ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (30^{th}cycle)

Evolution and Fragility of Mobile Automated Test Suites

By

Riccardo Coppola
******

Supervisor(s):
Prof. Maurizio Morisio

Doctoral Examination Committee:
Prof. Robert Feldt, Referee, Chalmers Tekniska Högskola AB
Prof. Paolo Tonella, Referee, Università della Svizzera italiana
Prof. Andrea Bottino, Politecnico di Torino
Prof. Fulvio Corno, Politecnico di Torino
Prof. Cristina Gena, University of Turin

Politecnico di Torino
2019

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Riccardo Coppola
2019

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

Acknowledgements

The PhD years are indeed quite a roller-coaster voyage. Nevertheless, I feel really enriched by the experience gathered in this time window and I want to express my gratitude to all the people that helped me to reach until its conclusion.

First and foremost, my thanks go to my tutor, Prof. Maurizio Morisio, for the trust and responsibility he gave me selecting me as a PhD candidate, and basically for making all of this possible; to Prof. Marco Torchiano, for being an important guidance and inspiration for my research interests; to Luca Ardito for the countless brainstorming sessions and precious advice. I would also like to mention (in strict alphabetical order) Alysson, Amir, Diego, Erion, Francesco, Iacopo, Rifat and all the other people who contributed to make Lab 1 the friendly and warm environment I have always been very happy to work in.

My thanks also go to the nameless reviewers that helped me improve my works, and to the many colleagues known in various locations during these years of workshops and conferences, with whom I had useful chats about (but not limited to) my research fields. In particular, I would like to mention Dr. Emil Alégroth, for the collaboration from which some researches detailed in this manuscript sparked.

Lastly but not less importantly, in a rapid glance outside the world of academia, I would like to wholeheartedly thank my family, for its long-distance yet continuous and fundamental support over all these years; the loyal fellows I have known back in the days of high school and bachelor's in my hometown, who still ring my phone albeit being spread at varied latitudes; the assorted bunch of folks populating with me the disastrously gloomy landscape of Turin, with whom I fuel the most heartfelt discussions about topics of very questionable importance. And of course Joy, for being an inexhaustible source of the indispensable feeling that her name itself suggests.

Abstract

Context. Android applications have reached a level of diffusion and complexity that was once exclusive of desktop computing, hence demanding a thorough validation and verification phase to ensure that they meet their requirements. Such need especially applies to their GUIs (i.e., Graphical User Interfaces) through which most of the interaction with the final user is performed. However, although many approaches, techniques and tools exist to test Android apps, evidence from the literature suggests that GUI testing is generally limited among practitioners.

Goal. The main goals of the studies reported in this thesis have been: (i) assess the usability of existing GUI testing techniques applicable to Android apps; (ii) quantify the adoption of existing tools by developers from the open-source environment; (iii) investigate the amount of effort needed in maintaining test suites during their co-evolution with the AUT (Application Under Test) and the principal reasons behind the interventions; (iv) identify the main issues faced by Android testers, and provide guidelines and tools to aid reducing their impact on testing design and maintenance.

Method. These goals were pursued by performing five different studies, after a preliminary exploratory study on a popular open-source app. An experiment with Graduate students and a survey with developers from the industry were conducted in order to gather qualitative information about the usability of testing tools and techniques, and to understand what are the needs and difficulties felt by different categories of users of those tools. Starting from a mining of all tested Android applications published on GitHub, a set of metrics for quantifying testware evolution and maintenance and a taxonomy of maintenance reasons were defined and validated. Finally, a tool that leverages the benefits of two GUI testing techniques (Layout-based and Visual) was implemented and validated.

Results. The studies showed that the diffusion of testing tools is limited on the set of projects mined from GitHub, the largest context for this kind of quantitative experiments up to date. It is deduced that GUI testing frameworks are characterized by a steep learning curve and are often considered imprecise by developers. Additionally, test suites suffer from many types of fragility, requiring a relevant maintenance effort (estimated as 5% of total development effort). To mitigate the fragility issues, the TOGGLE tool has been developed. The implemented translational approach proved to be able to relieve the testers from part of the effort in maintaining test suites, and mitigate the drawbacks of the two approaches that were considered in the studies.

Conclusion. In summary, Automated GUI testing frameworks for Android are still far from being widely adopted by either open-source or industry developers, and a relevant reason for this missing diffusion is their fragility to the evolution of the tested AUT. The results of the study gathered in this thesis suggest that there is still room for improvement of existing testing techniques to mitigate their current drawbacks. The proposed translational approach can serve as a first effort in reducing the complexities of co-evolving mobile apps along with their testware.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

At its ninth release, the Android operating system has affirmed as the choice of nearly 90% of mobile users as of Q2 2018[1], and the most popular among all OSs regardless of the platform, as of Jan 2019[2]. Today's mobile apps have reached a complexity that is comparable to that of desktop applications, and that should encourage a thorough Verification and Validation phase, with a significant focus posed on testing their GUIs (i.e., Graphical User Interfaces), since most of the interactions with the final users are carried through them.

At the same time, in industrial practice, there is a continuous push for faster delivery of software, with the agile trend and continuous integration seen in any domain of software development [10]. To facilitate these practices, automated validation and verification is nowadays a necessity, and companies apply automation to many levels of testing.

However, regardless of the centrality of the GUIs and the push for continuous integration, there is evidence in the literature about a lack of adoption of automated GUI testing techniques for mobile apps. While automated lower-level (i.e., unit or integration) testing is a widespread practice, GUI testing is performed most of the times manually, and at high expense [21]. Even though there is evidence that relevant players of the industry perform structured and automated testing of the GUIs of their applications [4], several studies in academic literature showed that open-source mobile developers rarely adopt automated testing techniques in their projects [67].

[1] https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[2] http://gs.statcounter.com/os-market-share

This lack of adoption may be justified by several characteristics that are proper of mobile apps, like the sensitivity to many context events, the large amount of devices with which the apps must provide compatibility (i.e., *device fragmentation*), the fast evolution of projects, the existence of many different frameworks for app development that make the same automated tools inapplicable to all projects [83], and finally the issues of fragility of test cases, that are easily breakable even by small changes in the AUT (i.e., Application Under Test) [43]. At the same time, intrinsic drawbacks are exhibited by GUI testing frameworks themselves, which may contribute to resilience in adopting them [57].

Albeit several analyses of the adoption of automated GUI testing frameworks were available in the literature at the beginning of the work documented in this thesis, no prior effort in quantifying the characteristics of testware (in terms of relevance with respect to production code, amount of maintenance needed during the evolution of the projects, and fragilities of test cases) was applied to large-scale software repositories. The issue of fragility of test cases, already discussed for web-based applications [62], was still not described for mobile apps, and no discussion of the most common causes of test breakage and maintenance was available. Finally, while the research for new techniques of automated GUI testing had proved the effectiveness and benefits of different approaches (e.g, layout-based or visual testing tools), and existing tools have proved the feasibility of a combined translational approach for web-based tests [63], no combined use of the techniques has been studied in detail for mobile apps.

Starting from a case study on a popular Android app, and with exploratory studies to assess the issues faced by developers and master's students when approaching GUI testing frameworks, the work detailed in the remainder of this thesis has been designed to fill the research gaps discussed above.

1.1 Research Goals and Questions

The objective of this thesis can be explained by defining four main *goals*:

- **Goal 1 - Perception and Usability:** Investigate the ease of use of existing Android testing tools, and the perception that potential users have of them.

Study	Research Question	Goals Addressed
S1: Survey with mobile developers from the industry	RQ 1 - What is the perception of GUI testing for Android apps among practitioners from the industry?	G1, G3, G4
S2: Controlled experiment with Graduate Students	RQ 2 - How usable are GUI testing tools and what is the productivity of graduate students using them?	G1, G4
S3: Measures of Diffusion and Evolution of Testware in OS projects	RQ 3 - What is the adoption and typical evolution of test suites with automated GUI testing frameworks among Android open source projects?	G2, G3
S4: Taxonomy of fragility causes	RQ 4 - Why and with which frequency fragilities occur in tested Android projects?	G3
S5: Layout-based vs Generated visual test cases: An experiment with TOGGLE	RQ 5 - What is the dependability and performance of visual test cases generated by translation?	G3, G4

Table 1.1 Studies detailed in the thesis

- **Goal 2 - Adoption and Size:** Quantify the adoption of such tools by industry and OS developers, and investigate the size and relevance of testware in tested Android apps.

- **Goal 3 - Evolution and Fragility:** Quantify the effort needed in maintaining testware during the evolution of an Android project, and identify the main causes of test fragility.

- **Goal 4 - General Android testing issues:** Identify common challenges in performing Android testing, and find possible guidelines to mitigate such challenges.

To purse the four main goals, five different studies – plus a preliminary exploratory case study – were performed. Each study was tailored to answer a high-level Research Question, which pertained one or more of the Goals.

Table 1.1 summarizes the studies described in this dissertation, and emphasizes the link between the Research Question answered in each study and the high-level goals of the thesis.

The first study wanted to capture the perception of automated testing among developers from the Italian industry, and to understand what are the factors pushing them towards its adoption or, on the other hand, discouraging it. The study was performed by means of a qualitative survey subministrated to the identified respondents, about the perceived advantages and disadvantages of automated testing and its maintenance, and about the perceived peculiarities of Android and mobile testing. The second study had the aim of understanding how usable are GUI testing tools of different nature that can be applied to mobile applications; to this purpose, a controlled experiment was conducted to measure the productivity of graduate students approaching the different techniques. Again, a qualitative survey was subministrated to the students, to capture their perception about the techniques and mobile application testing.

To provide a quantitative counterpart to the analysis of the interviewed developers' responses, a set of metrics has been defined, to characterize (i) the diffusion of available testing tools among open-source Android projects, (ii) the average size of test suites developed with a given testing tool, and (iii) the evolution and the amount of maintenance needed by test code during the normal evolution of the tested software project. The metrics represent a novel contribution to the general field of software evolution and maintenance, being at a finer level of detail than other change metrics already available in the literature [96]. The study involved the mining of Android open-source repositories from the GitHub platform, and allowed the collection of a corpus of diff files that could be analyzed to understand the motivations behind maintenance operated on test cases. The fourth study presented in this thesis was an application of the Grounded Theory technique on the set of collected diff files, to infer a taxonomy of reasons for modifications in test code. Taxonomy of change reasons have been already derived for web-based applications [50]; however, inferring a new taxonomy specific to Android test code changes is justified by the different nature of Android applications and of the way their GUIs are described.

While all other studies followed mainly an empirical approach, the last study detailed in this dissertation followed an engineering approach to provide a solution to some of the issues related to Android testing, and included the full design and development of a tool for aiding Android GUI testing. Specifically, the results of the interviews to the developers suggested that the frequent maintenance needed and the high device fragmentation (i.e., the necessity of porting the same test cases to many devices) are among the biggest issues faced by testers of mobile applications. The frequent maintenance was also quantitatively confirmed by the analysis of the history of Android repositories hosted on GitHub, along with a relatively scarce diffusion of automated testing tools due to those inherently high costs for adoption and maintenance. Additionally, the taxonomy of modification causes highlighted several trivial changes that happen quite frequently during the normal evolution of an Android app, and that can have severe impacts on the maintenance needed by either scripted or visual test suites. The final objective of the thesis was hence to design a tool to fulfill the following purposes:

- Leverage the benefits of two different technologies of testing tools to counter their drawbacks;
- Partially automate the creation of tests by allowing the reuse of existing ones written with other tools;
- Reduce the impact of device fragmentation on the total testing effort;
- Reduce the needed maintenance effort introduced by failing test locators.

The aims above led to the definition and design of the proof of concept of an Android-specific tool, implementing a translational approach from layout-based to visual test cases, and vice-versa. The tool partially performs a knowledge transfer of the PESTO tool – defined for the translation of layout-based web test scripts to visual tests [63] – to the Android domain. The applicability of the approach to real test cases on popular Android applications and the provided benefits were finally measured in the experiment detailed in Study 5.

1.2 Dissertation Structure

The remainder of the present thesis can be summarized as follows:

- **Chapter 2** provides positioning for the work described in this thesis. Definitions of System testing, GUI testing and Mobile testing are provided, along with basic concepts of Android development and its issues.

- **Chapter 3** provides a description of the studies adopted during this PhD, and describes the preliminary parts of the study that were common to all the following sections of the thesis.

- **Chapter 4 to 9** describe the individual studies that were performed. For each study, its design is described in detail, along with the decomposition of the high-level Research Question that the study answers. Results and threats to the validity of the individual studies are also discussed.

- **Chapter 10** summarizes the outcomes of all the studies performed, providing comparisons with the current state of the art and related literature.

- **Chapter 11** concludes the thesis, with a roadmap of future work and a summary of contributions that can be leveraged immediately by practitioners and researchers working in the field of Android development and testing.

Chapter 2

Background

The work presented in this thesis is positioned in the areas of software development processes, software Validation and Verification, Automated Software Testing, and Mobile Application Development.

This section provides background information about the basic concepts of End-2-End testing, especially when the software is tested interacting with it through its user interface. An overview of the different *generations* and techniques for performing GUI testing is also provided.

Then, the characteristics of Android apps and of the Android application frameworks are detailed, along with the main peculiarities of Android testing and a review of the techniques and tools for testing Android apps through their GUIs that are available in the literature.

Finally, the concept of fragility is introduced, along with a discussion of prior analysis performed in the literature about testing fragility in other domains, leading to the definition that has been adopted for all the studies that constitute the present thesis.

2.1 System and GUI Testing

System testing is the testing step that is performed between Integration testing and Acceptance testing (see figure 2.1). Its purpose is to execute test cases to verify the whole software system. Typically, System Testing is a form of Black Box testing.

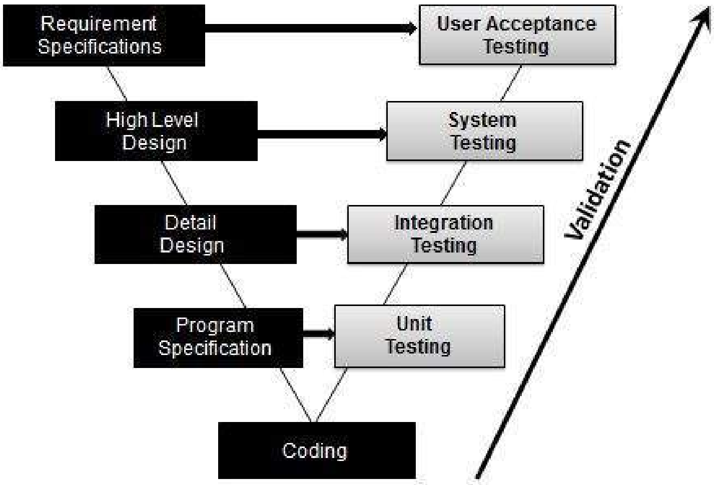


Fig. 2.1 V-Model for Software Development Process

The main activities performed by System testing are to test each input provided to the application in order to verify the respective output, and to test the user scenarios with the software product. When other peripherals and other software systems are tested along with the Software Under Test (SUT), the procedure is typically referred to as *End-To-End Testing*.

There are many different forms of System Testing, e.g. *Usability testing* (testing the ease of use of the complete software as perceived by its final users), *Load testing* (checking how a software behaves under real-life loads), *Functional testing* (verifying that each function of the software is compliant with the specified requirements).

GUI Testing is a form of system testing that can be applied to software which is provided with a Graphical User Interface. A general definition of GUI testing is provided by Banerjee et al. [19] as:

> *GUI testing is System Testing of software that has a graphical user interface (GUI) front-end. During GUI testing, test cases – modeled as a sequence of input events – are developed and executed on the software by exercising the GUI widgets.*

The first and most immediate option for performing GUI Testing is to manually execute test cases on the SUT. The information shown by the GUI, the flow between the different screens of the SUT and the responses to the provided inputs are hence verified by human testers for conformity with the requirements. As discussed by Kropp et al. [59], the manual execution of test cases is rarely exhaustive and error-prone, and the performed tests are not easily reproducible. Furthermore, it requires relevant effort from testers, especially with the capabilities offered by modern user interfaces.

On the other hand, automated GUI testing techniques may define sets of scripts to exercise exhaustively – in a quick and repeatable way - all the main functionalities of a GUI. In addition to that, automated test scripts can be also used to test the presence of regressions, in the transition between two consecutive releases of an application.

As reported by Tramontana et al. [97], even the automation of simple tasks in the field of GUI testing can make feasible the execution of complex testing processes, that are instead too expensive for manual testing approaches.

2.2 Classification of Automated GUI testing techniques

Several approaches have been explored in the literature for performing GUI testing of applications of any domain. A systematic mapping of available GUI testing techniques has been performed by Banerjee et al. [13]. Automated GUI testing tools can be classified according to various aspects: the type of oracles that they use, the use of models (and the specific model adopted) for the description of the user interfaces, the way the inputs for the GUI are generated, the language that is used to write test scripts for re-execution of the same test cases.

For being defined as *automated*, GUI testing tools do not need to automate the entire workflow of testing, but at least some of its phases, e.g. the execution of test scripts. Regarding the generation of test cases, several techniques still rely on the manual creation of test scripts by the tester/developer in a specific syntax.

Capture & Replay testing tools, instead, rely on recording the operations performed on the GUI from a tester, that are translated into a test script by an engine and are then replicable on the AUT, to mimic human usage.

Model-based testing (MBT) techniques are black-box approaches that rely on models of a system under test and/or its environment to derive test cases [98]. Test cases are generated based on an abstraction of the SUT, according to a specific *test selection criteria* (e.g., coverage of the structural model or coverage of the requirements of the SUT). Most typically, the models used by these approaches are finite-state machines, state chars or UML diagrams. The model can either be generated by the tester/developer, provided as part of the requirements, or automatically reverse-engineered from the AUT.

Random (also called fuzzy) testing techniques rely on aleatory sequences of inputs that are fed to the application, in order to trigger potential defects and crashes. Models of the GUI can be used by random test tools to distribute the inputs in structured ways that can resemble typical human interactions with the SUT.

Automated GUI testing techniques may adopt different testing oracles (i.e., mechanisms to understand whether a test case has passed or failed). State references are one of the most common forms of oracles: in this case, states of the GUI extracted during a first execution of the AUT – known to be correct – are used to verify further executions of test cases. In Crash Testing no explicit oracles are used, and instead a test case is marked as failing if the AUT crashes during its execution, otherwise is considered passing. Formal verification methods use models or specifications of the AUT, to verify the correctness of the output of a test case.

GUI testing techniques are a relevant aid to verify the dependability of applications whose features are strictly tied with their graphical appearance, and that are based on constant interaction with the final user. Many efforts, for instance, have been devoted to applying the concepts of model-based testing for web applications, where models are used to describe the transitions between different screens of the SUT and their content [15].

2.2.1 Evolution of GUI testing tools

Automated GUI testing tools – according to a definition formulated by Alegroth et al. and adopted for the remainder of this thesis – can also be classified in different *generations* [8], according to the level of abstraction they use for interacting with the GUI when defining sequences of commands or executing them:

- *First Generation* (or Coordinate-based) testing tools uses exact coordinates on the screens of the AUT for identifying the places where interactions have to be performed. Coordinates are recorded during manual interaction with the AUT.

Coordinate-based testing tools have no knowledge of the components of the screens of the application.

- *Second Generation* (or Layout-based) testing tools are based on a model of the graphical user interface that is decomposed in layouts and hierarchies of components. Properties and values are associated with each component of the GUI, thus allowing to identify them. For instance, IDs in the definition of the screen layout hierarchies can be leveraged to unambiguously identify elements of the user interface on which interactions have to be performed.
- *Third Generation* (or Visual) testing tools use image recognition in order to find the elements of the GUI on which to perform interactions, and to provide assertions about the correctness of the SUT after the execution of a given sequence of interactions. Exact screen captures of the components are hence used at each step of the test cases.

Coordinate-based testing tools, nowadays, are rarely adopted because of their scarce adaptability to even minor changes in the GUIs and hence their lack of robustness. Layout-based testing tools and Visual testing tools exhibit different benefits and drawbacks to testers. Visual testing is more appropriate to test the actual appearance of the application, better recreating the usage of a final user; on the other hand, Layout-based testing defines every interaction based on the properties of the GUI components and not on their appearance, hence it is more appropriate to verify a proper composition of the screens of the application, and a proper functioning of the screen hierarchy.

Studies in the literature have proved the advantageous applicability of Visual GUI testing tools to industrial contexts [7][3] and the benefits of approaches combining Layout-based and Visual techniques [8]. However, the adoption of those tools is hampered by lesser robustness and performance if compared to Layout-based tools.

2.3 The Android application framework

Android is an open-source Operating System paired with an application development platform that is based on the Linux Kernel. The full Android Software Stack[1] is

[1] https://developer.android.com/guide/platform/

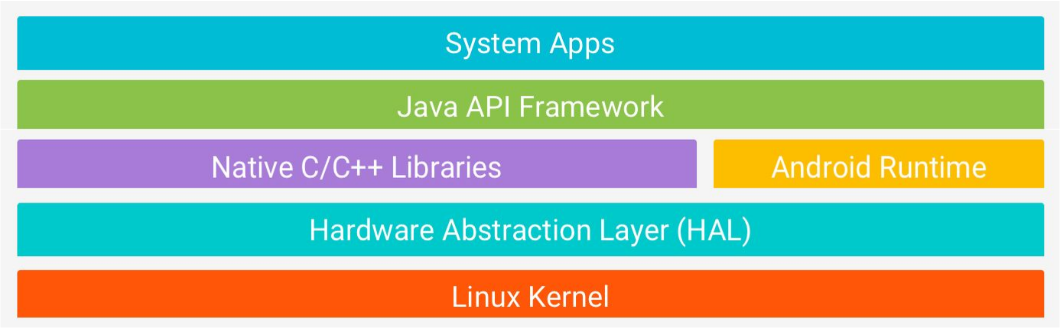


Fig. 2.2 Android Software Stack

shown in figure 2.2. The Linux Kernel is leveraged by higher-level elements of the Android platform for functionalities like threading, memory management and security features.

Upon the Linux Kernel stands the Hardware Abstraction Layer, a set of standard interfaces that is used to abstract the hardware capabilities of the device to the upper layers of the platform. To use specific functions offered by the operating system, an application will have to require the related permission on a specific file that is located in the main directory of any Android project, namely the Manifest XML file.

An instance of the Android Runtime (ART) is launched for every application, to which it is also associated a process of its own. The Android Runtime is paired with a set of Native C/C++ Libraries, needed by many components and services of the Android operating system.

The API Framework is the set of classes that is available to Android developers for building their apps. The View System provides developers the possibility of building the GUIs of their applications, through which all the interactions from the users are gathered and most functionalities are exposed. Placed on top of the Android stack, System Apps constitute the set of core default apps with which every release of the Android framework is equipped.

2.3.1 Android Apps

A general definition for a *Mobile App* has been provided by Muccini et al. as *”an application running on mobile devices and or/taking in input contextual information”*[83]. According to this definition, a Mobile Application adds to its *mobile* nature (i.e., it can run on a movable electronic device) also a *context-aware* nature, in the sense that the application is constantly adapting and reacting to the computing environment in which it is run.

According to the way they are programmed and the way they leverage the components offered by the specific frameworks they are developed for, Mobile Apps can be classified into three different categories: Native, Web-based and Hybrid apps.

Native Apps are written in a specific programming language, for a specific device platform. Java (recently paired with Kotlin) is used to write Android Native Apps.

Web-based Apps are applications that are loaded in web browsers and that provide the user with functionalities and interactions that are specifically intended to be used by a mobile device.

Hybrid Apps combine the principles of Native and Web-based apps, leveraging components that are specific to a specific platform to load, at run-time, content from the internet. On Android, the dynamic loading of the content of Hybrid apps is performed with the usage of WebViews.

The Android development platform provides four basic components with which Native apps can be built. Each component has a specific life cycle, which is driven by the operating system with the invocation of a set of methods (e.g., the *onCreate* function which is the first one invoked by a new instance of a component). Components belong to the following classes:

- *Activities* are in charge of building the user interface. Typically, each activity is dedicated to a screen or a use case of the application. Activities handle all responses that are triggered by user inputs.
- *Services* manage long-running background operations carried by the app, which do not need any interaction by the user (e.g., management of network connections).
- *Content Providers* manage the data stored by the application, and the sharing of information with other applications of the system.
- *Broadcast Receivers* respond to events that are sent by the Android system and manage the way the app must respond to them.

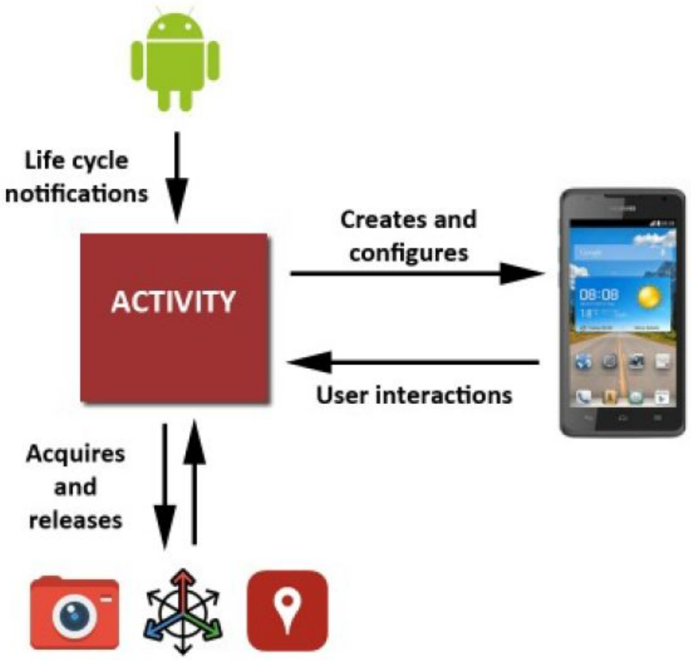


Fig. 2.3 Relationship between activities and GUI in the Android OS.

Activities (see figure 2.3) are hence the main components of any Android app. Each activity defines and builds a user interface, composed by Views arranged according to a particular layout. A layout is defined either programmatically or statically inside an XML layout file, which is then inflated in the first operation performed when transitioning into an Activity. In addition to the relative disposition of the Views inside the device screen, layouts attach properties to the elements of the user interface, e.g. unique *ids* that can then be used by the application to retrieve the elements of the GUI on which to perform operations. Callbacks can also be attached to the elements of the layouts, in order to trigger specific behaviours in response to interactions performed by the user.

From Android API 12 Fragments have been introduced, in order to manage more easily interfaces that must adapt in complex ways to different device screen size, orientation, density and format.

The strict coupling between activities and screens of the user interface of a Native app leads to an association of GUI testing for Android application of the practice of testing the Activities of an app, and their life cycle.

2.4 Mobile and Android App testing

Mobile Testing, as done by Gao et al. [41], can be defined as *"testing activities for native and Web applications on mobile devices, using well-defined software test methods and tools, to ensure quality in functions, behaviours, performance and quality of service"*.

There are different peculiarities of Mobile applications when it comes to testing them, and scopes that are specific to the mobile scenarios. For instance, Anureet et al. [16] list compatibility testing, performance testing, and security testing as primary needs for Mobile applications. Several sources identify GUI testing as a prominent testing need for all mobile applications, since GUI malfunctions for a Mobile app can seriously hamper the experience provided to the user.

Several of the described approaches for automated GUI testing have been adapted to the domain of Mobile applications in general, and specifically to Android apps. Many studies in the literature tackle the challenge of automating the whole testing procedure for Android apps, or parts of it (e.g., generation of models or execution of test cases on multiple different devices).

2.4.1 Peculiarities of Mobile testing tools

In a systematic mapping study by Tramontana et al. [97] it is underlined that 122 of the 131 examined papers about mobile testing are about system testing, with a relevant portion leveraging GUI-based approaches. The possible dimensions for the characterization of automated testing tools proposed by the authors are shown in the mind map in figure 2.4.

According to the mapping, studies in the literature can be classified according to the procedures they automate: test case generation, test case execution, definition and evaluation of oracles; the techniques detailed in about 20% of the examined papers provided automation for the full testing process. About 10% of the papers, on the other hand, illustrated techniques that provided automation for a single activity of the testing process.

Testing techniques can make use of different artifacts to create testing tools. Both white box testing tools, leveraging the source code of the application, and black box testing tools, based on executables or bytecode of the apps, are available in the

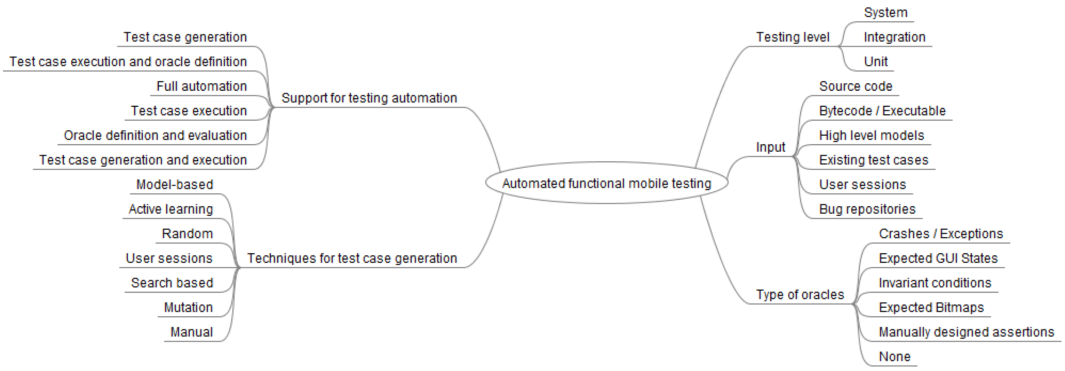


Fig. 2.4 Concepts about Automated functional testing tools for mobile applications (Tramontana et al. [97])

literature. Other techniques leverage models of the user interface (i.e., model-based testing techniques), collections of test cases, previously recorded user sessions or repositories of common bugs to derive test cases.

As test oracles, studies in the literature used principally the detection of crashes or exceptions as implicit oracles for test cases. Models of the behaviour of the application, that can be obtained also by reverse engineering of the application, can be used to identify the expected state for a given execution of the AUT. Visual oracles (i.e., bitmap captures of the expected screens at the end of the test case execution) are also considered.

Concerning the generation of the test cases, many testing tools existing in the literature adopt a dedicated syntax to create repeatable test scripts that the testing tool itself can launch. Some tools allow exporting the generated test cases in order to make them launchable with other testing engines (e.g., JUnit).

2.4.2 Categories of Mobile testing tools and services

An overview of the testing tools that are available for Automated Mobile App Testing, not only limited to GUI testing, has been provided by Linares-Vasquez et al. [65][68], who subdivided the tools in the following categories: Automation APIs/Frameworks, Record and Replay Tools, Automated Test Input Generation Techniques, Bug and Error Reporting/Monitoring Tools, Mobile Testing Services. More details about the typologies of tools, along with relevant works from literature presenting examples of them, are given in the following.

