.3.2 and 3.3.3, respectively. After its description, will be presented in the following section a specific kind of test, GUI testing, that is the main purpose of this study.

3.5 GUI Testing

When a Android application's GUI is tested, both of some functional (system, regression, user acceptance) (3.2.1), and non-functional (stress, usability) (3.2.2) tests are combined and several persons from the software, infrastructure and design fields are involved. Furthermore, different types of testing methodologies and analysis are performed, as shown in the previous sections.

[8] Tcpdump.org. *Tcpdump/libpcap public repository*. 2015. [Accessed: Feb. 24 2016]. URL: http://www.tcpdump.org.

Generally, GUI tests are made when the software product is already developed, but not released to the market, yet. Usually, it is done manually and black-boxed, but nowadays, there are several ways to automate this kind of tests. In this way, the following two subsections present the two different approaches to test an Android application's GUI.

3.5.1 Manual Testing

One of the ways most known and used by software companies to make product's GUI tests is manually. To perform them, there are some different methodologies, as reported in [40], such as:

- Heuristic Methods: requiring a group of UI specialists testers that studies the software in order to find problems. This technique identifies many more and serious problems, in a short time, however it requires many and well payed specialist testers;
- Guidelines: this technique is based on recommendations about software and UI previously defined, however it can miss some serious problems;
- Cognitive walkthrough: This technique is characterized by developers/testers trying to act like a real user, defining the main tasks, and its goals/expectations about the UI's interactions.
- Usability tests: The software is used by real users, under controlled conditions, collecting and identifying problems, reporting to the company the improvements that can be made before the product's release.

This is the most reliable way to test applications, because more bugs can be found, if performed by good specialists, and bugs could provide hints to find new ones that an automated test is unable to understand yet. Another benefit of manual testing, is the detection of usability problems. However, this technique has also drawbacks, such as: the effort and time required, a weak coverage, a regression test must be repeated and it is difficult to find a good tester. In this way, the following subsection an automate approach to test application's GUI.

3.5.2 Automated Testing

As reported in [17], test automation has some number of goals that, nowadays, software companies want to accomplish. One of them is the reduction in cost and effort done to make tests, since every single release of new software always requires human resources and a huge amount of time to perform them. Another goal is to speed up the bug reporting process, allowing faster corrections by developers. Finally, the last goal is to reduce the errors made by humans during the testing phase.

But the real question is, from what point does the test automation offsets? The answer to this question can be found in [41] and [42], that detail the costs of automation. In these documents, they specify the cost with a single automated test (Aa), represented in equation 3.1, where V represents the costs with the test specification and implementation, D represents the cost for the

single test execution and n is the number of test executions. They use the same calculation to represent the cost for manual test execution(Am), as can be seen in equation 3.2, and to calculate the break-even($E(n)$), i. e., the point where the automated costs are equal to manual costs, simply comparing the both costs, as can be seen in 3.3 and 3.2.

$$Aa := \quad Va + n * Da \tag{3.1}$$

$$Am := \quad Vm + n * Dm \tag{3.2}$$

$$E(n) := \quad \frac{Aa}{Am} \quad = \quad \frac{(Va + n * Da)}{(Vm + n * Dm)} \tag{3.3}$$

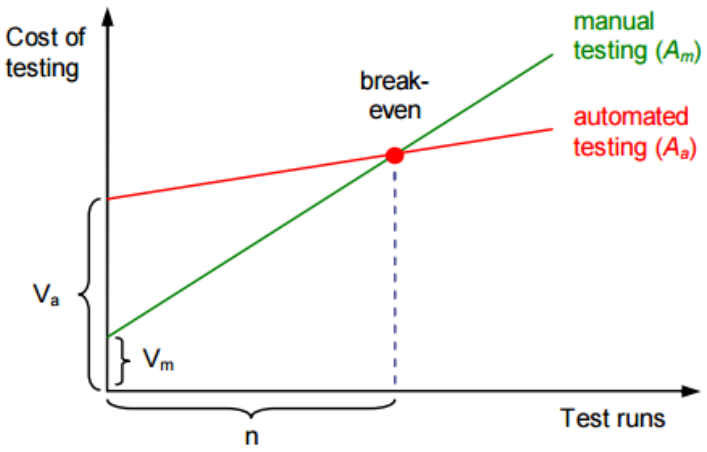


Figure 3.2: The cost of manual vs. automated testing.

According to this model and with the growing of tests' number *(n)* necessary to perform before each release, it's empirically proven that test automation not only is necessary but also, if not implemented, may incur higher costs as the number of tests grow.

Test automation can be applied to a different pyramid's levels, as presented in figure 3.3, that goes from the bottom, deep level of automated tests, to the top, manual tests. Considering that GUI's automated tests are the closer layer to the manual tests, it means that, among all automated tests, this type requires more supervision. Usually, as further closer to the top of pyramid, more black boxed tests are.

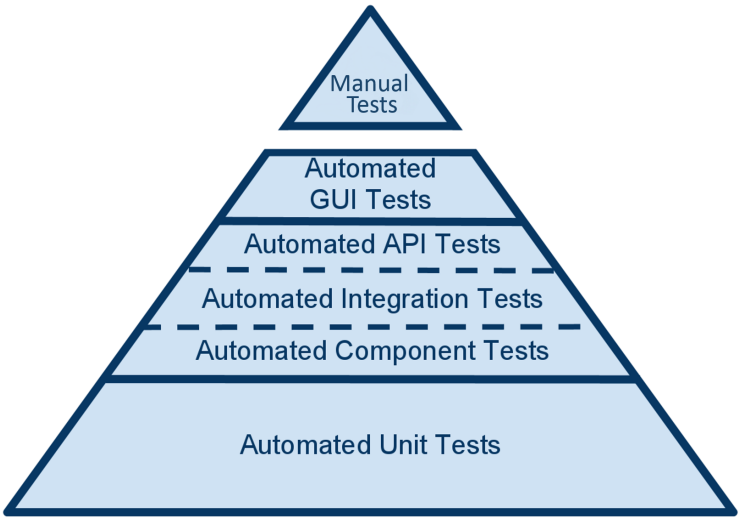


Figure 3.3: Automated testing pyramid [9].

To conduct GUI's automated tests can be used different approaches, as reported in [43], such as:

- Capture and replay: the tester can track the entire interactive session, recording user's inputs, gestures and keys pressed, saving it in a log file. Then, it is possible to replay with accuracy the entire session without human's interaction. This approach can be very powerful in regression testing, however, it only can be used when there is a product and when there is no layout changes between product's versions. *RERAN*, reported in [44], is one of these tools;

- Random inputs: acting like a monkey, random and unpredictable inputs are generated automatically, making a kind of stress test. Although this approach is good to find system crashes, the bugs' finding and reproduction can be a hard task. *UI/Application Exerciser Monkey*, is one of these tools;

- Unit testing: generally based in *JUnit* testing framework, the tests can be automatically executed in order to test the application's GUI. The tests can be written before the development phase, however the test cases have to be programmed manually and usually require more lines of code than the application itself. *Robotium* and *UIAutomator* are examples that take advantage of this approach;

- Model-based: makes an abstract representation of states and behavior of a *System Under Test* (SUT). Then, based on it, test cases are derived, generally using only black box methodologies. This approach is characterized by its adaptability to changes, automatic test case

[9] Alister Scott. *Introducing the software testing ice-cream cone (anti-pattern)*. WatirMelon. January, 2012. [Accessed: Feb. 12, 2016]. URL: http://watirmelon.com/2012/01/31/introducing-the-software-testing-ice-cream-cone/.

generation and its testing exhaustion. However, it has some drawbacks, such as: the model complexity, the huge amount of generated test cases and its management issues. *GUI Ripping*, reported in [45], is one tool that uses this approach;

Despite these four different approaches, all can be categorized in two software development processes, known as *Behavior-Driven Development* (BDD) and *Test-After Development* (TAD). BDD is an extension of *Test-Driven Development* (TDD) and is characterized by writing behavior tests before the development phase. In other hand, TAD is characterized by writing tests after the development phase, having a prototype or a final product to understand its behavior.

Nowadays, there are also some tools based on *Domain-Specific Language* (DSL), as described in [8]. This is a very popular approach to automate GUI tests and is used to express the behavior of an application and the expected outcomes in a script using natural language sentences. It can be used for both BDD and TDD approaches. Tools as Cucumber, Appium and Robotium take advantage of a DSL, and will be described further.

Currently, there are a set of tools available on the market that will be described and analyzed in the next subsection.

3.5.3 Test Automation Tools

As described in [8] and mentioned in previous subsection, *Cucumber* is a command-line and a DSL tool that reads natural language text files, as a simple algorithm script, taking advantage of *Gherkin* parser. Each file is interpreted as a feature that contains several scenarios, and it is tested against the program or system. *Cucumber* uses *Ruby* to convert the natural language into native programming language with the help of a library, that translates it into the application's domain.

As mentioned in the previous subsection, *Appium* [10] is an open-source automation and cross-platform tool for mobile applications, both for Android, iOS and FirefoxOS. It takes advantage of *Apple* UI Automation for iOS, *Google UIAutomator* and Instrumentation for Android. This tool joins all of these tools in a web-driver API that acts in a client-server architecture. It uses a RESTful web service and *Hypertext Transfer Protocol* (HTTP) protocol to application's testing. Due to huge amount of libraries included, this tool offers some flexibility in terms of programming language chosen.

Since the main focus in this dissertation will be Android, would be good to make an analysis exclusively about the automation tools of this mobile OS. Currently, *Google* provides some support in the Android testing automation, including and developing classes/libraries to do it, such as *Robotium*, *monkeyrunner*, *Monkey*, *UI automator* and *Espresso*, as reported in [46], [47] and [48].

Robotium [11] was developed in 2010 and is one of the main test automation frameworks for Android mobile applications. With this tool it is possible to a tester the faster creation of test cases, eliminating the boring task and error-prone of manual scripting. Robotium requires a minimal

[10] Appium. *Automation for Apps*. Sauce Labs, 2016. [Accessed: Feb. 27 2016]. URL: http://appium.io/index.html.

[11] Renas Reda. *Robotium*. 2010. [Accessed: Feb. 10 2016]. URL: http://robotium.com/.

knowledge of the application under testing, since it is black boxed and it handles with both native and hybrid applications. Although the code is available to the community, Robotium has a paid version to record Android UI tests, but can be used in *Android Studio* and *Eclipse* with a free plugin.

On the other hand, there is the *monkeyrunner* [12], a black boxed tool that provides an API to control an Android device or emulator. This tool allows the users' inputs simulation and generation, take screenshots or install an Android application. monkeyrunner requires the writing of Python scripts, and uses Jython to convert them into Java programming language.

As mentioned in the previous subsection, the *UI/Application Exerciser Monkey* [13] generates pseudo-random streams of users' inputs, simulating clicks, touches or gestures and system-level events. This is a command-line based tool that performs stress tests in Android applications, allowing the user to choose which application he wants to test, the time of testing, the events, constraints and the debugging cases.

The *Espresso* [14] testing framework, is used to automatic simulate users' inputs in the application's UI, assuring that the users don't have a bad experience when interacting with the application. It is very similar to monkeyrunner.

UI Automator [15] is an instrumentation-based API that allows the simulation of users' inputs across multiple applications, regardless of which activity is focused. It has also UI dump capabilities, can extracting the elements presented on the Android device's screen.

Another existing tool is *Selendroid* [16], a script-based tool for Android native and hybrid applications, both in mobile devices or emulators. The tool has a web-driver to conduct tests on the browser, and supports a large number of Android target APIs.

As the detailed description of the tools is done, now is necessary define some metrics to compare them. Major part of this job was already done by Linus Esbjörnsson in [49], and can be found in tables 3.1 and 3.2, with some corrections and another tools analysis.

[12] Android Studio. *monkeyrunner*. Google. [Accessed: Feb. 10 2016]. URL: https://developer.android.com/studio/test/monkeyrunner/index.html.

[13] Android Studio. *UI/Application Exerciser Monkey*. Google. [Accessed: Feb. 10 2016]. URL: https://developer.android.com/studio/test/monkey.html.

[14] Android Developers. *Testing UI for a Single App*. Google. [Accessed: Feb. 10 2016]. URL: https://developer.android.com/training/testing/ui-testing/espresso-testing.html.

[15] Android Developers. *Testing UI for Multiple Apps*. Google. [Accessed: Feb. 10 2016]. URL: https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html.

[16] Ebay Software Foundation. *Selendroid*. 2012. [Accessed: Feb. 10 2016]. URL: http://selendroid.io/.

Table 3.1: UI automation tools comparison - general metrics

General metrics						
Metric/-Tool	***Robotium***	***Selendroid***	***UI Automator***	***Espresso***	***Appium***	***monkeyrunner***
API	Very simple	Moderate	Simple	Very simple	Very simple	Simple
Logging support	Average	Average	Bad	Good	Average	Average
Capture/replay support	Yes	Yes	No	No	Yes	Yes
Documentation	Good	Bad	Good	Good	Good	Good
Testing method	Gray-box	Black-box/Gray-box	Black-box/Gray-box	Gray-box	Black-box	Black-box
Random event generation	No	No	No	No	Yes	No
Emulator/real device support	Both	Both	Both	Both	Both	Both
Version support	API 8+	API 10+	API 18+	API 8+	API 17+	API 8+
Version control support	No	No	No	No	Yes	No
IDE support	Eclipse, Android Studio	Eclipse	Eclipse, Android Studio	Eclipse, Android Studio	Eclipse, Android Studio	Eclipse, Android Studio

From these tables and the previous description of each tool, can be concluded that there is a larger number of tools for test automation and some of them can be very useful in the solution's implementation, as *UIAutomator*. However, it may noticed that no one of them make a security and dynamic analysis of the applications' network traffic/logs generated.

Table 3.2: UI automation tools comparison - technical metrics

Technical metrics						
Metric/-Tool	***Robotium***	***Selendroid***	***UI Automator***	***Espresso***	***Appium***	***monkeyrunner***
Resume application	No	Yes	Yes	No	Yes	Yes
Suspend application	Yes	Yes	Yes	Yes	Yes	Yes
Finish activity	Yes	Yes	Yes	Yes	Yes	Yes
Change device settings	Limited	Limited	Yes	No	Yes	Yes
Scrolling	Yes	Yes	Yes	Yes	Yes	Yes
Simultaneous keypresses	No	Yes	Yes	No	Yes	No
Re-launch application	Limited	No	Yes	Limited	Yes	Yes
Delay support	Yes	Yes	Yes	Automatic	Yes	Yes
Condition based testing	Yes	Yes	Yes	Yes	Yes	Yes
Support for device keys	Yes	Yes	Yes	Yes	Yes	Yes

Nowadays, there is a truly security concern in mobile devices' World and it is very important to approach due to the huge amount of malicious or insecure applications existent in Android official and unofficial markets. Furthermore, there is no consistent automated tool that generates test cases for all possible patterns of the application.

After the detailed analysis of the current existing tools available in the market to automate tests, it becomes mandatory analyze some implementation attempts of tools similar to the solution presented in this dissertation. In this way, the related work will be approach in the next section.

3.6 Related Work

Some related work is already done in this area of study, and provides really important information for the implementation of the solution.

The paper [50] suggests a tool called *Dynodroid* that generates a set of randomness touching and scrolling events as *Monkey* does, to perform applications stress testing. Besides this tool generates significantly more concise input sequences, it was more slower than *Monkey*, doesn't support asynchronous operations and is only supported by Gingerbread Android version.

In [51] is proposed a tool called *Swift-hand* that uses a machine learning approach, generating sequences of users' inputs and, at the same time, learning all the application's possible patterns, visiting unexplored states of the application and also unreported crashes/bugs. The main limitations are that the tool requires much time to restart the applications, re-exploring the paths already discovered. Furthermore, it is only supported by Android 4.1 or higher versions, and it cannot handles apps that use a remote server to store data through Internet connectivity.

In [35] is proposed the *AppPlayground* to automate the Android application's security analysis, using some techniques to discover malicious applications installed in the device. However, this tool cannot work well when an app contains embedded web pages and applications that require login or registration.

In [45] is suggested *Android GUI Ripper*, an automated testing tool based on instrumentation of Android application's source code, using an event-driven approach. This tool creates a tree model from the GUI objects and generates test cases in a Robotium format. This tool does not provide support for some events such as hardware sensors and networks, and didn't was tested for real applications with huge size and complexity.

Troyd in [52] is a integration testing framework, that simulates the user's inputs and records the interactions in order to generate Ruby testing scripts that can be used for regression or compatibility tests. However, this tool only control the application's components, reacting bad to browser launches. Furthermore, the main drawback of it is the delay between user's actions and its effect in the application.

3.7 Summary

Firstly, it can be concluded that there are many tools on the market that perform static and dynamic analysis of other applications, either using black, gray, or white box methods.

On the other hand, there is as huge amount and diversity of tests that require several hours of manual labor, hence increase the cost or planning, executing and maintaining them. Henceforth, it is necessary its automation when the number of test cases executions is high.

This chapter presents also a decent amount of testing tools available on the market that can be used to automate tests, improving the Android applications' robustness and liability. Although many of those tools actually help the software testers' performance while in-job, none of them fully satisfies their needs. There's a common requirement throughout the software testing field that requests increased test automation, as reported in [17].