Automation APIs/Frameworks

Automation APIs are tools that provide means for interacting with the GUI or for obtaining GUI-related information to describe the content of the screens and to verify the state of the AUT. Testers typically leverage those APIs to manually write down test scripts, that can then be launched and verified automatically. Many of the automation frameworks leverage white or *grey box* approaches, which extract high-level properties of the app (e.g., the list of the activities and the list of UI elements contained in each activity) in order to generate events and traverse the GUI [26].

While being among the most powerful tools for the expressivity of developed test scripts and for the possibility of using such scripts also for Regression Testing, the main shortcoming of GUI Automation Frameworks – as it will be detailed later – is the very high maintenance cost during the normal evolution of the App to which tests are associated.

Two testing tools officially developed by Android, Espresso[2] and UI Automator[3] are among the most widespread Automation Framework and APIs. UI Automation[4] is a counterpart to UI Automator designed for testing the GUI of iOS apps; its automation engine is now the basis for Appium[5], a cross-platform testing tool able to automate both iOS and Android apps. Other open-source and widely-adopted alternatives in the literature are Robolectric[6] [88] and Robotium[7] [102].

Several commercial Automation APIs, like Calabash[8] [52], Quantum[9] and Qmetry[10], offer to testers the possibility of writing test cases in Natural Language.

[2] http://developer.android.com/training/testing/ui-testing/espresso-testing

[3] http://developer.android.com/training/testing/ui-automator

[4] https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/09-ui_testing.html

[5] http://appium.io

[6] http://robolectric.org

[7] https://robotium.com

[8] http://calaba.sh

[9] http://community.perfectomobile.com/posts/1286012-introducing-quantum-framework/

[10] http://qmetry.github.io/qaf/"

Record and Replay Tools

Record & Replay tools offer tester/developers the possibility of generating test scripts by capturing the interactions performed on the AUT during an execution of its usage scenarios to be tested.

The important advantages exhibited by Record & Replay tools are the registration of test cases from the final user's perspective, the low effort required for creating test scripts if compared to manual writing down scripts using GUI Automation APIs, and the possibility of creating tests without having information about the implementation of the application. As well, these techniques expose several shortcomings when it comes to the accuracy and the portability of the generated test scripts.

Several of the available Record & Replay Tools are conceived as extensions of existing GUI Automation APIs, to provide another way of creating test scripts: it is the case of Espresso Test Recorder[11], Robotium Recorder[12], Xamarin Test Recorder[13]. Other examples of testing tools cited in the literature that leverage such approach are RERAN [45], VALERA [53], Mosaic [48], Barista [39], ODBR [79], and SPAG-C [64].

Automated Test Input Generation Techniques

Many tools available in the literature are used to generate sequences of inputs for the applications to test. Most of the time, in these cases implicit oracles (i.e., triggering crashes in the AUT make test cases considering as failing) are used. Automated Input generation can be seen as a way to reduce the effort and cost of manually writing test scripts for GUI Automation frameworks, or for capturing sequences of interactions with the AUT. However, as reported by Choudary et al. [26], the tools available in the literature still expose several issues especially in terms of efficiency in finding bugs.

The generation of inputs, in its simplest form, can be random. Monkey[14] is official random testers provided by Android; another example from literature is Dynodroid [70].

[11] https://developer.android.com/studio/test/espresso-test-recorder
[12] https://robotium.com/products/robotium-recorder
[13] http://www.xamarin.com/test-cloud/recorder
[14] http://developer.android.com/tools/help/monkey.html

Input Generation Techniques are instead said Systematic when the inputs are not generated randomly, but in order to maximize some coverage function (e.g., code coverage, or Activity coverage). Examples from literature of Systematic Input Generation Testing Approaches are AndroidRipper [13] and CrashScope [82].

Model-Based Input Generation techniques define sequences of inputs according to a model of the user interface, that can be provided by the tester/developer or obtained automatically by the tool itself. Examples from literature are MobiGUItar [14] or Swifthand [25]. Advanced testing generation is not limited to model-based approaches, with examples of tools featuring different input generation approaches.

Several recent tools have adopted the search-based approach to software testing, adopting meta-heuristics (such as Genetic Algorithms) to automate or partially automate testing tasks like the generation of test data or test sequences [75]: Mahmood et al. have presented EvoDroid, that leverages an evolutionary algorithm to generate test cases [72]; Jabbarvand et al. have described two algorithms for energy-aware test suite minimization [54]; Mao et al. introduced SAPIENZ, a multi-objective search-based approach for the minimization of test sequence length, fault revelation and coverage [73]. The SAPIENZ tool has had significant industrial impacts and has been officially adopted by Facebook to test their mobile app[15].

Bug and Error Reporting/Monitoring Tools

Under this category are considered all the tools used to track the unexpected behaviour of Mobile apps, either via user's reports or through automated crash detection mechanisms.

Examples of these tools are ODBR [79], which leverages the UI Automator framework for documenting sequences of inputs that can lead to crashes in a given Android app, and FUSION [80], which links information about the user experience with the app provided by the user itself with program analysis performed automatically.

[15] https://code.fb.com/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/

Mobile Testing Services

Lastly, several online testing services are available for Mobile apps, typically intended for tackling problems like device diversity and OS compatibility, and the generally high cost and required effort for testing Mobile apps. Mobile Testing Services typically use sets of different devices on which automated tests are executed, and can be used for traditional functional testing, but also for verifying non-functional properties of Mobile apps like usability of the GUI, security, energy consumption, localization.

To underline the importance for the Mobile industry of such services, it is worth reporting that both Google (with Android Robo Test[16]) and Amazon (with Fuzz Test[17]) recently released cloud service for the automated testing of Android apps.

2.5 Challenges in Mobile app testing

Developers typically face a specific set of challenges when building apps for one or more mobile platforms. A study performed by Joorabchi et al. [55] includes among the most relevant difficulties in the mobile development practices: the selection of the proper nature of the app to be developed (i.e., Native vs. Web or Hybrid applications); the limited capabilities of the typical device for a given platform; the choice between reusing others' code or writing from scratch; the multiplication of time, effort and budget due to the multitude of devices and platforms the apps must be able to run on (i.e. *Fragmentation*); the rapid changes of requirements and typically rapid life cycle of an average Mobile app.

Those difficulties experienced when developing mobile (and, specifically, Android) apps are reflected, if not magnified, in the practice of testing. Moreover, Mobile app testing also has to take into account several aspects that can be completely overlooked when testing traditional desktop applications. Muccini et al. [83], Kirubakaran et al. [56] and Kaur et al. [16] identified a set of characteristics of Mobile apps leading to specific forms of non-functional testing: mobile connectivity scenarios (i.e, coping with unreliable Wi-Fi or 3G connections) and rapid changes of

[16] http://firebase.google.com/docs/test-lab/robo-ux-test

[17] http://docs.aws.amazon.com/devicefarm/latest/developerguide/test-types-built-in-fuzz.html

connectivity type; limited resources of devices; data intensivity of the applications; constant interruptions caused by system; very short time to market; very high amount of multi-tasking and communication with other apps.

Due to the listed difficulties, there is a substantial unanimity in the literature about a general tendency of Android developers to neglect automated testing, and to rely instead on manual testing only. As it emerges from sets of interviews to contributors to open-source projects performed by Linares-Vasquez et al. [67] and by Kochhar et al. [57], the time constraints, lack of properly documented testing tools, and high costs for developing and managing test artifacts are the main reasons for such preference towards manual testing procedures.

2.5.1 Fragmentation

The Fragmentation concept (also known as Device Diversity) encompasses, for the Android ecosystem, two different concerns [51]. *Hardware-based fragmentation* refers to the fact that devices based on the same Android operating system run on different processors, graphic cards, screen sizes, pixel densities. According to a report of August 2015[18], more than 24 thousand different devices, built by more than 12 hundred vendors, were existing at the time, and many display sizes, ratios and pixel densities could be found (see figure 2.5).

Software-based fragmentation refers to the fact that several versions of the Android OS exist in parallel and that, at the same time, vendors and carriers may offer customizations for apps and OS GUIs. Device fragmentation is a specific issue of the Android ecosystem and not of Mobile development in general, being the number of available devices and maintained versions of the OS for iOS apps very limited.

Device fragmentation has relevant impacts on many aspects of Mobile development. First of all, for coping with software fragmentation of the OS, developers must cope with deprecated or even removed methods of the framework they use; hence, apps may need to be developed differently for different versions of the OS [49][99]. Software-based fragmentation, especially for what concerns the customized versions of the OS by different vendors, also creates concerns about the security of the apps [104].

[18] https://opensignal.com/reports/2015/08/android-fragmentation/

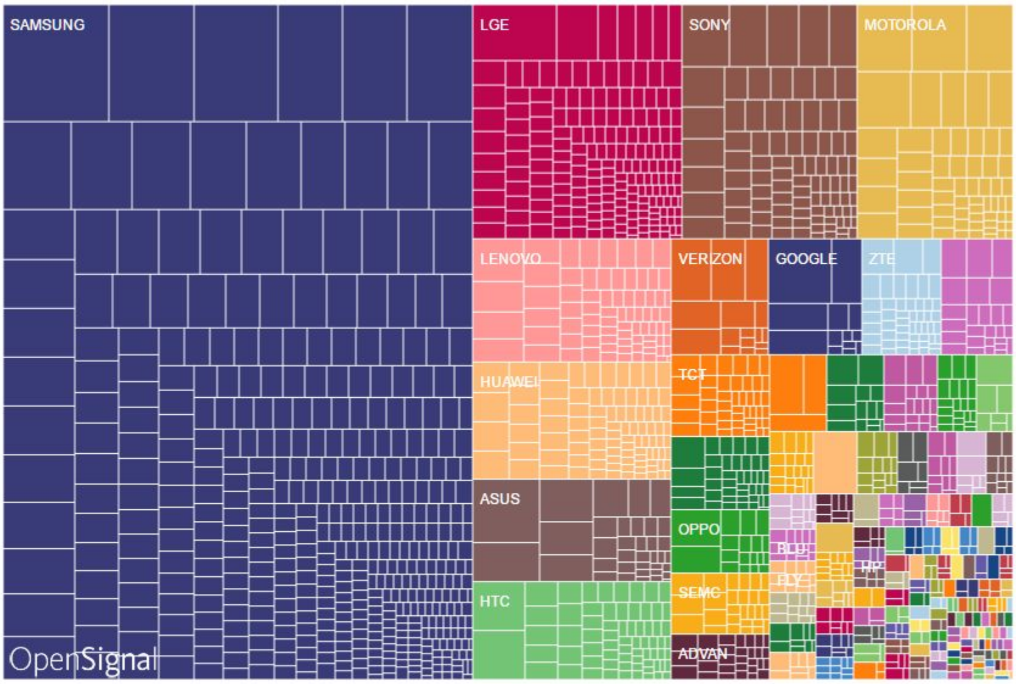


Fig. 2.5 Relative screen sizes of Android devices available at August 2015 (source: https://www.xda-developers.com)

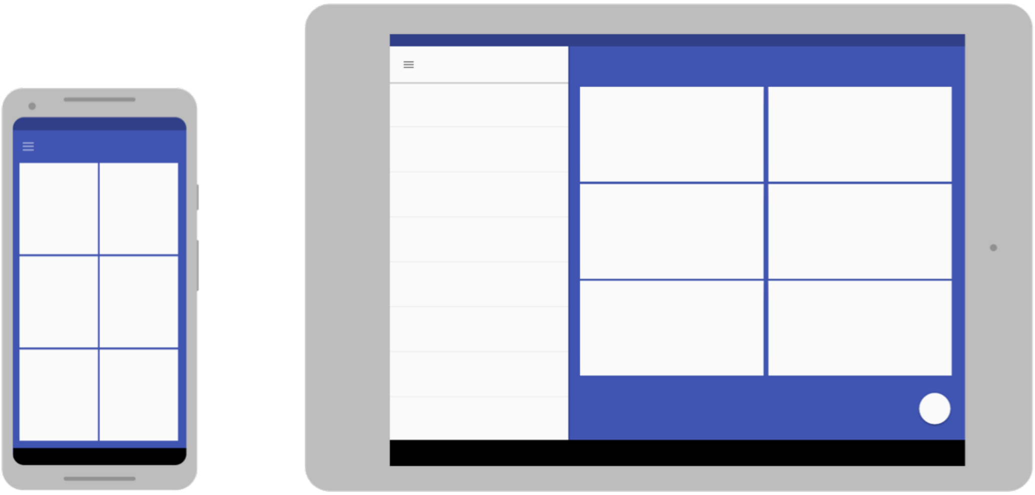


Fig. 2.6 Different layouts inflated for the same Activity on different devices (source: https://developer.android.com/training/multiscreen/screensizes)

From the point of view of system and GUI testing of Android apps, fragmentation translates specifically to the need of verifying the correct inflation of the layouts, which can be different to comply with the screen size, density and orientation of the device (since the same Activity can adopt different layouts according to the current orientation of the device: see figure 2.6). Layout-based, or 2^{nd} generation, testing techniques must hence consider the possible different widgets of the layouts that are used to populate the Activities. Visual, or 3^{rd} generation, testing techniques, must cope with the elements of the interface possibly not appearing on the screen, or appearing at different resolutions and hence invalidating the correct recognition of the screen captures used as visual identifiers.

2.5.2 Testing Hybrid and Web-Based applications

The development of hybrid applications comes in handy when small teams are not capable of building and maintaining several code bases for apps that are intended to work on different platforms. Several frameworks are available for building hybrid apps, e.g. React, Ionic, and Google's Flutter.

While less prone to the fragmentation problem, Web apps, even when already tested for regular browsers, should be specifically tested in mobile environments for what concerns battery usage, connectivity issues, and performance [2]. GUI testing of web applications is also fundamental, in order to verify that all the elements of the user interface are visible, aligned and rendered in the same way as in the browser interface. A limitation for testing Web applications in the Android environment is the partial inapplicability of GUI Automation Frameworks specific to Android, and in general of Layout-based testing tools, with many of the most widespread testing tools (like Espresso or Robotium) providing limited or no support to what is loaded inside WebViews.

As reported by Ahmad et al. [1], lack of access to platform features, changes in contextual or environmental factors of the apps, integration and conformance issues, and diversity in user interface add other layers of complexity and fragmentation when testing hybrid Apps. United approaches, like Squish[19], are typically at the level of abstraction of the GUI only ($3^{rd} generation$ tools), and leverage Capture & Replay approaches for test case generations.

[19] https://www.froglogic.com/squish/

2.6 Maintenance of Automated tests

One of the biggest challenges of test automation – in general – is to keep pace with the changes of the Application Under Test [69]. Maintenance constitutes a significant cost for all forms of automated testing, that grows with the size of the test suite and the coverage of the features of the AUT. Maintenance of test scripts is required to keep them aligned with the varying requirements of the AUT [20], to its design or simply to its expected behaviour. GUI test cases are expected to face relevant maintenance costs, since (i) GUI tests are impacted by modifications in all abstraction layers that underlie the GUI and (ii) test scenarios performed through the GUI are typically longer than unit tests based on individual classes or modules of the applications.

Testware maintenance has a growing cost with the amount of code changed in the AUT, and often requires a certain level of knowledge of the changed implementation details, in order to make the test cases perform the same tests on the new release of the AUT [91]. Additionally, the maintenance burden of automated testware typically increases with time, since the relative amount of test code with respect to production code tends to increase with time [86]. As pointed out in a work by Berner et al. [22], who also defined a theoretical cost model for the maintenance of automated testing, testware maintenance can also depend on factors different than the amount and frequency of changes performed on the AUT: among them, they cite missing architecture for the test software, reimplementation of repetitive actions through test cases instead of reuse, poor organization of the test code and test data, and missing verification of testware itself. Alegroth et al., who performed an empirical analysis of factors contributing to maintenance of GUI test cases, identified a set of thirteen factors, ranging from technical (e.g., test case similarity, test case length and presence of loops and flows in the test cases) to human (e.g., sequential mindset to test case description), that contribute to those costs. The applicability of a testing technique to a real context, hence, mainly depends on the likelihood of a positive Return on Investment (ROI) when the AUT undergoes a reasonable amount of changes. In industrial contexts, moreover, a proper education of developers and testers to the possible costs of testware maintenance is fundamental: as Fewster reports, if the maintenance of testware automation is ignored and not performed as soon as possible to comply with the changes of the AUT, the update of an entire test suite can cost as much or even more than re-performing all the test cases manually [40].

Several studies in the literature have measured the costs and benefits of state of the art GUI testing techniques and compared them to the costs of manual testing. A case study by Andersson and Pareto, for instance, compared the needed maintenance costs by Capture and Replay testing techniques for GUI testing of desktop applications [90]; it was highlighted that, if the applications have frequent releases, the adoption of automated testing techniques may become a burden for developers rather than an advantage, because of a lower cost-effectiveness when compared to manual regression testing. Several related manuscripts in the literature have investigated the applicability of various testing techniques in industrial settings: examples are the work by Borjesson and Feldt [23] and by Alegroth et al. [6], who evaluated the advantages of adopting Visual GUI Testing techniques respectively at Saab and Siemens; Nguyen et al. considered maintenance costs among the most important factors in the design of their model-based tool GUITAR [85].

2.6.1 Definition of Fragile GUI Tests

Fragile tests, as defined by Garousi et al. [43], are tests that are broken during the evolution of an application by changes that are not related to the test logic itself or to the specific features that they exercise. Test fragility represents a significant maintenance issue for testware of all domains, and has been largely explored in the literature.

Grechanik et al. [47] and Memon et al. [76] propose approaches for automatically fix broken test cases for GUI-based applications; Gao et al. developed SITAR [42], a technique to automatically repair test suites, modeling and repairing test cases using Event-Flow Graphs (EFGs); Leotta et al. [61][62] reported the outcomes of an empirical study about the fragility of GUI tests for web applications.

Through the studies reported in this thesis, the following definition has been adopted for fragile GUI tests:

> A GUI test case is fragile if it requires interventions when the application evolves (i.e., between subsequent releases) due to any modification applied to the Application Under Test.

GUI testing differs significantly from testing of traditional software. Being system level tests, test cases developed with GUI automation frameworks may be affected by variations in the features of the app, but also from even small interventions in the appearance and presentation of the screens by which the GUI is composed. A large number of test case executions can be lead to malfunctions if they refer to GUI elements that have been renamed, moved or otherwise altered [74, 77].

The adopted definition distinguishes (as fragile vs. non-fragile) tests that need interventions because of any type of change in the AUT from tests that do not require changes because of the evolution of the app, but that instead are just subject to variations in the test logic or in the functions that are proper of the GUI Automation Frameworks adopted.

The assumption behind the analysis presented in this thesis is that mobile test cases – for which fragility was not previously explored by large-scale academic studies – may be heavily subject to fragility, because (i) mobile apps mostly rely on their GUIs for all the interactions with the users and data presentation; (ii) mobile GUIs are subject to frequent changes during the app's lifespan; (iii) albeit having several similarities with web-based apps, mobile GUIs are described in specific ways and their layout feature properties that are not present in other domains..

Chapter 3

Research Design and Approach

The work presented in this thesis mainly follows an empirical experimental approach, using controlled experiment designs, surveys and mining from software repositories. In the latest phase of the study, this empirical approach was paired with the development of a testing tool, whose requirements and design were based on the results of the previous works.

This chapter provides a summary of the studies that were performed to pursue the three high-level goals of the thesis, that were detailed in section 1.

The selected testing tools and the mining from repositories, that were common to most of the different studies, are also detailed in the remaining sections of this chapter.

3.1 Overall study design

Five different studies have been performed in order to pursue the goals of this thesis. Each study, except for a first exploratory case study, was tailored around a principal research question, which was linked with one of the three high-level goals of the study. Individual research questions were then subdivided into sub-research questions, that are presented in detail in the individual chapters dedicated to the studies.

Table 3.2 reports the names of the studies and the main Research Questions they responded, along with the goal(s) they were linked to (a summary of the goals is reported in table 3.1 for ease of reading); table 3.3 reports the characteristics of said studies; table A.1, in Appendix A, reports the full list of sub-research questions of all the studies. A brief high-level description of the studies is provided in the following:

- **Study 0 - Case study with K-9 mail:** an exploratory case study on an open-source Android application, in order to assess – through qualitative analysis – the main issues faced during the maintenance of suites created with automated testing tools for Android GUI testing;
- **Study 1 - Survey with mobile developers from the industry:** a qualitative survey conducted with IT professional from the Turin area, with the objective of assessing the issues faced by developers and testers when developing and maintaining test suites for Android applications;
- **Study 2 - Controlled experiment with Graduate students:** an experiment conducted to gather quantitative evidence of the productivity of graduate students with automated GUI testing tools and techniques. The gathered data are paired with a structured interview, and examined with qualitative analysis, thematic analysis and descriptive statistics;
- **Study 3 - Measures of Diffusion and Evolution of Testware in OS projects:** a quantitative evaluation of the issue of maintenance and fragility among open-source Android projects. The study involved a data mining experiment, and the extracted data were analyzed with quantitative analysis and through the definition of a novel set of metrics;
- **Study 4 - Taxonomy of Fragility causes:** application of the Grounded Theory technique to a set of diff files relative to the evolution of open-source Android projects;
- **Study 5 - Layout-based vs Generated visual test cases: An experiment with TOGGLE:** description of the architecture and implementation of a tool for a translational approach (from layout-based to visual test scripts) for Android GUI testing. The tool is applied for a confirmative study on two Android open-source applications, to compare the performance and quality of generated visual test suites.

Table 3.1 Goals of the thesis

Goal number	Goal name	Goal description
G1	Perception and Usability	Investigate the ease of use of existing Android testing tools, and the perception that potential users have of them.
G2	Adoption and Size	Quantify the adoption of such tools by industry and OS developers, and investigate the size and relevance of testware in tested Android apps.
G3	Evolution and Fragility	Quantify the effort needed in maintaining testware during the evolution of an Android project, and identify the main causes of test fragility.
G4	General Android testing issues	Identify common challenges in performing Android testing, and find possible guidelines to mitigate such challenges.

Table 3.2 Performed studies and main research questions

Study	Research Question	Addressed Goals
S1: Survey with mobile developers from the Industry	RQ 1: What is the perception of GUI testing for Android apps among practitioners from the industry?	G1, G3, G4
S2: Controlled experiment with graduate students	RQ 2: How usable are GUI testing tools and what is the productivity of graduate students using them?	G1, G4
S3: Measures of diffusion and evolution of testware in OS projects	RQ 3: What is the adoption and typical evolution of test suites with automated GUI testing frameworks among Android open source projects?	G2, G3
S4: Taxonomy of fragility causes	RQ 4: Why and with which frequency fragilities occur in tested Android projects?	G3
S5: Development and validation of TOGGLE, a Layout-based to Visual test case translator	RQ 5: What is the dependability and performance of visual test cases generated by translation?	G3, G4

Table 3.3 Details of performed studies

Study	Type of Study	Data collection	Data analysis	Context
S0	Exploratory case study	Data repository	Qualitative analysis	OS Project
S1	Qualitative survey	Interview	Qualitative analysis	Seven IT companies
S2	Controlled Experiment	Observation Interview	Qualitative analysis Descriptive statistics Thematic analysis	Master’s students
S3	Data mining experiment	Data repository	Quantitative analysis	GitHub software repository
S4	Grounded Theory study	Data repository	Qualitative analysis	GitHub software repository
S5	Confirmative case study	Observation	Quantitative analysis Descriptive statistics	Two OS projects

Table 3.4 Characteristics of the selected Layout-based GUI Testing Frameworks

Framework	Black Box	Non-native app testing	Multi-app	C&R	Multi-OS	Level	Image Recognition
Espresso	No	Partial	No	No	No	GUI-Level	No
UIAutomator	Supported	Partial	Supported	No	No	GUI-Level	No
Robolectric	No	No	No	No	No	Unit-Level	No
Robotium	Yes	Supported	No	Supported	No	Unit-Level	No
Selendroid	Yes	Supported	No	No	No	GUI-Level	No
Appium	Yes	Supported	No	Supported	Yes	GUI-Level	Supported

3.2 Selected testing tools for the studies

The studies that are described in the following have taken into consideration a set of testing tools that can be used for testing Android applications. 2^{nd} and 3^{rd} generation GUI testing tools were taken into account. The main inclusion criteria for testing tools to be considered was their open-source nature, and, for Layout-based testing ones, the possibility of generating test scripts in a common language, to make the application of size metrics on produced test code possible. Java has been chosen as the common script generation language for the selected tools.

3.2.1 Selected Layout-based testing tools

Regarding the Layout-based testing technique, six testing tools have been selected, in order to cover all the possible features offered by such generation of GUI testing. Information about the six selected Layout-based testing tools, previously reported in [32], is given in the following.

A classification of the main peculiarities of the tools is given in table 3.4. In particular, the table reports the type of testing that can be performed using a given tool (either white box or black box testing); the presence of multi-platform capabilities; the possibility of using the tool for other forms of testing rather than GUI testing; the support to other ways of generating test cases (possibly with add-ons of the basic tool) in addition to manual scripting (i.e., through Capture and Replay and/or Image Recognition); the possibility of testing multiple applications at the same time or to span the GUI of the Android operating system in addition to the application activities.

Based on the characteristics listed in table 3.4, and on similar investigations available in literature [68], the selected sample of six tools is evaluated as representative of the category of Layout-based testing tools. Several other frameworks are based on the six considered testing tools, and hence can be represented by them: RaceDriver [95] or Barista [39] are based on Espresso; T+ [66] or Fusion [81] build their testing approaches over the UI Automator APIs; Segen [84] (as Appium) works on the Selenium WebDriver framework; many advanced tools for ripping interactions with the GUI of Android apps and recording Capture & Replay tests are based on Robotium (examples are A2T2 [11] and AndroidRipper [12]).

Espresso

The Espresso automation framework is part of the Android Instrumentation Framework, and it is the tool officially supported by Android for testing the GUI of a single application, without taking into consideration the host OS. The used approach for the generation of test cases is defined gray-box, meaning that the tester/developer can develop test scripts without knowing the internal implementation of the activities, but needs access at least to the definition of the appearance of the activities, and to the declaration of the internal disposition of the elements of the activities. Coding test scripts of Espresso is typically carried in the Android Studio IDE, hence having full access to the layout and resource files of the application. An expansion of Espresso, namely Espresso Test Recorder, allows the definition of test scripts through the Capture & Replay technique, executing the desired sequence of operations on an Android Virtual Device. Espresso can be used to test native and hybrid applications, leveraging the EspressoWeb extension[1].

UI Automator

The UI Automator framework is part of the Android Instrumentation Framework, and it is the tool officially suggested for testing multiple applications, also spanning the GUI of the Android operating system. The tool can also perform operations on the GUI of the operating system, and operate on the system and display settings (e.g., enabling the WiFi, working on the settings for fonts and colors of the screen). As for Espresso, support for the hybrid applications is given, through the possibility of automating WebViews. Tests written with UI Automator are based on the elements of the app that are exposed through its user interface, hence the tool is considered as a black-box testing tool.

[1] https://developer.android.com/training/testing/espresso/web.html

Robolectric

Robolectric is a white-box unit testing tool for Android, that allows testing on the Java Virtual Machine directly, without the need (as for all the other testing tools considered) of a real or emulated Android device. The checks and assertions used by Robolectric are mostly at code-level, so it cannot be used for testing the actual appearance of the user interface of an Android app, but at most its definition and instantiation. The optional emulation of a device can be enabled to test interaction with a full Android environment.

Robotium

Robotium is an extension of the JUnit framework for the definition of unit tests of Android apps; it has been very popular among Android developers before the release of Espresso and UI Automator. Tests created with Robotium can either be black-box or white-box, and can be deployed on any kind of Android app (either native, web-based or hybrid), with the limitation of testing a single application at a time and without the possibility of exercising the operating system user interface. An extension, namely Robotium Recorder, allows the creation of test scripts with the Capture & Replay technique.

Selendroid

Selendroid[2] is conceived as the Android counterpart of Selenium, a very popular tool for automated testing of web applications. The tool allows the execution of automated test scripts on native, hybrid and web-based applications (thanks to the integration with Selenium WebDriver). The generated test scripts are black box, since the widgets of the application are retrieved without having any access to the source code; the AUT is instrumented through the use of the Android Instrumentation Framework.

[2] http://selendroid.io/

Appium

Appium is a testing tool based on Selenium WebDriver and Selendroid, to create black-box tests for both Android and iOS. The test cases can be created through manual scripting, and exported in a series of scripting language (ranging from Python to C# and Java), or with the use of an Image Recognition extension based on the SikuliX libraries.

3.2.2 Selected Visual GUI testing tools

Two different Visual testing tools have been considered for the further steps of this doctoral work: Sikuli and EyeAutomate. Both the considered tools are not specifically developed for working with Android applications, hence they have been used on an emulated device on the screen of the desktop PC. The AVD provided by Android[3] or the Vysor[4] tool have been used for that purpose. The main characteristics of the tools are described in the following.

Sikuli

Sikuli[5] is an open-source image recognition tool, presented originally by Yeh et al. [100] and then no longer maintained in its original branch. The version of the testing tool using for the following work is Raiman's SikuliX[6]. The tool is powered by OpenCV for its image recognition functions, and it can run on any desktop computer platform. The supported languages, for the generation of test scripts, are Python, RobotFramework, Ruby, JavaScript and any Java aware programming and scripting language. The tool contains modules for OCR (i.e., Optical Character Recognition) powered by the Tesseract library, and is paired with an IDE that makes a set of basic commands easily reachable for the tester/developer.

[3] https://developer.android.com/studio/run/managing-avds
[4] https://www.vysor.io/
[5] http://www.sikuli.org/
[6] http://sikulix.com/

EyeAutomate

EyeAutomate[7] is an open-source image-recognition library, developed in Java by Auqtus and subject of several studies in literature about the advantages of Visual testing with respect to Layout based testing techniques, and the feasibility of a Visual testing set up on industrial settings. It comes paired with an IDE, EyeStudio, which has embedded many simple to complex commands that can be performed on the screen to emulate the mouse and keyboard of a desktop pc. The EyeStudio editor can record and save test scripts in plain text format. The pairing algorithms for the provided screenshot is based on pixel and vector-based image recognition, and AI-based features are provided by some of its components.

3.3 Mining of Android repositories from GitHub

The following parts of this study required a statistically significant sample of tested Android applications whose usage of GUI testing, evolution and fragility of test code could be inspected and measured. The natural choice for finding a context of open-source Android app was the GitHub repository, which has been mined for searching Android applications through an automated procedure which is described in detail in the following.

3.3.1 Search for Android projects

The first operation to conduct in order to obtain the context of tested open-source Android projects was the extraction of all Android projects hosted on GitHub. To that extent, the GitHub Repository Search API[8] has been leveraged. Such API allows extracting all the repositories that contain a given keyword in their names, readme files or descriptions. The Search API also allows, with the *language* parameter, to filter the projects also according to the programming language they are written into. The *created* parameter of the API allows instead to filter the GitHub repositories that have been created in the interval between the two dates passed as parameters. Since the GitHub API has an upper limit to 1000 maximum repositories returned by each

[7] http://eyeautomate.com/eyeautomate.html
[8] https://developer.github.com/v3/search/

call, the *created* parameter has been used to cycle with different queries over a set of disjoint date ranges, in order to obtain fewer than 1000 results for each of them.