This work focuses on enhancing the automation of GUI tests by making a sophisticated Android crawler, similar, but improved, to what are detailed in the some of related work tools and that can serve as an iterative contribution to the field.

The following chapter will introduce and detail the proposed solution.

Chapter 4

Android GUI Crawler

This chapter introduces the problem's definition, and details the proposed solution of a Android GUI crawler-based and black-boxed automatic tool, for test case generation. In every section are placed some important questions and its respective answers for a better understanding of the dissertation's purposes. Firstly, it is made a brief introduction and it is described the way which the interaction with device was done. Then, the ADB tool is detailed as well as the used data model. Furthermore, the crawler is specified and finally the test case generation and capture modules are explained. Finally, it is also described the traffic capture module.

4.1 Introduction

Nowadays, mobile applications are in constant change. Software companies have to develop new products and upgrade the old ones to assure the users' needs. In this way, every single release, there is a huge amount of tests that needs to be planned, executed and maintained, requiring human resources and consuming many time. But, the main problems are the costs associated to manual tests, that will just increase with time. Because of that, there is a need to automate tests, eliminating the boring and error-prone task of manually test it, simplifying and reducing the tester's work.

To solve the problem, it is proposed a GUI crawler-based automatic testing tool, that will be used to derive test cases and to applications' mapping. Different from another solutions described in section 3.6, it is proposed a totally black boxed solution, with any knowledge of applications flow so as to act like an end-user. This solution is a desktop-based Java application and one of the main goals of it is covers the maximum number of Android applications' possible patterns, constructing a control-flow tree, extracting screenshots, names of activities and the events that trigger the transitions. Furthermore, another goal is the test cases generation in natural programming language that will be useful to perform regression tests.

Finally, there is a concern in the evidences' capture to report bugs and existent problems, namely in terms of security. In this way, all device's network traffic and logs generated are captured during the crawling process as well as the screen recording, in case that is revealed as a possibility.

All this contents will be approached in the following sections, representing all sequential steps that are made during this dissertation.

4.2 Interaction with device

In a first step, some questions appear and some decisions had to be taken about the way that the solution will interact with device, such as:

- **Q1:** Will the solution use real or virtual devices?
- **Q2:** Will the solution be an Android application, a desktop application or a mixture of both ?
- **Q3:** What are the benefits and limitations of each option?
- **Q4:** What is the best option and what is its cost, in terms of time and complexity of its implementation?

In order to test the solution, currently there are available some virtual devices as *Android Virtual Device* (AVD) and *Genymotion* emulators, that simulate a large number of real devices with different Android versions, screen sizes and resolutions. However, this type of environment is very isolated, not counting with other external factors such as 3G connections, background running applications, services, processes and memory issues. In this way, it was chosen the use of real devices to completely simulate the end-user device.

In case the final solution would be an Android application, it would be necessary to develop a service that will take off advantage from Android Linux-based command line, and will be running on the device OS background during the running of the test intended application. With this approach implemented, there would be no need of any cables and push/pull files to another systems, improving the speed of crawler. However, to implement this solution wouldn't allow the applications in several devices to be tested at the same time, requiring direct interaction with the device to start it, which sometimes becomes difficult due to touchscreen size and responsiveness.

If the solution was a mixture of Android and desktop applications, it would be necessary to develop a desktop application and Android application service, acting as client/server architecture, taking off advantage from local Wi-Fi connection. This solution requires the service's installation in each device, its running, and the desktop application requires a multi-threading server socket implementation, with push/pull of files from device, increasing the implementation's time and complexity.

If the solution was only a desktop application, it would take advantage of a tool already developed and already mentioned in 3.4.2, the ADB. This tool only requires a USB cable or Wi-Fi connection with the devices and the push/pull of files from the device, increasing the time of crawling, however it is lower than the previous proposed solution. This solution doesn't require any direct interaction with the devices, taking off advantage from the access to the command line

via ADB. Furthermore, the main drawback of this solution was the insufficient number of USB ports in the desktop to all pretended test devices, that can be solved with a simply USB switch, or using wireless ADB capabilities.

After the analysis of these tree possibilities, the solution chosen was the last one, because it was the one that offers more scalability and benefits considering the pros and cons. Furthermore, it provides a great relation between time and complexity of implementation as well as the possibility of usage of Android emulators. It was represented in figure 4.1, where *N* ranges from 2 to 15 that is the maximum connection through ADB, as described in 3.4.2.

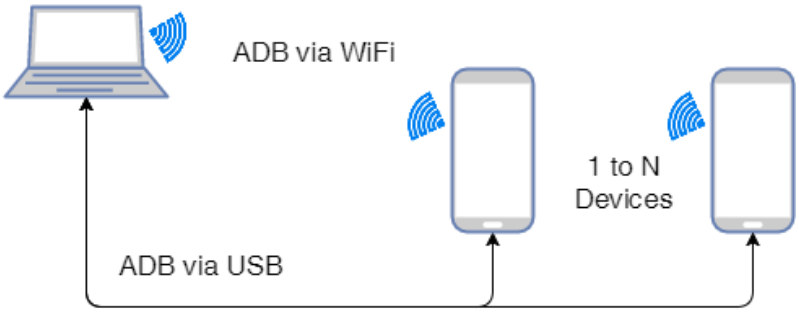


Figure 4.1: Connection with devices.

After performing this choice, it became important to analyze the tool's features used to make the device's connection, ADB, which will be approached in the next section.

4.3 *Android Debug Bridge* (ADB)

To make the connection between the desktop and Android device or emulator, there are some pre-conditions to set up it. These conditions are established to give some security to the device's user, because of the possibility of device remote control through ADB. In this way to run ADB it is necessary to:

- Set the path of ADB executable file as environment variable of the system, in order to execute ADB commands independently of the working folder;
- Plug device to the computer via USB;
- Activate developer options in the device, by generally tapping ten times in the compilation number of the device available on about settings. It is considered as an Easter egg, as described in 2.2;
- In the device, activate USB debugging on developer options;
- If needed, install devices's ADB drivers. Currently Android SDK provides a generic USB driver that works for almost all devices.

As said in the previous section, the solution takes advantage of ADB client on the desktop, that can be used via USB cable or via Wi-Fi through local network. In this way, it was made a representation of its architecture in figure 4.2, based on 3.4.2.

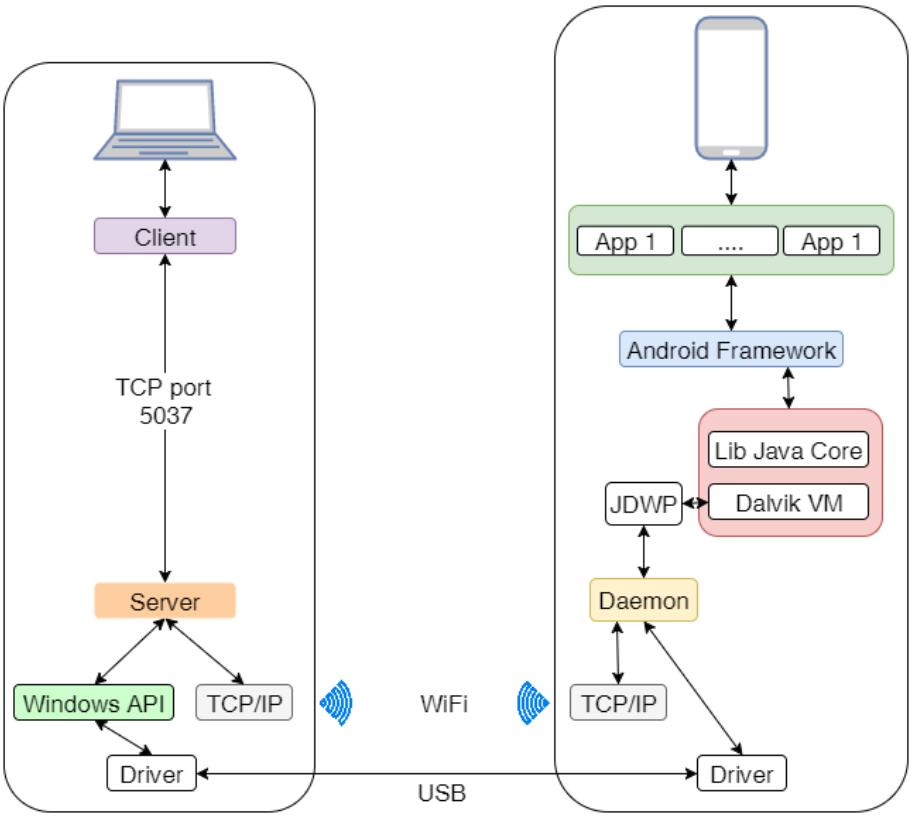


Figure 4.2: ADB operation diagram - Wi-Fi and USB options.

In order to have some abstraction about this level of the solution, it is developed a very complete Java API, to send desktop ADB commands, presented in section B.1, and device shell commands, presented in section B.2, of appendix B.

Regarding the device shell's commands, they take advantage of some existent tools in the Android OS, such as:

- *dumpsys*: that provides information about the status of system services. In the solution it is used to verify the state of applications, to know what activity is displayed and focused at the moment, to verify if keyboard is enable or not, screen resolution, status of battery and numerous information about the device;

- *getprop*: this command provides all information about the properties of device, including the model, Android version, API level, *Internet Protocol* (IP) addresses, default gateway, *International Mobile Subscriber Identity* (IMSI), brand, language, region and others. The outputs of this tool are used to connect to device and to create some pre-conditions to run another tools, like Android version and IP addresses;

- *screenrecord*: this command allows the video record of device's screen. It is a feature only available to devices with Android API level 19 or higher, and is possible to define its bit rate, size, rotation, duration and destination of record file. This tool is used to record, if the user desires, the crawling process, that can be useful to future analysis and bug report. Unfortunately, the record's time limit is only 180 seconds (Android limitation), but it can be surrounded using threads;

- *screencap*: that allows to take screenshots of the device's screen and put it in a file. This tool is used to capture all screenshots of all different activities and dynamic elements, that can be useful for analysis and bug report;

- *cmp*: is used to compare the content of two selected files. It was very important to verify if layout has changes after an input action on the screen;

- *pm*: also known as package manager, this command is used to install/uninstall applications, clear its cache and get the system/3rd party applications installed on device;

- *monkey*: as described in subsection 3.5.3, it generates random events to device's screen, but this tool could be also used to open application by given a package;

- *input*: is a tool to interact remotely with the device. It is used to send key events, taps, swipes, text strings, presses and rolls;

- *tcpdump*: this tool is used to capture device's network traffic to a *pcap* file. It execution is only possible in rooted devices, because it needs superuser permissions, and sometimes the *tcpdump* binary file need to be pulled to the device, because some rooting tools don't have it. If device is rooted, the traffic captures will starts and when the crawling process finish, it is stopped and the *pcap* file is pulled from device to desktop;

- *uiautomator*: is the key to the solution works. This is a tool dumps the current screen layout to a XML file, according the layout's hierarchy, containing several UI elements, such as: relative layouts, linear layouts, buttons, edit texts, spinners and others. Then the XML file needed to be pulled to the desktop, to make its parsing into a tree data model.

These tools are not run at the same time, and are used in different states of the crawling process, that will be better explained in the description of the crawling algorithm. Before this is done though, it is required an explanation about the model used, being it the subject of the following section.

4.4 Data Model

Before the development of the crawler's logic, it was necessary to develop a model to deal with all the UI elements, actions and activities provided by the XML file dumped from device, in every iteration. In this phase of work, new questions appear to deal with these elements, such as:

- **Q1:** What is the best model that fits better in these elements representation?
- **Q2:** What is the complexity of implementation of each model?

In order to answer these two questions, firstly it is tried the implementation of a simply model based on lists. However, after some time lost on it, it is concluded that it wasn't the best model to implement this type of crawler, because several lists will be required, the complexity and entropy will be huge, and the test case generation will be difficult to specify after the crawling process ends. Furthermore, the time and efficiency of the crawler will be penalized because of that.

After this unsuccessful route taken, it was chosen a more logic model that fits better in the proposed solution, a tree model, similar to another tools already mentioned in related work section.

A tree model had been constantly used in several fields of science, mostly in prediction and decision models. In this case, a tree is used to map an Android application, with all activities, all clickable elements and all actions performed to achieve new patterns. This tree can has only one root node and can't have any cyclic pattern.

The model used in this work is composed by nodes, that can be a root or a normal node, and each node can be a different type of element: an activity, an UI element or an action. In addition, each node has a status, if it is already visited or not, that can be important for crawling process, and it is represented in figure 4.3.

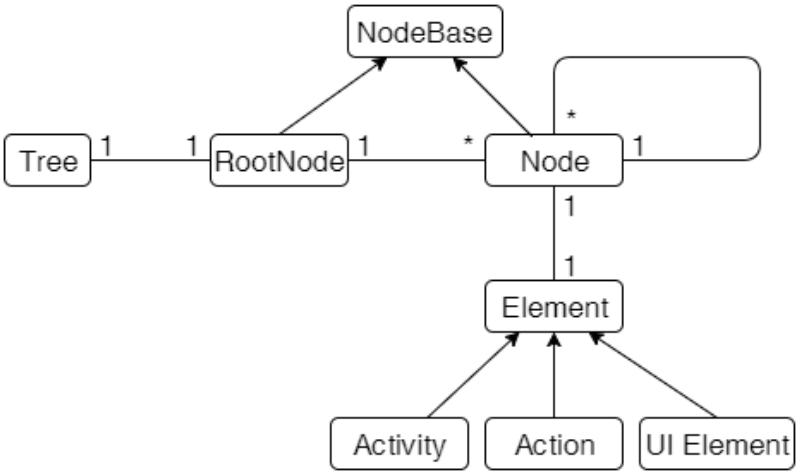


Figure 4.3: Model - Class Diagram.

The main goal of choosing this model is to reduce the complexity of implementation and representation, however it has some drawbacks, namely with the complexity of search. This work takes advantage of a well-known type of search called *Breadth-First Search* (BFS), in order to increase the time, efficiency and coverage of crawling process.

In first step, the BFS is used to find four main actions available in the activity (swipe left, swipe right, scroll down and scroll up, as represented in figures 4.6 and 4.5), to search and add all superficial elements to the activity node.

Then it is started the second phase, making tap actions (represented in figure 4.4) in the clickable elements and looking for new ones, evaluating the new layout or activity that appears in the same way.

Figure 4.4: Tap Gesture

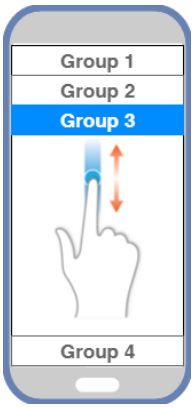


Figure 4.5: Scroll Gesture

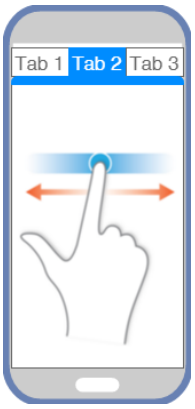


Figure 4.6: Swipe Gesture

Regarding the tree representation, it is used a very simple and complete open sourced library named *Graphstream*. This is a *Java* library for dynamic graphs representation and analysis, allowing the developer to observe, in real-time, the nodes being added by the crawler, interact with them and, in the end, to export them to some different file formats (DGS, DOT, GML, GraphGML). This type of representation was very important for application's debugging process and to generate the test cases more easily.

After the understanding of the tools used to make the connection to the device and its selected data model, it became necessary to understand the proposed solution's core, the crawler, that will be described in the next section.

4.5 Crawler

Also known as "spider" or "bot", a crawler is a program that travels automatically and with no knowledge, through pages of websites. Millions of people use a specific web crawler every single day without knowing it by this name.

Nevertheless, in this case, the concept of crawler is quite different. It was implemented an automated program that tries to travel into all possible patterns of Android applications, based on a tree model, with the purpose of generating automatically test cases for every possible pattern, capturing some evidences such as the network traffic and the logs, during its execution.

This is a very new and innovative concept in the mobile applications crawling field, and as can be seen in the literature review, there are some tentatives to doing this, generally based on gray box and white box methodologies. Regarding these techniques, this dissertation proposes a totally different implementation of a crawler, based only on a black box methodology and on the GUI of Android applications. Without any access to the source code, this technique nowadays can be very useful for companies that have its software products development outsourced. However, some

questions appear, calling for an answer and requiring a more detailed explanation to understand the purpose of this work, such as:

- **Q1:** Why crawling Android applications?
- **Q2:** What device should be used for the crawling process?
- **Q3:** What are the advantages/disadvantages of the exclusive usage of black box methods?
- **Q4:** What are the tools the crawler requires to function properly?

The answer for the first question can be difficult to understand at the beginning. Google was one of the first companies that saw the importance of indexing and crawling the web, due to the huge amount of data and difficulty to search something. Why not doing it also for Android applications? Today, there are millions of Android application available to the users, and crawling them could be useful to get data for business intelligence processes, and to index all the data inside applications and make it available for search. It also can be useful for testing all available patterns in order to find bugs and vulnerabilities.

With this approach, it is possible to monitor constantly the quality of the application and test faster its GUI without the tedious task of doing it manually. Furthermore, it has the main purpose to automate the test case generation. It also can be used to perform a dynamic analysis of captured evidences. These are the main reasons why applications should be crawled.

Regarding the second question, because of what was mentioned in section 2.2, due to different Android versions, screen sizes and densities of devices, each application will have different behaviors. In this way, it is important that the tool has the ability to use different devices for crawling applications. Once the crawler requires the use of *UIAutomator*, the device must have the Android's API Level 18 or greater, as described in table 3.1.

This approach takes advantage of black box only methods to crawl each application. The main advantage of it is that it does not require any access to the source code, as described in 3.3.2. However, using only this method can be a harder task to perform the crawling process as efficiently as the gray/white box methods, due to the lack and precision of information to do it.

Finally, in order to crawl an application there are some dependencies required, such as the ADB executable file and *Windows* API *Dynamic-Link Library* (DLL) file to perform the connection to the device, the *Android Asset Packaging Tool (AAPT)* executable file, that is a tool used to extract information about the Android application. The *tcpdump* binary file is also required to capture the network traffic and all of the other tools, already mentioned in subsection 4.3, will be needed.

After answering these questions, now it becomes important to fully describe the solution in a detailed time sequence.

Firstly, all the devices should be connected via USB to the desktop. When the program starts, the ADB server is started, and it is obtained all the connected devices. For each device, it is obtained all system and third-party applications, as well as all important information about the

device, as presented in section D.1 of appendix D, and as can be seen in algorithm 1, appealing to *getprop* and *dumpsys* capabilities.

Algorithm 1 Program Startup

```
1: startADBserver();
2: List<Device> listOfDevices ← getDevices()
3: for (Device device : listOfDevices) {
4:     Device dev ← getDeviceInfo(device)
5:     dev.listOfSystemApps ← getSystemApps(device)
6:     dev.listOf3rdPartyApps ← get3rdPartyApps(device)
7:     initialConfigurationThread(dev)
8: }
```

At this state, it is possible to set up the Wi-Fi connection, if desired, simply by clicking in a button and disconnecting the USB cable. After the user chooses the application that he wants to crawl, the APK file is extracted to the desktop and using *AAPT* it is also extracted to a file some important data about it, such as application's name, version code and name, target SDK, permissions, supported screens, densities and languages supported, as presented in section D.2 of appendix D. After that, three main threads are started, in case the user configures it so, one for network traffic capture, other to *LogCat* capture and another to video record the process, as can be interpreted in the algorithm 2, that can be useful evidences for bug reporting, control flow and security testing, respectively. Taking advantage of the execution of multiple processes by a single or multi-core processor parallely, known as multithreading, was fundamental to perform this evidences' capture during the application's crawling.

Algorithm 2 initialConfigurationThread (Device *device*)

```
1: if wifi.isClicked() then
2:     setUpWifiConnection(device)
3: App app ← getSelectedApp(device)
4: app ← getAppInfo(app)
5: if login.isSelected() then
6:     app ← getLoginCredentials()
7: if trafficCapture.isSelected() then
8:     setUpTrafficCaptureThread(device)
9: if recordScreen.isSelected() then
10:    setUpRecordScreenThread(device)
11: if logCat.isSelected() then
12:    setUpLogCatThread(device)
13: startCrawler(device, app)
```

Then, when the crawler starts, firstly the tree is initialized, the app is opened, using *monkey* capabilities, and the initial activity node is added to tree. The activity's name is obtained using *dumpsys* capabilities, and parsing the command line output string using the *findstr* tool provided

by *WindowsOS*. Subsequently, the UI is dumped, using *UIAutomator*, to a XML file, then is parsed and interpreted, and all new nodes are added to the activity node, as represented in algorithm 3.

Algorithm 3 startCrawler(Device *device*, App *app*)

```
1: Tree tree ← initializeTree()
2: openApp(device, app)
3: String activity ← getCurrentActivity(device)
4: Node activityNode ← createActivityNode(activity)
5: tree.addChildren(activityNode)
6: dumpUI(activityNode)
7: crawling(activityNode.getChildren())
8: return tree
```

In this step, the main difficulty was to parse existing elements without ID and any text, per example RelativeLayout boxes, and several elements with same ID inside different activities, per example EditText boxes. Regarding the first case, it only happens with ViewGroups elements, i. e., a container that can contains several Views in it, such as, RelativeLayout, LinearLayout, FrameLayout and others. To solve this problem, if the ViewGroup is clickable, the clickable attribute of its children will be set to true and the clickable possibility of the father is removed, as can be seen in figure 4.7, where the green boxes means that the element is clickable and the white and dashed boxes means that the element is not clickable.

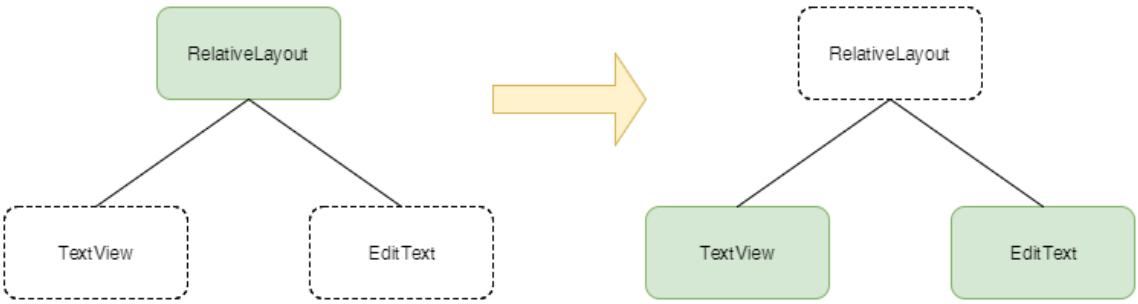


Figure 4.7: Resolution of the problem with elements without ID and any text.

Regarding the elements with same ID, first of all it will be take for granted that for a given activity, any element of it has a different ID. In this way, for each element is associated the activity that it belongs, in order to avoid same elements to be represented in the tree data model.

After that, the crawling process entered in a recursive method, performing the crawling. Recursion was the way found to simplify the problem, being the key of this algorithm. With it, it is possible follow the approach "*Divide and Conquer*" and divide a bigger problem in a smaller ones, having less and cleaner code. In the end, a final tree object is returned, with all available nodes and patterns of the application identified by the crawler and that will be used for test case generation.

Regarding this recursive method of the crawler, described in algorithm 4, firstly it is necessary to copy the list of children nodes to another list, to prevent concurrency and stack overflow exceptions during the crawling process. Then, it is used an iterator to run across the list. For each node on the list it is verified if it was already visited and it is a clickable node, and if not it is marked as visited.

Algorithm 4 crawling(List<Node> *children*)

```
1:  List<Node> listOfNodes ← new List<Node> (children)
2:  Iterator it ← listOfNodes.iterator()
3:  while it.hasNext() do {
4:      Node node ← it.next()
5:      if !node.wasVisited() && node.isClickable() then {
6:          if nodeExistsInUI(node) then {
7:              tap(node.x, node.y)
8:              if keyboardIsEnable() then
9:                  keyboardDisable()
10:             if activityIsLoading() then
11:                 wait()
12:             if appWasChanged() then
13:                 back()
14:             if node.IsEditText() then
15:                 insertText(node);
16:             if activityIsTheSame() then {
17:                 if layoutChanges() then
18:                     node.addNewNodes()
19:             } else {
20:                 String activity ← getCurrentActivity(device)
21:                 if tree.hasActivity(activity) then {
22:                     node.setVisited(true)
23:                     node.addActivity(activity)
24:                     Node activityNode ← tree.getActivityNode(activity)
25:                     activityNode.addNewNodes()
26:                     node ← activityNode
27:                 } else {
28:                     Node activityNode ← new Node(activity)
29:                     node.addChildren(activityNode)
30:                     activityNode.addNewNodes()
31:                 }
32:             }
33:         } else {
34:             markFailedNode(node)
35:             failCount++
```

```
36:                 if failCount == 2 then {
37:                     failCount ← 0
38:                     restartApp()
39:                 } else
40:                     node ← node.getFatherActivity()
41:             }
42:         } else
43:             node.setVisited(true)
44:
45:         if node.hasUnvisitedChildren() then
46:             crawling(node.getChildren())
47: }
```

At this state of work, many problems appear namely because of dynamic elements presented in the GUI of the applications. Because of that, before tapping in a UI element is done a verification assuring that the element is really presented in the GUI. This search contemplate actions like swipe and scroll, as described in 4.4, performing BFS. If not, the node is marked as failed, and is given another opportunity to find it. Only in the last resort the application is restarted, and the crawler make the tracking to the last state of application is before the node be failed.

If the node effectively exists in the UI, it is clicked and are contemplated some verifications, that will be explained next.

Firstly, is verified if the keyboard appears. This is done because the dump of *UIAutomator* isn't capable to identify if the keyboard is enabled or not, having the possibility of it overlaps the UI elements. Per example, if the next clickable node is behind the keyboard, the crawler will "*think*" that it is clicking in the node, but, in reality, it is clicking on the keyboard, producing undesirable consequences. To solve this problem, if keyboard was enable, it is dismissed using *dumpsys* capabilities.

In second place, is verified if the activity is loading. Many application has asynchronous operations (due to waiting time from the response of REST or SOAP web service), and because of that, is necessary wait for the integral loading of the activity and its elements, before clicking in any element. To solve this problem, the dump of *UIAutomator* is filtered, and if it contains an UI element that characterize the loading (generally a progress bar or a resource ID with "*loading*" keyword) and if the layout was in constantly change, is necessary to wait till the layout changes stop.

Many application take advantage of other integrations, such as Email, Contacts and Gallery, in this case is necessary back to the selected application. This condition is also contemplated by the crawler.

Then, is verified if the UI element is a EditText. In this case, is necessary to verify if the activity contains a login form. There are a huge amount of applications that require login credentials, and this can be difficult to solve in this process. To make this verification, is checked if the activity contains the keyword "login". This was the solution found to solve in this problem. However, can be used other approaches, such as verify if the activity has, at least, one TextView filled with

"login" keyword or even the verification the existence of two EditTexts and one of that must has the password flag enabled.

Another element that must be specially considered is the WebView. A WebView element is an UI element provided by Android to display *HyperText Markup Language* (HTML) web pages inside the Android applications. Because there are another tools to test web applications and its content doesn't belongs to the Android application natively, this kind of element will be considered as a "dead end", and only be performed actions as scroll and swipe to verify what will be displayed in the WebView.

Subsequently, after an action, is performed the main verification, that is if the activity changes or not. When a button is clicked or a scroll/swipe action is triggered there are four possibilities:

1. activity change;
2. layout change and activity is still the same;
3. activity already exists in the tree;
4. layout and activity are still the same;

These four conditions are covered by the algorithm, but only for the first three precautions are taken, because the last one does not produce any traceable changes, as represented in figure 4.8. The blue node represent the root of tree, the red nodes represent activities, the yellow nodes represent the UI elements of each activities and the green nodes represent the actions. These representations are also used in figures 4.9, 4.10 and 4.11. The nodes with dashed lines represent visited nodes.

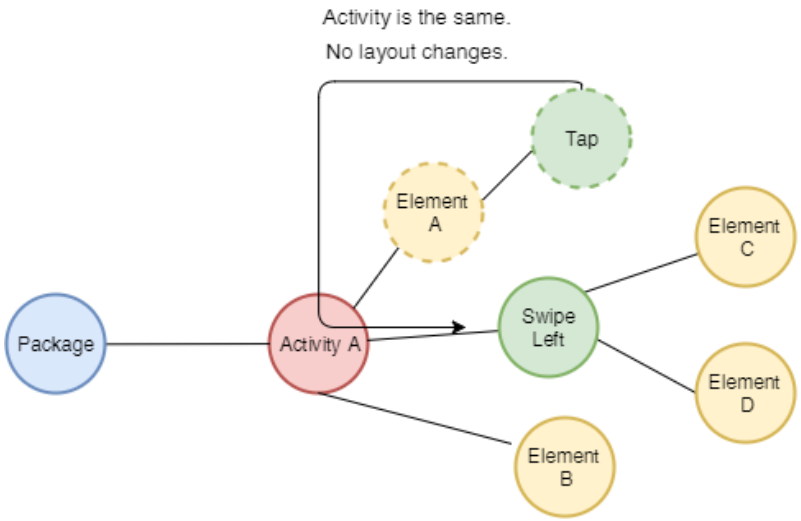


Figure 4.8: No changes in the layout and activity.

When the layout changes and the activity is still the same, is performed a new dump of the GUI, and the new nodes, not present yet in activity node, are added as children of current node, as represented in the figure 4.9.

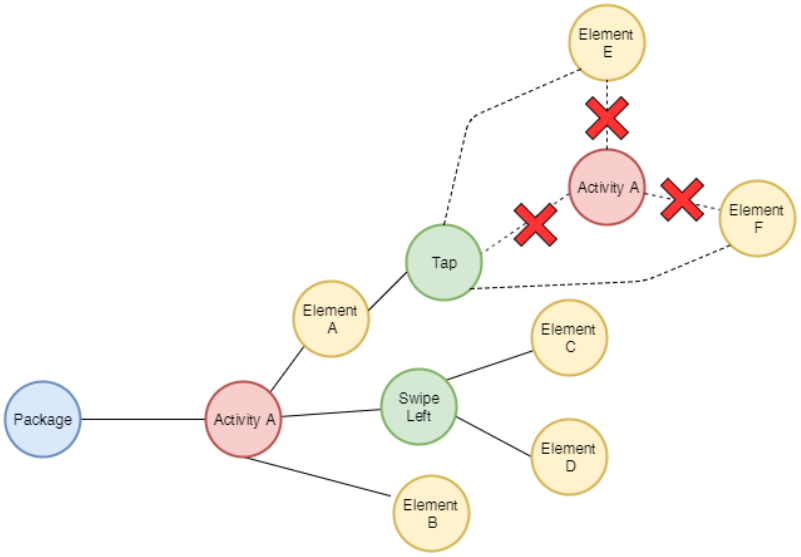


Figure 4.9: Changes in the layout, activity is still the same.

When the activity changes, firstly is verified if the activity already exists in the tree. In affirmative case, the activity node is extracted from tree, and compared with the new one. This kind of approach prevent infinite loops inside the application, and the activity name is stored into Node object, to represent this possible pattern, as represented in figure 4.10.

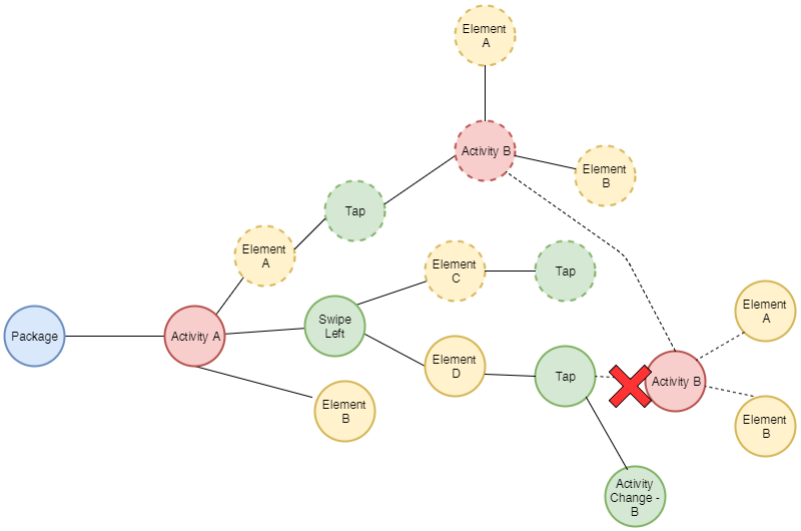


Figure 4.10: Activity already exists in tree.

In another hand, in a negative case, is created a new activity node, that is child of the current node, and are added the new nodes in the GUI to it. This is represented in figure 4.11, when the Element A of Activity A is tapped and new Activity appears (Activity B).

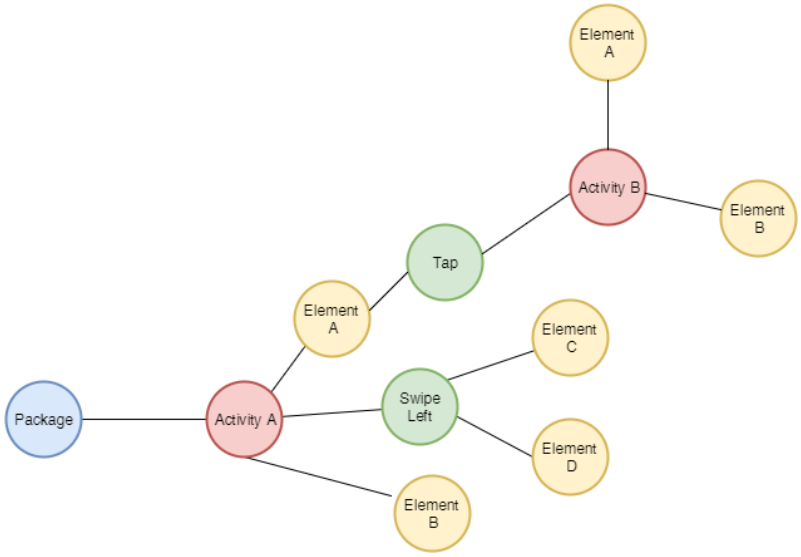


Figure 4.11: New activity appears.