The GitHub API has been used inside a bash script, using the cURL bash function, and the results (which are provided by the API in Json format) have been examined automatically using the jsawk tool[9]. The data mining process has been performed between September and December 2016.

The resulting search string, using the GitHub search API, is the following one:

```
curl −x, −u $USER:$PASSWORD −H ’Accept: application/vnd
    .github.v3.text−match+json’ ’https://api.github.com/
    search/repositories?q=android+language:java+created:
    "’$CURR_DATE_RANGE’"&sort=stars&order=desc&page=’
    $CURRENT_PAGE’’
```

A second filtering step has been then applied to the Android repositories, in order to cut out from the context all the repositories which did not have a release history. This has been done because the final aim of the studies on the context of Android projects was to track the evolution of test code, and the occurrences of fragilities inside it. Hence, projects without at least another tagged release in addition to the *master* were removed for the context, because they did not allow even for a single comparison between two consecutive releases. The Git Tags API[10], which outputs the names of all the tagged releases of a given GitHub repository, has been leveraged to this purpose, with the exclusion from the selected set of all projects which returned a single tagged release.

A set of repositories obtained by searching the word “Android” may include actual Android applications, but also spurious results, e.g. libraries, utilities and applications for other systems that have interactions with Android devices. Hence, a heuristic was needed to filter out those spurious projects automatically and as much accurately as possible. The method used by Das et al. [36] was adopted: a given repository was considered an actual Android application if it contained one (or more) Android Manifest file. A Manifest file[11] is a mandatory file (with that exact name) for any Android app, since it contains essential metadata for the building, installation and

[9] https://github.com/micha/jsawk

[10] https://developer.github.com/v3/git/tags/

[11] https://developer.android.com/guide/topics/manifest/manifest-intro

use of the application (e.g., all the components of the app, the required permissions and hardware features are specified in it). Hence, repositories that do not contain any Manifest file are cut out from the context. Repositories with multiple Manifest files (that may suggest either the presence of multiple builds for a single app, or the inclusion in the same repository of multiple apps) have been evaluated as single projects for the subsequent investigations, since the metrics described in following sections apply to whole repositories and not necessarily to single apps.

To cut out the projects without Android manifest files, the GitHub Code Search API[12] was used. The API offers the *filename* parameter, to search for keywords inside files with a given filenames.

The GitHub Code Search API has some limitations, which however have been considered not very relevant in the context of our study. As explained in its documentation, (i) only the default branch (the master branch in most cases) is considered for code search; (ii) only files smaller than 384 kb are searchable; (iii) only repositories with less than 500,000 files are searchable. The second and third issues were not considered a concern for searching Android applications, since the typical size of such projects is not particularly big.

For the purposes of this study, the GitHub Code Search API has been parameterized using the keyword *manifest*, and AndroidManifest.xml as the required filename, as in the following:

```
curl -x , -u $USER:$PASSWORD -H 'Accept:application/vnd
    .github.v3.text-match+json' 'https://api.github.com
    /search/code?q=manifest+filename:AndroidManifest.
    xml+repo:ligi/passandroid' | jsawk 'return this.
    items' | jsawk 'return this.path'
```

To limit the context only to applications that were provided with an actual GUI, a second heuristic was adopted, cutting out all the projects that did not feature at least an occurrence of a call to the setContentView method, or a declaration of a FragmentTransaction object. The selection of those methods has been done because setContentView is typically the first method called in the onCreate() method of any Activity, and has the role of populating the screen with the widgets described in a layout resource. The FragmentTransaction, on the other hand, is used to handle

[12] https://developer.github.com/v3/search/#search-code

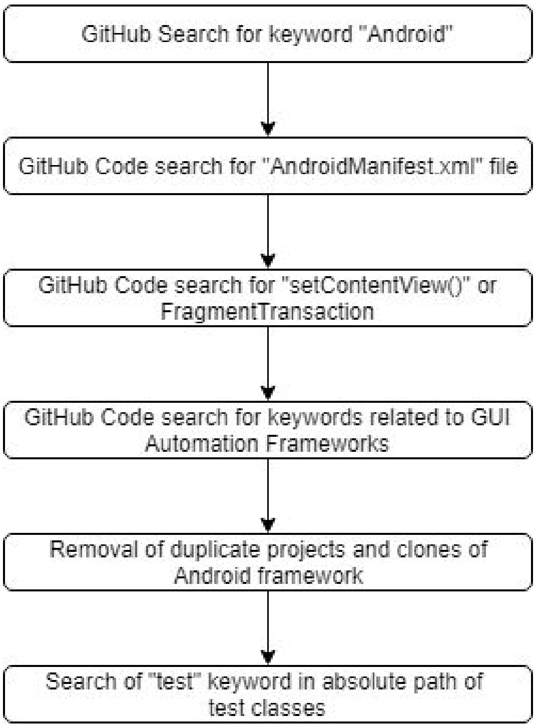

GitHub Search for keyword "Android"
GitHub Code search for "AndroidManifest.xml" file
GitHub Code search for "setContentView()" or FragmentTransaction
GitHub Code search for keywords related to GUI Automation Frameworks
Removal of duplicate projects and clones of Android framework
Search of "test" keyword in absolute path of test classes

Fig. 3.1 Search procedure for Android projects and test classes associated with the considered testing tools

the creation of a Fragment to populate the activities instead of using the static function to load a layout. Such filtering was performed, again, leveraging the GitHub Code Search API, passing the names of the described function and object as search keyword.

3.3.2 Search for Testing Tools code

The further step in the project selection from GitHub was a search for projects featuring the six Layout-based testing tools detailed in section 3.2.2. The names of the testing tools themselves have been considered as legitimate keywords to search for their adoption, since they are part of include statements that are needed for the tools to work. The only exception has been made for Espresso, since the term *espresso* is a common word that was found in a number of projects which had no connection to the Espresso testing framework. Hence, the keyword *test.espresso*, which is contained in most of the includes needed to use the tool, was searched.

To search for the usage of testing tools, the GitHub Search Code API was leveraged as well. Any Java class featuring a keyword related to a given testing tool was considered as a class associated with the tool (for instance, a class featuring the statement “import static android.support.test.espresso.Espresso.onView” is considered as a class featuring Espresso). After this search phase, the projects were divided into six sets, based on the testing tools that they featured. The sets of projects were not considered as mutually exclusive: a repository featuring keywords associated with multiple testing tools has been added to all the respective repository sets.

After an examination of the individual sets of repository names related to each of the tools, it could be noticed that multiple projects were clones of the Android sdk, or sets of frameworks or sdks that were of no interest for the study. All the projects named after a combination of the keywords *android-platform*, *android-framework*, *framework* and *base* were removed from the context. Also, duplicate projects were removed from the sets.

To avoid considering in the results also projects that contained spurious usages of the testing frameworks and considering as test classes also normal application classes that included import statements related to the tools, a final filtering has been performed by analyzing the names of the classes identified as associated with the tools. The classes were considered as “test classes” only if their absolute path featured the keyword *test*. All the test classes with keywords contained only in comments were also removed from the sets.

The whole procedure adopted for searching Android repositories on GitHub and test classes associated with the six selected testing tools is summarized in image 3.1.

Chapter 4

Study 0: Case study with K-9 Mail

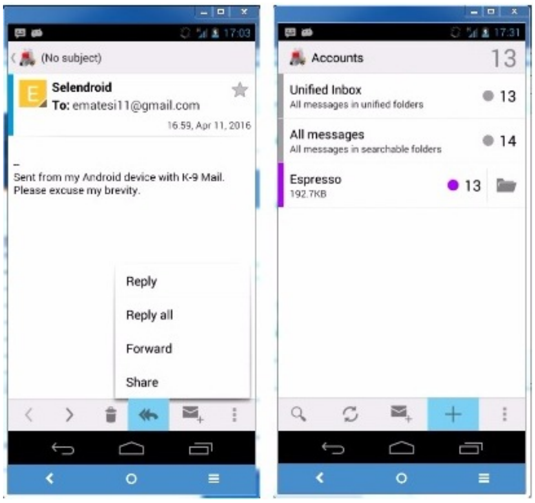
The first study of this research revolved around the analysis of the selected testing tools for Android applications with a popular app released on the Play Store, to perform an exploratory assessment study of the difficulties encountered when using automated testing techniques for the Android platforms.

The case study aimed at assessing the advantages and disadvantages showcased by popular testing tools for Automated GUI testing, and at measuring, on a simple test suite developed manually on different versions of the same applications, the needed amount of maintenance and the occurrence of fragilities. A preliminary classification of the causes of such fragilities was also provided.

The case study design and the results reported in this section have been originally published as a workshop paper at INTUITEST ‘16 [34].

4.1 Study Design

The selected app for the case study was K-9 mail, a widespread (more than 5 million downloads) e-mail client for Android devices. Figure 4.1 shows two sample screen captures of the application. The application, whose original release dates back to 2009, had reached at the time of the conduction of the experiment the release 5.010. The application was open-source, and publicly available on the GitHub repository. The open-source nature of the application was fundamental for the aims of the study, since instruments like Espresso and UI Automator have a strong connection to



--
Sent from my Android device with K-9 Mail.
Please excuse my brevity.

Fig. 4.1 Screen captures from K-9 Mail, release v5.010

Table 4.1 Test cases defined for K-9 mail

ID	Description
TC1	Successful authentication
TC2	Send a message
TC3	Reply to a message
TC4	Delete a message
TC5	Add an user account
TC6	Delete an user account
TC7	Delete account data
TC8	Restore account data
TC9	Export account settings
TC10	Import account settings

the application code (for instance, for the retrieval of identifiers used to select the elements of the user interface to interact with). The application was selected after a search for a case study with a long enough lifespan, and a significant code base.

A test suite of ten test cases has been designed based on the latest stable version of the App at the time of the experiment. Test scenarios were designed to test all the main features of the applications, and are listed in table 4.1. Espresso, UI Automator and Selendroid were selected as testing tools for the definition of test scripts. In addition to them, the Sikuli testing tool was used, as an example of the application of third generation testing tools to Android testing.

Development and execution of Espresso and UI Automator test suites were performed in the Integrated Development Environment (IDE) Android Studio 1.1. The Eclipse Mars IDE was used for the development of Selendroid test cases, while Sikuli was used with its IDE. The tests have been executed on two different Android devices, using Android API 23 and Android API 19 (to comply with the limitation of Selendroid 0.10.0, working only with Android versions prior to 20). Finally, the Vysor tool was used to mirror the screen of the Android device on which the tests were run on the desktop PC.

After their definition, tests then have been reproduced, when possible, for five previous releases of the application. The last stable minor releases belonging to three different major releases were selected: v2.995 (April 2010); v3.993 (December 2011); v4.804 (June 2014). Two other releases of the application were selected randomly: v2.102 (November 2009); v3.309 (November 2010). Tests were based on the localized version (in Italian) of the app. Then, the test cases developed for one version of the application were executed on the next selected release, to understand whether they were still working, or they needed maintenance due to the verification of fragilities. The manual repair performed on non-working test cases, and the examination of changed test code, posed the basis for a first assessment of the possible causes of fragilities for Android applications.

4.2 Results

This section describes the experience gathered from the K-9 Mail case study, the amount of maintenance to perform on 2nd and 3rd generation test suites, and the resulting preliminary categorization of fragility issues for Android applications that were derived.

4.2.1 Implementation of test cases in different releases

In the following, the principal aspects of the implementation of the tests for individual versions of K-9 mail are described, along with the fragilities that occurred when they were executed on following releases. The changes that had to be made in order to adapt test cases to subsequent releases are highlighted. Since the AUT requires an authentication phase, tests were intended to run sequentially, so that just one authentication had to be performed (in TC1). The app was brought back to its original state after each full execution of the test suite.

A total number of 50 distinct test cases were developed, in addition, a few of them had to be modified to adapt to the different releases. Since earlier versions of the application did not offer all the functionalities that were listed after an examination of release v5.010, only the available test cases have been developed for them (for instance, only seven test cases could be applied to version v2.1012).

K-9 Mail v2.102

Seven tests out of the ten test cases of table 4.1 could be written for version v2.102 of K-9 Mail, according to the features it provided. The implementation of the scripted versions of test case TC1 (*Successful Authentication*) leveraged unique identifiers, for the detection of specific buttons and text boxes to interact with. By converse, TC2 (*Send a message*) and TC5 (*Add User Account*) needed the interaction with a menu that could be opened only by pressing the physical Menu button of the device. Additionally, in the menu, the individual text buttons had no unique identifier, and hence they had to be detected through their textual description. Text boxes for composing the e-mail message could e detected using IDs specified in the activity layout. Unique identifiers are missing also for the development of test cases TC6 (*Delete User Account*), TC7 (*Delete Account Data*) and TC8 (*Restore Account Data*).

K-9 Mail v2.995

Eight tests out of ten could be written for release v2.995 of K-9 Mail. Test cases from TC1 to TC4 were completely compatible with the ones written for release v2.102. The functionalities for the deletion of account data and for the deletion of a user account were moved to an advanced options menu, hence test cases TC5 and TC7 had to be rewritten. The release featured the possibility of restoring the data of an account, hence TC8 could be developed for it.

The textual description of the feature for adding user accounts (TC5) had a slightly different color and dimension with respect to the previous release considered. The test case, hence, had to be re-recorded with Sikuli, even though the Espresso, UI Automator and Selendroid counterpart had not shown any fragility.

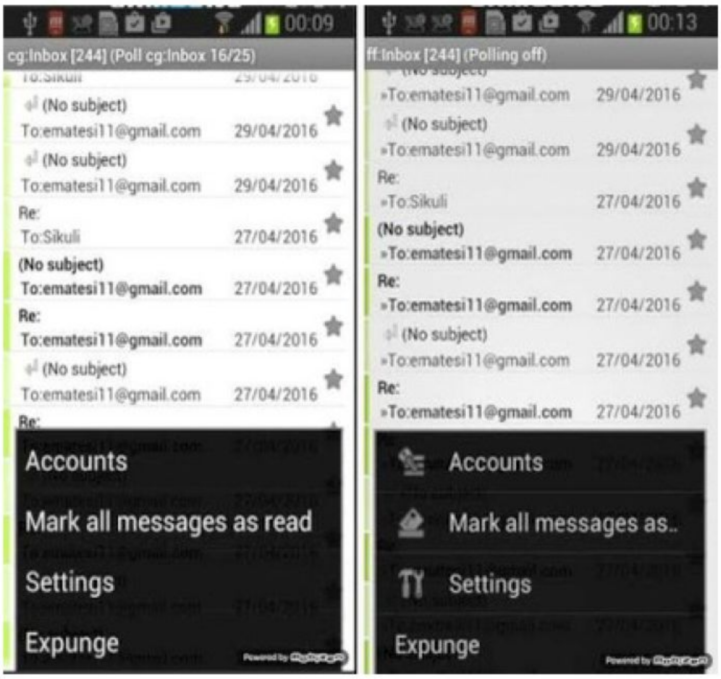

00:09
cg:Inbox [244] (Poll cg:Inbox 16/25)
(No subject)
To:ematesi11@gmail.com 29/04/2016
(No subject)
To:ematesi11@gmail.com 29/04/2016
Re:
To:Sikuli 27/04/2016
(No subject)
To:ematesi11@gmail.com 27/04/2016
Re:
To:ematesi11@gmail.com 27/04/2016
(No subject)
To:ematesi11@gmail.com 27/04/2016
Accounts
Mark all messages as read
Settings
Expunge
00:13
ff:Inbox [244] (Polling off)
»To:ematesi11@gmail.com 29/04/2016
(No subject)
»To:ematesi11@gmail.com 29/04/2016
Re:
»To:Sikuli 27/04/2016
(No subject)
»To:ematesi11@gmail.com 27/04/2016
Re:
»To:ematesi11@gmail.com 27/04/2016
(No subject)
»To:ematesi11@gmail.com 27/04/2016
Re:
»To:ematesi11@gmail.com 27/04/2016
Accounts
Mark all messages as..
Settings
Expunge

Fig. 4.2 User interface differences between release v2.995 and v3.309 of K-9 Mail.

K-9 Mail v3.309

Eight tests out of ten could be written for release v3.993 of K-9 Mail. All tests written for v2.995 were compatible with this version, except for the Sikuli version of test case TC5. Once again, a slight difference in the appearance of the widgets creates the need for the new capture of Sikuli test cases (see figure 4.2).

K-9 Mail v3.993

Eight tests out of ten could be written for release v3.993 of K-9 Mail. Test cases TC1 to TC6 developed for release v2.995 could be executed seamlessly on release v3.993, without any maintenance, since all widget identifiers and textual contents did not change between the two releases. The buttons to perform the operations of test cases TC7 and TC8 had their text content changed, so the test cases developed for release v2.995 were invalidated by the release transition. This version of the application still uses physical buttons of the Android devices, and many widgets of the activities traversed by the test cases are not described by unique identifiers.

Table 4.2 Test suite implementation on various versions of K-9 Mail, with Espresso, UIAutomator and Selendroid.

Test case	v2.102	v2.995	v3.309	v3.993	v4.804	v5.010
Authentication	n	o	o	o	o	o
Send a message	n	o	o	o	x	o
Reply to a message	n	o	o	o	x	o
Delete a message	n	o	o	o	o	o
Add user account	n	o	o	o	x	o
Delete user account	n	x	o	o	x	o
Delete account data	n	x	o	x	x	o
Restore account data	-	n	o	x	x	o
Export account settings	-	-	-	-	n	o
Import account settings	-	-	-	-	n	o

’-’ feature not supported, ’x’ test had to be modified, ’n’ new test written, ’o’ previous version of test still working

K-9 Mail v4.804

All the ten described test cases (including T9, *Export Account Settings*, and T10, *Import Account Settings*) could be developed for this release of the application. The app underwent a significant re-organization of the menus with respect to the previous major release. Hence, most of the tests (six out of the already developed eight) had to be re-written. The principal addition to the previous user interface was the addition of a toolbar in the bottom part of the screen, whose contained widgets were provided with individual unique IDs (thus making the writing of test cases easier).

K-9 Mail v5.010

All functional tests developed for release v4.804 were still functioning when applied on release v5.010, since all the graphics and the properties of the layouts were unchanged between the two releases.

Table 4.3 Tests compatible with previous versions, with Espresso, UIAutomator and Selendroid.

	v2.995	v3.309	v.3993	v.4804	v5.010
Number of unbroken tests	5/7	8/8	6/8	2/8	10/10
Percentage of unbroken tests	71%	100%	75%	25%	100%

Table 4.4 Causes of fragilities in broken test cases.

Cause	v2.995	v3.993	v4.804
Text change	0/2	2/2	3/6
Identifier change	0/2	0/2	3/6
Deletion or relocation	2/2	0/2	3/6
Physical buttons	0/2	0/2	3/6

4.2.2 Changes in Test Suite

A summary of the need for maintenance of scripted (Espresso, UI Automator, Selendroid) test cases in the transitions between the considered releases is shown in table 4.2. The same is done for Sikuli test cases in table 4.5.

Table 4.3 shows the percentage of tests that, for each release of the application, could be maintained as they were developed for the previously considered release. The same is done for Sikuli test cases in table 4.6.

As it was expected, in correspondence with tangible changes in the GUI (as it happens between the third and fourth major release), the majority of test cases had to be rewritten. On the other hand, in the transition towards the latest releases – which mainly corrected bugs without performing any intervention in the app appearance – no test cases were broken.

This first case study proved that, even for a very small sample test suite for a single application, the occurrence of fragility may require significant intervention (re-writing up to the 75% of scripted test cases of a test suite, and up to the entirety of a visual test suite) in existing test suites. The evolution of the K-9 Mail application was characterized by fragilities in both the definition of the user interface and its appearance. Table 4.4 summarizes the main causes of the fragilities found for the Espresso, UI Automator and Selendroid test suites, with individual test cases that

Table 4.5 Test suite implementation on various versions of K-9 Mail, with Sikuli.

Test case	v2.102	v2.995	v3.309	v3.993	v4.804	v5.010
Authentication	n	o	o	o	x	o
Send a message	n	o	o	o	x	o
Reply to a message	n	o	o	o	x	o
Delete a message	n	o	o	o	x	o
Add user account	n	x	x	o	x	o
Delete user account	n	o	o	o	x	o
Delete account data	n	o	o	x	x	o
Restore account data	-	n	o	x	x	o
Export account settings	-	-	-	-	n	o
Import account settings	-	-	-	-	n	o

'-' feature not supported, 'x' test had to be modified, 'n' new test written, 'o' previous version of test still working

Table 4.6 Tests compatible with previous versions, with Sikuli.

	v2.995	v3.309	v.3993	v.4804	v5.010
Number of unbroken tests	6/7	7/8	6/8	0/8	10/10
Percentage of unbroken tests	85%	87%	75%	0%	100%

could be weakened by multiple different causes. The encountered fragilities were due to the following reasons:

- **Changed identifiers**: modification of the identifiers that were used by test cases for the identification of interface widgets.
- **Changed text**: modification of the textual content, when such text is used to identify widgets in test cases (in absence of unique identifiers).
- **Deletion or relocation**: changes in the arrangements of the widgets in the layouts of the app activities.
- **Usage of physical buttons**: older Android apps made use of physical buttons, deprecated since Android 4.0 and replaced with the use of Action Bars. New releases of applications after that version of the o.s. needed to change the interaction paradigm used accordingly, thus invalidating all test cases developed beforehand with the use of physical buttons.

Chapter 5

Study 1: Survey with mobile developers from the industry

In order to perform a first investigation about the adoption of automated GUI testing techniques among IT professionals, and to have insights about the testing practices performed by IT professionals having mobile and web apps in their portfolio, a set of semi-structured interviews were conducted with representatives of seven medium- and large-sized companies of the Turin area.

This study allowed answering to the first research question of the study: ***RQ1*** *- What is the perception of GUI testing for Android apps among practitioners from the industry?*

5.1 Study design

The interviewed representatives are described in table 5.1. The companies were selected based on the location of the interviewers at the moment of the conduction of the experiment. All the interviewed representatives were involved in the development of mobile or web applications. The interviews have been conducted between June and December 2017.

RQ1 can also be split into three sub-questions, related each to a different topic covered by the whole survey subministrated to the developers. First, to understand the testing practices adopted by developers, insights were gathered about the most

Table 5.1 Interviewed developers from the industry

Interview ID	No. of representatives	Company and project
A	1	Distributor of testing tools for various typologies of applicatives.
B	2	Test factory for third party applications and test consulting.
C	1	Insurance company: web and mobile apps for insurers and customers.
D	2	Insurance company: platform for insurance management.
E	1	Test factory for third party applications and test consulting.
F	1	Full-stack development of mobile applications for multiple platforms.
G	2	Test factory for consulting of test and test management for banking applications.

common tools and techniques they adopted, and the typologies and levels of testing performed. Hence, RQ1.1 could be formulated as:

> **RQ1.1** : Are mobile applications tested by the interviewed sample of industry practitioners? How? To what extent?

As discussed in the introduction section, mobile apps showcase a set of peculiar aspects when it comes to testing them, that must be taken into account by testers/developers wanting to design and execute automated test cases. The second section of the survey aimed at understanding which aspects of mobile applications were considered crucial by professionals adopting automated testing, and at the same time which peculiarities were seen as a deterrent from the adoption of automated testing techniques. Hence, RQ1.2 could be formulated as:

> **RQ1.2** : What are the most peculiar properties to test in mobile applications according to the interviewed sample of industry practitioners? What aspects of mobile apps discourage them from adopting automated testing?

Finally, questions were asked to characterize the interest in emerging testing techniques, and to summarize the principal difficulties felt by developers, along with the amount of perceived fragility of developed test cases and the needed human labour to maintain existing test suites. Hints were gathered for possible research directions to aid developers and testers from the industry in overcoming those difficulties. Hence, RQ1.3 could be formulated as:

Table 5.2 Structure of the survey to developers from the industry

RQ	Number	Question
RQ1.1	1	Do you use to test your application code?
	2	If you test your apps, do you take advantage of manual or automated testing techniques?
	3	If you use automated testing techniques, which kind of technique do you prefer? Why?
	4	What are the typologies of testing you perform?
RQ1.2	5	In your experience, what are the differences that you have found between testing mobile applications and web/desktop applications?
	6	Which aspects of mobile applications encourage you to adopt specific testing techniques?
	7	Which aspects of mobile applications discourage you from testing them?
RQ1.3	8	What are the main difficulties you encounter in testing mobile applications?
	9	Have you ever needed additional effort to keep test suites up to date with the evolution of the application GUI?
	10	What is your attitude towards new generations of testing like model-based testing and visual recognition?
	11	Which directions should take the academic research on mobile testing?

> **RQ1.3** : What are the main challenges felt by developers from the industry performing automated testing, and the directions research should take according to them?

To gather answers to the survey, a set of questions arranged in three different groups (each pertaining to one of the three subquestions of RQ1) were subministrated to the interviewed developers/testers. The interview sessions lasted around 30 minutes each, and a transcript was obtained at the end of every session based on the minutes taken during the interview. The findings were cataloged and organized, to answer the individual questions of the survey. All the questions of the survey, detailed in table 5.2, were open. In each interview, the motivation of the study was clearly stated at the beginning, and the definition of testing fragility for mobile applications was provided, according to the formulation defined in the introduction section. Hypotheses were not stated, neither implicitly nor explicitly, at the beginning or during the interviews, in order to avoid any bias.

5.1.1 Threats to Validity

Threats to external validity

The findings are based on seven interviews, conducted only with developers of industry, working on a limited set of application domains (principally, banking, insurance and leisure) and developing mainly Android and web applications. It is not sure whether the findings can be generalized to all mobile developers, e.g. open source developers, iOS developers, and so on. The difficulties and needs perceived by the interviewed developers may be strongly subjective and not representative even of developers working on the same platform and in the same domain.

Threats to construct validity

For what concerns the fragility definition, GUI fragile test classes are linked to any modification in the interface requiring adaptations in the test suite. The definition of fragility may vary from other ones used in literature, and its relevance among the issues faced by developers may hence be different.

5.2 Results

This section showcases the results gathered from the interviews to developers, subdivided according to the subsresearch question that they answer.

5.2.1 Adoption of mobile testing techniques and tools

Adopted Testing Techniques

The considered respondents were active in both web and mobile application development. All of them highlighted a priority put in testing web applications. Only three of them performed structured and automated testing on their applications. The formalized and structured testing procedures for their web applications were only loosely applied to the mobile apps of the respondents' portfolio, even when such apps were the direct counterparts of the tested web apps.

Capture and Replay testing was adopted by more than half of the respondents. Representatives of company C adopted Capture and Replay in a data-driven fashion (with parameters about the personal information taken from .csv files to populate test cases) to perform regression testing.

Scripted testing was performed by three respondents; company B used scripted tests on a limited number of different devices, and company E stressed the test of device diversity with tests run on many multiple devices, leveraging dedicated services in the cloud.

With the exception of respondent F, mobile testing was typically performed on actual devices and not on emulators. Respondent F was also the only one fully leveraging techniques of random/monkey testing.

Company B adopted techniques of Mobile APM (i.e., Application Performance Management), capable of evaluating the compliance to non-functional requirements on the application after its release, monitoring the usage of the application running on the users' devices.

Company G adopted techniques for static source code analysis and reporting instruments, while the formalized testing procedures only involved manual tests recorded through Capture and Replay techniques.

Typologies of testing

Among the respondents, mobile testing is executed principally at system and acceptance level, either using manual or automated testing techniques. A certain amount of unit and integration testing is performed with automated techniques by all respondents, except B and E that, being test factories for third party projects, leave low-level test practices to the developers of the software. The rapid development life cycle of mobile applications, and the frequent addition of new functionalities, is seen as a deterrent for the adoption of structured regression testing. For what concerns non-functional testing, the main focus of the respondents is on usability and performance.

Table 5.3 Survey with developers from industry: tools used by the respondents

Tool	Type	R.
MicroFocus UFT / QTP	Regression and functional testing	4
Selenium	Scripted web-based app testing	4
Appium	Multi-platform mobile app automated testing	3
PerfectoMobile	Scripted cloud-based app testing	3
JMeter	Load and performance testing	3
Silk Mobile	Capture and Replay testing	2
JUnit	Java unit testing	2
AppliTools	AI-based Visual Testing	1
Monkey	Random testing	1
Qualitia	Scripted testing and GUI modeling	1
TestComplete	Capture and Replay testing	1

Adopted tools

For what concerns the most used testing tools for mobile applications, Selenium is the most used for test scripts of web-based and hybrid mobile applications; Selenium IDE is also used for the creation of Capture and Replay scripts, with a component that allows the tester to record the test case and generate the code automatically.

Some commercial tools were cited by the interviewed developers: four respondents cited HP UFT, used for web-based applications; two cited Silk Mobile, that can be used to perform tests on native applications; three cited PerfectoMobile, which can be used to perform cloud-based functional tests on real devices, using scripts created by Capture and Replay.

Other test frameworks, like the official ones provided by Android with the Android Instrumentation Framework, were cited by some respondents. Only one respondent used a tool leveraging the new paradigm of AI-based visual testing tool, namely AppliTools. Table 5.3 enumerates the testing tools adopted by the interviewed developers.

Answer to RQ1.1: All the respondents to the survey performed manual testing on their mobile applications. Among the automated testing tools used, the interviewed developers mostly relied on Capture & Replay test, with a rare adoption of scripted testing tools.

5.2.2 Peculiarities of mobile application testing

Several differences have been highlighted by the interviewed developers in the procedure of testing mobile applications, with respect to testing traditional desktop and web applications.

Apps may be subdivided under three different categories: Native, if they are engineered for a particular os/platform; Web Apps, if they are typical web applications that are customized for being loaded by browsers on mobile devices; Hybrid, if they have a native part that loads dynamically web pages with contents and functionalities. As respondent B pointed out, the testing procedures for the three categories of apps vary significantly in terms of adopted instruments, and test case definition.

For mobile applications, the complexity of the testing procedure has increased dramatically principally because of device diversity. Mobile applications must ensure compatibility with a set of different device types, screen sizes, pixel densities, resolutions, screen orientations. If the applications are multi-platform, they must also be tested on the principal operating systems. Finally, apps must comply with the design characteristics and functionalities of new releases of the operating system they work on -that may be published rather rapidly while guaranteeing retrocompatibility with past versions. As respondent B highlighted, *"device diversity is a relevant enabler for test automation, because it is impossible to execute manual tests on many devices; to select the devices on which to run test cases, we pick the devices that are sold the most in the market, and the ones that are used the most by the final users, also taking into account geographic statistics."*; respondent F also pointed out that *"device diversity and form factor are the fundamental variables to take into account, and influences mobile app testing much more than web application testing, for which it is sufficient to test on the principal browsers."*

Different non-functional properties are peculiar of mobile applications and require specific testing procedures. The topic of Green Energy, as pointed out by respondent C, is a very perceived issue for the test of mobile applications: ensuring a battery consumption that is adequate to the typology of the app is fundamental for the users' satisfaction. In general, the usage of the device resources (that in some cases can be very limited) is a stringent non-functional requirement for mobile application testing: the same reasoning made for energy testing can be applied also to the usage of network and data connection, the usage of CPU and memory, and the possibility of overheating. Being mobile applications strongly GUI-based, the rendition of the graphics on the device screens is a crucial element of acceptance testing. However, as pointed out by respondent F, the usage of preliminarily defined and tested mock-ups, already working before the functionalities are implemented, may relieve the developers from testing the final appearance of the app once it is deployed.