After this detailed explanation about the algorithm based on pieces of pseudo-code and diagrams, all interactions and operations between desktop and device can be grouped and represented in a timeline, making a division between tools and outputs in the device, as represented in figure 4.12. Regarding the command line strings output, is important to report that the operative system used was *WindowsOS*. Other OSs are not contemplated by the crawler, that imply a few modifications namely in the tools of search and filter command line strings, per example, replacing *findstr* by *grep*.

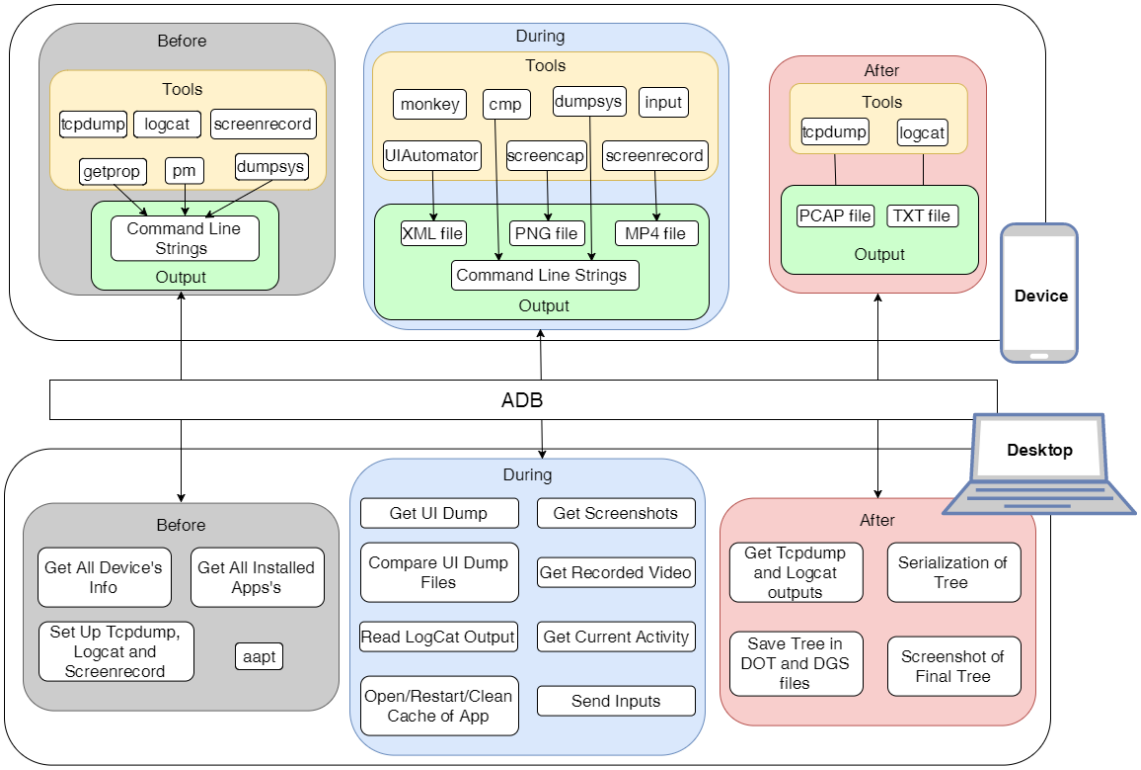


Figure 4.12: Crawling process.

At the end of crawling process, the tree is stored in a different formats to be used in the test case generation module. Furthermore, this tree object is serialized into a XML file and saved in a different formats, taking advantage of *Graphstream* library, to be simply parsed and used for another tools. A screenshot of final tree is taken, the *pcap* and log file are extracted, as the cache of the app is also cleared. Then, the tree is used by the test case generator module that will be described in the next section.

4.6 Test Case Generation

Before the explanation of this module, is necessary to answer some important question for a better understanding of the solution, such as:

- **Q1:** Why generate test cases?
- **Q2:** What is the best format of a test case?

Generate test cases is the best way to automate the testing phase, that usually require a huge amount of time, however, write test cases also require it. In this way, automate this process becomes mandatory, primarily to reduce the time spent with regression tests. Furthermore, write these tests allow to improve the quality of the final product.

A test case can be written using some different programming languages, that sometimes requires technical and specific skills, as unit tests. These skills usually are owned by developers, but not by the testers. Because of that, is necessary to use a understandable language by both sides. As reported in 3.5.2 and in 3.5.3, a solution that best fits to this is a DSL based tool as *Cucumber*. Using a natural language programming allow any person to write and understand every test case with a low learning curve. Furthermore, when is necessary, is more easier make changes in the test case.

After the response to these question, now is important to understand how the test case generator module was implemented.

As said in the previous section, the final tree is saved in a XML file. Now, in this module, is necessary to read, parse, and make a graphical representation of it. In this phase, is made a leaves extraction, i.e., for each extremity of tree (known as a leaf), is extracted the full pattern from the root to the leaf, as represented by green nodes in figure 4.13.

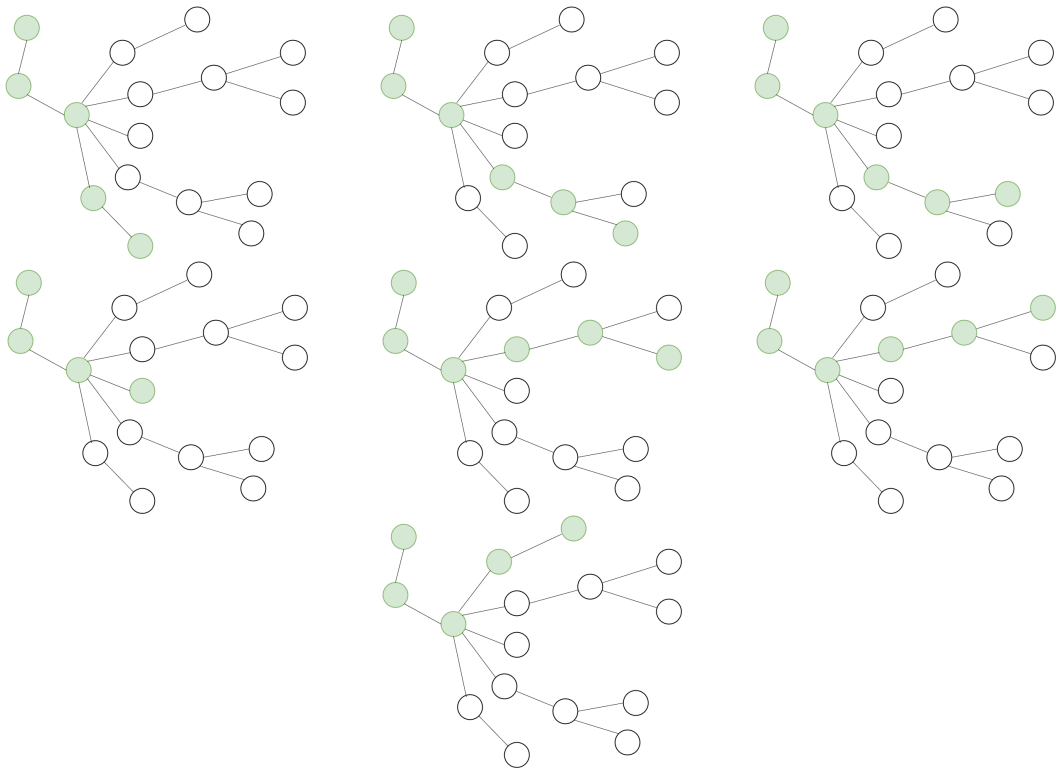


Figure 4.13: Leaves full pattern extraction.

Each pattern corresponds to a test case, however, it translation is not so simple. In this way, to make this parsing was developed a data model, based on best practices of *Gherkin* language, composed by features, scenarios, steps and its description. As can be seen in figure 4.14, a feature is composed by a name, description, a stakeholder that describes the correspondent user story, a goal that describes the purpose of test case, and one or more scenarios. Each scenario is composed by a unique identifier (ID), by a name, one or more steps and tags. Each step contains also an

unique identifier and a string that describes it. At the end of translation is generated a .feature file with full definition of the feature, and that can be run in automation platforms as *Jenkins* [1].

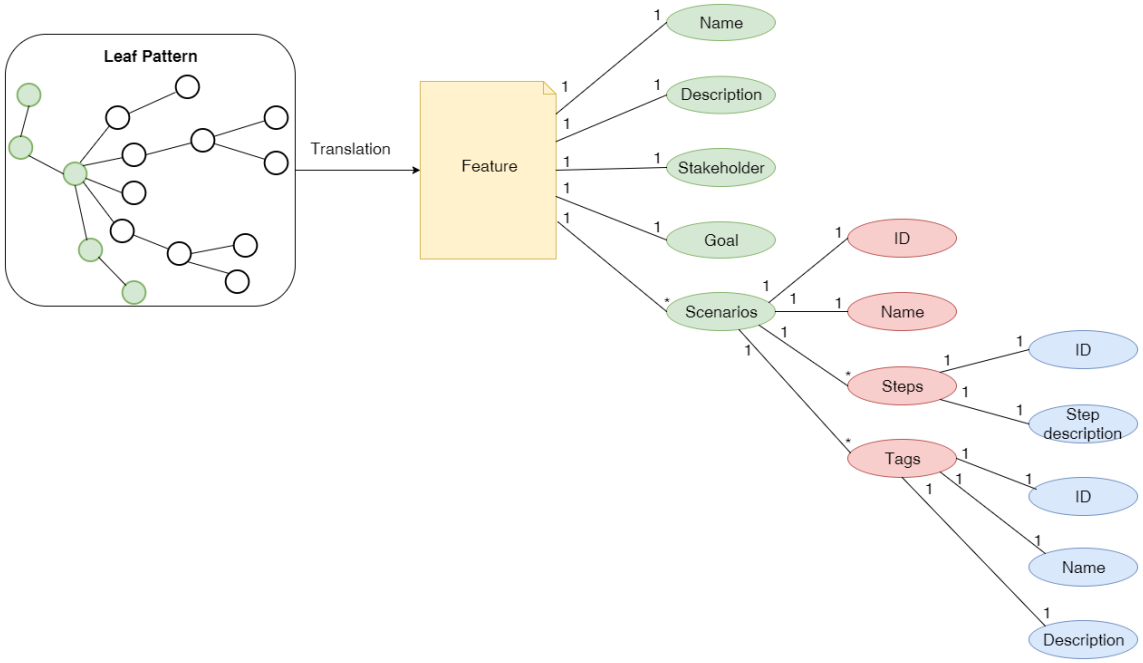


Figure 4.14: Test Case Generation.

At this step of the work, the user can see the full pattern, its test case and can make some configuration based on description of test case. The user can edit each feature, scenario and step description. Furthermore, it is also possible add new or make changes in the scenarios and steps, if desired. Then, it is possible to save the feature in a file, similar to the test case presented in algorithm 5, or discard it, bringing to the next pattern.

Algorithm 5 Test Case Example

```
1:  Feature: <Test76>
2:  In order to <Meet Some Goal>
3:  As a <type of stakeholder>
4:  I want <Feature Description>
5:
6:      @perf @obsessive_screenshots
7:      Scenario: <Scenario0>
8:
9:          Given the app <com.vodafone.mCare> is running
10:         When I start measuring "<Scenario0>"
11:         Then I press the "Aceitar" button
12:         Then I press the "Rejeitar" button
13:         Then I enter "910106596" into text field number 0
```

[1] *Jenkins*. April, 2016. [Accessed: Apr. 04 2016]. URL: https://jenkins.io/.

```
14:     Then I enter "vodafone2016" into text field number 1
15:     Then I press the "Login" button
16:     Then I wait for "910106596" to appear
17:     Then I touch "910106596" text
18:     Then I wait for 13 seconds
19:     Then I wait for "Logout" to appear
20:     Then I press the "Logout" button
21:     Then I finish measuring "<Scenario0>"
```

Here appears a big problem, the test case explosion, already exposed in description of model based tools in section 3.5.2. This is a unavoidable problem, that this model still not succeeded in resolving. To solve this problem, a new module is included in this dissertation, that will be presented in the following section.

4.7 Test Case Capture

Using a similar and a half approach to capture and replay type of tools, already reported in section 3.5.2, as *RERAN*, with this model, a test case can be captured manually, if the user desires a single test case, resolving the test case explosion problem reported in the previous section. This module use the same data model described in figure 4.14, generating a similar test case to the test case example 5.

During the recording process, *UIAutomator* and *getevent* tools capabilities are used to extract all the elements present in the GUI of the application and to extract the action performed, respectively. Cross-linking this data, is possible to find the element clicked, in case of tap action, or, in case of swipe (left/right) and scroll(up/down) actions, is possible to decode it, and translate them to a natural language step, defining a specific scenario and feature, as can be seen in figure 4.15.

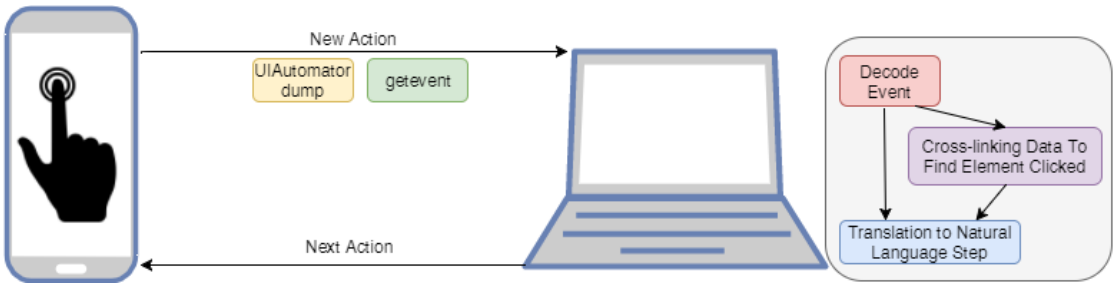


Figure 4.15: Test case capture process.

The event decoding process have three different possibilities, depending of the action performed or key clicked, as can be seen in figure 4.16. In case of key clicked, it is possible to track the back, homepage and menu button, and this kind of event has a specific *signature* that can easily be translated. Regarding the actions performed as tap, swipe and scroll, they have the same start and end *signature*, and similar position *signature*. With the flags ABS_MT_POSITION_X and ABS_MT_POSITION_Y is possible to extract the position on the screen where the user clicks.

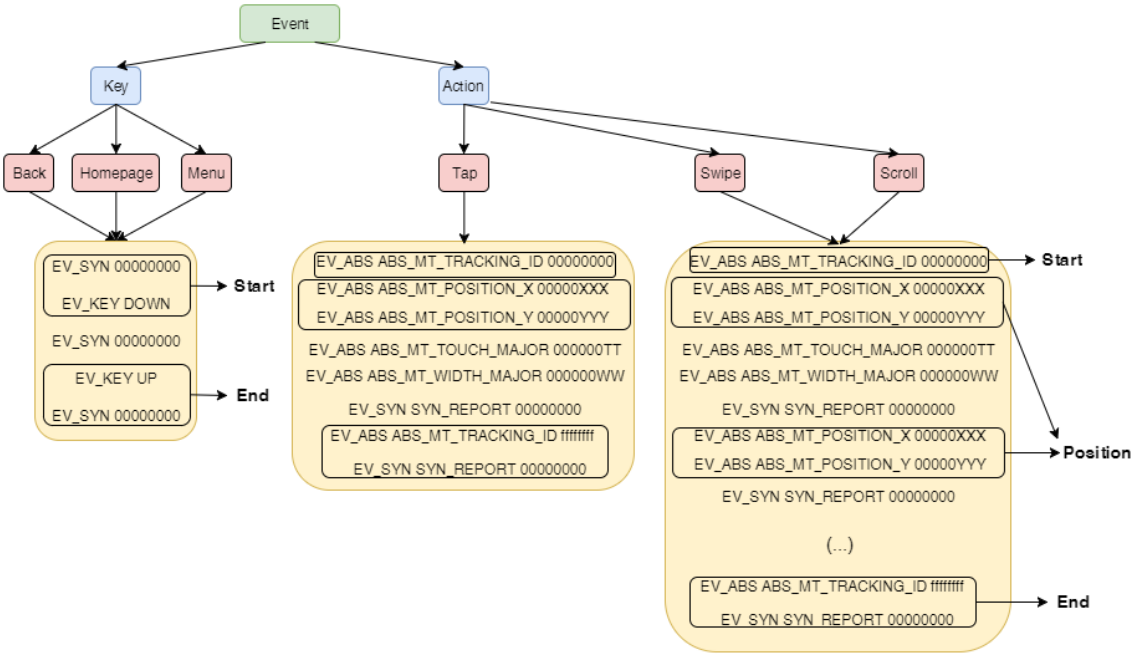


Figure 4.16: Event Decoding process.

To stop the capture of the test case, the user can define and choose on the beginning one specific key of three: back, homepage and menu. By default, the menu key is defined as stop button because, in almost all applications, between the others, this button less influence the behavior of the application. The other flags not used as EV_SYN, ABS_MT_TOUCH_MAJOR and ABS_MT_WIDTH_MAJOR, are used to divide the events and to give some information about the cross-sectional area of the touch contact and of the tool itself, respectively.

The only difference between the track of tap actions and swipe/scroll actions is that the second ones have a huge variety of positions till the end. In this specific case, only the start position and the last position will be considered to extract two different points in order to decode them in a swipe or scroll action.

To make a distinction between these two actions and its direction, is calculated the gradient based in start and end positions, as can be seen in equation 4.1.

$$gradient = \frac{start.y - end.y}{start.x - end.x} \tag{4.1}$$

Then, the angle of the action is obtained, based in equation 4.2, that will be useful to make a distinction between scroll and swipe actions, as presented in algorithm 6.

$$angle = abs(toDegrees(atan(gradient))) \tag{4.2}$$

Algorithm 6 Swipe and Scroll Distinction

```
1:  if(angle > 45.0) {
2:      if(start.y > end.y)
3:          step = SCROLL_UP;
4:      else
5:          step = SCROLL_DOWN;
6:  } else {
7:      if(start.x > end.x)
8:          step = SWIPE_LEFT;
9:      else
10:         step = SWIPE_RIGHT;
11: }
```

Regarding the tap action, the position of the touch contact is gathered as the dump of the application's GUI. Then, the correspondent node based on the position is extracted and translated in a natural language step. After this, a new event will be expected. It is possible to perform this even via USB or Wi-Fi, costing almost always some time of *UIAutomator* dump processing.

Despite of this module can be very useful to capture a specific test case or a test case that the crawler can't covers, it has some limitations that at this state of work were impossible to avoid, such as in EditText UI element is impossible to capture the inserted text on it, and time restrictions, because the user have to wait until the authorization was given to him by the module to perform the next step, because the UI of the application must be dumped and parsed before, requiring some waiting time to do it.

After the analysis of the test case capture module, it becomes important to analyze another important module included in this work, the traffic capture, that will be described in the following section.

4.8 Traffic Capture

As described in section 4.3, before traffic analysis is required to capture all traffic generated by the device using *tcpdump* capabilities and it is only possible in rooted devices, due to superuser permissions required, and in applications that have Internet permissions defined in Android manifest file.

Because, one of the main goals of tool is to be the more black-boxed possible, to verify if device is rooted is used a simply try of running superuser commands in *ADB* shell, like *su*, and to verify if the Android application has the Internet permission is used an auxiliary tool called *AAPT*, that can extract the permissions of a given APK file.

In this way, before the crawler process starts, this verification is done, and in affirmative case, is pushed to device the *tcpdump* binary. Then, is run the proper command to start the capture, as described in appendix B.2. When the crawler ends, the process that runs *tcpdump* is killed and the *pcap* file with capture is pulled from device to desktop, as can be seen in figure 4.12.

Before the traffic analysis, a pre-defined file is read, where the user can places important words such as username, password, email, *Personal Identification Number* (PIN), *Mobile Station International Subscriber Directory Number* (MSISDN) and others, acting like a dictionary.

After this phase, to perform the analysis of capture file is used and external library called *jnetPcap* [2]. With this library is possible to decode the *pcap* file and obtain almost all content of each packet with some protocols available such as IP, TCP, *User Datagram Protocol* (UDP), Ethernet, HTTP requests, responses and its payload.

In case of HTTP packets, the dictionary file previously filled is used to verify if the payload of this packet contains any word of it. If this happen, means that the Android application has encryption vulnerabilities, compromising user's data.

In the end of the analysis, all packets and its content is dumped to a .txt file, in a understandable way, and some stats about traffic such as number of packets of each protocol, duration of file analysis and duration of capture.

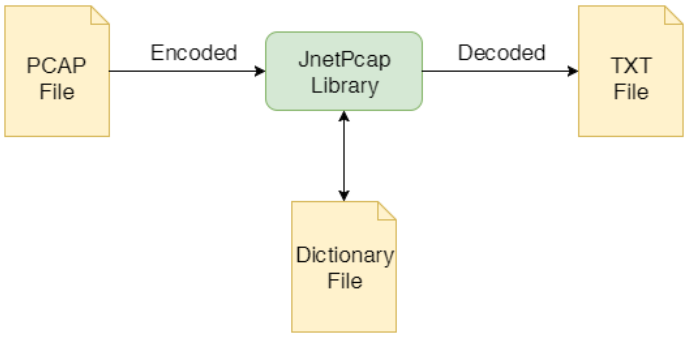


Figure 4.17: JnetPcap Decoding.

However, the library used has some drawbacks. Many Android application send/receive XML and *JavaScript Object Notation* (JSON) objects to its correspondent REST or SOAP web services, that many times contains important data about the application's user, and is impossible to decode with this library. The other drawback is that this library does not support all type of protocols.

This chapter provided a detailed overview of how the proposed solution is built and how it should work for several possible use cases. The next chapter focuses on demonstrating how the solution performs on across several applications and devices, and its impact in the software testing process.

[2] Sly Technologies Inc. *JnetPcap*. 2014. [Accessed: Feb. 02 2016]. URL: http://jnetpcap.com/.

Chapter 5

Crawler Evaluation

This chapter will focus on the performance evaluation of the proposed solution. Firstly, it is defined the criteria to select the application's dataset. Then, it is explained the different methodologies carried out to validate the proposed solution. Finally, the results are shown and discussed regarding a case of study, the solution expansion to another applications and the evidences' analysis captured during its process.

In order to assess the performance and completeness of the proposed solution, it was conducted several tests to answer the following questions:

- **Q1:** Can the GUI crawler cover all of the application's patterns?
- **Q2:** How expansible is the crawler to another applications ?
- **Q3:** How useful can the solution be to find bugs and vulnerabilities ?
- **Q4:** Is the tool faster generating test cases than have them manually written?

These questions, define the goals of this assessment, and will be answered after the analysis of results, that can be seen in the following sections.

5.1 Apps' Dataset

Since February 2016, Google Play Store has more than 2 million apps available for download [1]. This number of apps is only indexed for the current Android version (6.0 - Marshmallow), being more reduced for others version. Due to this huge amount of apps, it is impossible to test and achieve results for each application, timely. In this way, to have a good sample of applications' diversity it was selected a set of 20 apps for study, based on different criteria, such as:

- Different applications' categories, in order to demonstrate the dynamism of the proposed solution;

[1] Statista. *Number of available applications in the Google Play Store from December 2009 to February 2016*. February, 2016. [Accessed: May 11 2016]. URL: http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

- Free and freemium applications, excluding paid applications, to avoid extra costs. Freemium applications are free applications that include paid additional resources or features in it.
- Popularity, to test apps commonly used among the World;
- Size of app, to test apps with different range of sizes.

Based on these metrics, it can be chosen a set that fully represents all the Android applications available on the market. Due to this fact, it was chosen the applications presented in table 5.1.

Table 5.1: App Dataset

ID	**App's Name**	**App's Package**	**Category**	**Size**[2]	**Down-loads**[3]
1	My Vodafone	com.vodafone.mCare	Productivity	7,11	500k+
2	Placard	pt.scml.placard	Sports	15,99	500k+
3	Calculator	com.android.calculator2	Productivity	0,10	System's App
4	Terminal Emulator	jackpal.androidterm	Utilities	0,54	10M+
5	Uporto	com.moofwd.uporto	Education	27,59	1k+
6	Tinder	com.tinder	Life Style	14,96	50M+
7	Airbnb	com.airbnb.android	Travels	43,39	10M+
8	Farmácias Portuguesas	pt.anf.farmaciasportuguesas	Health and Fitness	57,85	10k+
9	JN	pt.civ.jn	News and Magazines	10,97	100k+
10	Facebook Messenger	com.facebook.orca	Communication	35,33	1B+
11	Juasapp	es.mrcl.app.juasapp	Entertainment	5,86	5M+
12	Instagram	com.instagram.android	Social	15,04	500M+
13	2048	com.androbaby.game2048	Games	3,20	5M+
14	Boomerang	com.instagram.boomerang	Photo	4,88	10M+
15	PT Airports	com.innovagency.ana	Transport	14,23	100k+
16	Empregos	com.indeed.android.jobsearch	Companies	1,46	10M+
17	eBay	com.ebay.mobile	Shopping	18,12	100M+
18	Disney CR	com.disney .disneycrossyroad_goo	Familly	48,00	5M+
19	Spotify	com.spotify.music	Music and Audio	27,00	100M+
20	Me-teo@IPMA	pt.ipma.meteo	Wheather	5,06	100k+

[2]Size represented in megabyte (MB) units.
[3]k - Thousand (10^3), M - Million (10^6), B - Billion (10^9), + - or plus. Extracted from *Google Play Store.*

After the choice of the applications to be analyzed, now it is important to define the methodology used to achieve the results. This is the subject that will be approached in the following section.

5.2 Methodology

In order to measure and assess the developed solution, the performance and completeness terms, it was chosen three distinct metrics: *Activity Coverage* (AC) (absolute and relative) to evaluate its completeness, duration time to evaluate its performance and its time variance between each connection mode, and *Mean Time per Test Case* (MTTC) spent to prove or not the utility of this tool. This choice was made because from these metrics can be extracted some good conclusions to answer the previously placed questions in the chapter's introduction.

Absolute AC is defined as the ratio between the number of *Number of Activities Found Automatically* (NAFA) during execution of crawler and *Total Number of Activities* (TNA), as represented in equation 5.1.

$$\mathrm{AC}(Absolute) = \frac{\mathrm{NAFA}}{\mathrm{TNA}} \tag{5.1}$$

The number of NAFA can be only obtained dynamically from the crawling process, however TNA can be only retrieved statically. As described in 2.2.2, Android has a manifest file, where all included activities in the application are defined, and this can be used to retrieve TNA. But, to read the manifest file is necessary to decompile the application, that can be a hard task. To speed up this process, it was used a reverse engineering tool, already described in section 3.4.1, called *APKtool* that can used to gather access to the source code, assets and manifest file by given APK. Having the manifest file, the TNA can be extracted counting the number of existing activity tags.

Sometimes it is impossible to find all of the activities described in the manifest file, because they are hidden or are not being used. The majority of these activities are used to show ads, to make integrations with another apps or frameworks or simply to make tests. An activity does not always have an associated GUI. In this way, it was developed a Java program to extract the *Number of Activities Found Manually* (NAFM), during a tester session, where the tester tries to run across all patterns available. So, it was specified the relative AC, that was defined as the ratio between the number of NAFA during execution of crawler and NAFM, as represented in equation 5.2.

$$\mathrm{AC}(Relative) = \frac{\mathrm{NAFA}}{\mathrm{NAFM}} \tag{5.2}$$

From both equations 5.1 and 5.2 it can be concluded, intuitively, that as higher the AC is, the higher the number of application's patterns that can be explored.

To evaluate the speed of the solution, it is important to measure the duration of the crawling process. This calculation can be achieved by using a simple formula, represented in equation 5.3.

$$Duration = EndTime - StartTime \tag{5.3}$$

Intuitively, the smaller the duration is, the faster all of the patterns of the application will be discovered. Regarding this duration, it will be made a comparison between the performance of the crawler in USB versus Wi-Fi.

After this detailed description about the methodologies used to evaluate the solution, now, in the following section will be presented the results and all important data obtained.

5.3 Case Study: My Vodafone App

At first step, the crawler was developed bearing in mind an Android application with sufficient complexity and diversity of operations. So this criteria be fulfilled, it was chosen the Portugal's version of the My Vodafone App for solution's initially testing. Its general characteristics can be seen in the table 5.2 and they are the reason of this choice.

Table 5.2: My Vodafone app characteristics.

ID	App's Name	Version	TNA	Authentication	Asynchronous Operations	WebViews
1	My Vodafone	2.6.2	138	Yes	Yes	Yes

This application contains at total 138 activities (statically extracted and parsed from manifest file), requires authentication to access all app's features, has asynchronous operations (communication with web services/servers), and contains WebViews. If all of these situations can be covered, it can solve some problems of the related work tools, already described in section 3.6.

After this superficial analysis, now in the following subsection it will be presented the obtained results of the solution for this application.

5.3.1 Results

To perform these tests it was used the device Vodafone Smart Ultra 6, via USB, with Android version 5.1.1 (Lollipop), not rooted, unencrypted, with screen size 1080x1920 and with a *Subscriber Identity Module* (SIM) card associated to Vodafone Portugal operator.

The results of these tests are presented in the table 5.3 and 5.4.

Table 5.3: My Vodafone App - Crawler results.

Duration[4]		NAFA	NAFM	AC(%)		Generated Tests	Restarts
USB	Wi-Fi			Relative	Absolute		
03:02:47	03:24:54	49	68	72,06	35,51	237	25

Table 5.4: My Vodafone App - Crawler UI elements results.

UI Elements	EditTexts	TextViews	ImageViews	Buttons	Others
502 (100%)	24 (4,78%)	239 (47,61%)	69 (13,75%)	107 (21,31%)	63 (12,55%)

As expected, the duration of crawling process via Wi-Fi is more slow than USB connection ($\approx$ 10% slower), due to packet loss, limited band, interferences from other sources and limited range. Another external and uncontrollable factor that must have into consideration is the server response, that could be slower at hours of greater affluence of clients to the service.

At this step, can be concluded that solution requires a huge amount of time (more than 3 hours). This duration time is due to the fact of loadings, restarts, search of element in the GUI, UI dumps pulling from the device and parsing.

Considering the generated tests from the crawling process, it was generated 237 test cases automatically using, in the best case, only 03:02:47 hours, corresponding to a single test case generated by 46 seconds, approximately, that can be considered faster than do it manually.

Furthermore, regarding table 5.4, can be concluded that the solution use a huge amount of UI elements, catching approximately 87,5% of them (EditText, TextViews, ImageViews and Buttons), doesn't interpreting the other ones.

After this kind of analysis, now, in the next subsection the crawler's performance and expandability to another devices will be evaluated properly.

5.3.2 Devices' Comparison

The goal of this test is to take some conclusions about the performance of the solution across different devices. An Android application can has different behaviors depending on the Android version and screen resolution. Due to that, these results will show what kind of device will be the faster to run the crawler.

In this test, only be considered Android devices with versions API 18 or higher, as already described in table 3.1, because these are the versions that supports *UIAutomator*.

As can be seen the crawler performs better in devices where screen is bigger, independently of the Android version. It can be justified due to the number of the scroll actions necessary to execute in the devices where the screen is smaller. To search new existing elements in a single activity, the crawling process will be more fast in this kind of devices.

The other conclusion that can be taken from these results is that, on average, the crawler is 13,09% slow via Wi-Fi than connected via USB, due to what was already described in section 5.3.1.

The device more fast in both connection modes is Samsung Galaxy S6 and the slowest is Vodafone Smart 4 Turbo. Regarding the difference between Samsung Galaxy S6, with API level

[4]Time format - hour, minute, second (hh/mm/ss)

Table 5.5: Crawler results using different devices.

Device	Resolution	Version	API Level	Duration		
				USB[4]	Wi-Fi[4]	Variance
Samsung Galaxy S6	1440x2560	6.0.1	23	03:00:40	03:23:07	11,05%
Vodafone Smart Prime 7	720x1280	6.0.1	23	03:15:22	03:44:32	12,99%
Vodafone Smart Ultra 6	1080x1920	5.1.1	22	03:02:47	03:24:54	10,79%
Alcatel One Touch Idol3	720x1280	5.0.2	21	03:11:45	03:37:04	11,66%
Huawei Ascend P7	1080x1920	4.4.2	19	03:05:33	03:34:42	13,58%
Vodafone Smart 4 Turbo	480x854	4.4.2	19	03:30:45	04:12:23	16,50%
Huawei Ascend G6	540x960	4.3	18	03:20:35	03:56:07	15,05%
			Mean:	03:12:30	03:41:50	13,09%

23, Vodafone Smart Ultra 6 and Huawei Ascend P7 with same resolution screen, although the resolution of the first is bigger than the others two, the difference between its result is very low, due to an additional step that the crawler have to pass on Android Marshmallow (permissions acceptance). This happens also with Vodafone Smart Prime 7 and Alcatel OneTouch Idol3. This additional step added in Android version 6 increases the crawling process time on each restart of the application.

Before take some conclusions about the solution, to evaluate it is necessary to collect data based in a different application's dataset. In this way, all of the applications will be specified, described and analyzed in the following section, presenting the results for each one.

5.4 Expandability To Another Applications

To perform this test it was used the device Vodafone Smart Ultra 6, via USB, with Android version 5.1.1 (Lollipop), not rooted, unencrypted, with screen size 1080x1920 and with a SIM card associated to Vodafone Portugal operator.

The results for the dataset described in table 5.1 are presented in the table 5.6.

Table 5.6: Crawler measures for USB connection. MTTC represented in seconds. "*" means that the crawler has undetermined end, due to loops and always new nodes appears.