Answer to RQ1.2: The respondents to the interviews underlined several aspects that are peculiar to mobile vs. desktop or web development. The two main characteristics that required specific forms of testing – according to the interviewed developers – were resource and battery saving and adaptation to different devices and display sizes.

5.2.3 Challenges and desires of mobile app testers

Factors limiting mobile application testing

Several difficulties have been identified by the respondents as problems hampering the practice of mobile application testing. For commercial applications, the companies may want a fixed and strict time-to-market, and compromises are necessary to find an optimal balance between cost and quality. Respondent B considered that *"clients want the application to be published anyhow, and often the quality aspect is sacrificed, even offering limited or malfunctioning features. The quality of the app then grows with time, in parallel with the number of the users that use them, and thanks to their feedback."* Respondent E added that *"rarely projects have a test strategy which is carefully defined, validated and approved, with a reasonable time to perform it; testing, also, typically suffers from delays in the previous phases of the development, even though it is supposed to guarantee in any case the same quality."*

Many of the interviewed developers underlined that the culture of testing mobile applications is still limited with respect to other kinds of applications. Respondent A highlighted that, in large companies producing software, a huge difficulty in the adoption of automated testing is due to the fact that the testing department is managed by members of the business department, who have a different perception of testing with respect to people of ICT extraction. Therefore, manual testing is often preferred both for web and mobile applications and, in general, it is difficult to go particularly far beyond C&R techniques. Still according to respondent A, *"the mobile device, from the business point of view, is mainly seen as a proxy to access services that are located on the web."* Respondent D confirmed that the focus is often kept on the core of the applications, without particular interest from the business department towards application testing. Respondent B pointed out that *"mobile application testing is still not treated with sufficient maturity, and clients are just beginning to see the return of investment that test automation can guarantee; in general, only companies creating apps that manage sensitive and economically critical data tend to adopt automated testing for them"*.

Respondents B and F also highlighted that a relevant limitation for mobile application testing, especially for small testing teams possibly not used to scripting, is a scarce dissemination and documentation of automated testing tools. The problem appears to be amplified for open-source tools.

Maintenance and fragility

All respondents, except respondent F, experienced issues with fragility of test cases, that was seen as the main cause for maintenance over existing test suites.

Respondent A, who used C&R techniques, had to completely re-register test cases when the interface was modified between a version of the application and the subsequent one. Respondent C underlined scarce adaptability of Selenium, and found that identifying the individual elements of mobile app was easier and less prone to fragility using commercial tools; this developer estimated the work in modifying existing scripts as about 30% of the total testing work, and identified this reason as one of the most relevant in deciding to not develop huge test suites (no more than 250 scripts).

Respondent B experienced fragilities in the regression testing of its applications, with an estimate that 20-30% of his total testing effort belongs to maintaining test scripts. This respondent identified changes in the ID and in the text of elements of the interfaces as the most relevant causes of fragilities of test case: the problem is particularly felt when the names assigned to the elements of the interface are not semantic, and are assigned by automated tools, thus possibly changing between every pair of builds. This respondent added that *"changes in the user interface are* *an aspect that significantly hamper the adoption of test automation, and the issue is amplified when it is not the same company performing developing and testing”*.

Respondent D defined fragility as a *“problem that is perceived and that has to be fought on a daily basis: test suites must be maintained daily”*. For projects, fragility is identified as a critical problem, especially for possible shortages of time: it is often not possible to do complete maintenance of test cases that fail even though they should not. This developer identified the effort for the maintenance of test suites as two days every twenty days of testing.

The estimate of the cost of fragility was even higher for respondent E, which identified the cost of maintenance of already present test scripts as 60% of the total maintenance cost. The developer pointed out that *“the impact of fragility is higher for mobile applications, because mobile interfaces and functionalities evolve more rapidly than traditional applications. The investment in the maintenance of test cases is mandatory and grows with time, even though the modifications in the user interfaces are limited.”*

Requests to academia.

All the respondents considered that a solution, possibly automated, to the problem of fragilities of automated tests, especially the ones related to GUI, should come in handy to companies performing testing in both web and mobile applications.

Respondent C expressed a desire for a more direct and extractable link between incidents in tests, or in running applications, to the defects running in the source/system. Respondent B highlighted the problem of test prioritization. They already use static analysis of source code, to extract info about code and classes changed. Knowing what is the code coverage, when the new version of the app is available it is possible to know what changed, and it should be desirable to launch only tests that maximize the coverage on modified source. This can be useful if resources and time for testing are limited, because it theoretically maximizes the amount of useful testing performed. Obviously, the model should also take into account a sort of higher-level functional prioritization, in addition to coverage prioritization. Still about coverage, respondent D identified the need for a finer way to calculate the code coverage for web/mobile test suites, with fine-grained metrics able to represent the actual economic value of the testing procedure.

Automation in the development of test cases was a need expressed by respondent F, who pointed out that *"an algorithmic creation of test cases during the definition of the application logic (e.g. the definitions of API and accesses to databases) to an even limited coverage of functionalities to be tested, would be very happily welcomed by developers, who still see writing test cases as an overhead."*

Model-based testing is not felt as a primary need, or something that can be useful at least in the near future. Only respondent A showed enthusiasm towards the possibility of adopting model-based testing, expressing the need for a *"trustable mobile ripper, capable of semantically interpret everything that happens during the exploration of an interface, proposing test cases to the business department."* Respondent B, which represents a consulting company performing outsourced test, reported that they actually sell their experience to their clients, so they're not proposing something that is not yet fully understood and managed. They highlighted that it is indeed an interesting topic for academia, but at a first approach the model-based testing techniques that have emerged from research are too complex and require too much knowledge to be actually adopted by companies. Respondent E was the only one actually performing a sort of modeling of apps for the definition of test cases, and pointed out that an extended modularity of tests should be encouraged by research.

Answer to RQ1.3: All the interviewed mobile developers found that the maintenance of test cases for mobile applications is one of the most relevant challenges that prevent the adoption of automated testing techniques. Almost all respondents expressed the desire for better ways to manage fragilities and to reduce the effort for making the test suites evolve. Little enthusiasm was instead shown towards new paradigms of testing explored by literature, like model-based testing and visual recognition testing.

Chapter 6

Study 2: Controlled experiment with Graduate Students

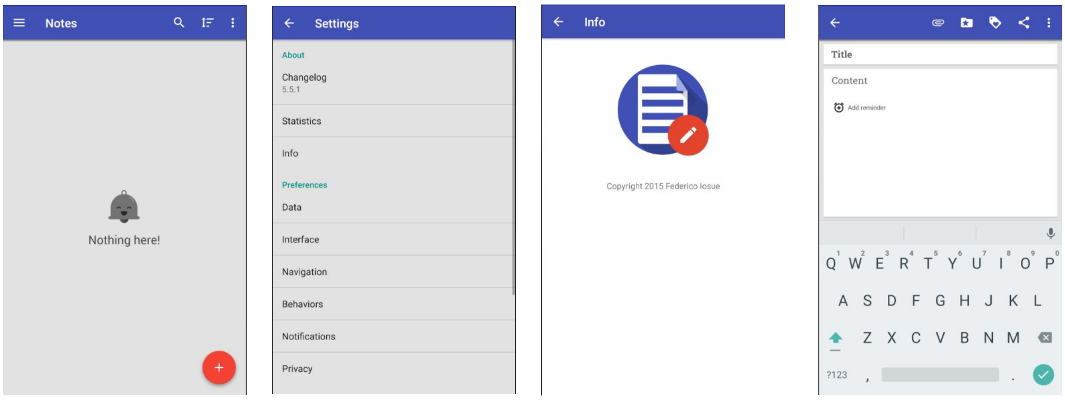
To assess the possible difficulties encountered by practitioners at the first experience with Android Testing Tools, and at the same time to estimate the usability of two different approaches to automated GUI testing for Android applications, a controlled experiment with graduate students has been conducted. The experiment allowed to answer the research question ***RQ2*** *- How usable are GUI testing tools and what is the productivity of graduate students using them?*

The design and results of this study have been accepted for presentation at the EASE 2019 conference.

6.1 Study design

Two different testing tools were chosen for the experiment: Espresso, as a representative of 2nd generation (Layout-based) testing tools, and EyeStudio, as a representative of 3rd generation (Visual) testing tools.

The human subjects of the experiment were a sample of undergraduate computer students, all enrolled in the Computer Science program, at the Polytechnic University of Turin, and attending the Software Engineering course. The participation in the lab sessions devoted to the empirical experiment was fostered by assuring additional



(a) Main Activity (b) Settings Act. (c) Info Activity (d) Note Activity

Fig. 6.1 Screens and Activities of Omni Notes app

points in the final grade for the course. Automated GUI testing was introduced in two lectures that were part of the course.

The selected software source of the experiment was release 5.5.1 (released May 11, 2018) of Omni-Notes, an open-source and well-reviewed (100+ thousand downloads, and 4.4 star rating on the PlayStore) Android app for writing and storing small text notes. Screen captures of the app activities are shown in figure 6.7. The source code of the application is available on GitHub. The experiment was based on the English-localized version of the app. The device used for the experiment was an emulated Android Virtual Device (AVD) on which to launch (and capture, in the case of Visual test cases) a Nexus 5X mounting API 25 (Android 7.11), with visible device frame and activated keyboard inputs.

The analysis of the experiment results allowed answering three different subquestions of RQ2, listed in the following:

- **RQ2.1 - Productivity:** What is the productivity of inexperienced developers when approaching to Layout-based and Visual GUI testing tools?
- **RQ2.2 - Quality:** What is the percentage of working test scripts produced by undergraduate programmers using Layout-based and Visual GUI testing tools?
- **RQ2.3 - Obstacles:** What are the perceived difficulties in approaching visual and layout-based GUI testing techniques?

The metric used for the productivity of the participants was the raw count of delivered test cases after each session. To measure the quality of the test suites, instead, the number of working test scripts on the total written by each student was taken into account. All the delivered test cases were re-executed on the original version of Omni-Notes, in the research environment. Non-working test cases were found analyzing the execution reports generated by EyeStudio, or the presence of exceptions in the Java logger of the JUnit engine.

The final subquestion was answered after a manual analysis of the answers given by the participants to a survey, and a classification of the errors found inside delivered test scripts.

6.1.1 Experimental procedure

The productivity of the sample of students was evaluated by asking them to develop the same test suite for the selected application, with both the considered testing tools. The individual scripts of the required test suites covered a set of usage scenarios that were deemed of principal interest for the use of the Omni-Notes application. Such usage scenarios are described in table 6.1.

The experiment was subdivided into two different lab sessions. In the first one, the participating students were asked to produce scripted test cases, using the Espresso testing tool inside the Android Studio development environment. The students were allowed to use any possible way to identify the widgets of the application inside the test cases, including recording the interactions with the Espresso Test Recorder tool, manually checking layout files from the *res* folder of the apps, leveraging the Layout Inspector, Hierarchy Viewer and UI Automator Viewer built-in tools for finding properties of the widgets. In the second session, the students were asked to produce Visual test cases, with the use of the EyeAutomate library and the EyeStudio IDE. Both the sessions lasted 1.5 hours long.

At the end of the second session, students were asked to fill a structured survey, to gather their impressions about the testing practices they performed, and their considerations about the tools used and their usability. Demographic information was also collected, to perform statistical evaluations of the results obtained by the students in developing the test suites. The structure of the survey is reported in table

Table 6.1 Description of use cases for the empirical experiment with graduate students

Name	Description
Open info screen	The user clicks on the menu button (upper-left corner in any activity), then clicks on the Settings button to go to the Settings screen, then on the Info menu voice to move to the AboutActivity. In the AboutActivity, the icon of the application and the copyright notice have to be properly shown.
Add note	The user clicks on the plus button (lower-right corner in the MainActivity, the principal activity that is opened at start-up at every launch of the application), then selects “Text Note” and inputs title and content for a test note. The user then goes back, clicking on the back button. The inserted note has to be shown in the MainActivity.
Search a note	The user first creates two or more notes, with different titles and contents. Then, he clicks on the search button and inputs in the search bar a keyword that is contained in the title or content of one of the notes. The application must show as search result the note containing the keyword, while the other one(s) should be hidden.
Check available languages	The user clicks on the menu button (upper-left corner in MainActivity), then clicks on the Settings button to navigate to the SettingsActivity, then clicks on the Language menu voice to list all the featured languages by the activity. The list must feature the English, Italian and French language.
Delete a note	The user creates a note with test text for title and content, then goes back to the MainActivity. The user long-clicks the newly created note, then clicks on the Overflow Menu Button and selects Delete. The note must not be shown anymore in the list of notes of the MainActivity.
Restore a note	The user creates and deletes a note as in the previous use case, then – through the drawer menu – navigates to the trashed notes, long clicks on the deleted notes and restores it through a click on the Restore button. Then, the user moves back to the MainActivity. The application must correctly show the deleted and then restored note.
Add category	The user creates a note, and – in addition to the input of test text – selects "Add Category" for the new note, selecting for it a name and a color. The application must show in the drawer menu the name and the color of the newly created category.

Table 6.2 Questions of the survey for undergraduate students

Question number	Question	Question type
1.1	Student ID	Open
1.2	Age range	Multiple choice
1.3	Have you ever worked as a Java professional?	Closed (Yes - No)
1.4	How many years of experience do you have in Java programming?	Open
1.5	How many years of experience you have in Android programming?	Open
1.6	Do you have previous experience in Java application testing?	Closed (Yes - No)
1.6b	Which tools have you used for Java testing?	Open
1.7	Do you have previous experience in Android application testing?	Closed (Yes - No)
1.7b	Which tools have you used for testing Android applications?	Open
2.1	The user scenario descriptions were clear to me	Likert
2.2	Implementing the test suite with EyeAutomate was easy and intuitive	Likert
2.3	The EyeStudio IDE was helpful in the creation of test scripts	Likert
2.4	It was easy to identify elements with the visual recognition technique	Likert
2.5	What were the principal issues that you found in identifying elements of the screen using the Visual GUI testing approach?	Open
2.6	The implementation of the test suite with the Espresso framework was easy and intuitive	Likert
2.7	The Android Studio IDE was helpful in the implementation of the test suite	Likert
2.8	It was easy to identify elements of the screen using the layout-based technique	Likert
2.9	What were the principal issues that you encountered in identifying elements of the screen using Espresso?	Open
2.10	Which tool would you choose if you had to perform visual testing again? And why?	Multiple Choice + Comment

6.2, with the questions subdivided into two groups: demographic information (group 1) and opinions about the tried testing tools and techniques (group 2).

To measure the quality of the test suites that were submitted by the participants to the experiments, all the tests were executed on the app, on the same configuration used by the students in the lab sessions. Layout-based test scripts were considered incorrect when they triggered executions during their executions, or when they did not feature any assertion to check the actual state of the application. Visual test scripts, as well, were considered incorrect when they were not able to reach the end of the related usage scenario, and when no visual check on the GUI was performed. The quality of the provided test suites was computed as the fraction of the correct test cases over the total number of provided test cases. Statistical tests were then applied to the sets of measures about productivity and quality, to understand whether there were statistically significant differences in the obtained data regarding the two considered tools.

6.1.2 Threats to Validity

Threats to Internal Validity

An internal validity threat may be due to learning effects, during the first development of the test suite, with the students that could become familiar with the proposed usage scenarios and app, and hence produce test suites of better quality in the second session. It was tried to limit learning effects giving a basic knowledge of the app and of both tools before the first session of the experiment. It is also assumed that the low complexity of the proposed usage scenarios could not constitute a relevant obstacle for the first of the two lab sessions. The close values for the answers to questions 2.4 and 2.8 show however that in both sessions the participants encountered very similar difficulty level in identifying the GUI components to be used in the test scripts. This supports our belief that the order of treatments had no noticeable effect.

Threats to Construct validity

The principal threat is that the productivity and quality metrics that were defined are not the most suitable for measuring the learnability and ease of use of testing techniques and tools. Typical measures for productivity like the delivered LOCs, however, were not applicable for a comparison between the two tools, since the delivered test scripts were written in different syntax (plain text for visual test scripts, Java code for Layout-based test scripts).

Researcher bias is another possible threat to the validity of this study since all test suites delivered by the participants were manually examined by the involved researchers. However, the researchers were not involved in the development of any of the two approaches or tools and had no reason to favour any particular approach neither are inclined to demonstrate any specific result.

Threats to Conclusion Validity

Non-parametric tests were adopted to account for the non-continuous and non-normal distribution of the variables. Conclusions were drawn using the customary 5% type I error threshold.

Threats to External Validity

A real-world open-source application was considered, thus selecting a realistic context for the application of GUI testing techniques to Android apps.

The results of this work are applicable only to the considered testing tools – i.e. EyeAutomate and Espresso – and it is hence unsure if they can be generalized to other testing tools belonging to the same generations of GUI testing, and generating test scripts using the same approaches. The two tools, however, are quite widespread in literature and have several similarities to other common testing tools, like Selenium or UI Automator for the Layout-based generation, and Sikuli for the Visual generation.

Another threat related to the generalizability of this study is that the results are based on 78 participants which had little to no experience in Android development and testing. The sample might not be representative of all new users of Android testing techniques or newly hired practitioners. Results, in general, are not generalizable to experienced Android App developers and testers.

Table 6.3 Experiment with graduate students: delivered and working test cases

	Delivered Scripts		Working scripts		Quality
Technique	Mean	Median	Mean	Median	Average
Visual	5.38	7	4.2	4	0.76
Layout-based	5.51	7	3.51	3	0.63

6.2 Results

In this section, the results of the experiment with graduate students are reported. The results are based on the analysis of the submissions performed by 78 students, the ones who completely performed all the three parts of the experiment (namely, the development of the two test suites and the survey).

6.2.1 Demographic characteristics of the sample

Information about the considered sample of students was collected from the answers they gave to the demographic questions of the survey. The most frequent age range for the student was between 22 and 26 years old (89.7% of the respondents). A large percentage (89.7%) of the participants did not have any previous professional experience in Java development at the time of the experiment, with few years of experience in Java programming (average 2.05 years, with median equal to 2). The experience gathered by the students in Android programming before the experiment was little: only 41% had developed for such platform beforehand, with an average number of years of experience equal to 0.6.

For what concerns the experience in testing practices, only 21.5% of the sample had experience with Java testing of any kind (principally with JUnit), and only 3.8% of respondents had used testing tools designed specifically for testing Android applications (namely, Espresso, UIAutomator and MonkeyRunner).

6.2.2 Productivity and Quality of developed test suites

To measure the productivity of the participants to the experiment the count of the number of test scripts that they delivered at the end of each session was taken into account. The rationale between the link of this measured productivity and the

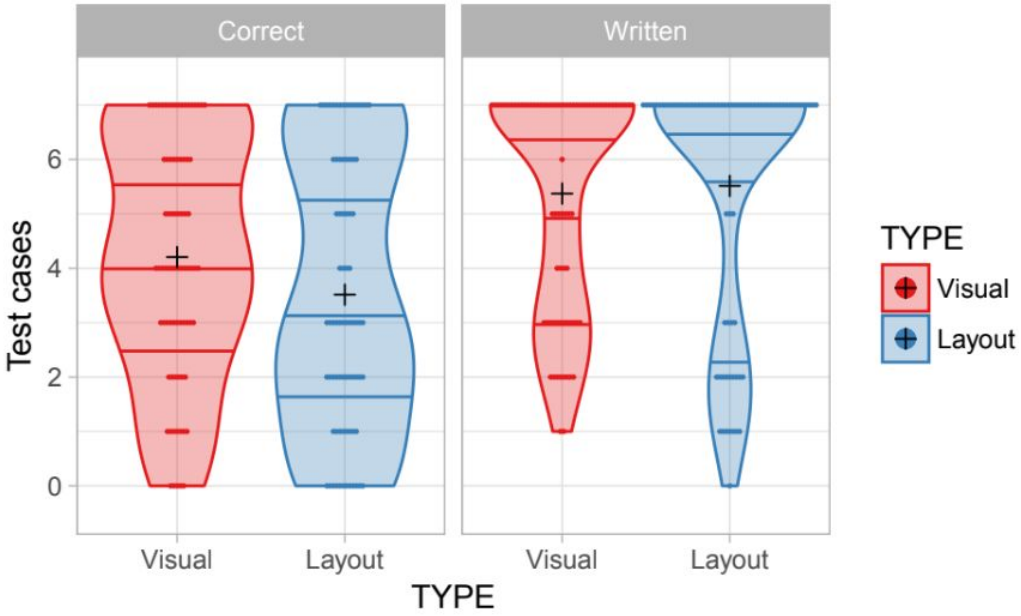


Fig. 6.2 Experiment with graduate students: Violin plot of delivered and working test cases

Table 6.4 Experiment with graduate students: null hypotheses about Productivity and Quality

Name	Description	p-value	Result
H_{p0}	There is no significant difference in the measured productivity between the test suites developed with EyeAutomate and the test suites developed with Espresso	0.6252	Accept
H_{q0}	There is no significant difference in the measured quality between the test suites developed with EyeAutomate and the test suites developed with Espresso	0.02509	Reject

learnability and usability of testing tools lies in the fact that an easier to understand testing tool should lead practitioners to develop more test scripts in the same time.

Table 6.3 shows the average and the median number of test cases that were delivered by the respondents, using the Visual or Layout-based testing tools. Violin plots in 6.2 show the distribution of the number of delivered and working test cases submitted by the respondents for both techniques. Those results show that, although a relevant portion of all the respondents tried to build the entirety of the test suite (being the median value of delivered test scripts equal to 7 for both the techniques) the number of working test cases was way lower than the total number of delivered ones.

Table 6.5 Experiment with graduate students: statistics about recorded and not recorded layout-based test suites

	Delivered Scripts		Working Scripts		Quality
	Average	Median	Average	Median	Average
Recorded	6.69	7	3.69	3	0.57
Not Recorded	4.13	4	3.30	3	0.71

Wilcoxon paired signed rank tests were applied to the pairs of distributions about productivity and quality, to verify whether the difference between the measured metrics for Layout-based and Visual testing tools was statistically significant or not. The null hypotheses in table 6.4 were validated through the computation of the p-value of the pairs of distribution.

Being the p-value for H_{p0} equal to 0.6252, the hypothesis cannot be rejected, and hence it can be stated that there is no statistically significant difference in terms of the number of tests written with the use of a Layout-based testing tool, or a Visual testing tool.

Answer to RQ2.1: No statistically significant difference has been found between the respective productivity obtained using EyeAutomate or Espresso. It can be deduced that the learnability of the two tools is similar, for non-professional developers approaching them.

Being the p-value for H_{q0} equal to 0.02509, the hypothesis can be rejected, then it can be stated that there is a statistically significant difference in the number of working test scripts written with the use of a Layout-based testing tool, or a Visual testing tool. However, such difference can be considered small.

Pertaining to layout-based test cases, an additional inspection has been performed, to subdivide the delivered test suites between the ones that were written down manually by the participants, and those that were obtained through the use of the Capture & Replay tool Espresso Test Recorder. Test scripts were manually inspected to find patterns that are typical of the automated generation of a test script through such tool. 42 of the 78 participants made use of the tool to create their test suites. The average and median values for delivered and working test cases were computed also on the subsets of test scripts that were written down manually and on those that were obtained automatically; they are shown in table 6.5. It can be seen that when the Espresso Test Recorder is used, the average number of delivered test cases is higher than the one for Layout-based test scripts delivered without using it, and even than the one for Visual test scripts; this can be justified taking into account the slower process of identifying manually the properties with which to identify the widgets of the user interface, or to take one screen capture at once to compose the test script, instead of using a capture and replay technique.

Even though the average values of delivered test cases are very distant for recorded and not recorded test suites, the average values of working test cases are very close (with equal median values). On average, 70.6% of the test cases created without the use of the Recorder were working, while only the 56.7% of the recorded test cases were working. This can be seen as evidence of lower dependability of test scripts obtained using the Recorder tool with respect to manually written test cases.

Answer to RQ2.2: A statistically significant difference has been found between the respective quality obtained using EyeAutomate or Espresso. In particular, test suites developed with EyeAutomate had a higher quality than the ones that the participants developed with Espresso.

6.2.3 Errors performed in test scripts

Regarding the development of Visual test scripts, the most common errors performed by the participants were related to the capture of images containing variable elements (e.g., another note created beforehand, or the time in the upper-right corner of the Android device) that made the tests completely not reproducible. Some test scripts failed due to missing explicit sleep instructions, that come in handy when the operations performed by the app take more time than a few seconds. Some respondents experienced issues in choosing the proper command from the EyeAutomate command library.

However, many test scripts (or entire suites) failed because some inputs to the GUI were hardly reproducible, using the EyeAutomate tool, even when the screen capture was taken correctly. This issue could be due to an intrinsic difficulty of the image recognition library itself in identifying images with specific characteristics in terms of size and content. For instance, the menu button in the upper-left corner of

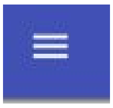
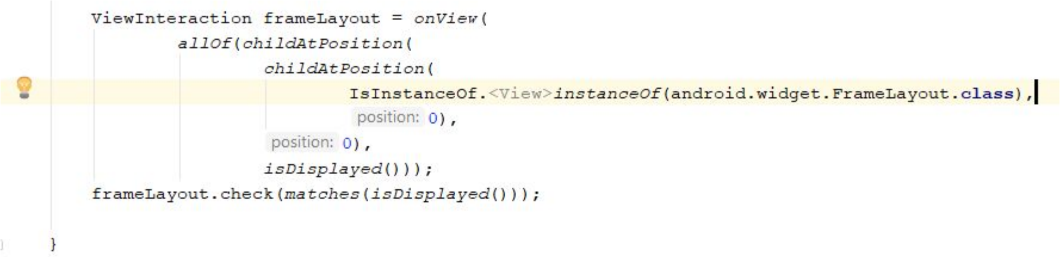


Fig. 6.3 Omni-notes menu button



```
    ViewInteraction frameLayout = onView(
            allOf(childAtPosition(
                    childAtPosition(
                            IsInstanceOf.<View>instanceOf(android.widget.FrameLayout.class),
                            position: 0),
                    position: 0),
                    isDisplayed()));
    frameLayout.check(matches(isDisplayed()));

}
```

Fig. 6.4 Non-working assertion generated by the Espresso Test Recorder

any activity (image 6.3) was almost never recognized by the EyeAutomate engine. A possible solution to this issue was to capture a larger portion of the screen (not limited to the square menu button) and then select the exact position where to click inside the screen capture.

Regarding the development of Layout-based test scripts, several tests failed due to the intrinsic limitations of the Espresso Test Recorder tool itself, which was used by the majority of respondents, without performing any editing on the scripts after their generations. At the time of the conduction of the experiment, the Espresso Test Recorder was still incapable of detecting long clicks on widgets (hence, all the test scenarios involving such operation were not reproducible) and several generated assertions were based on properties of the widgets that were not unique, hence triggering an exception when tests were executed. An example of such failing assertions is reported in figure 6.4.

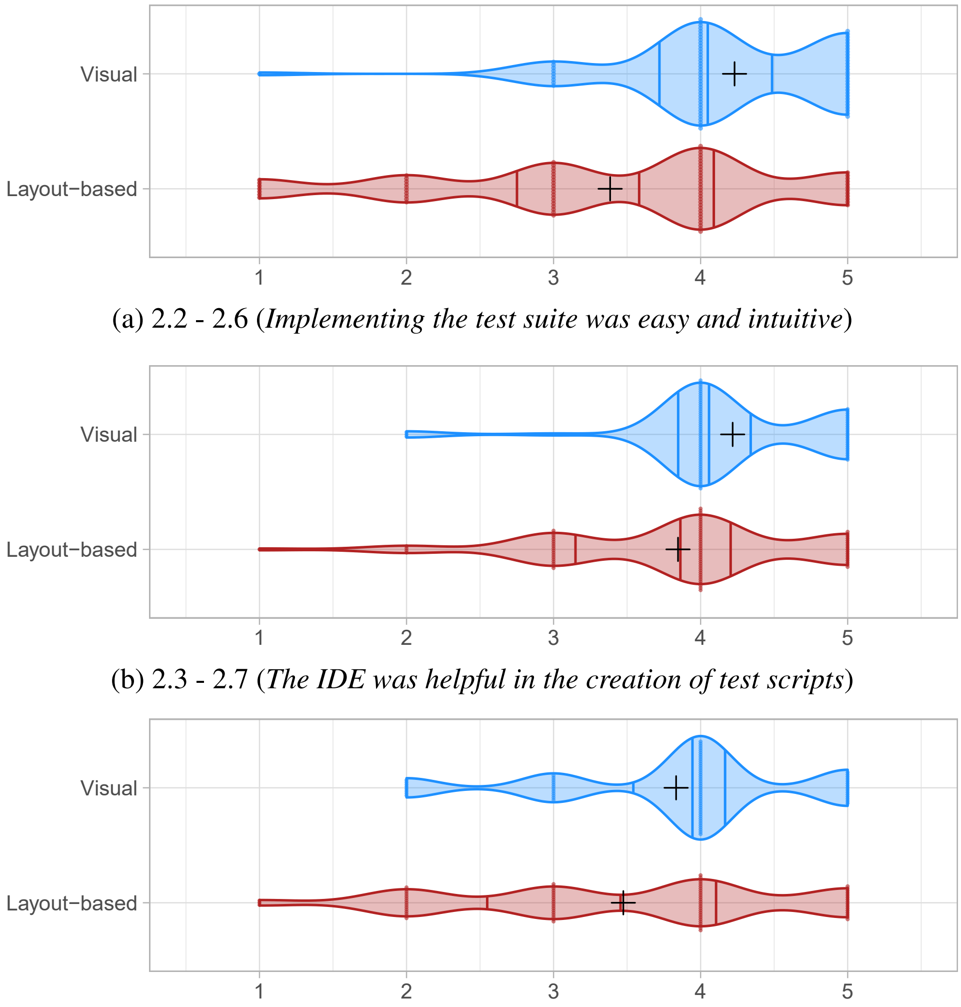
6.2.4 Usability of testing tools

The second part of the survey was aimed at understanding the difficulties experienced by developers when using the two selected testing tools, and their opinions about them.

With question 2.1 (distribution of answers are reported in figure 6.6), participants were asked whether the description of the user scenario was sufficiently clear: 83.3% of respondents answered Agree or Strongly Agree, with Agree as the median value.