ID	Duration			Authen-tication	Restarts	NAFA	NAFM	TNA	AC(%)		Genera-ted Tests	MTTC
	USB[4]	Wi-Fi[4]	Variance						Relative	Absolute		
1	03:02:47	03:24:54	10,79%	Yes	25	49	68	138	72,06	35,51	237	46,27
2	01:05:41	01:12:02	8,82%	No	35	3	4	6	75	50	80	49,26
3	00:27:31	00:30:24	9,48%	No	0	1	1	1	100	100	33	50,03
4	00:01:57	00:02:04	5,65%	No	0	1	3	8	33,33	12,50	5	23,40
5	00:04:10	00:04:54	14,97%	Yes	0	3	20	24	15	12,50	13	19,23
6	00:01:05	00:01:18	16,67%	Yes	0	1	7	23	14,29	4,35	4	16,25
7	00:03:01	00:03:22	10,4%	Yes	10	2	25	125	8,00	1,60	10	18,10
8	00:08:14	00:08:31	3,33%	Yes	1	2	4	6	50	33,33	10	49,40
9	00:08:16	00:08:24	1,59%	No	0	2	2	2	100	100	15	33,07
10	*	*	*	Yes	*	*	25	153	*	*	*	*
11	00:39:23	00:40:11	1,99%	No	4	9	9	14	100	64,29	67	35,27
12	*	*	*	Yes	*	*	10	22	*	*	*	*
13	00:02:10	00:02:14	2,99%	No	1	1	1	2	40	50	3	43,33
14	00:10:29	00:11:21	7,64%	No	1	1	2	2	33,33	50	5	125,80
15	00:09:07	00:09:43	6,17%	No	2	2	5	17	19,44	11,76	30	18,23
16	00:00:55	00:01:01	9,84%	Yes	0	1	3	4	33,33	25	2	27,50
17	*	*	*	Yes	*	*	36	122	*	*	*	*
18	00:02:04	00:02:10	4,62%	No	1	2	6	29	33,33	6,90	2	62,00
19	00:02:12	00:02:21	6,38%	Yes	0	2	12	80	8,33	2,5	5	26,40
20	00:02:04	00:02:22	12,68%	No	0	1	1	1	100	100	7	17,71
Mean:	00:21:50	00:23:57	7,88%	-	4,18	4,88	12,20	38,95	55,35	38,84	31,06	38,90

Firstly, regarding these results, can be concluded that the crawler achieves, on average, more than 50% activities of an application, according to what a tester can achieves manually (NAFM), having a more lower absolute AC, approximately 40%, according to the number of activites stated in the manifest file of an application (TNA).

Then, can be concluded that the restarts of the applications increase and penalize significantly the time performance of the tool. As much the number of the restarts is higher more probably the duration time could be.

As already stated in the last section, in these results the time performance of the crawler is better through USB connection than Wi-Fi.

Some social and e-commerce apps (Instagram, Facebook Messenger and eBay) presents a huge amount of possibilities, increasing exponentially the time of crawling process, and because they have constantly updates of the data on some activities, the crawler's algorithm is incapable to detect these loops. In this way, this algorithm calls for improvements similar to *PageRank* algorithm, a search tool of *Google*.

In the authentication required apps, the login is the "master key" to achieve more activities, however the crawler cannot detect all of login forms of some applications, as happen in the Spotify, eBay, Farmácias Portuguesas, Uporto and Tinder apps. At this step, some app authentication integrations, as Facebook, could "help" the crawler to achieve more higher results, as happen in the Instagram app. The crawler calls for improvements in the *à priori* definition of the login.

Some bugs and crashes decrease significantly the AC of the crawler, as happens in the Tinder app. Furthermore, the number of test case are usually bigger as the NAFA and the duration are.

On average the MTTC is usually smaller than 60 seconds, that can be interpreted as much more faster than specify and write each test case manually.

5.5 Evidences' Analysis

Before describe and detail the vulnerabilities found in the applications, and make its risk assessment, it is necessary define three related and important aspects in this area according [53]: likelihood, impact and risk.

5.5.1 Risk Assessment

The first one keyword is defined as the probability of a potential vulnerability being exploited by a threat-source, that can be a hacker, cracker, computer criminal, terrorist, insider or an industrial spy. In this specific case can be a malware installed on the device or *sniffer* that makes passive or active eavesdropping attacks. To evaluate this parameters is necessary take into account the motivation and capability of threat-source, the nature of vulnerability, and the existence/effectiveness of current controls. These factors describe likelihood in three levels: low, medium and high.

To understand impact is necessary divide it in two type: tangible impact and non-tangible impact. The first one, refers to the commitment of a system infrastructure and security mechanisms,

and should be based on the combination of integrity, availability and confidentiality losses. This impact can be quantitatively measured according the market losses, cost of repairs or the effort to replace the availability of the system. The second one, refers to reputation, credibility and confidence losses, and can be measured qualitatively, normally appealing to market surveys and others types of internal analysis. As likelihood, we will consider the same qualitative classification of three levels of impact: low, medium, high.

Regarding risk, as quoted in [53], it was defined as "*the net mission impact considering (1) the probability that a particular threat-source will exercise (accidentally trigger or intentionally exploit) a particular information system vulnerability and (2) the resulting impact if this should occur.*"

In this way, is necessary a model to classify the risk assessment based on likelihood and impact, one could be the risk classification model defined in [54]. This model classify risk in three qualitative levels according quantitative attributes, thus, to the low risk level corresponds 1, to the medium risk level corresponds 2 and 3, and to the high risk level corresponds 4 and 5, demonstrated in table 5.7.

Table 5.7: Risk levels classification.

		Likelihood		
		Low	Medium	High
Impact	Low	1	2	3
	Medium	2	3	4
	High	3	4	5

After the analysis of the concepts, now is a concern describe the main and common risks in terms of security that can be found in the analyzed applications. The following risks include only medium and high level risks and will be detailed with some examples of how to perform the respective attack.

5.5.2 Results

To perform this test it was used the device Vodafone Smart 4 Turbo, via USB, with Android version 4.4.2 (KitKat), rooted, unencrypted, with screen size 480x854 and with a SIM card associated to Vodafone Portugal operator.

In this section it is presented all bugs, crashes, log errors and potential security issues found in table 5.8 and 5.9 by the use of crawler or its modules separately.

Table 5.8: Issues found.

ID	Bugs Found	App's Crash	App's Log Errors
1	3[E.2]	2[E.1]	Yes
2	0	0	No
3	0	0	No
4	0	0	No
5	0	0	Yes
6	1[E.6]	0	Yes
7	0	0	No
8	2[E.4]	0	No
9	1[E.3b]	1[E.3a]	Yes
10	0	0	Yes
11	0	0	Yes
12	0	0	Yes
13	0	0	Yes
14	0	0	No
15	0	0	No
16	0	0	No
17	0	0	Yes
18	0	0	No
19	0	0	No
20	1[E.5]	0	Yes

Table 5.9: Number of apps with potential vulnerabilities issues and its risk assessment levels

Potential Vulnerabilities	
Nº of Apps with Issues	**Risk Assessment**
3	5
1	4
2	3
0	2
3	1

During the run of crawler some blocking bugs are detected that freeze the application and avoid the solution to go further, this is the case of application 8 and 20. Furthermore, some app crashes were detected, and for this specific case, the crawler can overtake it, restarting the application and mark the node that trigger the event as visited.

Regarding the potential vulnerabilities found, it requires an analysis of the log and capture files, generated by the solution, looking for vulnerable data and app errors.

All bug and crash evidences can be found in appendix E. Furthermore, there are a huge number of applications that print errors in the log file, during the navigation through the app, and it must be corrected in the next release of the app.

Regarding the table 5.9, 9 potential vulnerabilities were found, and almost all were reported to the respective company, assuring a responsible disclosure policy. In this way, it is only presented in this dissertation the number of apps where a potential vulnerability was found and its respectively risk assessment. Before the issues be solved, it was decided to do not expose the sensitive data found, because it can be a false positive, or an issue already reported or even a serious security issue that can impacts the company's business and its client's reliability.

After the execution of this test cases, now it is possible to answer the questions placed in the beginning of this chapter and will be approached in the next section.

5.6 Discussion

Regarding all of this dataset of results and its analysis some conclusions can be taken in order to answer the questions stated in the beginning of this chapter. In this way, it will be presented, again, the answer and it respectively answer.

Q1: Does the GUI crawler can cover all application patterns ?

Regarding the case of study, the crawler has a good percentage of AC relative (72,06%). It can identifies more than 87% of UI elements and overtake some issues of related work tools, such as authentication, asynchronous operations and WebViews.

Q2: How expansible is the crawler to another applications ?

The crawler can be expansible to another applications, however the algorithm calls for improvements, namely regarding the way that the loadings, dialogs, apps' bugs, number of UI elements are detected, login forms filling and dynamic elements (notifications and ads) identification. With all of these improvements the duration time, even through USB and Wi-Fi, can be reduced significantly.

Q3: How useful can be the solution to find bugs and vulnerabilities ?

Regarding the bugs, crashes and security issues, the crawler and its modules can be very useful for bug and vulnerabilities reports and analysis, been possible to implement and integrate them into another tools or frameworks.

Q4: Is the tool more faster generating test cases than manually ?

As can be seen in the results, the specification of a test case could be much faster than do it manually, and the test case can be generated in two possible modes:

- The slow mode: here the user have to wait for the discovery of all application's patterns by the crawler. At the end of this process the user can choose and edit every test case in the way that he wants. The main drawback it is the huge amount of test cases generated, called test case explosion.
- The fast mode: where the user can capture a test case manually. The test case will be specified according the navigation of the user across the application's patterns.

In conclusion, these results shows that this tool can be very useful, even for test case generation as well as for runtime analysis.

So, after the description of the proposed solution and its evaluation, now, in the next chapter, will be presented all conclusions taken from this dissertation, the contributions of this work even to the science as well as to the company, and future work that can be done based on solution's core.

Chapter 6

Conclusions

In this chapter it will be presented conclusions about all the work done in this dissertation, as well as detailed the contributions and specified some suggestions of future work to be made after the completion of this dissertation.

6.1 Contributions

In this dissertation, it was introduced the importance of software testing for product's reliability and customer's attraction purposes. It was made a wide description about some of the existent software development models, focus on mobile development, more concretely, Android OS. This operative system was properly analyzed, specified and described regarding its architecture and components.

Furthermore, it was presented a deep analysis about the numerous and different existent test types, considering both the function and non-functional tests, the different types of testing methodologies and the different types of analysis that can be performed, enumerating some existent tools for each one.

Then, it was made an overview about GUI testing and a comparison between manual and automated testing and its different methodologies for each. Regarding the second one, some existing tools were properly described.

Regarding the solution, firstly it was presented some related work. In this section are presented gray and white box tools, that are characterized by its speed in test case generation and required only the APK file, however this type of approach is not used in this work because of the possibility of obfuscation of source code and the impossibility to capture evidences such as screenshots, video, logs, bugs, crashes and network traffic to perform user acceptance, system, regression, security tests.

Due to drawbacks identified in these tools, a sophisticated black-boxed Android GUI crawler was developed in order to automate the test case generation and runtime analysis.

The crawler takes advantage of the ADB's capabilities, in order to test several devices at the same time (even through USB and Wi-Fi connection), and the *UIAutomator* tool, in order to dump the UI elements presented on the Android device's screen and to build a tree data model.

In this way, the solution tries to map every application, testing every possible pattern available. Regarding the results, it has, on average, more than 50% of completeness, regarding to the number of activities that can be found manually, and a lower AC for activities that can be found in the manifest file.

The results show that the tool can be expanded to several devices, since the 18 to 23 Android API Level, independently of its brand and model. Furthermore, it can be also expanded to another applications, independently of its size and category's family. This automated tool can helps a tester, primarily in test case specification following a TAD approach, and can be useful to find untested patterns, facilitating the bug and crash reports to the application's support team.

After all the test cases be generated by the tool, its implementation, management and maintenance in an automated tool as *Jenkins*, they might be very useful in order to reduce the huge amount of regression tests that must be performed before a new release of an Android application, translating into a reduction of company's costs. Furthermore, the tool can replaces a common tester in easy test cases, such as scroll, action/reaction of buttons and its expected outputs, and so on. It can be also used to find bugs and crash situations as shown in table 5.8 and appendix E.

Considering the time of the crawling process and its generated test cases, the specification of each one is much faster than do it manually (less than 1 minute per test case, on average).

Regarding the evidences' analysis, the tool can provides a good help for a tester in order to improve the security issues analysis, that normally are being left behind. As presented in the results, based on potential vulnerabilities found, for a dataset of 20 applications, 9 of them have potential security issues, considering the presented risk assessment classification. The majority of them are classified as serious risk, and from these results can be concluded that there are yet few security concerns in mobile development, calling urgently for improvements.

However, the tool is only a prototype and a proof of concept. Many improvements can be done in the time performance and completeness of the algorithm. Its main drawbacks are duration time, the incapacity to go forward in some forms such login, being unable to detect different ways of loading (asynchronous operations), the application's restarts that increases significantly the duration time and the use of not all UI elements.

So, it can be improved in several ways, prospectively. In this way, were proposed in the following section some future improvements, that can take advantage of the solution's potentialities and of what was already developed.

6.2 Future Work

During this dissertation, new ideas, possibilities and different approaches emerged in order to take advantage of the developed solution. Mainly, due to time restrictions, decisions had to be made and possible new ideas were held back.

Regarding the developed solution, some improvements can be made, mainly related with:

- the crawling algorithm: currently the algorithm doesn't cover all possibilities, as results show, calling for new improvements, such as:
  - Detection of loadings, taken advantage of the unused data provided by the *dumpsys* tool of Android OS;
  - Handle with dynamic elements that appears with no reason in the GUI of application. This can be solved using a separately thread that dumps the UI and verify if the layout changes even without no interaction. If it happens new nodes should be added to the respective node;
  - Decrease of restarts, currently the search of the element is only done in the present activity. It could be improved implementing a better search of the UI element using back button to find it in other activities;
  - Increase of the UI elements identified, appealing to use reverse engineering and static analysis tools to improve it;
  - Better identification of login forms and the button that triggers this event. Currently, the detection of login forms is not so efficient. The identification of two EditTexts and one of that must has a password flag enabled, might be better for identification of authentication forms.
- the UI: beside the application has already an GUI, it is very rudimentary, calling for some improvements, more specifically it can be converted to a web interface, that can improve the user experience using some new technologies such as *AngularJS*, *React* and *D3JS*;
- a database: currently, the data is stored in files. It will be necessary store all important information in a relational or non-relational database for data's persistence;
- the search: given an application, can be implemented a search based in a keywords, showing patterns where this keyword can be found, and giving the possibility to generate the test for it pattern;
- the testing runner module: can be implemented a module that can runs Cucumber tests. With this module, the system can be centralized in only one tool, eliminating the dependency of tools and frameworks that can't be controlled directly in the source code.

Apart from these improvements, other new ideas appeared, and based in what was developed can be proposed new approaches using the core of this solution, such as:

- Cloud testing system: that is a system were many mobile devices are connected, and where the solution is running, giving to the user the possibility of runs crawler in every device, having access to all information captured during it. Furthermore, can be given to the user the possibility of interact with the device, through a web page, like he wants;

- Automated capture and replay apps: this tool can be very useful, namely to perform regression tests. The test or a dataset of tests can be captured and before the release of the product, all of it can be executed automatically without human intervention, saving time, human resources and consequently money. Furthermore, currently there are some apps that do this, however almost all require a rooted device. With this approach it can be made without superuser permissions;

- Automated scroll apps: currently there are a huge amount of social and news apps that requires many scroll actions to the user see new information. Based in the developed solution, can be made an app, that scrolls other apps without user intervention automatically, and it also possible adjust its speed, without root. It can be useful also for big files and books reading. As the previous idea, there are already in the market apps that do this, however almost all require a rooted device;

- Image analysis: it can be made a image analysis of the activities and UI elements found in the application, in order to improve the crawling algorithm and to take conclusions about user experience;

- Apply *Google* search to Android apps: this is the most utopian idea. The purpose of this idea is to apply search to all application's data inside every application of *Google Play-Store*, extracting the data model and all captured evidences that can be important to report to developers in case of errors, building a test system available for everyone.

In conclusion, the problem was thoroughly analyzed and a solution was presented and detailed. After a series of tests, it can be concluded that the proposed solution does in fact improve the software test automation and is a subject with potential that should be further explored.

Appendix A

Activity Lifecycle

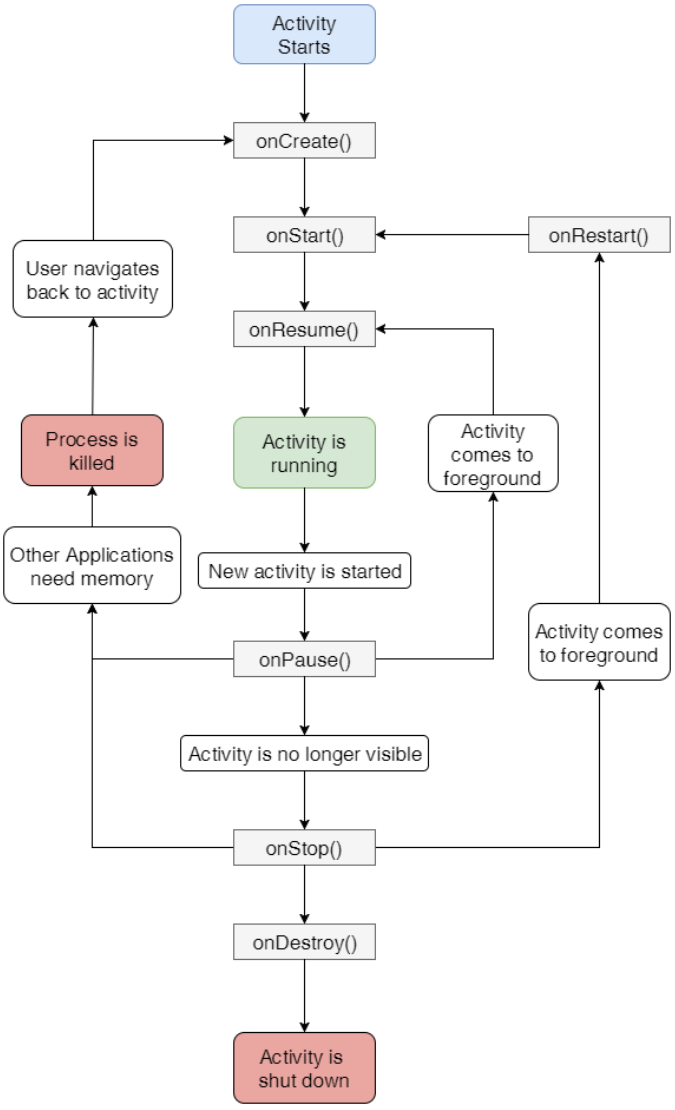


Figure A.1: Activity lifecycle.

Appendix B

Commands

After conclusions and before references, are presented in this appendix the commands used by the Java API developed, where '>' or '$' in the beginning means the command line of Windows and Android, respectively, the '$WORD' must be replaced by the correct information, and the proper '#Comment' that explains what the command does.

B.1 ADB

```
>adb start−server #Start adb server
>adb kill−server #Stop adb server
>adb devices #list devices connected
>adb shell #access shell device − one device connected
>adb −s $SERIAL shell #access shell device − many devices connected
>adb −s $SERIAL pull $SRC $DEST #Pulling files from the device to desktop
>adb −s $SERIAL push $SRC $DEST #Pushing files from the desktop to device
>adb −s $SERIAL tcpip 5555 #Set up TCP port to connect shell via Wi−Fi
>adb −s $SERIAL connect #Connect shell via Wi−Fi
>adb −s $SERIAL install $APKPATH #Install app from desktop
```

B.2 Device's Shell

Note : to run the following commands from desktop, they must be preceded by:

```
>adb −s $SERIAL shell
```

```
$getprop ro.product.model #Get device model
$getprop ro.build.version.sdk #Get device API version
$getprop ro.build.version.release #Get device Android version
$getprop ro.product.brand #Get brand of device
$getprop ro.product.locale.language #Get language of device
$getprop ro.product.locale.region #Get region of device
```

```
$getprop | grep wlan0.ipaddress #Get Wi−Fi IP address of device
$getprop persist.sys.timezone #Get Timezone
$getprop ro.nfc.port #Get NFC port
$getprop ro.crypto.state #Get crypto state
$getprop sys.usb.state #Get USB state
$getprop gsm.operator.numeric #Get operator numeric
$getprop gsm.sim.state #Get SIM state
$getprop gsm.operator.isroaming #Get roaming state
$getprop gsm.operator.alpha #Get operator
$getprop ro.build.date #Get build date
$getprop persist.radio.sim.imsi #Get IMSI
$getprop net.dns1 #Get gateway
$dumpsys window windows | grep mSurface=Surface(name=InputMethod) #Verify if keyboard is enable
$dumpsys package $PACKAGE #Dump info about a application
$dumpsys window windows | grep mCurrentFocus #Get current package/activity displayed
$dumpsys window displays | grep init #Get screen size
$dumpsys battery | grep level #Get battery level of device
$dumpsys input_method | grep mInteractive #Verify if device is locked
$dumpsys | grep SurfaceOrientation #Get screen orientation
$screenrecord −−bit−rate $BITRATE −−size $SCREENSIZE −−time $TIME $DEST
$pidof #Verify if device is rooted
$pm uninstall −k $PACKAGE #Uninstall app
$pm clear $PACKAGE #Clear cache of app
$pm list packages −3 #Get list of 3rd party installed apps
$pm list packages −s #Get list of system installed apps
$pm path $PACKAGE #Get local path of the APK file
$monkey −p $PACKAGE −c android.intent.category.LAUNCHER 1 #Run app
$am force−stop $PACKAGE #Stop app
$ls /system/bin/uiautomator #Verify is uiautomator is installed
$rm $FILEPATH #Remove file
$uiautomator dump $FILEPATH #Dump current activity to a XML file
$screencap −p $FILEPATH #Take screenshot
$su −c ./$PATH/tcpdump −vvvXSs 0 −w $DEST #Run traffic capture
$cmp $FILE1 $FILE2 #Compare the content of two files
$mv $SOURCE $DEST #Move file
$input tap $X $Y #Tap in screen
$input swipe $X1 $Y1 $X2 $Y2 $TIME #Swipe in screen
$input text $TEXT #Write text in input fields
$input keyevent $NUM #Send key events to device
$content insert −−uri content://settings/system −−bind name:s:user_rotation −−bind value:i:1 #Change
    orientation to landscape
$content insert −−uri content://settings/system −−bind name:s:user_rotation −−bind value:i:0 #Change
    orientation to portrait
$netcfg | grep rmnet0 #Get 3G IP address
$netcfg | grep wlan0 #Get Wi−Fi IP address
$getevent −lt /dev/input/event7 #Dump touch events
```

Appendix C

UIAutomator Dump Example

This is the file output obtained by running UIAutomator dump command, supporting all crawler logic.

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<hierarchy rotation="0">
  <node index="0" text="" resource-id="" class="android.widget.FrameLayout" package
      ="jackpal.androidterm" content-desc="" checkable="false" checked="false"
      clickable="false" enabled="true" focusable="false" focused="false" scrollable
      ="false" long-clickable="false" password="false" selected="false" bounds="
      [0,0][1080,1920]">
  <node index="0" text="" resource-id="android:id/decor_content_parent" class="
      android.view.View" package="jackpal.androidterm" content-desc="" checkable="
      false" checked="false" clickable="false" enabled="true" focusable="false"
      focused="false" scrollable="false" long-clickable="false" password="false"
      selected="false" bounds="[0,0][1080,1920]">
  <node index="0" text="" resource-id="android:id/action_bar_container" class="
      android.widget.FrameLayout" package="jackpal.androidterm" content-desc=""
      checkable="false" checked="false" clickable="false" enabled="true" focusable=
      "false" focused="false" scrollable="false" long-clickable="false" password="
      false" selected="false" bounds="[0,75][1080,243]">
  <node index="0" text="" resource-id="android:id/action_bar" class="android.view.
      View" package="jackpal.androidterm" content-desc="" checkable="false" checked
      ="false" clickable="false" enabled="true" focusable="false" focused="false"
      scrollable="false" long-clickable="false" password="false" selected="false"
      bounds="[0,75][1080,243]">
  <node index="0" text="" resource-id="" class="android.widget.Spinner" package="
      jackpal.androidterm" content-desc="" checkable="false" checked="false"
      clickable="true" enabled="true" focusable="true" focused="false" scrollable="
      false" long-clickable="false" password="false" selected="false" bounds="
      [48,122][299,195]">
  <node index="0" text="Janela 1" resource-id="" class="android.widget.TextView"
      package="jackpal.androidterm" content-desc="" checkable="false" checked="
      false" clickable="false" enabled="true" focusable="false" focused="false"
      scrollable="false" long-clickable="false" password="false" selected="false"
      bounds="[48,122][251,195]" /></node>
```

```
9  <node index="1" text="" resource-id="" class="android.widget.LinearLayout" package=
       "jackpal.androidterm" content-desc="" checkable="false" checked="false"
       clickable="false" enabled="true" focusable="false" focused="false" scrollable="
       false" long-clickable="false" password="false" selected="false" bounds="
       [672,75][1080,243]">
10   <node index="0" text="" resource-id="jackpal.androidterm:id/menu_new_window"
         class="android.widget.TextView" package="jackpal.androidterm" content-desc="
         Nova janela" checkable="false" checked="false" clickable="true" enabled="true
         " focusable="true" focused="false" scrollable="false" long-clickable="true"
         password="false" selected="false" bounds="[672,87][816,231]" />
11 <node index="1" text="" resource-id="jackpal.androidterm:id/menu_close_window"
       class="android.widget.TextView" package="jackpal.androidterm" content-desc="
       Fechar janela" checkable="false" checked="false" clickable="true" enabled="true
       " focusable="true" focused="false" scrollable="false" long-clickable="true"
       password="false" selected="false" bounds="[816,87][960,231]" />
12 <node index="2" text="" resource-id="" class="android.widget.ImageButton" package="
       jackpal.androidterm" content-desc="Mais opcoes" checkable="false" checked="
       false" clickable="true" enabled="true" focusable="true" focused="false"
       scrollable="false" long-clickable="false" password="false" selected="false"
       bounds="[960,87][1080,231]" /></node></node></node>
13 <node index="1" text="" resource-id="android:id/content" class="android.widget.
       FrameLayout" package="jackpal.androidterm" content-desc="" checkable="false"
       checked="false" clickable="false" enabled="true" focusable="false" focused="
       false" scrollable="false" long-clickable="false" password="false" selected="
       false" bounds="[0,243][1080,1920]">
14   <node index="0" text="" resource-id="jackpal.androidterm:id/view_flipper" class="
         android.widget.ViewFlipper" package="jackpal.androidterm" content-desc=""
         checkable="false" checked="false" clickable="false" enabled="true" focusable=
         "false" focused="false" scrollable="false" long-clickable="false" password="
         false" selected="false" bounds="[0,243][1080,1920]">
15   <node index="0" text="" resource-id="" class="android.view.View" package="jackpal
         .androidterm" content-desc="" checkable="false" checked="false" clickable="
         false" enabled="true" focusable="true" focused="true" scrollable="false" long
         -clickable="true" password="false" selected="false" bounds="
         [0,243][1080,1920]" /></node></node></node>
16 <node index="1" text="" resource-id="android:id/statusBarBackground" class="android
       .view.View" package="jackpal.androidterm" content-desc="" checkable="false"
       checked="false" clickable="false" enabled="true" focusable="false" focused="
       false" scrollable="false" long-clickable="false" password="false" selected="
       false" bounds="[0,0][1080,75]" /></node></hierarchy>
```

Appendix D

Extracted Data Example

This example shows an example of the information extracted by the solution for both device and application.

D.1 Device

Information extracted from Vodafone Smart Ultra 6 device.

```
DEVICE:
  Serial: 77a1c109
  Folder: C:\Users\freitasm\Documents\vodafone\Crawler\VodafoneGUIcrawler\VodafoneSmartultra6
  Model: Vodafone Smart ultra 6
  API Level: 22
  Android Version: 5.1.1
  Brand: Vodafone
  Language: pt
  Region: PT
  Timezone: Europe/Lisbon
  Battery Level: 100
  NFC Port: I2C
  Crypto State: unencrypted
  Build Date: Tue Feb 16 15:42:58 CST 2016
  IMSI: 268011201001962
  Gateway: 192.168.1.254
  USB State: mtp,adb
  Rooted: false
  Mobile Country Code (MCC) + Mobile network code of operator (MNC): 26801
  SIM State: READY
  Roaming State: false
  Network Type: LTE
  Operator: vodafone P
  Wireless IP Address: 192.168.1.133/24
  3G IP Address: 0.0.0.0/0
  Screen Size: 1080x1920
APPS:
```

```
  Last App Tested: com.airbnb.android
  Initial Activity of Last App Tested: es.mrcl.app.juasapp.ConfirmActivity
  Last Activity of Last App Tested: es.mrcl.app.juasapp.ConfirmActivity
 System Apps:
  com.google.android.youtube
  com.android.providers.telephony
  com.google.android.googlequicksearchbox
  com.android.providers.calendar
  com.android.providers.media
  com.android.wifidirect.test
  com.zte.camera
  (...)
  com.android.bluetooth
  com.qualcomm.timeservice
  com.google.android.androidforwork
  com.android.providers.contacts
  com.android.captiveportallogin
  cn.com.zte.settings.patch
 3rd Party Apps:
  com.vodafone.mCare
  com.indeed.android.jobsearch
  com.innovagency.ana
  (...)
  com.tinder
  com.ebay.mobile
  com.moofwd.uporto
```

D.2 App

Information extracted from the MyVodafone app.

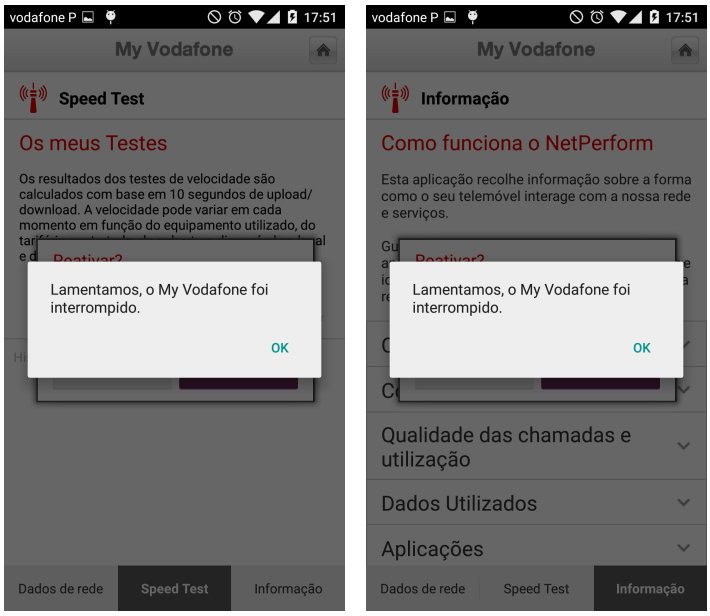
```
APP:
 Package Name: com.vodafone.mCare
 Target Sdk: 23
 SDK Version: 9
 Version Name: 2.6.2
 Version Code: 177090000
 Platform Build Version Name: 6.0-2166767
 install Location: auto
 Resource Path: /data/app/com.vodafone.mCare-1
 APK Location: /data/app/com.vodafone.mCare-1/base.apk
 First Install Time: 2016-04-20 14:31:27
 Last Update Time: 2016-04-20 14:31:27
 Label: My Vodafone
 Icon: res/drawable-mdpi-v4/icon.png
 Launch Activity: com.vodafone.mCare.Main
```