Table 6.6 Experiment with graduate students: wilcoxon tests for Likert answers of the survey

Name	Description	p-value	Result
H_{l1-0}	There is no significant difference in the respondents' perceived easiness in using EyeAutomate or Espresso	2.053e-06	Reject
H_{l2-0}	There is no significant difference in the respondents' perceived ease of use of EyeStudio or Android Studio	0.001087	Reject
H_{l3-0}	There is no significant difference in the respondents' perceived easiness in finding properties to discriminate widgets, using the image recognition-based or the layout-based approach	0.01812	Reject



(a) 2.2 - 2.6 (*Implementing the test suite was easy and intuitive*)

(b) 2.3 - 2.7 (*The IDE was helpful in the creation of test scripts*)

(c) 2.4 - 2.8 (*It was easy to identify elements of the screen to interacte with*)

Fig. 6.5 Experiment with graduate students: distributions of answers to Likert questions

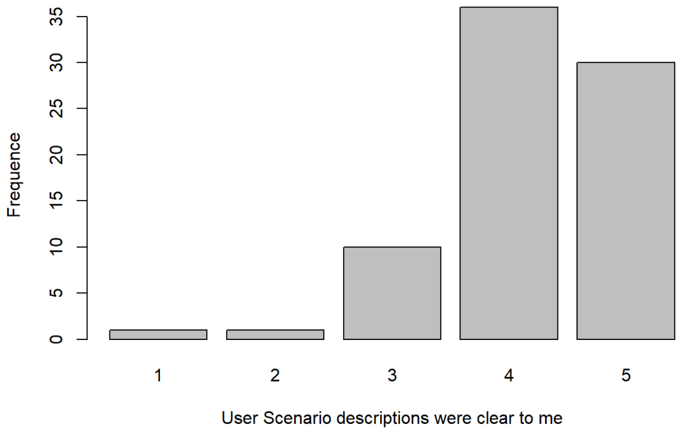


Fig. 6.6 Experiment with graduate students: answers to question *The user scenario descriptions were clear to me*

This result suggests that the errors and difficulties experienced during the development phase were related to the tools themselves, and not to an obscure description of the task to perform.

Regarding the perceived difficulty in implementing the test suites with both the tools, question 2.2 (*Implementing the test suite with EyeAutomate was easy and intuitive*) and question 2.6 (*Implementing the test suite with Espresso was easy and intuitive*) both had Agree as median answer. However, the distributions of the answers (see violin plots in figure 6.5a) suggest a higher perceived easiness for the Visual testing tool. A comparison between the two distributions has been made through the application of the Wilcoxon test, validating the hypothesis of distributions with the same median. The obtained p-value (2.053e-06) leads to a rejection of the null hypothesis. Hence, it can be assumed that the respondents perceived the development of test cases with the visual approach as easier with respect to the layout-based approach with Espresso.

Regarding the perceived difficulty in using the respective IDEs for the two tools (EyeStudio and Android Studio), questions 2.3 (*The EyeStudio IDE was helpful in the creation of test scripts*) and question 2.7 (*The Android Studio IDE was helpful in the creation of test scripts*) both had a prevalence for Agree or Strongly Agree answers. The distributions of the answers (see violin plots in figure 6.5b) suggest a higher perceived easiness when using EyeStudio rather than Android Studio. A comparison between the two distributions has been made through the application of the Wilcoxon test, validating the hypothesis of distributions with the same median. The obtained p-value (0.001087) leads to a rejection of the null hypothesis. Hence, it can be assumed that the respondents perceived the EyeStyudio IDE was considered as more helpful for the creation of test scripts than Android Studio. This result was rather expected, being EyeStudio specifically designed for the creation of visual test scripts, and Android Studio a full development environment that may be more difficult to master for inexperienced Android developers.

Regarding the easiness in identifying the properties to select the widgets of the screen with the two techniques, question 2.4 (*It was easy to identify elements with the visual recognition technique*) and question 2.8 (*It was easy to identify elements of the screen using the layout-based technique*) had both average values around Agree. The distributions of the answers (see violin plots in figure 6.5c) suggest a higher perceived easiness in obtaining screen captures rather than finding useful information in the layout files of the application. A comparison between the two distributions has been made through the application of the Wilcoxon test, validating the hypothesis of distributions with the same median. The obtained p-value (0.01812) leads to a rejection of the null hypothesis. Hence, it can be assumed that the respondents had found easier the collection of working captures of the interacted widgets instead of the search for individual unique properties.

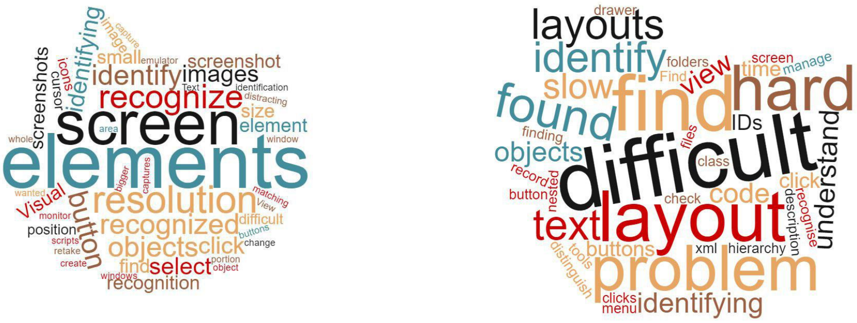
Table 6.6 reports a summary of the results of the Wilcoxon tests applied to distributions of answers to Likert questions.

Obstacles to test case development with EyeAutomate

Responding to question 2.5 (What were the principal issues that you found in identifying elements of the screen using the Visual GUI testing approach?) 27 of the respondents did not point out any difficulty faced in using the Visual approach with EyeAutomate.

The Word Cloud in figure 6.7a has been created upon all the textual responses given by the respondents to question 2.5. Figure 6.8 shows a classification of the answers given by the respondents.

The most perceived difficulty faced by the participants while using the EyeAutomate library through EyeStudio was the difficulty of capturing the right screen



(a) Issues with Visual approach

(b) Issues with Layout-based approach

Fig. 6.7 Experiment with graduate students: Word Clouds based on the answers to questions 2.5 and 2.9

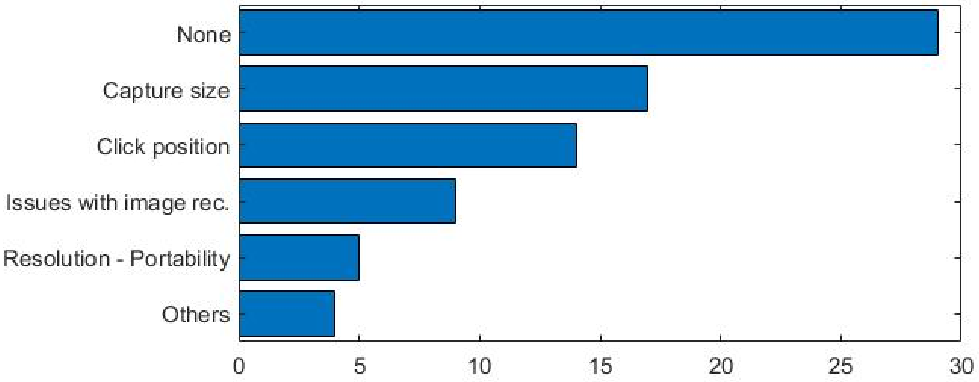
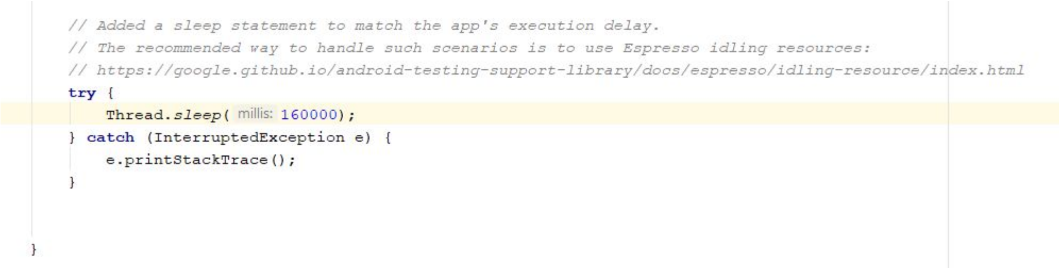


Fig. 6.8 Experiment with graduate students: perceived Obstacles to Visual Testing

captures to create working test cases. Sometimes, in fact, the image recognition algorithm is not able to interact with a given element of which the screen capture is correctly provided in the script, especially if the element is small and relatively simple. Some respondents highlighted this difficulty, as in the following comments: *"In some cases it was required to frame a bigger portion of the screen to make the test work."; "Sometimes it didn't recognize the objects"; "It seems that EyeStudio performs image recognition on the entire screen: this could lead to some problems in case the virtualized device screen is surrounded by distracting UI elements, that could interfere with the image recognition process. For instance, I had no problems since I was testing on an HD-ready monitor with a solid pitch-black wallpaper: a*



```
    // Added a sleep statement to match the app's execution delay.
    // The recommended way to handle such scenarios is to use Espresso idling resources:
    // https://google.github.io/android-testing-support-library/docs/espresso/idling-resource/index.html
    try {
        Thread.sleep( millis: 160000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }


}
```

Fig. 6.9 Sleep instructions generated by the Espresso Test Recorder

few colleagues of mine, instead, had several problems due to other distracting UI elements."

Some respondents experienced issues with the exact position at which the interactions with the GUI were performed. Several respondents had significant issues in identifying the drawer menu button of the application (on the top-left corner): many of them solved this issue by oversizing the screen capture, including the whole Title bar in it.

One relevant issue that has been found when collecting the results of this experiment has been the necessity of recapturing some (or even all) images for a test script, when replicating it on a different configuration – with unvaried resolution – than the original one on which the test scripts were taken. The test scripts worked correctly when the same screen captures were collected again, launching the application anew on the second device.

Obstacles to test case development with Espresso

The Word Cloud in figure 6.7b has been created upon all the textual responses given by the respondents to question 2.9 (*What were the principal issues that you encountered in identifying elements of the screen using Espresso?*). Figure 6.10 shows a classification of the answers given by the respondents.

28 respondents did not point out any difficulty faced in using the Layout-based approach to testing, with the Espresso tool. This result was expected since a significant number of respondents leveraged the Espresso Test Recorder without checking the correctness of the generated test suite, and hence did not find any evident problem in the developed test scripts. Several respondents criticized the insertion of delays – that can be up to 160,000 milliseconds, see figure 6.9 – performed after nearly

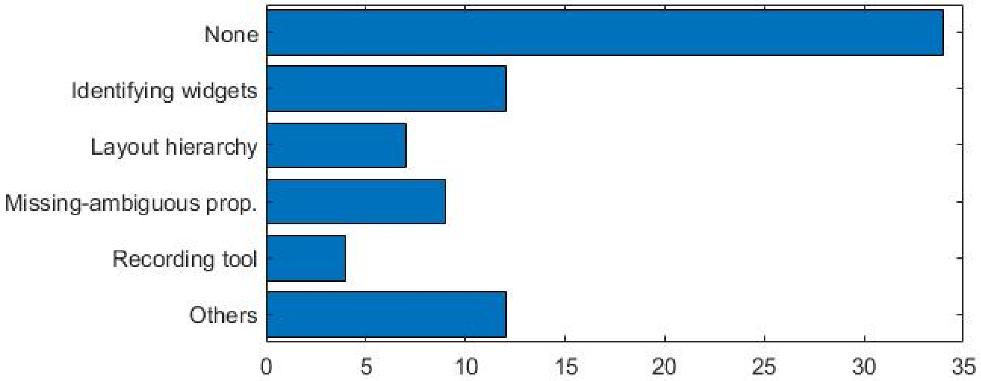


Fig. 6.10 Experiment with graduate students: perceived Obstacles to Layout-based Testing

every interaction by the Espresso Test Recorder: *"Manual layout search is too slow unless the element has a xml file with an obvious name. The layout inspector was faster but it still had some problems (I couldn't find a fast way to retrieve IDs, only text and description, which in many cases were null/not present). Also, I didn't find any "reload" button to update the Android screen in the layout inspector in a timely manner. The Espresso Test Recorder was the best, even though it needs some corrections after generation, it's just a bit slow when typing text into the android virtual device (1 character per second), and the pauses between actions are way oversized (>20 seconds, sometimes even 40)"; "The automatically-generated code is full of useless thread sleeps and it could be way simpler, in my opinion."*

Among those who did not leverage the Espresso Test Recorder, most found it difficult to find properties able to discriminate unambiguously between elements of the screen, and to clearly identify the values for those properties using the layout .xml files and the Layout Inspector: *"I found that, even with a small background of programming in Android, unless the project was specifically designed to be testable (all elements have ID property), it wasn't easy to track a specific element. The easiest way that I found was to use the Espresso Recorder to obtain the test case and then modify or adapt it to check that it actually matched my intentions. In fact, from the layout inspector, it wasn't clear to me what properties were useful to identify objects and this was even more difficult from the view layouts (since they were many and it was difficult even just to understand which layout corresponded to which view on the app)."*

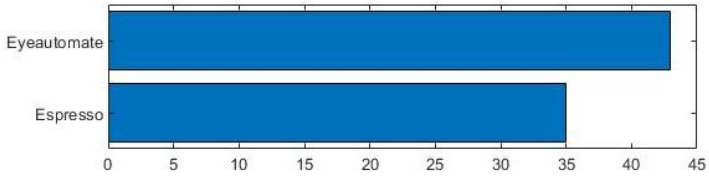


Fig. 6.11 Experiment with graduate students: answers to question 2.10 *Which tool would you choose if you had to perform visual testing again?*

Most difficulties were perceived in testing ListViews, the DrawerMenu of the application, and for executing the long click instruction on the Notes for the deletion and restoration scenarios.

6.2.5 Preference towards Layout-based or Visual GUI testing tools

Of the 78 complete surveys that were collected, students expressed a slight preference (see figure 6.11) towards the Visual testing technique with respect to the Layout-based GUI testing technique. Namely, 43 students (the 55.1% of the total sample) indicated EyeAutomate as the tool that they would use again if they had to perform GUI testing, whereas 35 (the 44.9%) indicated Espresso.

The explanations given by the students for their preference towards one or another testing tool seemed to be principally moved by individual preference rather than actual advantages or disadvantages exposed by the tools. For instance, the answer *"it is easier to use"*, was frequently given by respondents for both the testing tools.

Most of the respondents who selected EyeAutomate as the preferred way of testing Android apps gave the easier generation of test cases – guaranteed by the Capture & Replay functionality of the EyeStudio IDE – as the main explanation of such choice. Some respondent also recognized that EyeAutomate is more indicated when the tester is not the developer itself, and when the focus of the testing is not strictly on the functionalities of the applications but principally on the actual appearance provided to the user. Other respondents praised the portability and possibility to work with multiple platforms of the Visual approach to GUI testing

On the other hand, some students who preferred Espresso pointed out that the tests developed with such tool felt more precise and reliable than the ones obtained through screen captures and image recognition. Emphasis was also put on the characteristics of layout-based testing tools to be not fragile to minimal changes in the graphics of the application. Several students highlighted as an advantage the greater control of the application that layout-based testing tools are able to provide. The fact that Espresso is specifically designed for Android testing – instead of being a general purpose tool for GUI applications applied to an emulated Android device on a desktop PC – was also highlighted as a reason for preferring Espresso to test an Android app.

In general, however, a significant number of participants leveraged the automated Espresso Test Recorder for the creation of test scripts, instead of manually writing down the operations for identifying views and executing operations on them. It can be assumed that many respondents' answers about the preference towards Espresso were largely influenced by the availability of such an add-on for the Android Studio IDE.

Answer to RQ2.3: The respondents to the experiment found slightly easier the development of test suites using the EyeAutomate library, in the context of the EyeStudio companion IDE, with respect to developing scripted Espresso test cases in the Android Studio IDE. The respondents identified the imprecision of the image recognition library, and the difficulty in finding individual ids for the widgets, the most problematic aspects of, respectively, the proposed Visual and Layout-based testing tools.

Chapter 7

Study 3: Measures of Diffusion and Evolution of Testware in OS projects

In order to give a quantitative evaluation of the issue of maintenance and fragility among open-source Android projects, measurements were performed on a set of repositories mined according to the procedure detailed in section 3.

This study allowed answer the third research question of the study: ***RQ3*** *- What is the adoption and typical evolution of test suites with automated GUI testing frameworks among Android open source projects?*

The design and results of this study have been presented in a workshop paper at INTUITEST 2017 [29], in a conference paper at PROMISE 2017 [28], and in a journal paper published in IEEE Transaction on Reliability [32].

7.1 Study design

A set of metrics have been defined, in order to (i) quantify the adoption of testing tools on sets of Android app projects; (ii) quantify the evolution needed by the test suites, during the normal evolution of an Android app project.

Hence, RQ3 is split into the following subquestions:

- **RQ3.1 - Adoption and Size**: What is the level of adoption of a set of automated testing tools among open-source Android projects?

- **RQ3.2 - Evolution**: How much are GUI test classes associated with the analyzed sets of tools modified through consecutive releases of an open-source Android project?

To answer the two subquestions, a set of 12 metrics has been introduced. The metrics are subdivided in two different groups, and are based on an input consisting of test classes (of a single release, or belonging to two consecutive releases of the same project), production code classes and .txt difference files (from now on, *diff files*) computed between consecutive versions of the same file. The metrics have been defined for Java code, and are hence applicable to any kind of Java application provided with test code based on Java itself, not limited to Android applications.

Change metrics have been proposed by several works available in literature. A popular example is the set of metrics defined by Tang et al. [96], which were defined to describe the amount of bug-fixing change histories in source files. The metrics proposed by Tang et al. are subdivided in pure size metrics (e.g., the amount of added or removed lines of code between two releases of the same file), atomic metrics (e.g., boolean metrics that are equal to one if a test class features added methods or deleted methods), and semantic methods (e.g., counting the number of changed dependencies inside a test class).

Most of the metrics defined in this work are instead relative metrics because they aim at measuring the co-evolution of test code along with the normal evolution of the project the tests are associated with. The metrics are also normalized, in order to allow the comparison between different projects, with production code and test code bases of different sizes. The normalization of the metrics makes them inapplicable to testing tools that create test scripts in languages that are different from pure Java, or to apps containing fragments of code in different languages (e.g., Kotlin for Android programming).

Table 7.1 summarizes all the defined metrics, the acronyms that are used in the remainder of the paper, and the research question they contribute to answering. The formulas for computing them are provided in the following. In the definition of the metrics, *production code* indicates all the code of the application, including both program code and test code.

Table 7.1 Defined metrics for the computation of diffusion and evolution of test suites for Layout-based GUI testing

Group	Name	Explanation
Diffusion and size (RQ1)	TD	Tool Diffusion
	NTR	Number of Tagged Releases
	NTC	Number of Test Classes
	TTL	Total Test LOCs
	TLR	Test LOCs Ratio
Test evolution (RQ2)	MTLR	Modified Test LOCs Ratio
	MRTL	Modified Relative Test LOCs
	MRR	Modified Releases Ratio
	TSV	Test Suite Volatility
	MCR	Modified Test Classes Ratio
	MMR	Modified Test Methods Ratio
	MCMMR	Modified Classes with Modified Methods Ratio

7.1.1 Adoption and size metrics

Five metrics have been defined for the measurement of the adoption of the testing tool in a sample of Android projects and for the relative size of the test suites developed with such tools (hence, for answering RQ3.1).

This set only comprises static metrics, i.e. they can be measured for a single release of the given project, without the need for comparisons with the precedent or subsequent one.

TD (Tool Diffusion) It is defined as the percentage of projects, among the context considered, featuring a given testing tool. In the context of this research, it is used to provide the percentage of Android projects from GitHub that are provided with test scripts generated with automated GUI testing frameworks.

NTR (Number of Tagged Releases) It is defined as the number of releases of a repository. In the context of this experiment, it can be used to give statistics about the lifespan of the considered Android GitHub projects, discriminating between long-lived and maintained projects and tryouts or smaller applications with a short lifespan.

NTC (Number of Tool Classes) It is defined as the absolute number of classes that can be associated with a given testing tool, according to the heuristic explained in section 3.3.2. In the context of this experiment, this metric allows computing the total number of classes generated with the considered six testing tools for tested Android projects hosted on GitHub.

TTL (Total Tool LOCs) It is defined as the absolute number of lines of code contained in the test classes associated with a given testing tool, inside a repository. In the context of this experiment, this metric is used to compute the absolute size of test suites generated with the considered six testing tools for tested Android projects hosted on GitHub.

TLR (Tool LOCs Ratio) It is defined as:

$$TLR_i = TTL_i / Plocs_i,$$

with $Plocs_i$ being the total number of production lines of code in release i, and TTL_i being the Total Tool LOCs metric defined before. The metric lies in the $[0,1]$ interval and allows to quantify the relevance of the code associated with a given testing tool with respect to the total production code of a given project. In the context of this experiment, it is used to compute the relevance of the six selected testing tools in tested Android projects hosted on GitHub.

7.1.2 Test Evolution metrics

Seven metrics have been defined to evaluate the amount of evolution needed by test code, and hence to answer RQ3.2.

The metrics belonging to this set are computed on consecutive releases of the same project, or on the entire set of releases belonging to a project (called *lifespan* hereafter).

MTLR (Modified Tool LOCs Ratio) is defined for each release i of a given project as

$$MTLR_i = \frac{Tdiff_i}{Tlocs_{i-1}},$$

where $Tdiff_i$ is the number of added, deleted or modified lines of code in all the test classes in the transition between release $i-1$ and i, and $Tlocs_{i-1}$ is the total number of lines of code associated with the studied tool in release $i-1$. The metric gives an indication about the amount of intervention that is performed on the existing test code, comparing it with the total size of test code in the previous release of the project. The resulting value can be higher than 1, if the number of additions and modifications of existing lines of code is higher than all the test lines of code in the previous release. In the context of this experiment, this metric quantifies the amount of maintenance needed by the six selected testing tools, with respect to the size of existing test code of Android GitHub projects.

MRTL (Modified Relative Tool LOCs) is defined for each release i of a given project as

$$MRTL_i = \frac{Tdiff_i}{Pdiff_i},$$

where, considering the transition between release $i-1$ and i', $Pdiff_i$ is the number of added, deleted or modified lines of code in all the production classes, and $Tdiff_i$ is the number of added, deleted or modified lines of code in all the test classes associated with a given testing tool. This metric is computed only for releases who feature test code (i.e., $TRL_i > 0$) and is defined only if $Pdiff_i > 0$, meaning that any modification has been applied on the repository between release $i-1$ and i. The metric belongs to the range $[0,1]$, since the added, deleted or modified test LOCs are a subset of the complete set of added, deleted and modified LOCs for the whole project. A value close to 1 of this metric imply that a relevant portion, among the total effort for maintaining a given project, is devoted to the update of test code. In the context of this experiment, the metric measures the amount of relative maintenance required by Android GitHub repositories featuring the six selected automated GUI testing frameworks.

MRR (Modified Releases Ratio) is defined for a given project as the ratio between the number of tagged releases featuring at least one modification in test code associated with a given testing tool, and the total number of tagged releases of the project (i.e., the *lifespan* of the project). The metric belongs to the $[0,1]$ interval. Values close to 1 of this metric implies that the majority of the releases of the project involved maintenance in the test code, implying high coupling between test code and application code.

TSV (Test Suite Volatility) is defined for a given project as the ratio between the number of test classes that were modified at least once during the lifespan of the project, and the total number of test classes that are encountered during the evolution of the project (also test classes that are deleted at some point in the release history). The metric is a companion to the MRR metric, but from the point of view of the test classes instead of that of the releases of the project. The metric lies in the $[0,1]$ interval. Values close to 1 of the metric imply that the majority of the test classes had to be modified at least once during the project lifespan, meaning that most of the test classes covered features of the application that had to be modified during its evolution. On the other hand, values close to 0 of the metric may imply scarce coverage of the test classes of the features of the application, or abandonment of developed test classes after their insertion in the repository.

MCR (Modified test Classes Ratio) is defined for each release *i* of a project as

$$MCR_i = MC_i/NTC_{i-1},$$

where NTC_{i-1} is the number of classes associated with the given testing tool in release $i-1$, and MC_i is the number of the test classes that have been modified in the transition between releases $i-1$ and i. The metric is not defined when release $i-1$ of the project does not feature test classes associated with the given testing tool (i.e., $NTC_{i-1} = 0$). The metric takes a snapshot of the number of test classes requiring modifications in a single release of the project, and tracking its evolution may give indications about which modifications on application code have higher effects in test code maintenance.

MMR (Modified test Methods Ratio) is defined for each release *i* of a project as

$$MMR_i = MM_i/TM_{i-1},$$

where TM_{i-1} is the number of Java methods contained by test classes associated with a given testing tool in release $i-1$, and MM_i is the number of methods among those that have been modified in the transition between releases $i-1$ and i. The metric is not defined when release $i-1$ of the project do not feature any test class associated with the given testing tool (i.e., $NTC_{i-1} = 0$).

MCMMR (Modified test Classes with Modified Methods Ratio) is defined for each release i of a project as

$$MCMMR_i = MCMM_i/NTC_{i-1},$$

where NTC_{i-1} is the number of classes associated with the given testing tool in release $i-1$, and $MCMM_i$ is the number of test classes that have been modified in the transition between releases $i-1$ and i, and that contain modifications inside test methods. This metric is not defined when release $i-1$ of the project does not feature test classes associated with the given testing tool (i.e., $NTC_{i-1} = 0$). The metric is upper-bounded by MCR, since by definition of the intermediate metrics $MCMM \leq MC$.

7.1.3 Metrics computation

Based on a given repository of git projects, the described set of metrics were computed applying the procedure detailed in the following.

Computation of diffusion and size metrics

The TD metric has been computed for each of the six considered testing tools, as the ratio of the projects obtained at the end of the filtering phase described in section 3.3.2, and the total number of mined valid Android projects.

Static metrics defined for answering RQ3.1 have been computed for each project on the *master* release only. The code search for keyword related to one of the six considered testing tools resulted in the computation of the NTC metric for each tested repository.

For each Java test class associated with a given testing tool, the lines of code have been counted using the *cloc* bash tool[1]. The sum of such count for all test

[1] http://cloc.sourceforge.net/

classes allowed to compute the TTL metric. The count has been repeated also for the complete set of production classes of each project, to find P_{locs} of the master release; the ratio between TTL and P_{locs} allowed to find the TLR metric for each project.

The NTR metric was obtained, for each considered repository, by means of the *git tag* command.

Computation of evolution metrics

To compute the values of the metrics designed for answering RQ3.2, for each pair of consecutive tagged releases, the total amount of modified LOCs was computed. Then, the total amount of LOCs added, removed or modified in classes associated with the featured testing tool was computed. To perform such computations, the *git diff* command was used. The command gives the possibility, by using the -M parameter, to identify files that have been renamed or moved in the transition between subsequent releases, without considering such operation as the combination of a deletion and an addition of a file. Since the -M command is able to work with a maximum size of the git repository, it was decided to not use such option to obtain the same way of computing the number of changed LOCs for all considered projects, and hence moved files were considered as different files in all release transitions.

The measurement of changed lines of code, paired with the TLR_i metric measured for each release of each project, allowed to compute the derivated metrics $MTLR_i$ and $MRTL_i$ for each tagged release of the project. Once the exploration of the lifespan of a given project was completed, final averaged values were obtained as $TLR = Avg_i\{TLR_i\}$, $MTLR = Avg_i\{MTLR_i\}$, $MRTL = Avg_i\{MRTL_i\}$, with $i \in [1, NTR]$ being NTR the number of tagged releases featured by the project.

Volatile classes (i.e., classes featuring modifications throughout their lifespan) were flagged during the computations described above, so that it was possible to compute for each project its TSV value.

The evolution of individual test classes and methods was tracked with the use of an automated Java class examiner, based on the open-source JavaParser tool[2]. The tool, given two releases of the same test classes as parameters, is capable of extracting all methods defined in both releases, finding whether the methods are present in only a release of the two, and for those who appear in both versions

[2] https://github.com/javaparser/javaparser

of the class finding whether they have been modified or not. The application of this automated examiner to each class, for each release transition in the project lifespan, allowed for the computation of the MCR_i, MMR_i and $MCMMR_i$ metrics. Also in this case, average values have been computed for each project at the end of the exploration of its lifespan, as $MCR = Avg_i\{MCR_i\}$, $MMR = Avg_i\{MMR_i\}$, $MCMMR = Avg_i\{MCMMR_i\}$, with $i \in [1, NTR]$.

A running sample of the computation of the set of metrics, on a real application, is provided in Appendix B.

7.1.4 Threats to Validity

Threats to internal validity

The test class identification process was based on some keywords specific to each testing tool: any file containing one of those keywords and containing the word *test* in its absolute path is considered as a test file without further inspection. This procedure may miss some test classes, or consider a file as a test class mistakenly. Evaluated on a sample of 100 classes that were manually examined, the proposed heuristic guaranteed a precision (measured as the amount of true positives, i.e. classes extracted with the code search that were actual test classes, over the number of classes extracted using the heuristic) of about 90%. Better procedures of test class extraction can be used, taking into account the presence (or absence) of calls to specific methods that are proper of a specific GUI Automation Framework (e.g., the use of the onView or onData methods, for the Espresso framework). The way the test classes of our study were generated was also not evaluated. This may introduce bias to our computed metrics, because test classes are re-generated in each version using automated tools (e.g., Capture Replay tools) higher amounts of modified LOCs are expectable, with respect to manual editing of existing test classes.

The number of tagged releases is used as a criterion to identify a project as worth to be considered for the executed investigations; it is not assured that this check is the most dependable one for pruning negligible projects.

The metrics have not been tested outside the scope of this study, hence the correctness of the assumptions based on them is not ensured . The reported evaluations are based only on files that contain pure Java code. Hence, code in other languages, that may be part of test suites as well as of production code of Android applications, does not contribute to the computations. This may add biases to the presented results.

Java files containing keywords pertaining to each tool were entirely associated to the tool, and all their lines were counted for the defined metrics. In addition to that, no discrimination has been made about the use that was made of the individual tools, while some considered testing frameworks can be used to perform not only GUI testing. A manual validation of the heuristics, performed on a set of 100 classes, resulted in about 70% precision in finding test classes testing the GUI of the respective AUT. Both threats may add biases to the results, if multiple different testing frameworks are used in the same Java classes, and if the testing tool to which the code is associated is not used to perform GUI testing.

Structure, provided coverage and quality of the developed test cases have not been controlled and taken into account by the automated procedure for computing the metrics. Hence, the effects that low-quality tests have on maintenance effort are not taken into consideration. The study was also conducted statically, meaning that test scripts were not executed before and after the transition between subsequent releases of the projects. Hence, the evaluation of the needed effort in test code modifications is based on the measurement of the modifications that were actually performed, without evaluating whether those were sufficient to adapt to the evolution of the app or not.

Threats to External Validity

Testing tools and techniques adopted by relevant industrial players may vary significantly from the ones discussed in this work, and by the related ones discussed in earlier sections. It is not assured that the findings, based on a very large repository of open-source projects, can be applicable to the development of commercial or closed-source projects.

Measures were collected for just six scripted GUI automation frameworks. It is not certain that such selection of tools is representative of other categories of testing tools or even different tools of the same category, which may exhibit different trends and fragilities throughout the history of their AUT.

Table 7.2 Acronyms used for Diffusion and Size Metrics

Name	Explanation
TD	Tool Diffusion
NTR	Number of Tagged Releases
NTC	Number of Test Classes
TTL	Total Test LOCs
TLR	Test LOCs Ratio

Table 7.3 *NTR*, *NTC*, *TTL*, *TLR* per testing tool: average and median (in parentheses) values for master release.

Tool	n	TD	*NTR*	*NTC*	*TTL*	*TLR*
Espresso	372	2.42%	13 (5)	3 (2)	418 (181)	7.63% (4.23%)
UI Automator	50	0.32%	19 (7)	3 (1)	523 (226)	7.35% (3.38%)
Robotium	129	0.84%	16 (6)	4 (1)	518 (196)	6.15% (2.96%)
Robolectric	631	4.11%	15 (6)	9 (3)	1,307 (331)	13.50% (8.47%)
Appium	12	0.08%	37 (27)	14 (3)	1,510 (927)	2.81% (1.27%)
Average			15	6	908	10.49%

The metrics can be applied only to testing tools who produce scripts in Java. Other tools producing test scripts in other languages cannot be evaluated using the provided metrics.

7.2 Results

In this section, the measures gathered for the metrics described in section 7.1 are described. The metrics have been computed on the set of projects mined from GitHub with the procedure described in section 3.3.

At the end of the mining procedure, an initial total amount of 280,447 GitHub repositories have been found, featuring the term *Android* in their names, readme files or description.

The first filtering phase, about the deletion of repositories which did not feature any Manifest file and at most a single tagged release (which did not permit to perform any comparison between subsequent releases, necessary for the computation of the Evolution metrics), cut out a significant portion of the original set, with 18,930 resulting repositories (the 6.85% of the original context).

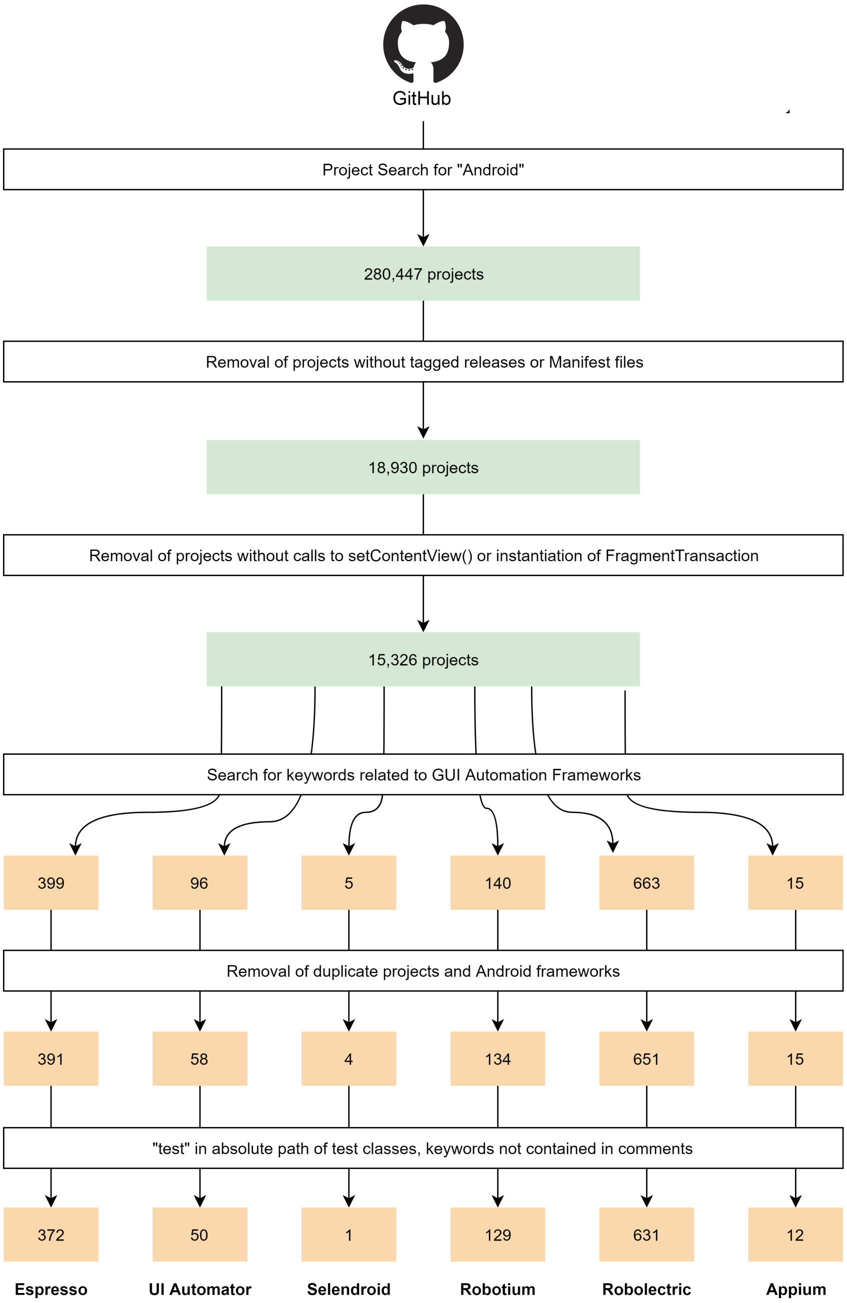

GitHub
Project Search for "Android"
280,447 projects
Removal of projects without tagged releases or Manifest files
18,930 projects
Removal of projects without calls to setContentView() or instantiation of FragmentTransaction
15,326 projects
Search for keywords related to GUI Automation Frameworks
399
96
5
140
663
15
Removal of duplicate projects and Android frameworks
391
58
4
134
651
15
"test" in absolute path of test classes, keywords not contained in comments
372
50
1
129
631
12
Espresso
UI Automator
Selendroid
Robotium
Robolectric
Appium

Fig. 7.1 Number of Android projects mined from GitHub and associated with the six considered testing tools, after each step of the filtering procedure

The final filtering phase on “Android” repositories, performed in order to cut out projects which were not likely to have any Graphical User Interface, led to a final set of 15,326 Android repositories (5.46% of the original context).

7.2.1 Diffusion and Size measures

After the filtering phase for the starting context of Android repositories with a history of releases, the application of the GitHub Code Search API for keywords related to the selected testing tools led to the definition of six different groups of repositories, with size ranging from 5 (for the set of repositories featuring Selendroid) to 372 (for the set of repositories featuring Espresso).

The additional filtering phases performed on individual sets (i.e., the removal of duplicate projects and clones of Android frameworks, and the removal of projects without test classes having the “test” keyword in their absolute paths) led to a reduction of the size of the six sets. The reduction was more substantial for the set of projects associated with UI Automator, principally for many clones of frameworks that figured inside it.

The graph in figure 7.1 shows the number of projects after each filtering phase. Table 7.3 summarizes the metrics that have been computed on the projects, to answer RQ3.1. The columns show: the total number of projects featuring the testing tools considered, at the end of the filtering procedure and the derivated the Tool Diffusion (TD) metric; the average and median values for the Number of Tagged Releases (NTR), Number of Test Classes (NTC), Total Test LOCs (TTL) and Test LOCs Ratio (TLR). The last two measures have been computed on the master releases of each project. The last line of the table shows the average value for all projects, weighted by the size of each set of projects. The table does not show the measures for the single Selendroid projects which lasted after the application of the whole filtering phase to the original set of projects featuring such tool.

To ease the reading of this section, the acronyms used hereafter for the metrics are reported in table 7.2.

The TD metric, computed as the percentual penetration of each testing tool among the extracted context of Android repositories, ranged from near 0% (for the set associated with Selendroid, containing just a single project) to 4.11% for the set of projects featuring Robolectric. The higher percentage of projects featuring

Robolectric may be justified with the fact that Robolectric has been available for a longer time than the other testing tools that have been considered, and it can also be used for other forms of testing in addition to automated GUI testing. The set of projects featuring Appium also proved to be rather small, with just 12 projects. Of the two testing frameworks that are part of the Android Instrumentation Framework, Espresso proved to be more widespread than UI Automator. This result may hint at Espresso being an easier testing tool to use for creating simple test suites for Android applications, whereas UI Automator was typically used for complex sets of applications or frameworks in which also the interaction with the OS user interface had to be tested. Also, the prevalence of Espresso test classes can be motivated by the fact that the tool is indicated by the Android Developer Guide as the official way to test individual activities of Android apps; additionally, the Android Studio IDE features a built-in plugin for the creation of Espresso test cases through Capture & Replay.

Overall, slightly less than 8% of the filtered set of Android projects featured tests belonging to at least one of the selected testing tools. The six sets of projects were not necessarily disjoint, since a single repository may contain references to multiple testing frameworks. This may create overlaps, and hence to an overestimation of the cumulated diffusion of the six considered testing tools. That considered, the resulting cumulated TD gives evidence of a lack of extensive adoption of automated GUI testing frameworks among Android repositories hosted on GitHub. As a limitation of this result, it must be considered that such value is limited to the six testing tools considered in this study, with the possibility of the presence of many scripted testing tools adopted by other Android repositories.

As a final comparison for the adoption of the considered testing tools, the same procedure of search for test classes was performed searching, this time, for the *JUnit* keyword. This search would result in a set of projects featuring any kind of unit testing classes developed with JUnit, along with classes associated with other testing frameworks using JUnit as an automation engine. The search resulted in 3,669 projects (with tagged releases and manifest files) featuring classes containing the *JUnit* keyword, for around the 20% of the total amount of extracted Android projects. This percentage, albeit significantly bigger than the combined percentage of adoption of the six considered testing tools, shows that the percentage of applications that are tested with any framework based on JUnit is still quite low, testifying a rather scarce adoption of any form of testing on Android open-source repositories hosted on GitHub.

NTR metric was used to give a statistic about the average history of the projects of each set. The averages went from 13 (for the projects featuring Espresso) to 37 (for the projects featuring Appium). The small average and median values for Espresso projects may suggest that Espresso is typically preferred for testing smaller applications with shorter lifespans, possibly thanks to their higher accessibility and to its integration (especially thanks to the Espresso Test Recorder tool) in the Android Studio IDE. In the case of Appium, the result may be heavily influenced by the small size of the set of projects that have been considered.

The average and median values for the NTC metric, useful for quantifying the typical size in terms of test classes of an automated GUI test suite for an Android project, were rather small for all the considered sets except for the set associated with Appium. This result may be a consequence of the usual coding patterns adopted when developing test classes for Android applications, with each test class associated with an Activity of production code. Most apps do not feature a high number of different screens to compose the interface shown to their users, and therefore they do not feature many activities to be tested. An investigation about the number of activities for the considered projects was performed, by computing the number of declared activities in the manifest .xml file, with a measured average number of 19 activities. The smaller average value for the NTC metric (6), suggests a partial coverage of the activities of the Android repositories, and hence small coverage of the production code by the test classes.

Average TTL values were very large for the sets of projects featuring Appium and Robolectric. However, it could be noticed that the TLR was very small for the considered Appium projects. This may suggest that the amount of test code written with Robolectric is typical more relevant, in the whole production code of a repository, than the amount of test code written with Appium. This result also suggests that, among all the considered sets of projects, Appium has been typically used for testing bigger projects in terms of production LOCs. The values of TTL and TLR were rather similar for sets of projects featuring Espresso, UI Automator and Robotium, suggesting that the testing tools are used in a similar way on projects of similar size. The set of projects featuring Espresso had the lowest TTL value. This measure can be explained with following reasons: (i) the fact that using a white-box

Table 7.4 Acronyms used for Evolution Metrics

Name	Explanation
TLR	Test LOCs Ratio
MTLR	Modified Test LOCs Ratio
MRTL	Modified Relative Test LOCs
MRR	Modified Releases Ratio
TSV	Test Suite Volatility
MCR	Modified Test Classes Ratio
MMR	Modified Test Methods Ratio
MCMMR	Modified Classes With Modified Methods ratio

Table 7.5 Measures of the evolution of test code (averages on the sets of repositories)

Tool	*TLR*	*MTLR*	*MRTL*	*MRR*	TSV	MCR	MMR	MCMMR
Espresso	6.30%	4.21%	3.17%	16.64%	19.42%	15.75%	3.83%	60.12%
UI Automator	5.84%	3.10%	1.14%	10.68%	21.46%	14.48%	3.42%	55.86%
Robotium	5.11%	5.09%	3.07%	16.50%	25.13%	17.40%	3.80%	58.41%
Robolectric	11.23%	5.30%	5.93%	20.39%	18.12%	14.91%	3.88%	55.36%
Average	8.78%	4.94%	4.54%	18.37%	19.43%	15.43%	3.83%	57.21%

testing technique allows to translate the same testing scenarios in direct operations instead of performing multiple operations on higher level widget descriptions; (ii) the accessibility of the Espresso framework even to non-experienced developers, and the availability of higher amounts of documentation with respect to the other testing tools; (iii) the integration of Espresso in the Android Development environment, that may make it the first choice for tryouts – later abandoned – of the practice of testing.

Answer to RQ3.1: The considered GUI testing tools reached a diffusion that is always lower than 4.11% individually, and a combined adoption of about 8% by the considered set of 15 thousand Android repositories hosted on GitHub. The projects that are tested with the considered tools are typically rather short-lived, with an average of 15 releases, and feature on average few very few test classes for around 10% of total production code devoted to testing.

7.2.2 Evolution measures

The measures gathered for the evolution of test suite, by comparisons of test classes in subsequent releases throughout the entire lifespans of the considered projects, are shown in table 7.5. The columns show, respectively, averages for Test LOCs Ratio measured on all the releases instead on the *master* release only (TLR), Modified Test LOCs Ratio (MTLR), the Modified Relative Test LOCs (MRTL), Modified Releases Ratio (MRR), Test Suite Volatility (TSV), Modified Classes Ratio (MCR), Modified Methods Ratio (MMR), Modified Classes With Modified Methods Ratio (MCMMR). The last row in the table reports the average of the individual averages for each testing tool, weighted by the size of the respective sets.

To ease the reading of this section, the acronyms used hereafter for the evolution metrics are explained in table 7.4.

The reported value for TLR show that – when present – the test code associated with the selected testing tools amounts on average to slightly less than 10% of the whole production code of the projects. Comparing the values in table 7.5 to those in table 7.3, it is evident that the averaged TLR value is smaller than the TLR measured on *master* release. This result may be evidence of the graduality of construction of test suites, or their absence in the initial release of Android projects. The measured average values ranged from 5.11% (for the set of projects featuring Robotium) to 11.23% (for the set of projects featuring Robolectric).

The average values for the $MTLR$ metric show that on a typical tested project, about 5% of the testing code associated with the six considered GUI testing tools is modified between consecutive releases of the project. The values for the $MRTL$ show that on average, when the selected testing tools are used, the 4.54% of the total modified production LOCs belong to test classes. This metric is unable to discriminate the reasons behind the maintenance performed in test code, but however gives an indication of the amount of intervention needed between subsequent releases by a typical test class written with one of the selected testing tools. Robolectric has shown the highest $MRTL$ values: this result, paired with the higher TLR for the same tool, may suggest a higher complexity of test suites written with Robolectric, that hence are of bigger size and need more maintenance.

The MRR metric was used as an indication about how often the developers of the inspected open source projects had to modify the individual classes associated

Table 7.6 Percentage of projects without modifications in test suites, classes and methods

Tool	Unmodified suites	Unmodified classes	Unmodified methods
Espresso	24.6%	57.0%	65.8%
UIAutomator	16.0%	40.0%	55.0%
Robotium	16.6%	44.1%	60.0%
Robolectric	15.8%	45.3%	53.3%

with the studied GUI testing tools. On average upon all projects, about 19% of the releases of the projects contained modifications in classes associated with the selected testing tools, with a maximum of 20.39% for projects featuring Robolectric and a minimum of 10.68% for projects featuring UI Automator. The TSV metric measures the occurrence of modifications from the point of view of the set of test classes associated with a given testing tool. Also in this case the resulting average over all the sets was of about 20% (with a maximum of 25.13% for Robotium, and a minimum of 18.12% for Robolectric), implying that about one every five test classes is modified during the lifespan of a project, and the other four are never modified after they have been inserted in the repository.

The average *MCR* metric shows that, on average, 15.43% of test classes are modified between consecutive tagged releases, in the set of Android repositories considered. Average values were rather similar for all the six sets of repositories, with the maximum value of 17.40% measured for projects associated with Robotium. The average value for *MMR* metric tells that 3.83$ of the test methods are modified between consecutive releases of the considered Android repositories. This percentage is, as expected, smaller than MCR, because individual test classes may contain multiple test methods, and just the modification of one method would make them count for the computation of MCR. Also in the case of the *MMR* metric all the individual values for the six sets were very close to the overall average value.

Not all modified test classes contained significant modifications, i.e. they could contain changed lines of code only due to syntax corrections, changed comments or changed imported files. The *MCMMR* metric was used to give a statistic about how many of the modified classes contained modified methods, instead of having changes limited to irrelevant sections of code. The measures for this metric showed that in almost 60% of the cases of modified test classes, the modifications were also lying inside test methods.

It is worth highlighting that the average values for the evolution metrics over the sets of projects featured quite a low variability: more specifically, the average values for Modified Classes Ratio and Modified Methods Ratio were very similar for all the considered testing frameworks. Since all the tools are layout-based and produce test code in Java, these results may suggest that they share similarities in terms of syntax, and hence are influenced to a similar extent by typical changes applied to an Android project.

Lastly, it must be taken into account that the values measured for the MCR, MMR and MCMMR metrics are heavily lowered if the test classes and methods are added at some point of the lifespan of a project, and then remain unmodified during its evolution. Table 7.6 shows statistics about the projects that have unmodified test suites (meaning that test suites are entirely unaltered for the whole lifespan of the project); those that only have unmodified test classes (meaning that no modifications in any test class are made during the whole lifespan of the project, but additions or removals of test classes are possible); those that have modified classes but no modifications in test methods (meaning that the changes inside the test classes are only limited to irrelevant portions of the code of test classes).

Answer to RQ3.2: An average 5% of testing code is modified between consecutive tagged releases of Android repositories hosted on GitHub featuring tests associated with the six selected testing tools. 4.54% of the whole maintenance effort on production code is limited to changes in classes that are identified as tests developed with the studied testing tools. On average, one every five release required efforts of maintenance on test classes, and one every five classes had to be modified at least once during the lifespan of a project. On each new release, an average 15.43% of test classes (3.83% of test methods) feature modifications.

Chapter 8

Study 4: Taxonomy of fragility causes

To understand what are the typical causes of fragility for Android projects, and to compute their frequencies of occurrence, the Grounded Theory technique has been applied over the full set of modified test methods gathered during the previous part of the study. This study allowed to answer the fourth research question of the study: ***RQ4*** - *Why and with which frequency fragilities occur in tested Android projects?*

A preliminary presentation of the application of Grounded Theory for the creation of a taxonomy of modification reasons of Android GUI tests has been given at the NEXTA 2018 workshop [31].

8.1 Study Design

This section contains a brief description of the Grounded Theory approach for the creation of taxonomy, and the way it was applied to git diff files for the construction from the bottom up of several distinct types of modifications triggering fragilities.

Previous definitions of categories of reasons for modifying test classes are available in the literature. For instance, Yusifoglu et al. [101] identified four types of maintenance activities that can be performed on test code:

- *Perfective maintenance:* modifications performed only to improve the quality of test code, e.g. refactoring;

- *Corrective maintenance:* modifications performed to fix bugs in test code;
- *Adaptive maintenance:* modifications performed to follow the evolution of the AUT;
- *Preventive maintenance:* modifications performed to remove smells or redundancies, and not after the actual detection of defects.

The element of novelty in the derivation of the taxonomy performed in this section of the study is the application of the Grounded Theory technique, and the derivation of a fine-grained set of categories for modification causes that are specific to Android development.

8.1.1 Grounded Theory and Taxonomies

The way this study has been performed follows the methodological guidelines and quality criteria described by Ralph [87] for obtaining process theories and taxonomies in Software Engineering. A critical review is also given by Stol et al. [93] about the usage of Grounded Theory as a tool for Software Engineering research, along with a literature review of studies leveraging such method.

The Grounded Theory approach has been primarily introduced by Glaser and Strauss [44] and is a qualitative method to generate theory through continuous and progressive analysis and comparison of available data. In its original form, the research carried using the Grounded Theory method does not contemplate answering a Research Questions which is defined a-priori, but the Research Question emerges itself during the data analysis steps.

Grounded Theory has then undergone a revision by Strauss and Corbin [94] that led to the Straussian Grounded Theory: this less stressed-out version of the technique allows the a priori definition of a Research Question, and the consultation of existing literature during the generation of the survey, instead of limiting only to the analyzed set of data.

Whatever the type of Grounded Theory adopted, the fundamental practices of the methodology are the *coding* activities: *Open Coding*, which based on line-by-line analysis of text data extracts the concepts and categories of the theory; *Axial Coding*, i.e., the process of analysing the categories that have been found to find structures and relationships between them; *Selective Coding*, i.e. the selection of a central category for the taxonomy, to which all other categories can relate.

8.1.2 Diff Files Analysis

For this study, the Straussian definition of Grounded Theory has been adopted, with the definition of a Research Question upfront as a follow-up of the previous parts of this study. In particular, the aim of this part of the study was answering the following two research subquestions to characterize the fragility issue for Android open-source projects:

RQ4.1 Modification Causes: what are the main causes behind the need for maintaining GUI test code in Android open-source projects?

RQ4.2 Fragility: how fragile are test methods and classes to modifications in the AUT or in its appearance?

To construct a taxonomy of modification causes from the bottom up, the Straussian definition of the Grounded Theory method has been adopted, with the Research Questions defined upfront following the previous part of the study, and not emerging from the research.

The *site* for the Grounded Theory studies, an organization or group in the original Straussian definition of the technique, can be interpreted in the case of Grounded Theory in Software Engineering as an artifact or a repository of artifacts. In this study, the chosen site is the repository of Android open-source projects mined in the previous step of the study.

Among the possible *Data Collection* strategies identified by Ralph in the guidelines for Grounded Theories in Software Engineering, the selected one for this study was the strategy of *Technical observation*, defined as *"accessing, creating or copying digital artifacts such as source code, unit tests, server logs or database entries"*. In particular, the copied digital artifacts were the diff files computed for each test class of the mined repositories, for each transition between consecutive releases in which they were featured.

Starting from modified lines in test methods, the corresponding production classes and layout files have been individuated and examined, to understand what was the underlying reason for each modification emerging from diff files. The inspection, hence, moved from the usage of widgets inside the test classes to the layout files where such widgets were defined, that were inspected to find changes in the definition, properties, and arrangement of the widgets; then, the activities in charge of inflating the identified layouts were also inspected, to understand whether the modifications in the layouts or test code were paired with changes in the production code. When, on the other hand, the modifications in test methods were not evidently linked to widgets of the user interface, the search for modified lines of code was not propagated to layout files and production code, and the modification was flagged as pertaining to test code only.

Following the described inspections, the categories of the taxonomy were generated incrementally through *Open Coding*, with each modified test method being classified under one or more classes of the taxonomy, that were thus not deemed as mutually exclusive (i.e., two or more different causes can concur to a single modification operated on a test method). The open coding procedure involved two iterations over the collected set of diff files. *Axial Coding* was used then to find macro-categories of modifications in the taxonomy.

The taxonomy building procedure was applied over four different sets of diff files, related to Android repositories that featured the Espresso, UI Automator, Robolectric and Robotium testing tools. The application of the taxonomy over a considerably high amount of diff files generated with four different tools proved also as a conceptual evaluation of the transferability of the taxonomy. Percentages of occurrence were gathered for each of the defined categories of the taxonomy, in order to find the most common causes for maintenance in Android test code (and hence answering RQ4.1).

Finally, the modification causes have been split between modification causes related to test code only, and modification causes related to changes in the AUT or, more specifically, to its GUI appearance or definition; the latter ones have been deemed as fragile according to our definition of fragile test cases of section 2.6.1. This way, an estimation of the fragility of the test suites obtained with the selected GUI automation frameworks was obtained (and hence, RQ4.2 was answered).

8.1.3 Threats to Validity

Threats to Internal validity

The analysis of diff files in existing Android projects has been conducted at a release granularity, considering all the tagged releases of projects hosted on GitHub. The commit granularity would take in consideration also smaller and/or temporary modifications; hence, the results in terms of frequencies of maintenance causes may vary sensibly.

The scripts and tools used for the inspection of diff files, and the individuation of modifications inside test methods, assume that there are no syntax errors inside test classes; the correctness of the extraction of modified methods – and thus of the diffs considered for the inspection of maintenance cause – is thus not ensured for any project.

Threats to External Validity

The findings are based solely on projects hosted on the GitHub open-source project repository. Even though the set of projects is very large and varied in terms of types of applications, it is not assured that the findings can be generalized to closed-source Android applications, or to other sets of open-source applications. This particularly applies to the frequencies of maintenance causes, that can vary significantly if test classes are produced using different testing tools.

8.2 Results

This section illustrates the taxonomy of modification causes that has been derived applying the procedure described in section 8.1.

The open coding procedure was applied on all the diff files containing modifications in test methods for Android open-source projects featuring code written with Espresso (819 diff files), Robotium (424 diff files) and UI Automator (59 diff files). The set of diff files of projects associated with Robolectric was instead sub-sampled, due to the size that was excessive for manual examination. This selection led to 422 randomly extracted diff files out of the full set of 4221 (10%). To sum up, the open coding procedure involved the manual examination of a total of 1724 diff files.

8.2.1 Modification Causes

All causes for modifications in test cases that have been found, and the macro-categories in which individual causes are grouped, are described hereafter. The 28 individual causes have been divided into nine macro-categories.

Only the first one of the categories, namely *Test Code Change*, is not related to the AUT. The macro-categories *Application Logic Change*, *Execution Time Variability*, *Compatibility Issues* are related to the AUT, but not specifically to its GUI. The remaining five categories, namely *GUI Interaction Change*, *GUI Views Arrangement*, *View Identification*, *Access to Resources* and *Graphic Changes*, are strictly related to changes in the graphical appearance of the AUT.

The taxonomy is shown graphically in figure 8.1, with the individual categories graphically divided in macro-categories, and the three groups of macro-categories described before depicted using different colors.

Test Code Change

To this macro-category of modifications are assigned all the changes that are performed in test code without any link to maintenance in the application code or in the GUI definition and appearance of the app. Those modifications are only related to how test cases are defined, set up and executed, and hence are cases – in the prior classification given by Yusifoglu et al. [101] – of Perfective and/or Corrective maintenance.

Test Logic Change. Modifications in the test code and in the usage of the GUI automation frameworks inside the examined test methods. For instance, different functions belonging to the adopted test framework can be used, or adaptations may be needed due to the natural evolution of the GUI testing frameworks used.

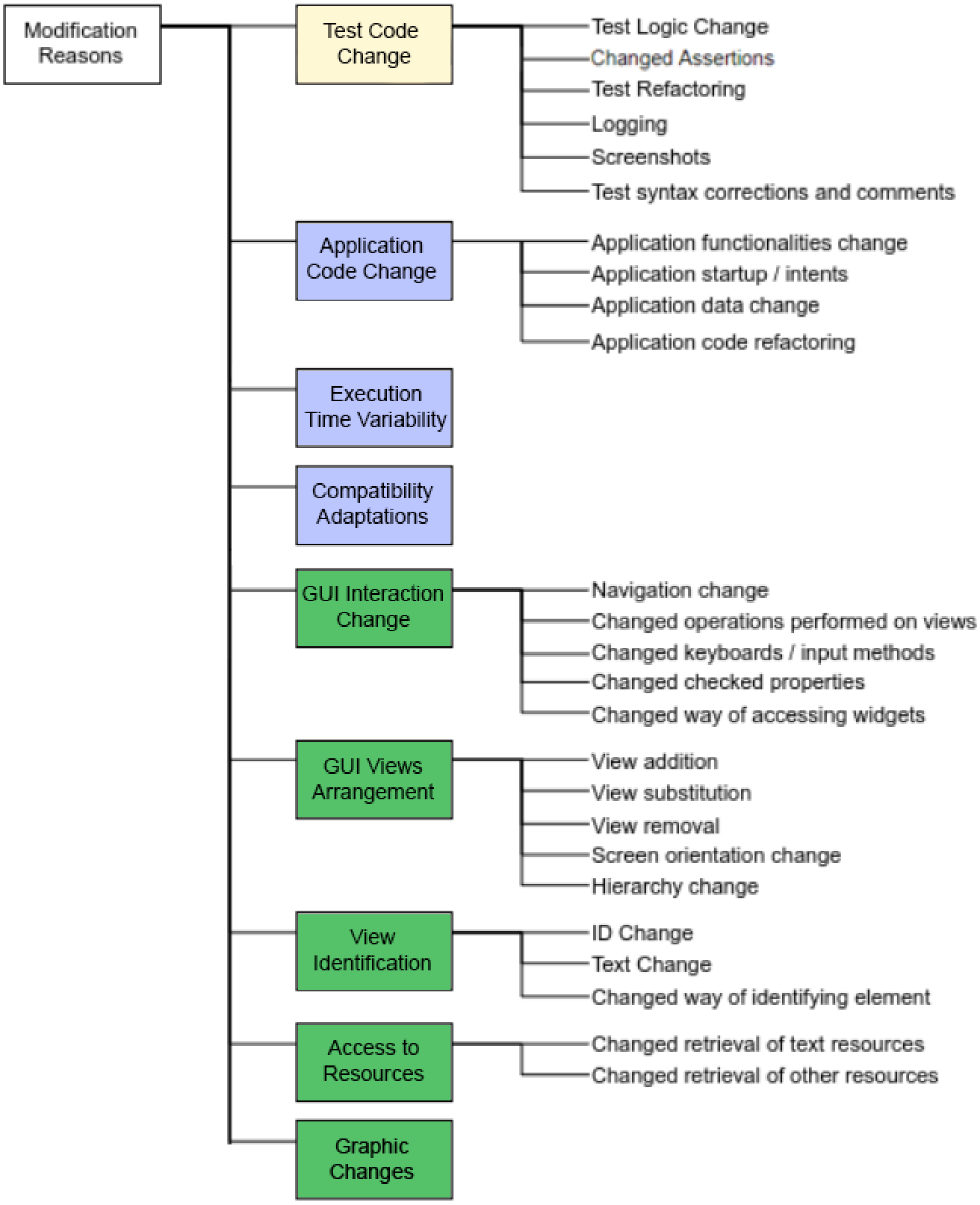

Modification Reasons
Test Code Change
Test Logic Change
Changed Assertions
Test Refactoring
Logging
Screenshots
Test syntax corrections and comments
Application Code Change
Application functionalities change
Application startup / intents
Application data change
Application code refactoring
Execution Time Variability
Compatibility Adaptations
GUI Interaction Change
Navigation change
Changed operations performed on views
Changed keyboards / input methods
Changed checked properties
Changed way of accessing widgets
GUI Views Arrangement
View addition
View substitution
View removal
Screen orientation change
Hierarchy change
View Identification
ID Change
Text Change
Changed way of identifying element
Access to Resources
Changed retrieval of text resources
Changed retrieval of other resources
Graphic Changes

Fig. 8.1 Graphic taxonomy of modification causes

Changed Assertions. Modifications in the assertions that are checked in the same test method, or in the sequence of oracles that are verified during the test case, as if the use case on which the test case is based is changed slightly.