```
App Location: C:\Users\freitasm\Documents\vodafone\Crawler\VodafoneGUIcrawler\VodafoneSmartultra6\com.vodafone.mCare\com.vodafone.mCare.apk
Pemissions:
  android.permission.INTERNET
  android.permission.ACCESS_NETWORK_STATE
  android.permission.READ_PHONE_STATE
  android.permission.CALL_PHONE
  android.permission.READ_CONTACTS
  android.permission.READ_EXTERNAL_STORAGE
  android.permission.WRITE_EXTERNAL_STORAGE
  android.permission.ACCESS_GPS
  android.permission.ACCESS_ASSISTED_GPS
  android.permission.ACCESS_LOCATION
  android.permission.ACCESS_COARSE_LOCATION
  android.permission.ACCESS_FINE_LOCATION
  com.google.android.providers.gsf.permission.READ_GSERVICES
  android.permission.ACCESS_WIFI_STATE
  android.permission.RECEIVE_BOOT_COMPLETED
  android.permission.GET_TASKS
  android.permission.READ_LOGS
  android.permission.READ_CALL_LOG
  android.permission.WAKE_LOCK
  com.google.android.c2dm.permission.RECEIVE
  android.permission.VIBRATE
  com.vodafone.mCare.permission.C2D_MESSAGE
  android.permission.GET_ACCOUNTS
Supported Screens: normal,large,xlarge
Supported Densities: 160,213,240,320,480,640
Locales: de,nl,cs,ca,da,fa,ja,nb,be,af,bg,th,fi,hi,vi,sk,uk,el,pl,sl,tl,am,in,ko,ro,ar,fr,hr,sr,tr,es,it,lt,pt,hu,ru,zu,lv,sv,iw,sw,fr−CA,lo−LA,en−GB,bn−BD,et−EE,ka−GE,ky−KG,km−KH,zh−HK,si−LK,mk−MK,ur−PK,sq−AL,hy−AM,my−MM,zh−CN,pa−IN,ta−IN,te−IN,ml−IN,en−IN,kn−IN,mr−IN,gu−IN,mn−MN,ne−NP,pt−BR,gl−ES,eu−ES,is−IS,es−US,pt−PT,en−AU,zh−TW,ms−MY,az−AZ,kk−KZ,uz−UZ
```

Appendix E

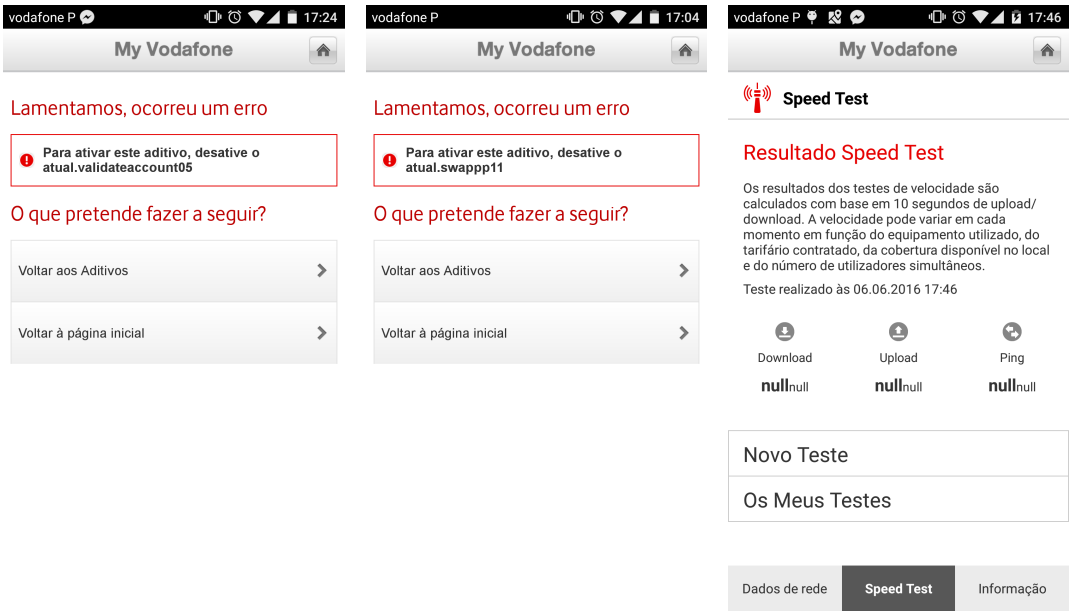
Evidences Found

E.1 Bug and Crash Evidences



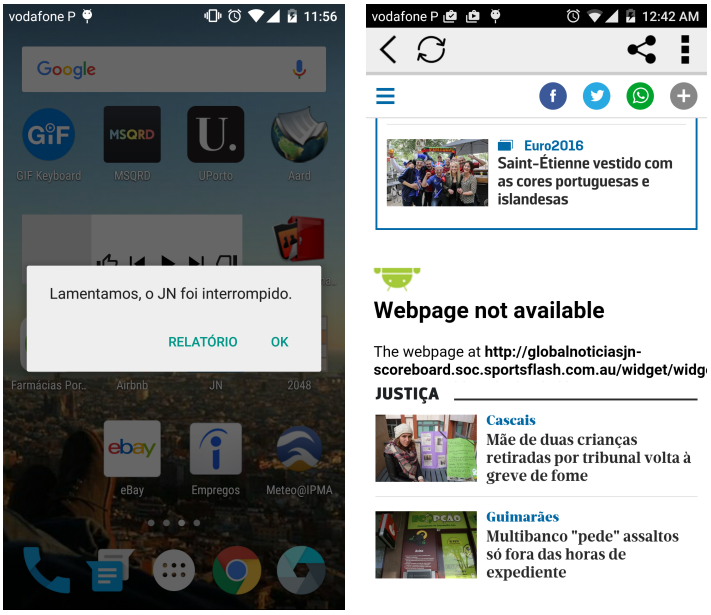
(a) Crash Speed Test. (b) Crash NetPerform.

Figure E.1: My Vodafone App - crashes found.



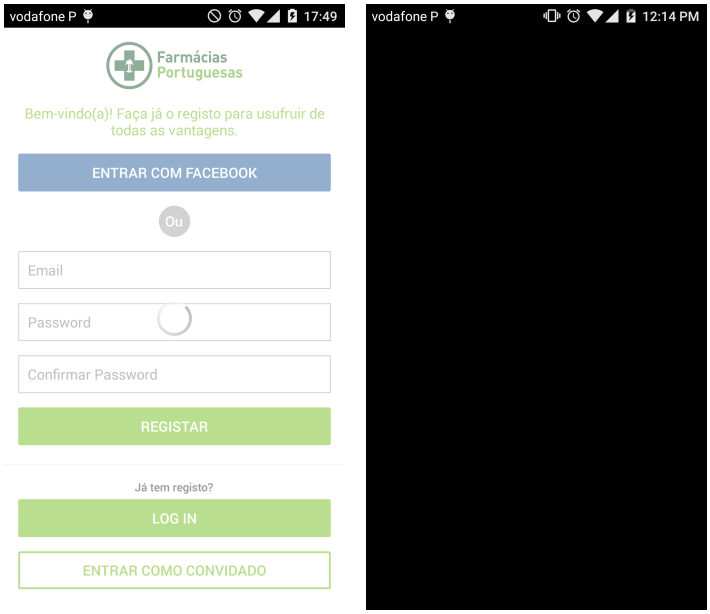
(a) Subcription of additive. (b) Subcription of additive. (c) Null Counters.

Figure E.2: My Vodafone App - bugs found.



(a) Crash App. (b) Resources Missing.

Figure E.3: JN App - crashes/bugs found.



(a) App Freeze. (b) Black Screen.

Figure E.4: Farmácias Portuguesas App - bugs found.

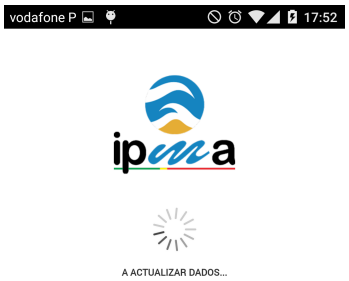


Figure E.5: IPMA@Meteo App Freeze.

Figure E.6: Tinder App Blank Screen.

References

[1] Winston W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 328 – 338. IEEE Computer Society Press, 1987. [Accessed: Feb. 05, 2016]. URL: http://dl.acm.org/citation.cfm?id=41765.41801.

[2] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001. [Accessed: Feb. 06, 2016]. URL: http://faculty.mu.edu.sa/public/uploads/1428308006.7514software%20engineering.pdf.

[3] Pat O'Sullivan and Joe Fitzpatrick. 21st century software development - an "on demand" software engineering process perspective. *IBM White Paper*, 2010. [Accessed: Feb. 06, 2016]. URL: https://www.engineersireland.ie/EngineersIreland/media/SiteMedia/groups/Divisions/computing/21stcentureysoftvev.pdf?ext=.pdf.

[4] Craig Larman. *Agile and Iterative Development: A Manager's Guide*. Pearson Education, 2003. [Accessed: Feb. 05, 2016]. URL: http://dl.acm.org/citation.cfm?id=861501.

[5] Alexander G. Mirnig, Alexander Meschtscherjakov, Daniela Wurhofer, Thomas Meneweger, and Manfred Tscheligi. A formal analysis of the iso 9241-210 definition of user experience. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '15, pages 437 – 450. ACM, 2015. [Accessed: Feb. 04, 2016]. URL: http://dl.acm.org/citation.cfm?id=2732511.

[6] International Organization for Standardization (ISO). *Ergonomics of human-system interaction: Human-centred design for interactive systems : ISO 9241-210*. Number pt. 210 in DIN EN ISO. ISO, 2010. [Accessed: Feb. 04, 2016]. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=52075.

[7] Susanne Furman, Mary Theofanos, Hannah Wald, and S. Chapman. *Human Engineering Design Criteria Standards Part 1: Project Introduction and Existing Standards DHS S&T TSD Standards Project*. National Institute of Standards and Technology (U.S.). Information Technology Laboratory, Material Measurement Laboratory, 2013. [Accessed: Feb. 04, 2016]. URL: https://books.google.pt/books?id=3ntOnwEACAAJ.

[8] Matt Wynne and Aslak Hellesøy. *The cucumber book : behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2012. [Accessed: Feb. 10, 2016]. URL: https://www.citeulike.org/user/pater/article/11814511.

[9] Niklaus Wirth. A brief history of software engineering. *IEEE Annals of the History of Computing*, 30(3):32 – 39, July 2008. [Accessed: Feb. 06, 2016]. URL: http://dl.acm.org/citation.cfm?id=2341280.

[10] Rational Software. Rational unified process : Best practices for software development teams. *Rational Software White Paper*, 1998. [Accessed: Feb. 05, 2016]. URL: https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf.

[11] Ritu Agarwal, Jayesh Prasad, Mohan Tanniru, and John Lynch. Risks of rapid application development. *Commun. ACM*, 43(11es), November 2000. [Accessed: Feb. 06, 2016]. URL: http://doi.acm.org/10.1145/352515.352516.

[12] James Martin. *Rapid Application Development*. Macmillan Publishing Co., Inc., 1991. [Accessed: Feb. 06, 2016]. URL: http://dl.acm.org/citation.cfm?id=103275.

[13] Janet B. Butler. *Rapid Application Development in Action', Managing System Development.* System Development, Applied Computer Research, vol. 14, no. 5 edition, 1994. [Accessed: Feb. 05, 2016]. URL: https://scholar.google.com/scholar?hl=en&as_sdt=0,5&cluster=4834007071832675334.

[14] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61 – 72, May 1988. [Accessed: Feb. 07, 2016]. URL: http://dx.doi.org/10.1109/2.59.

[15] Victor R. Basili and Albert J. Turner. Iterative enhancement: A practical technique for software development. *Software Engineering, IEEE Transactions on*, SE-1(4):390 – 396, December 1975. [Accessed: Feb. 07, 2016]. URL: http://ieeexplore.ieee.org/xpl/abstractAuthors.jsp?tp=&arnumber=6312870.

[16] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47 – 56, June 2003. [Accessed: Feb. 07, 2016]. URL: http://dx.doi.org/10.1109/MC.2003.1204375.

[17] Nick Jenkins. A software testing primer, 2008. [Accessed: Feb. 07, 2016]. URL: http://www.nickjenkins.net/prose/testingPrimer.pdf.

[18] Bruce Sterling. *Hacker Crackdown: Law and Disorder on the Electronic Frontier*. Bantam Books, Inc., 1993. [Accessed: Feb. 08, 2016]. URL: http://dl.acm.org/citation.cfm?id=529080.

[19] Cem Kaner, Jack L. Falk, and Hung Quoc Nguyen. *Testing Computer Software, Second Edition*. John Wiley & Sons, Inc., 2nd edition, 1999. [Accessed: Feb. 12, 2016]. URL: http://dl.acm.org/citation.cfm?id=553594.

[20] Pierre Bourque and Richard E. Fairley, editors. *SWEBOK: Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, version 3.0 edition, 2014. [Accessed: Feb. 07, 2016]. URL: http://www.swebok.org/.

[21] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall Press, 5th edition, 2010. [Accessed: Feb. 09, 2016]. URL: http://dl.acm.org/citation.cfm?id=1824151.

[22] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. [Accessed: Feb. 05, 2016]. URL: http://dl.acm.org/citation.cfm?id=983238.

[23] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007. [Accessed: Feb. 04, 2016]. URL: http://dl.acm.org/citation.cfm?id=1324770.

[24] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 1 – 8. ACM, 2007. [Accessed: Feb. 05, 2016]. URL: http://doi.acm.org/10.1145/1251535.1251536.

[25] Juho Lepistö. *Embedded Software Testing Methods*. PhD thesis, Helsinki Metropolia University of Applied Sciences, 2012. [Accessed: Feb. 06, 2016]. URL: https://www.theseus.fi/bitstream/handle/10024/46873/Lepisto_Juho.pdf?sequence=1.

[26] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27 – 38. ACM, 2012. [Accessed: Feb. 06, 2016]. URL: http://dl.acm.org/citation.cfm?id=2259056.

[27] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011. [Accessed: Feb. 06, 2016]. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.221.5311.

[28] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. In *ACM Sigplan Notices*, volume 43, N°10, pages 313 – 328. ACM, September 2008. [Accessed: Feb. 07, 2016]. URL: http://dl.acm.org/citation.cfm?id=1449790.

[29] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, August 2011. [Accessed: Feb. 08, 2016]. URL: http://dl.acm.org/citation.cfm?id=2028067.2028088.

[30] Damien Octeau, William Enck, and Patrick McDaniel. The ded decompiler. Technical report, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, September 2010. [Accessed: Feb. 08, 2016]. URL: http://siis.cse.psu.edu/ded/papers/NAS-TR-0140-2010.pdf.

[31] Paul Pocatilu. Android applications security. *Informatica Economica*, 15(3):163, September 2011. [Accessed: Feb. 10, 2016]. URL: http://revistaie.ase.ro/content/59/14%20-%20Pocatilu.pdf.

[32] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE '12, pages 6:1 – 6:11. ACM, 2012. [Accessed: Feb. 08, 2016]. URL: http://dl.acm.org/citation.cfm?id=2393600.

[33] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290 – 299. ACM, 2003. [Accessed: Feb. 05, 2016]. URL: http://dl.acm.org/citation.cfm?id=948149.

[34] Tuan Anh Nguyen, Christoph Csallner, and Nikolai Tillmann. Gropg: A graphical on-phone debugger. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1189 – 1192. IEEE Press, 2013. [Accessed: Mar. 11, 2016]. URL: http://dl.acm.org/citation.cfm?id=2486788.2486958.

[35] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209 – 220. ACM, 2013. [Accessed: Feb. 07, 2016]. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.294.9536.

[36] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67 – 77, May 2006. [Accessed: Feb. 07, 2016]. URL: http://link.springer.com/article/10.1007%2Fs11416-006-0012-2.

[37] Angela Orebaugh, Gilbert Ramirez, and Jay Beale. *Wireshark & Ethereal network protocol analyzer toolkit*. Syngress, 2006. [Accessed: Feb. 11, 2016]. URL: http://dl.acm.org/citation.cfm?id=1202316.

[38] Justin Clarke. *SQL injection attacks and defense*. Elsevier, 2009. [Accessed: Feb. 07, 2016]. URL: http://www.sciencedirect.com/science/book/9781597494243.

[39] Security Standards Council. Passing pci dss section 6 compliance. *HP*, 2010. [Accessed: Feb. 09, 2016]. URL: https://www.fortify.com/downloads2/user/Passing-PCI-Compliance-Section-6-6.pdf.

[40] Robin Jeffries, James R. Miller, Cathleen Wharton, and Kathy Uyeda. User interface evaluation in the real world: A comparison of four techniques. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 119 – 124. ACM, 1991. [Accessed: Mar. 05, 2016]. URL: http://doi.acm.org/10.1145/108844.108862.

[41] Rudolf Ramler and Klaus Wolfmaier. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 85 – 91. ACM, 2006. [Accessed: Feb. 11, 2016]. URL: http://dl.acm.org/citation.cfm?id=1138946.

[42] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. [Accessed: Feb. 11, 2016]. URL: http://dl.acm.org/citation.cfm?id=310674.

[43] Mark Blackburn and Aaron Nauman. Strategies for web and gui testing. Technical report, Software Productivity Consortium, 2011. [Accessed: Mar. 04, 2016]. URL: http://www.knowledgebytes.net/downloads/Strategies_Web_and_GUI_testing_spc-2004014-D-P.pdf.

[44] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 72 – 81. IEEE Press, 2013. [Accessed: Mar. 04, 2016]. URL: http://dl.acm.org/citation.cfm?id=2486788.2486799.

[45] Domenico Amalfitano, Anna R. 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. ACM, 2012. [Accessed: Feb. 10, 2016]. URL: http://doi.acm.org/10.1145/2351676.2351717.

[46] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. *SIGPLAN Not.*, 48(10):641 – 660, October 2013. [Accessed: Feb. 11, 2016]. URL: http://doi.acm.org/10.1145/2544173.2509549.

[47] Domenico Amalfitano, Anna R. Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Gennaro Imparato. A toolset for gui testing of android applications. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 650 – 653, September 2012. [Accessed: Feb. 11, 2016]. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6405345.

[48] 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. ACM, 2011. [Accessed: Feb. 11, 2016]. URL: http://doi.acm.org/10.1145/1982595.1982612.

[49] Linus Esbjörnsson. Android gui testing : A comparative study of open source android gui testing frameworks, 2015. [Accessed: Feb. 10, 2016]. URL: http://www.diva-portal.se/smash/get/diva2:820898/FULLTEXT01.pdf.

[50] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224 – 234. ACM, 2013. [Accessed: Feb. 10, 2016]. URL: http://doi.acm.org/10.1145/2491411.2491450.

[51] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. *SIGPLAN Not.*, 48(10):623 – 640, October 2013. [Accessed: Feb. 11, 2016]. URL: http://doi.acm.org/10.1145/2544173.2509552.

[52] Jinseong Jeon and Jeffrey S. Foster. Troyd: Integration testing for android. Technical Report CS-TR-5013, Department of Computer Science, University of Maryland, College Park, August 2012. [Accessed: Feb. 10, 2016]. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.306.3636.

[53] Paul E. Black, Michael Kass, and Michael Koo. Source code security analysis tool functional specification version 1.0. *National Institute of Standards and Technology*, May 2007. [Accessed: May 13, 2016]. URL: http://samate.nist.gov/docs/source_code_security_analysis_spec_SP500-268.pdf.

[54] International Electrotechnical Commission and International Organization for Standardization. *BS ISO/IEC 27005:2008*. British Standard. BSI Group, June 2008. [Accessed: May 13, 2016]. URL: https://books.google.pt/books?id=8I7ROQAACAAJ.