Test Refactoring. This category includes all the refactoring modifications that are performed on test code, without any influence from modifications (including refactoring) operated on production code. Examples of this category are modifications of the names of the variables and/or functions declared inside test methods, or the creation of helper functions in test classes to simplify the code of existing methods and make it less redundant.

Logging. Addition, removal or modification of logging operations inside test methods, using the built-in Logcat tool of the Android Development Bridge or third-party logging tools.

Screenshots. Screen captures are used to create test traces that can be analyzed after the execution of test suites. This category of modification causes includes additions, removals or changes of the places where screenshots are taken inside test methods.

Test Syntax Corrections and Comments. Modifications only in the syntax of test classes/methods (e.g., adding white lines or spacing inside brackets, adding or removing comments).

Application Code Change

All the changes related to production code – but not related to the appearance of the app – contribute to this macro-category of modification causes for test code. Here are considered all the functionality changes inside the app, like the addition or modification of methods inside activities, or changes in the data model managed by the app. This macro-category of modifications cover Yusifoglu's Adaptive maintenance category when the adaptations are limited to the app features and not to its GUI appearance and definition.

Application Functionalities Change. Changes in the functionalities provided by the application, in classes and methods that do not pertain to the graphical appearance of the app. An example of this category can be a modification in the way a connection to a remote server is performed.

Application Startup / Intents. Changes in the definitions of the activities, in the parameters exchanged between them, and in the operations that must be performed at the startup (or teardown) of each screen.

Application Data Change. Changes in the data models, classes and objects used by the activities of the app. Those changes must be reflected by test code if operations involving data have to be performed.

Application Code Refactoring. Refactoring operations performed in the code of the AUT, and that must be reflected by test code (e.g. changed names of activities, methods, data structures).

Execution Time Variability

This category of modifications is related to the adaptations to the changed amount of time required by the app for performing an operation. Changes in the execution time may be due to the behaviour of the app (e.g., time for establishing a network connection or to download a picture from a database) or to the appearance of the GUI (e.g., changed duration of an animation).

If the tests are performed on real handheld devices and not on emulated ones, this issue may become more serious because of the concurrence with multiple other applications in the same system.

Maintenance due to Execution Time Variability is identified in diff files of test classes by finding addition, deletion or modifications of Thread.sleep instructions, like in the following excerpt:

```
– Thread.sleep(500);
+ Thread.sleep(1000);
```

Compatibility Adaptations

Under this category are classified all the modifications in test code that are made in order to guarantee compatibility with different versions of the Android OS, or to take into account features of the new releases of the OS. Often, those modifications are in the form of adoption of new classes for the same widget because of deprecated previous ones.

For instance, the following diff file excerpt contains a modification in a test method due to a different orientation behaviour shown by the app to comply with the GUI of varying OS versions:

```
– rotateToPortrait(this);
+ if (VERSION.SDK_INT >= VERSION_CODES.JELLY_BEAN_MR2) {
+     rotateToPortrait(this);
+ }
```

GUI Interaction Change

Under this category are classified all the modifications in test code that reflect changed interactions with unaltered screen appearance, e.g., changes in the order of operations on the widgets, changes in the interactions supported by the same widget, changes in the way to access some specific view of the GUI.

Navigation Change. This category contemplates modifications in test code due to changed order interactions with the views of a tested activity, without the views themselves being altered.

The following diff file excerpt shows the effect that the necessity of an additional click on a second button – already present on-screen – has on the corresponding test method, developed with Espresso.

```
+ onView(withId(R.id.connectButton)).perform(click());
  onView(withId(R.id.startButton)).perform(click());
```

Changed Operations Performed on Views. Changes in test code due to different gestures to be performed on the same widgets (e.g., long clicks instead of normal clicks), without the widgets being altered.

In the following diff file excerpt, an example regarding the addition of a click operation on a widget, using the Espresso GUI Automation Framework, is shown.

```
– Espresso.onView(withId(R.id.fitnessProgramButton))
      .perform(ViewActions.scrollTo());
+ Espresso.onView(withId(R.id.fitnessProgramButton))
      .perform(ViewActions.scrollTo(), click());
```

Changed Keyboards / Input Methods. Modifications in the way the software keyboard of the application is accessed, used or removed from the interface.

For instance, in some diff files with Espresso test code, the call to a function for closing the software keyboard explicitly has to be added:

```
– InputMethodManager manager = (InputMethodManager) view
–     .getContext().getSystemService(Context.INPUT_METHOD_SERVICE);
– manager.toggleSoftInput(InputMethodManager.SHOW_FORCED, 0);
+ mCloseSoftKeyboard.perform(uiController, view);
```

Changed Checked Properties. Changes in the properties that are checked on the widgets of the user interface in different releases of the same test case. The properties need not be only graphic, but may also be related to the description of a widget.

In the following diff file excerpt, the properties that are checked for an element of the interface now include also the contained text and not only its position on the screen:

```
  onData(anything())
    .inAdapterView(withId(android.R.id.list))
–   .atPosition(24);
+   .atPosition(24)
+   .check(matches(withText("purus")));
```

Changed Way of Accessing Widgets. The category contemplates changes in the way the same type of widgets is accessed by specific operations in the user interface (e.g., show the widget from the context menu instead of normally inside the inflated layout of the activity).

In the following example, the way to access the menu of the application through Espresso functions is changed:

```
– onView(withId(R.id.console_flip))
    .perform(pressMenuKey());
+ openActionBarOverflowOrOptionsMenu
    (InstrumentationRegistry.getTargetContext());
```

GUI Views Arrangement

As opposed to the previous macro-category, related to changed operations on unchanged widgets, this macro-category covers all the modifications in the type and number of elements that compose the tested activities.

View Addition. It may be possible that new elements are added in the visual hierarchy of the activity to test, even though they are not essential for the completion of the tested functionalities. Those elements may need initialization values that may make test cases working on the activities fail. Modifications caused by View Additions have been identified by examining layout files relative to the tested Activities, and verifying that the operations added in the new release of the test class are on widgets that were not present in the previous version of the layout file.

A possible automated solution to this kind of modification is the creation of methods to fill automatically the newly added widgets in the tests with default values, if it is fundamental to populate them.

View Substitution. Views can be substituted between two consecutive releases of the application, with other ones having similar functionalities. For instance, a TextView may be changed to an EditText view, and the test code may need to be changed accordingly (e.g., in the retrieval of the pointer to the view).

Modifications caused by View Substitutions have been identified by examining layout files relative to the tested Activities, and verifying that the operations changed in the new release of the test class are on widgets whose type or characteristics have been changed with respect to the previous version of the layout file.

View Removal. Between different releases of the same app, it may occur that an element of a screen is removed or moved to another activity. Consequently, a test that has to use it is invalidated.

Modifications caused by View Removals have been identified by examining layout files relative to the tested Activities, and verifying that the operations removed in the new release of the test class are on widgets that were present in the previous version of the layout file, and that have been removed.

Screen orientation change. Operations with the orientation of the application may need to be added in test methods, to comply with similar modifications in the production code. The orientation change is considered among the GUI Arrangement Change macro-category because of the possibility that a change in the screen orientation may lead to a complete rearrangement of the screen shown by the current activity.

Hierarchy Change. Changes in the definition of layouts used by activities, and in the arrangement between widgets of the user interface. For instance, the same activity may be re-arranged using a ConstraintLayout instead of a RelativeLayout or LinearLayout in the passage to a new version, without modifications in the functionalities offered or in the widgets it contains; another example is the movement of a widget from one layout to another inside the same activity.

As in the following diff file excerpt, modifications in test methods due to Hierarchy Change can be linked to changed parents or views that are related to the widgets interacted in test code:

```
   expectVisible(viewThat(
-      hasAncestorThat(withId(R.id.attribute_symptoms_onset_days)),
+      hasAncestorThat(withId(R.id.attribute_weight)),
```

View Identification

This category applies to all modifications that are due to changes in the identifiers (either text-based or id-based) that are used for finding the widgets inside the currently inflated layout. Also, the cause of a modification is considered as View Identification if the chosen way to identify an individual view changes (e.g., id to text).

ID Change. Elements can be identified in visual hierarchies of the application through the use of the (optional) unique ID that can be attributed to them, either programmatically with Java code or in the layout .xml files. A test that detects elements by their identifier is invalidated if they are changed.

A first possible guideline to avoid fragilities due to changes in the IDs of the widgets is to use semantic IDs that clearly describe the functionalities of the widgets, and that are not related to their position in the layout arrangement or appearance, nor randomly generated. This way, even though the operations on a widget are changed, or the widget is moved inside the layout, it is unlikely that its ID will have to change.

The following diff file excerpt shows the effect that a variation in the ID of an unchanged element has on a test method developed with Espresso:

```
- onView(withId(R.id.morse_input_text_card))
+ onView(withId(R.id.morse_input_text_container))
        .check(matches(isDisplayed()));
```

Text Change. Elements that do not possess a unique identifier, but contain text, can be detected by their textual description. This case is frequent in menus where options have no individual identifier but obviously show distinct textual descriptions. This strategy is not robust for tests, because the textual attributes are more likely to change during the evolution of the app (and not only: for instance, they also depend on the device language) than identifiers, so tests must be modified at any change of the textual content of the widgets.

It is worth highlighting that image recognition testing tools – like Sikuli – which cannot rely on identifiers to discriminate between the elements of the GUI are particularly subject to this kind of fragility (as they are with pure graphical modifications).

In this category also fall the modifications of the text that is expected to be given as input to a text view of the user interface. Screen name changes are also subcases of the Text Change category.

A possible guideline to avoid this fragility is to always use String resources to identify text so that a modification in the String resource file has no impact in the management of test cases and classes.

The following diff file excerpt shows an example of modification in plain text used by a test case to identify a widget:

```
- onView(withText("No Account has been added yet"))
        .check(matches(isDisplayed()));
+ onView(withText("No account has been added yet"))
        .check(matches(isDisplayed()));
```

Changed Way of Identifying Elements. The way in which widgets are retrieved may need to change between consecutive releases of the app. For instance, it may be possible that a view, once referred by its ID, is now referred by text, or class name, or other properties.

In the following example, the original text contained in a text view is no longer set as *text* but as a *hint* in the new release; the diff file excerpt highlight the corresponding modification in the Espresso test method:

```
- onView(withText("Log In")).perform(click());
+ onView(withHint("Log In")).perform(click());
```

Access to Resources

Resources, mainly text, can be used as oracles and hence loaded and confronted with the proper appearance they should have inside test methods. The place and the identifiers with which the oracles (if there are any) are addressed may change between consecutive releases of the app, and hence test methods need to be changed accordingly.

Changed Retrieval of Text Resources. Text resources can be defined in several ways: Strings can be hardcoded, defined as constants inside Java classes, or as resources in the ”strings” .xml file in the ”res” folder of the Android project. This makes it difficult to maintain several classes that work on the same logical content (i.e., when a hardcoded string is modified in the product code, all the test classes using it have to be modified accordingly). Strings can also be defined as constants inside Java classes; if this approach is adopted, a proper refactoring and usage of constants can avoid fragility in relative test cases, when text values are changed. However, the best practice for identifying text resources is to use the ”strings” .xml file in the “res” folder of Android apps, so that in each (test) class of the app the proper string to be shown or tested can be referenced by its unique – and unchanging – ID.

When the way text resources are defined and accessed changes between two consecutive releases of the app, and even if the contained text does not change, it is likely that test classes have to be modified to reflect the modifications in the production code.

The following diff file excerpt shows the changes in an Espresso method due to the the change of access to a text resource through a String identifier, instead of the previously used hardcoded text:

```
- onView(withText("Coupon")).perform(click());
+ onView(withText(R.string.category_coupon))
      .perform(click());
```

Changed Retrieval of Other Resources. The way graphic resources are accessed in the production code may change (e.g. using the root view inside a fragment, or accessing them through identifiers declared in .xml resource files). This can apply, for instance, to colors used for the graphic appearance of the widget, to drawable images or to fonts used in TextViews.

In the following diff file excerpt, the way a graphic characteristic of the activity (a font size) is retrieved is changed, and the modification propagates to a test method using it:

```
- PreferencesState.getInstance().
     setScale(Constants.FONTS_LARGE);
+ PreferencesState.getInstance().
     setScale(getActivityInstance().getString(R.string.
     font_size_level2));
```

Graphic Changes

Even though the widgets are not entirely modified, small modifications in their appearance (e.g. animations, transparencies, themes, absolute coordinates, sizes) can invalidate tests, especially if they are based on graphic recognition, or on exact coordinates of the position of the widgets on the screen (i.e., tests are coordinate-based).

The following diff file excerpt shows the modifications that have to be performed when an element of the interface is identified through its exact coordinates, that are changed between two consecutive releases of the app:

```
- final float screenX = screenPos[0]
    + x * (view.getWidth() / gameSize);
- final float screenY = screenPos[1]
    + y * (view.getHeight() / gameSize);
+ final float screenX = screenPos[0]
    + (0.5f + x) * (view.getWidth() / gameSize);
+ final float screenY = screenPos[1]
    + (0.5f + y) * (view.getHeight() / gameSize);
```

Answer to RQ4.1: Examining a set of 1724 diff files related to Espresso, UI Automator, Robotium and Robolectric, 28 different possible causes were identified for modifications of test methods developed for Android apps with the use of GUI Automation frameworks. Nine different macro-categories of change reasons were identified: changes in the functions and logic of test code, changes in the application functionalities, changes in the interaction with the GUI, varied arrangements of the widgets of the layout, changed identification of views, changed retrieval of resources, pure graphic changes, execution time variations, and adaptations to provide compatibility with different OS versions.

8.2.2 Diffusion of Modification Causes and Fragility Occurrences

After the identification of the classes of the taxonomy of modification causes, an analysis of the frequency of occurrence of each type of modification cause was performed, in order to quantify the most common reasons for maintenance of test classes for Android applications.

Table 8.1 reports the absolute and relative frequency of occurrence for any category of modification causes, for four different testing tools (namely, Espresso, UI Automator, Robotium and Robolectric), upon the considered context of mined Android repositories. The first row (*Total Classes*) shows the number of diff files that have been considered, after the test class filtering operated in previous parts of the study and a subsampling of the test classes featuring Robolectric. By construct, the frequencies in the columns of the table do not necessarily sum up to 100%: this is due to the fact that the causes of modification were not considered as mutually exclusive, hence multiple different causes can concur to the same maintenance operation on a test method.

Test Logic Change was a fairly common modification reason for all the four testing tools that were considered. This highlights a high frequency of situations in which the maintenance made on test code is related to changes in the way the functions of the specific tools are used. Modifications caused by *Assertions Change*, related to changes in the design of test cases, happened more rarely, suggesting that the use cases of the apps – on which the tests are based – were rather stable during the evolution of the projects. *Test Refactoring* modifications, albeit being common for all the four testing tools, were more frequent for Robolectric and Robotium test

Table 8.1 Absolute (relative) frequency of occurrence of modification causes

		Espresso	UI Automator	Robotium	Robolectric
Total Classes		819	59	424	422
Test Code Change	Test logic change	130 (15.85%)	10 (17.95%)	87 (20.52%)	126 (29.86%)
	Assertions Change	21 (2.56%)	4 (6.78%)	19 (4.48%)	23 (5.42%)
	Test refactoring	41 (5.24%)	3 (5.08%)	76 (17.92%)	76 (18.01%)
	Logging	20 (2.44%)	1 (1.69%)	2 (0.48%)	5 (1.18%)
	Screenshots	20 (2.44%)	0 (0.00%)	4 (0.94%)	0 (0.00%)
	Test syntax corrections and comments	49 (5.98%)	9 (15.25%)	52 (12.26%)	53 (12.00%)
Application Code Change	Application functionalities change	146 (17.80%)	0 (0.00%)	42 (9.90%)	57 (13.44%)
	Application startup / intents	56 (6.83%)	2 (3.38%)	15 (3.53%)	19 (4.48%)
	Application data change	4 (0.49%)	1 (1.69%)	1 (0.23%)	14 (3.30%)
	Application code refactoring	24 (2.93%)	3 (5.08%)	49 (11.56%)	35 (8.25%)
Execution Time Variability	Sleeps add	29 (3.54%)	8 (13.56%)	33 (7.78%)	0 (0.00%)
	Sleeps change	28 (3.41%)	1 (1.69%)	12 (2.83%)	0 (0.00%)
	Sleeps removal	22 (2.68%)	2 (3.38%)	9 (2.12%)	1 (0.23%)
Compatibility Adaptations		8 (0.98%)	3 (5.08%)	0 (0.00%)	9 (2.12%)
GUI Interaction Change	Navigation change	76 (9.27%)	12 (20.34%)	43 (10.14%)	11 (2.59%)
	Changed operations performed on views	14 (1.71%)	0 (0.00%)	4 (0.94%)	2 (0.47%)
	Changed keyboards / input methods	12 (1.46%)	1 (1.69%)	1 (0.23%)	0 (0.00%)
	Changed checked properties	15 (1.83%)	0 (0.00%)	3 (0.71%)	2 (0.47%)
	Changed way of accessing widgets	63 (7.68%)	0 (0.00%)	16 (3.77%)	0 (0.00%)
GUI Views Arrangement	View Addition	14 (1.71%)	2 (3.38%)	4 (0.94%)	1 (0.23%)
	View substitution	6 (0.73%)	2 (3.38%)	4 (0.94%)	5 (1.18%)
	View removal	3 (0.37%)	1 (1.69%)	0 (0.00%)	0 (0.00%)
	Screen orientation change	3 (0.37%)	0 (0.0%)	1 (0.23%)	0 (0.00%)
	Hierarchy change	2 (0.24%)	1 (1.69%)	7 (1.65%)	0 (0.00%)
View Identification	ID Change	62 (7.56%)	0 (0.0%)	2 (0.48%)	12 (2.83%)
	Text Change	41 (5.00%)	9 (15.25%)	18 (4.24%)	4 (0.94%)
	Changed way of identifying elements	31 (3.78%)	4 (6.78%)	15 (3.54%)	0 (0.00%)
Access to Resources	Changed retrieval of text resources	35 (4.27%)	1 (1.69%)	12 (2.83%)	6 (1.41%)
	Changed retrieval of other resources	10 (1.22%)	0 (0.0%)	6 (1.41%)	5 (1.18%)
Graphic Changes		14 (1.71%)	1 (1.69%)	2 (0.48%)	11 (2.59%)

methods, suggesting a higher complexity of the test cases that would hence need more frequent fixes. A relevant amount of the modifications were only linked to changes in the syntax, documentation and comments of test code (more than 15% for diff files pertaining to UI Automator test classes).

The modification in test methods that were linked to *Application Code Change* had a minor frequency of occurrence than the ones related to *Test Logic Change*. The most relevant causes in terms of modifications triggered were *Application Functionalities Change* and *Application Code Refactoring*, while changes in *Application Startup/Intents* proved to be a frequent cause for test maintenance especially for Espresso test classes, which have a strong coupling with the activities of the tested application. The examination of diff files was performed without executing the tests, and taking into account only the last cause of a possible chain of different reasons for test maintenance. This could lead to possible overlaps between modifications associated with *Application Logic Change* or to *Test Logic Change*, in case of unclear link between the final modification in the test logic, and preexisting modifications in application logic.

Modifications linked to *Execution Time Variability* occurred rather rarely in the modified methods examined. The use of sleep instructions also had a different application for the considered testing tools: Espresso and Robotium wait by default for layouts to be populated and views to appear in their final state, so in test methods written with these tools the sleep instructions are inserted only to wait for long tasks (e.g., waiting for the response from a Service); on the other hand, explicit sleep instructions are needed by UI Automator to wait for the rendering of the screen. As expectable, hence, modifications due to *Execution Time Variability* were more frequent in test classes developed with UI Automator. On the other hand, Robolectric is used for developing test classes that are run on the Java Virtual Machine, without the need for instantiating any emulator or for connecting to a real device: this can be considered as the reason for missing occurrence of modifications due to *Execution Time Variability* for classes featuring such tool. *Compatibility Adaptations* were another quite rare motivation for the maintenance in test code. This is mainly due to the retrocompatibility that is guaranteed most of the time by new releases of the classes of the Android Frameworks.

Among the modification causes related to the GUI of Android apps, the most frequent were the one classified under the macro-category *GUI Interaction Changes*.

In particular, the most common cause among them was the one due to changes in the navigation inside the activities, i.e. the order of the operations performed on the widgets of the inflated layouts. Overall percentages of occurrence for the macro-category were high for all testing tools except Robolectric. In general, test code developed with Robolectric appeared to need way less maintenance after modifications performed in the GUI appearance or definition of the tested apps. Modifications in *GUI Views Arrangement* were less common that changes in the interaction with GUI elements. A possible explanation of fewer modifications in test methods in response to addition or removal of widgets can be that a new widget (or a deleted one) may be reflected in test suites with the addition of a new test method (or the deletion of an existing one).

ID Change was the most frequent modification causes among the ones related to changes in the way the widgets are retrieved. The modification cause proved very frequent, especially for Espresso test classes. On the other hand, the examined UI Automator test methods were modified only for changes in the text they contained. *Text Change* had generally a high frequency of occurrence, with the only exception represented by test classes featuring Robolectric (less than 1%). Changes in the *Access to Resources* (e.g., changes of the retrieval of a textual resource from using plain text to leveraging String resources) were not so common as causes of maintenance in test methods, with a maximum frequency of 4.27% again for Espresso test classes regarding the retrieval of text resources, and of 1.41% for Robotium test classes regarding the retrieval of any other kind of resources.

Graphic Changes, i.e. modifications that are related only to aesthetic variations in the graphical appearance of the app, was a rare macro-category of modifications for all the considered testing tools, with a maximum frequency of occurrence of 2.59% (for the set of diff files of Robolectric test classes). This low occurrence was also expected since all the considered tools work at a lower level of abstraction of the GUI, and hence should not be affected by modifications in the final appearance as it is shown to the user.

A higher-level statistic about the modifications triggering maintenance in test classes is reported in table 8.2, which shows, for each set of diff files of test classes associated with a Layout-based testing tool: the frequency of modification whose causes are not related to the AUT (i.e., modifications that have been classified as *Test Code Change*); the frequency of modifications whose causes are related to the

Table 8.2 Frequency of occurrence of modification causes

	Espresso	UI Automator	Robotium	Robolectric	Average
Causes not related to the AUT	32.80%	44.07%	53.53%	65.80%	46.36%
Causes related to the AUT	72.07%	69.49%	56.37%	39.86%	60.23%
of which GUI-related	69.07%	70.72%	46.44%	31.35%	54.32%

AUT; among the latter ones, the percentage of causes that are related to the GUI appearance. The table also reports, in the last column, the average value for the three fractions, weighted by the number of diff files examined for each of the considered testing tools.

Non-AUT related changes were frequent for all the sets of projects that have been considered, with an average 46.36% frequency of occurrence. Espresso had the lowest value among the testing tools considered, which can be justified with an easier development of test code with Espresso, and with an usage of the tool for simpler test suites, that have less need for maintenance than the ones developed with the other testing tools considered.

Causes related to the AUT had an average frequency of occurrence of 60.23%, with higher values measured for Espresso and UI Automator. The same trend was shown by the fraction of those modifications that were related to the GUI appearance. The smallest frequencies were measured for Robolectric test classes (40% frequency of occurrence for AUT-related modifications, and 31% of them concerning the GUI). This low frequency can be explained with the multi-purpose nature of the Robolectric framework, which can also be used solely for traditional GUI testing.

Answer to RQ4.2: A percentage of about 60% of modifications due to changes in the AUT, and hence of fragile classes, was measured. On average upon all the diff files examined, more than 50% of the modifications on test classes triggered by changes in the AUT were connected to the GUI of the app or to its appearance. However, test suites were modified often for reasons that were not connected to changes in the AUT: 46.36% of the modified diff files that were examined featured changes that were local to test code and that could not be backtracked to variations in the AUT.

Chapter 9

Study 5: Layout-based vs Generated visual test cases: An experiment with TOGGLE

As detailed in the previous sections, the available automated GUI testing techniques exhibit several shortcomings, especially in terms of the maintainability of test suites and usability perceived by testers/developers. The most advanced testing techniques, e.g. model-based ones, still fall short in being adopted widely in industrial settings, with developers mostly sacrificing test automation due to the required effort for creating test suites and their costly maintenance, and to the specific drawbacks of each testing technique or generation.

More specifically, 2^{nd} and 3^{rd} generation GUI testing techniques provide different benefits and drawbacks and are seldom used together because of the aforementioned costs, despite growing academic evidence of the complementary benefits.

This section describes the design of TOGGLE, a tool for a translational approach for GUI testing, that enables users to define and translate Layout-based to Visual scripts and vice versa, to gain the benefits of both generations, whilst minimizing the drawbacks. The details of the implementation of the translator from 2^{nd} to 3^{rd} generation test scripts are presented, along with a tentative architecture for the backward translator (from 3^{rd} to 2^{nd} generation). The motivating example and the first design of TOGGLE have been presented at the 2018 edition of the INTUITESTBEDS workshop [18].

The tool TOGGLE adopts several elements of the approach used by PESTO [92, 63], which implements the translation-based approach for Selenium test suites for Web Applications, translating them to Sikuli. The translation-based approach has not yet been explored for the mobile domain. Several differences in the approach used for testing native Android apps and web-based applications differentiate the two tools:

- The need for properly instrumenting the Android environment and the execution of Espresso tools;
- The specific syntax of Espresso test cases;
- The different way in which Android layouts are described;
- The significantly higher amount of interactions that can be performed on Android widgets with respect to those that can be performed on the elements of a web application.

9.1 Motivation

Visual GUI testing (3^{rd} generation) techniques are in general less common among practitioners from the industry, typically for their lack of robustness when compared to Layout-based testing (2^{nd} generation) techniques, and for their slowness when compared to other forms of automation techniques.

In contrast, Layout-based techniques work at a different level of abstraction of the GUI of the SUT, and hence cannot completely emulate a real user's interaction with the GUI of the application, or verify the app's appearance as shown to the human user.

Research, thus, has highlighted that a combined/hybrid approach is required, where both the generations are used in parallel by the researcher [9]. A hybrid approach, however, would need the tester/developer to have knowledge about both generations of test automation, and about the syntax used by tools pertaining to both the approaches. The proposed TOGGLE tool leverages instead an automated translation of test scripts, from one generation to the other, through systematic reuse of test logic and mapping of Layout-based locators to visual locators (and vice-versa). A translational approach would allow the tester/developer to focus on manually developing the test suite in one methodology only, and then automatically generate the counterpart.

The translational approach can mitigate challenges with Visual GUI test scripts such as their fragility to visual changes (e.g., size, resolution, color, etc.), whilst also mitigating challenges with Layout-based test scripts that are sensitive to changes in Java code and .xml definitions of layouts (e.g., IDs, text, descriptions of widgets, etc).

Instead, the techniques are respectively robust to changes in the definition of layouts, and in pure visual changes. When one generation of test scripts is broken due to specific fragility issues, test scripts of other generations – which could be still valid – can be translated into new valid test scripts. This allows a reduction of the overall fragility of developed test suites and by consequence a reduction of the cost of maintenance and repair of test cases throughout different versions of the same application, which comes particularly in handy when test cases are used to perform regression testing. The single case in which the translational approach would not be able to repair a fragile test case would be the simultaneous presence of changes in the graphic appearance and in the layout properties for the same elements of the user interface: in that case, both generations of test cases interacting with the same elements would be unable to retrieve it through the use of changed identifiers and graphics.

Existing literature has already explored the possibility of automatically repairing test scripts, without however performing translations to other abstractions of the GUIs: the works by Memon [76] and Zhang et al. [103] moved towards this direction.

In addition to the repair of fragile test scripts, a translational approach can provide many other advantages. In the following, a summary of the benefits of such an approach is given:

Automated creation of test scripts: Creation of Layout-based tests when only Visual tests are available for the SUT, and vice-versa. Such an automated creation is expected to reduce development costs of testware, since the use of the tool for the translation – even though some adjustments of the derived test scripts may be required – is expected to require significantly less effort than writing the same test suite with another tool.

Reduced maintenance for failing locators: The automated analysis of failing locators (either for Visual or for Layout-based test scripts) can lead to repair of parts of test scripts with locators that are invalidated by modifications in the layout definitions and appearance.

Reduced impact of fragmentation: Visual scripts for different devices/configurations can be generated from the same Layout-based tests, using the unchanged layout properties to obtain visual properties and visual oracles that are specific to the individual devices/configurations on which the applications will be deployed. A single test can hence be run on a set of emulated devices to obtain Visual GUI test scripts specific to each of them.

Enhanced testing of hybrid and web-based applications: Visual locators and oracles can be used in combination with Layout-based locators when the SUT contains WebViews or hybrid components with screens loaded at runtime from the web. The use of visual locators can be a solution to the missing native locators for components of the loaded web pages.

9.1.1 Motivating Example: a test script for K-9 Mail

A motivating example of the possibility of repairing layout-based or visual fragilities of an existing test suite through translation has been created with K-9 Mail (version V5.500-Snapshot).

In the following discussion, the original version of the application – as cloned from the GitHub repository – is called *v1*. A sample test case has been developed, to exercise the authentication feature of the application with two tools pertaining to different generations: Espresso for the generation of Layout-based testing tools, and EyeAutomate for the generation of Visual testing tools. All the tests have been run on an Android Virtual Device (AVD), namely a Nexus 5 with Android API 24 installed, with enabled device frame and hardware keyboard inputs.

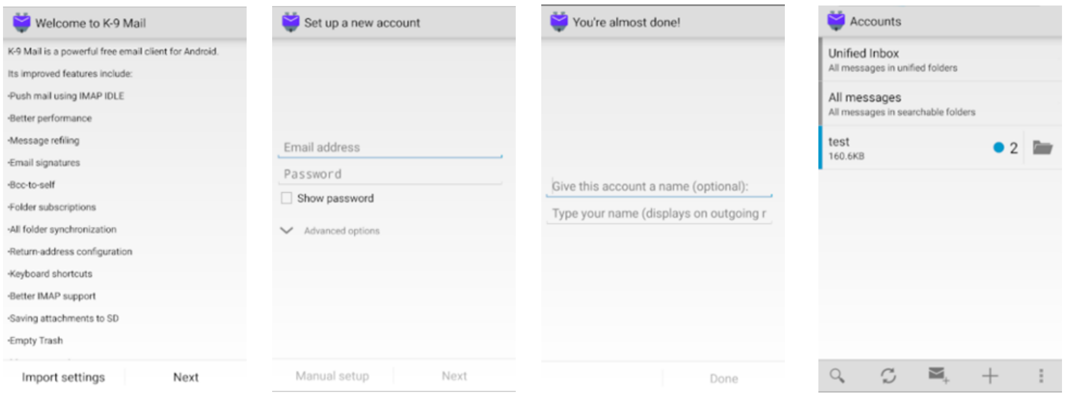
The considered use case is the wizard that is started at the first launch of K-9 Mail. Table 9.1 shows the steps of the use case, whilst figure 9.1 shows the four screens that are traversed by the use case, with the respective Activity names indicated in the captions. The use case traverses a first WelcomeMessage Activity; in the second Activity encountered, AccountSetupBasics, the user inputs his/her e-mail address and

Table 9.1 TOGGLE motivating example: Steps for the Authentication use case of K-9 mail

Step	Screen	Widget Description	Operation
1	s1	Next Button	Click
2	s2	Email account Form	Type test account email
3	s2	Password Form	Type test password
4	s2	Next Button	Click
5	s3	Account description Form	Type test account description
6	s3	Account name Form	Type test account name
7	s3	Done Button	Click
8	s4	Activity Title	Check that "Accounts" is shown
9	s4	Account List Item	Check that test name is shown

Table 9.2 TOGGLE motivating example: Retrieved IDs for Layout-based test case

Step	Object Description	Object ID
1	Next Button	next
2	Email account Form	account_email
3	Password Form	account_password
4	Next Button	next
5	Account description Form	account_description
6	Account name Form	account_name
7	Done Button	done
8	Activity Title	action_bar_title_first
9	Account List Item	description



(a) WelcomeMessage Activity (b) AccountSetupBasics Activity (c) AccountSetupNamesb Activity (d) Messages Activity

Fig. 9.1 TOGGLE motivating example: Screens and Activities traversed by the authentication use case

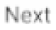
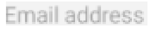
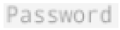
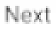
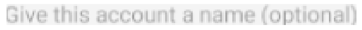
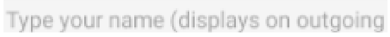
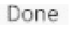
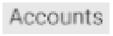
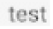
password; after going (through the use of the Next button) to the AccountSetupName Activity, the user inputs his/her desired account name and nickname for managing the e-mail account in the K-9 mail client; finally, the app shows the Messages activity (which is initially empty, if no message has not been downloaded yet by the client).

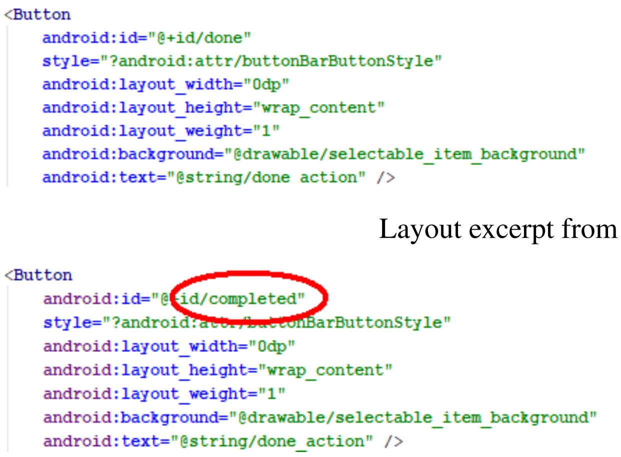
Test scripts definition for the original release

The Layout-based test script for the Authentication test script has been developed using the Android Studio IDE, and inside the K-9 Mail application project, having full access to the AUT production code and the .xml layout files describing its user interface. The resource IDs used in the Layout-based test script have been collected by launching the application and using the UI Automator Viewer tool, retrieving the "resource-id" field for any of the widgets that had to be interacted. Table 9.2 shows the retrieved ids for all the elements interacted throughout the execution of the test case. The test script, when executed on *v1*, runs to completion.

The test script has been developed also leveraging the Visual testing technique, with the EyeAutomate testing tool. Table 9.3 shows the retrieved images that were used as visual locator for executing the same test case on the app. All the reference images were gathered from a first manual execution of the use case on the emulated application, leveraging the image capturing tool embedded in the EyeStudio suite. The interaction points inside the visual locators were fixed to the center of the images

Table 9.3 TOGGLE motivating example: Retrieved images for the EyeAutomate test script

Step	Reference Image
1	 Next
2	 Email address
3	 Password
4	 Next
5	 Give this account a name (optional)
6	 Type your name (displays on outgoing
7	 Done
8	 Accounts
9	 test



```
<Button
    android:id="@+id/done"
    style="?android:attr/buttonBarButtonStyle"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:background="@drawable/selectable_item_background"
    android:text="@string/done_action" />
```

Layout excerpt from v1

```
<Button
    android:id="@+id/completed"
    style="?android:attr/buttonBarButtonStyle"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:background="@drawable/selectable_item_background"
    android:text="@string/done_action" />
```

Layout excerpt from v2a

Fig. 9.2 TOGGLE motivating example: Modification in the layout file between v1 and v2a

(the default location of the interaction for each screen taken with the image capturing tool). Sleep instructions have been added at every screen transition to prevent the visual testing tool to search for elements that were not yet been rendered on screen. Executed on *v1*, the Visual test runs to completion.

Layout-based fragility induction

To highlight the fragility of layout-based test scripts to changes in the properties of the interacted widgets, a simple modification on the definition of a layout of *v1* of the considered app has been performed. This way, the version that was called *v2a* is obtained. In particular, the resource ID associated with the *Done* button is changed from ”done” to ”completed” in the layout file (namely, wizard_done.xml) where the button is declared. The modification performed in the layout file is shown in figure 9.2.

In the considered Layout-based test scripts, the Resource IDs are inserted as constants inside the test script. If they are changed externally by a modification of the layout file (and no automated refactoring tools are applied to the application project) manual effort is required to fix the changed IDs in the test script. The same applies to any change in the textual properties of the widgets, like the contained text or the textual description of a button.

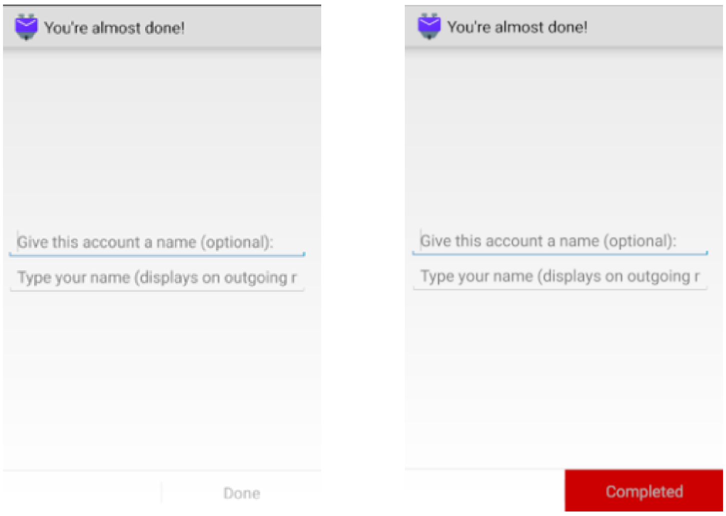
In all transitions like the one from *v1* to *v2a*, containing only modifications in the widget definition and properties, all test cases leveraging Layout-based properties and locators will fail and require maintenance from testers/developers. In the provided example, according to the taxonomy of modifications provided in the previous chapters, the test case fails due to an ID change fragility.

On the other hand, the Visual test runs to completion without errors, since no graphical property has changed for any widget in the screens traversed by the tested use case. Using the translational approach, the Visual test script, which is still valid, can be used to retrieve the actual (changed) id of the *Done* button when it is clicked, generating a valid companion layout-based test case.

Visual fragility induction

To cause fragility in the visual test script, we performed a couple of graphic modifications on the original version *v1*, leading to the version called *v2b*. In particular, the appearance of Screen 3 was changed, modifying the background color and the text of the *"Done"* button. Each of the two modifications performed would be sufficient, if applied alone, to invalidate a visual test case. The changed appearance of the screen between *v1* and *v2b* is shown in figure 6.3.

Without any modification in layout and widget definitions, there is no impact in scripted test cases, since the button to be clicked is unambiguously retrieved by its unchanged ID. On the other hand, the visual recognition test case that we developed experiences a failure, due to the inability to identify an element with the appearance of the Done button, still linked in the test script to the screen capture shown in table 9.3.


You're almost done!
Give this account a name (optional):
Type your name (displays on outgoing r
Done
You're almost done!
Give this account a name (optional):
Type your name (displays on outgoing r
Completed

(a)
Screen 3 in v1

(b)
Screen 3 in v2b

Fig. 9.3 TOGGLE motivating example: Graphic changes in Screen 3 between v1 and v2b

In the envisioned translational approach, using the coordinates of the interaction and possibly additional information extracted from layout files (e.g., size and padding of the interacted widget), a picture of the new appearance of the interacted widget can be cut from the capture of the whole current screen. Hence, a new working Visual test script can be obtained.

9.2 Layout-based to Visual Translator Architecture

Figure 9.4 shows the building blocks of the translator from 2^{nd} to 3^{rd} generation translator, along with the intermediate elements that are generated by each module and consumed as input by the following one. The translator works on Layout-based test cases, that are fed to an Enhancer and then to an Executor. Both those modules are testing tool-specific. The Executor runs in the Android environment, since it requires a connection through ADB to an Android Virtual Device, and feeds the results of the execution of the tests to a tool-agnostic Log Parser Java module. The Log Parser outputs a sequence of abstract interactions to be replicated in the test case, that can be fed to a tool-specific 3^{rd} generation test case creator to encode the test script in the desired syntax.

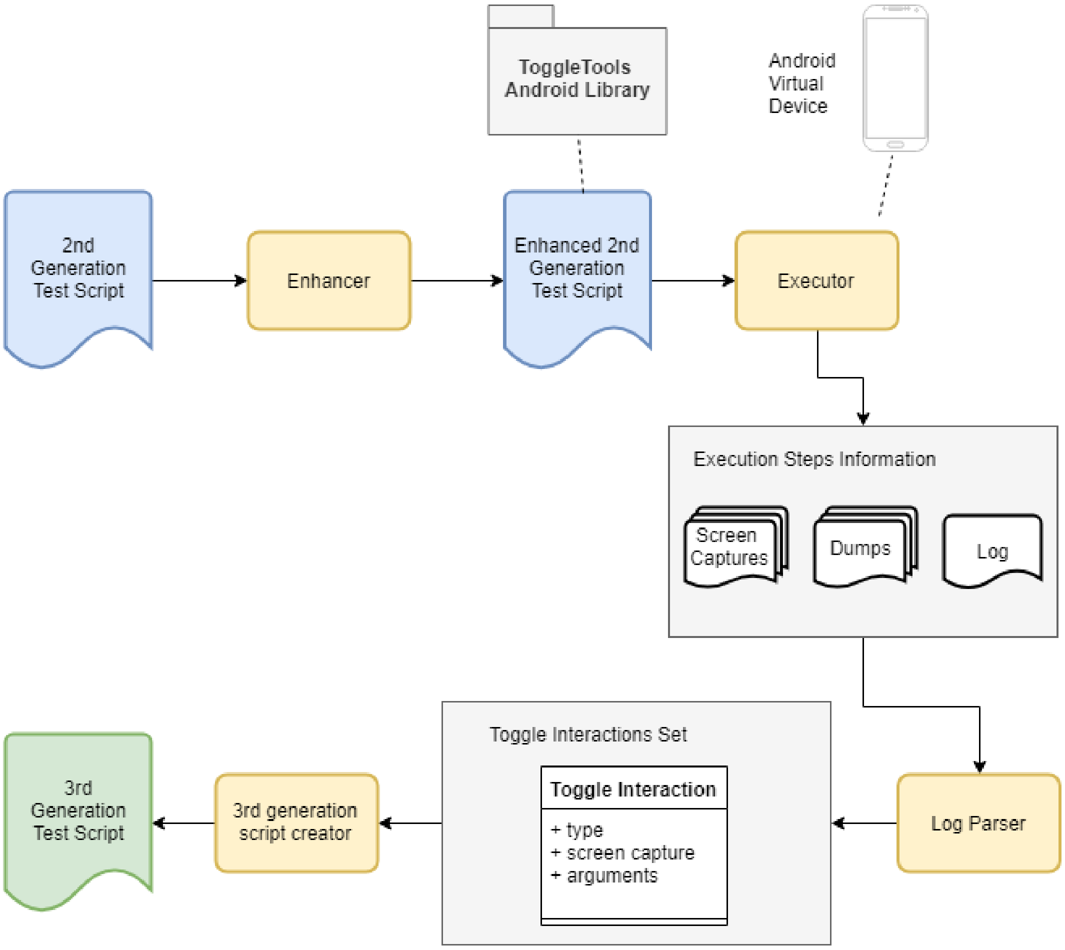


Fig. 9.4 TOGGLE: Architecture of 2nd to 3rd generation translator

In the following, the individual building blocks of the architecture are described in detail.

9.2.1 Enhancer

The Enhancer module receives as input a 2nd generation test script, written according to a given syntax, and parses it in order to find the operations that are performed inside it and add calls to the TOGGLETools Android library, to perform at execution time the extraction of the information needed for the translation.

The Enhancer is primarily tailored to identify Espresso interactions that are defined starting with an *onView* ViewInteraction, which is the primary interface – offered by the tool – to perform interactions and assertions on individual widgets of the GUI. The *onView* method receives as parameter a ViewMatcher, and each Matcher finds views in the visual hierarchy according to a specific property that can be defined for the view (i.e., IDs, text content, content description). After a view is identified, Espresso allows performing operations on it with the *perform* method, to which a ViewAction is passed. Each ViewAction is specific to a kind of interaction that can be performed on the identified view (e.g., click, double click, insertion or modification of contained text).

The inspection of Espresso test cases has been performed using the JavaParser library[1], identifying method calls corresponding to Espresso ViewMatchers, ViewActions, and ViewAssertions. Currently, the Enhancer supports most of the interactions (each defined by a ViewAction class) that are supported by Espresso, except for the ScrollTo and pressIMEActionButton commands. For what concerns the ViewAssertions, only the isVisible assertion (that checks if a given element appears on screen) has been taken, by now, into consideration by the Enhancer and hence by the following translation phase.

Each time an instruction starting with onView is identified in the code of the test script, the Espresso Enhancer adds calls to three different methods of the TOGGLETools library right before the execution of the operation, to capture information about the screen hierarchy and the actual graphic appearance of the current activity. Specifically, calls to the following methods are added:

TakeScreenCapture The method, whose prototype is

```
public static Bitmap TakeScreenCapture(Date curr_time ,
Activity activity),
```

receives as parameters the current time, and an instance of the current activity. It takes a capture of the current screen of the application, which is returned in a Bitmap file and saved in the external storage of the Android Virtual Device, named after the current timestamp of the Android device.

DumpScreen The method, whose prototype is

```
public static String DumpScreen(Date curr_time ,
UiDevice device),
```

[1] https://github.com/javaparser/javaparser

receives as parameters the current time, and an instance of the UiDevice class of the UIAutomator library. It takes a dump (a .xml description of the layout) of the current activity and saves it in the external storage of the Android Virtual Device, named after the current timestamp of the Android device.

LogInteraction The method, whose prototype is

```
public static void LogInteraction(Date curr_time,
String search_type, String search_keyword,
String interaction_type, String interaction_params),
```

uses the built-in LogCat tool to Log information about the operation that is performed, and the way the element of the screen has been identified. The *search_type* parameter identifies the type of search that has been performed (e.g., "id", "text", "content-desc"); the *search_keyword* is the specific keyword that has been searched in the layout to identify the view on which to operate; the *interaction_type* identifies the operation that is performed on the view (e.g., "click", "type-text"); the *interaction_params* string contains optional parameters (divided by a ";" character if multiple) that may be needed to describe the performed interaction (e.g., the text to type in case of "type-text" interaction).

The information is logged using the Log.d built-in function, adding the "TOGGLELOG" keyword to allow subsequent filtering of the lines of interest from the complete log that is relative to the application.

The logged arguments for the translated interaction types are reported in table 9.4. Appendix C reports details about the operations performed by the translated Espresso commands.

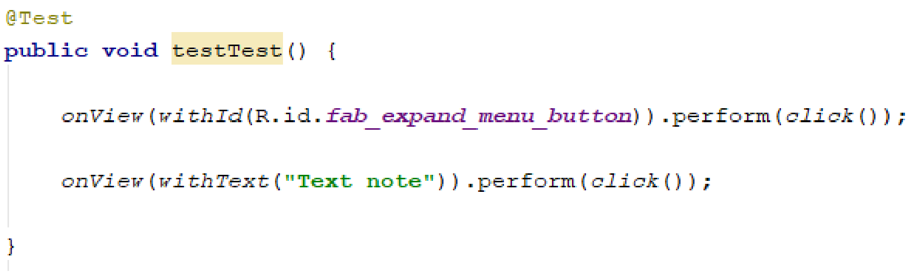
The output of the Enhancer module is an *Enhanced 2nd Generation Test Script*, which can be executed as a normal test on the application run in the Android Virtual Device, but that is now able to trigger the collection of the information to the translation, in addition to execute all the steps of the original test case.

Figure 9.5 and 9.6 show a sample excerpt of a test file, respectively in its original form and in the enhanced form after the use of the Enhancer module.

The Enhancer module also adds checks according to the check operations that are present in the original test case. A check for the appearance of the whole screen

Table 9.4 TOGGLE - Enhancer: Arguments for translated interaction types

Espresso interaction	Arguments
clearText()	i. Text length
click()	None
closeSoftKeyboard()	None
doubleClick()	None
longClick()	None
openActionBarOverflowOrOptionsMenu(Context context)	None
openContextualActionModeOverflowMenu()	None
pressBack()	None
pressBackUnconditionally()	None
pressKey(int keyCode)	i. Keycode
pressKey(EspressoKey key)	i. Keycode
pressMenuKey	None
replaceText(String stringToBeSet)	i. Text Length; ii. stringToBeSet
swipeDown()	None
swipeLeft()	None
swipeRight()	None
swipeUp()	None
typeText (String stringToBeTyped)	i. stringToBeTyped
typeTextIntoFocusedView(String stringToBeTyped)	i. stringToBeTyped



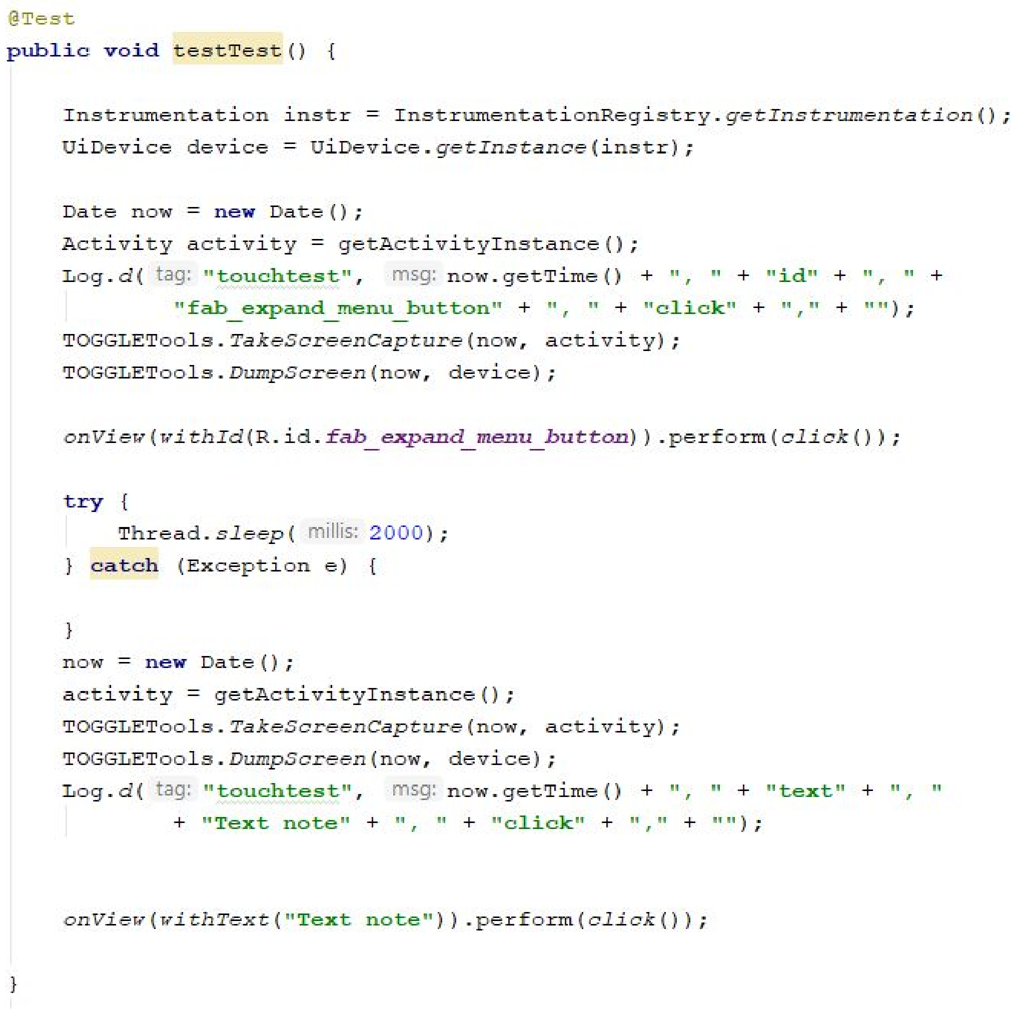
```
@Test
public void testTest() {

    onView(withId(R.id.fab_expand_menu_button)).perform(click());

    onView(withText("Text note")).perform(click());

}
```

Fig. 9.5 TOGGLE - Enhancer: Sample input Espresso test script



```
@Test
public void testTest() {

    Instrumentation instr = InstrumentationRegistry.getInstrumentation();
    UiDevice device = UiDevice.getInstance(instr);

    Date now = new Date();
    Activity activity = getActivityInstance();
    Log.d( tag: "touchtest",  msg: now.getTime() + ", " + "id" + ", " +
            "fab_expand_menu_button" + ", " + "click" + "," + "");
    TOGGLETools.TakeScreenCapture(now, activity);
    TOGGLETools.DumpScreen(now, device);

    onView(withId(R.id.fab_expand_menu_button)).perform(click());

    try {
        Thread.sleep( millis: 2000);
    } catch (Exception e) {

    }
    now = new Date();
    activity = getActivityInstance();
    TOGGLETools.TakeScreenCapture(now, activity);
    TOGGLETools.DumpScreen(now, device);
    Log.d( tag: "touchtest",  msg: now.getTime() + ", " + "text" + ", "
            + "Text note" + ", " + "click" + "," + "");


    onView(withText("Text note")).perform(click());

}
```

Fig. 9.6 TOGGLE - Enhancer: Sample enhanced Espresso test script

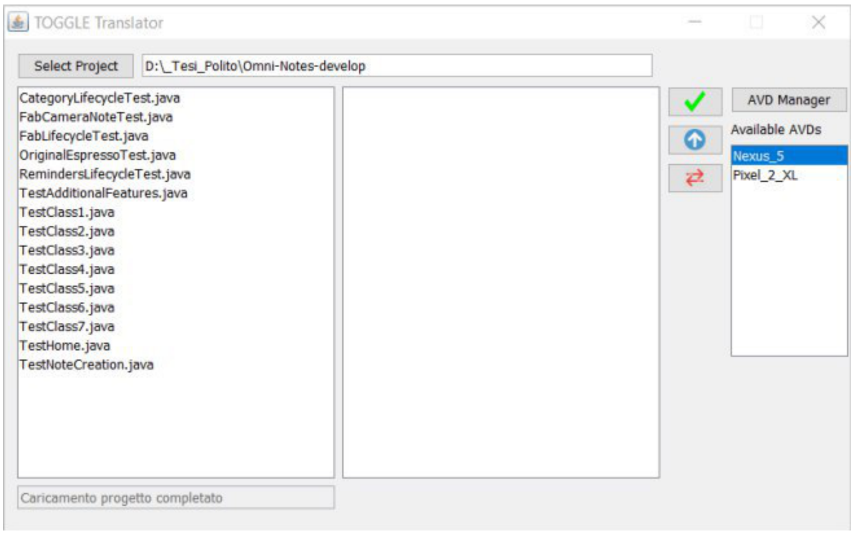
Name	Size (inches)	Resolution (pixels)	Density
Pixel XL	5,5”	1440x2560	560dpi
Pixel 2 XL	5,99”	1440x2880	560dpi
Pixel 2	5,0”	1080x1920	420dpi
Pixel	5,0”	1080x1920	xxhdpi
Nexus S	4,0”	480x800	hddpi
Nexus One	3,7”	480x800	hddpi
Nexus 6P	5,7”	1440x2560	560dpi
Nexus 6	5,96”	1440x2560	560dpi
Nexus 5X	5,2”	1080x1920	420dpi
Nexus 5	4,95”	1080x1920	xxhdpi
Nexus 4	4,7”	768x1280	xhdpi
Galaxy Nexus	4,65”	720x1080	xhdpi

Table 9.5 TOGGLE: devices supported by the Executor for test case execution

is added by default at the end of the test case. This is a design decision that has been made in order to have a check of the final state of the GUI at the end of each translated test execution. Visual testing tools, in fact, may perform mouse operations on wrong elements of the screen, if the image recognition engine they use mismatches a portion of the screen with a given screen capture. In those cases, if there are no explicit checks after those wrong interactions, the test case is considered passing. On the other hand, it is highly likely that interactions on wrong parts of the user screen lead to a final state of the application that is different from the one reached after the execution of the original 2nd-generation test script. A final check using the full screen of the app as an oracle is hence an added layer of robustness for the generated test case.

9.2.2 Executor

After the test scripts are enriched with calls to the TOGGLETools library with the Enhancer module, the Executor module is in charge of executing them on the selected Android Virtual Device. The Executor is in charge of launching an AVD, installing the AUT on it and executing the test cases. The device does not need to be rooted, given that the AUT is provided with the required storage permissions. Android Debug Bridge (ADB) commands are used to perform such operations. The module also ensures that the Android project is instrumented correctly and includes all



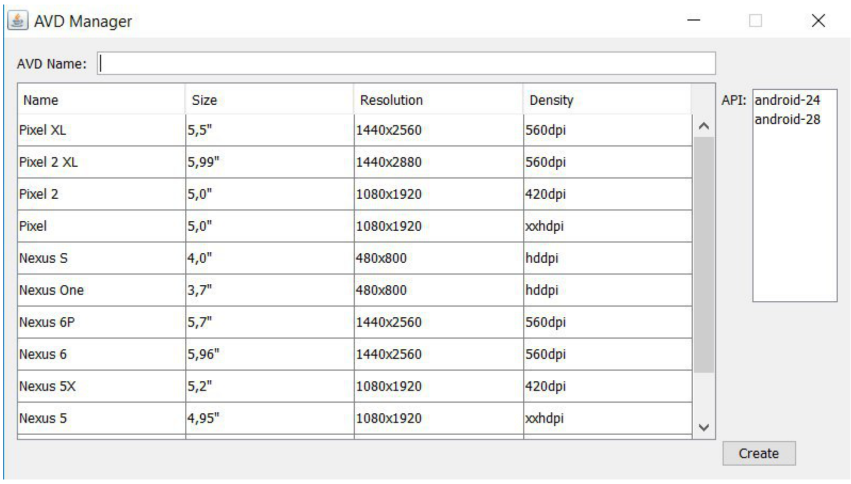
D:_Tesi_Polito\Omni-Notes-develop

CategoryLifecycleTest.java
FabCameraNoteTest.java
FabLifecycleTest.java
OriginalEspressoTest.java
RemindersLifecycleTest.java
TestAdditionalFeatures.java
TestClass1.java
TestClass2.java
TestClass3.java
TestClass4.java
TestClass5.java
TestClass6.java
TestClass7.java
TestHome.java
TestNoteCreation.java

Nexus_5
Pixel_2_XL

Caricamento progetto completato

Fig. 9.7 TOGGLE - Executor GUI: project selection



AVD Name:

Name	Size	Resolution	Density
Pixel XL	5,5"	1440x2560	560dpi
Pixel 2 XL	5,99"	1440x2880	560dpi
Pixel 2	5,0"	1080x1920	420dpi
Pixel	5,0"	1080x1920	xxhdpi
Nexus S	4,0"	480x800	hddpi
Nexus One	3,7"	480x800	hddpi
Nexus 6P	5,7"	1440x2560	560dpi
Nexus 6	5,96"	1440x2560	560dpi
Nexus 5X	5,2"	1080x1920	420dpi
Nexus 5	4,95"	1080x1920	xxhdpi

API: android-24
android-28

Fig. 9.8 TOGGLE - Executor GUI: AVD creation

libraries required by Toggle. Table 9.5 reports the Android Virtual Devices supported by the Executor at the current state of development. The Executor module has been provided with a GUI that allows to select an Android project from the file system, identify all the test cases available inside it, and proceed to launch and translate them individually (see figure 9.7). Another screen of the executor is used to instantiate Android Virtual Device through the GUI, instead of defining them through ADB commands manually (see figure 9.8).

During the execution, the TOGGLETools methods are called, and hence the screen captures and dumps are collected inside the device external storage, along with the collection of the information about the list of interactions inside the log of the application. The interactions are stored in an intermediate, and tool agnostic, script format that can be translated to any syntax required by the 3rd generation tools currently supported by Toggle. Theoretically, this tool agnostic list of interactions could be used, if needed, to translate the sequence of interaction to another 2nd generation test script as well.

The Executor also checks the outcome of the original 2nd generation test: if the test triggers any exception (failed test), the developer is notified and the translation process is aborted. This feature is added to minimize translations of invalid tests.

Figure 9.9 shows a sample screen capture, taken for the Main Activity of the Omni Notes Android app. In the screen, the button for opening the menu to add a new text note in the library can be noticed (the red button with the "+" icon in the bottom-right corner).

Figure 9.10 shows a sample dump, taken for the same activity shown in figure 9.9. In the dump provided by the UIAutomator library, each view in the visual hierarchy is identified by a node in the .xml file, and the contained attributes allow to identify the view among the others. For instance, in the dump excerpt, the id of the menu button (*fab_expand_menu_button*) is highlighted. Each node also contains the coordinates (top, left, bottom, right) of the rectangle that is occupied by the view on the current visual hierarchy,

Figure 9.11 shows an excerpt of the output of the Logcat after the execution of a test script starting from the Main Activity of Omni Notes, and after the filtering of the Log (searching for the "TOGGLELOG" keyword).

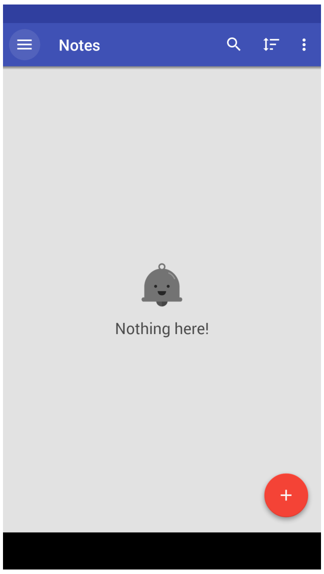
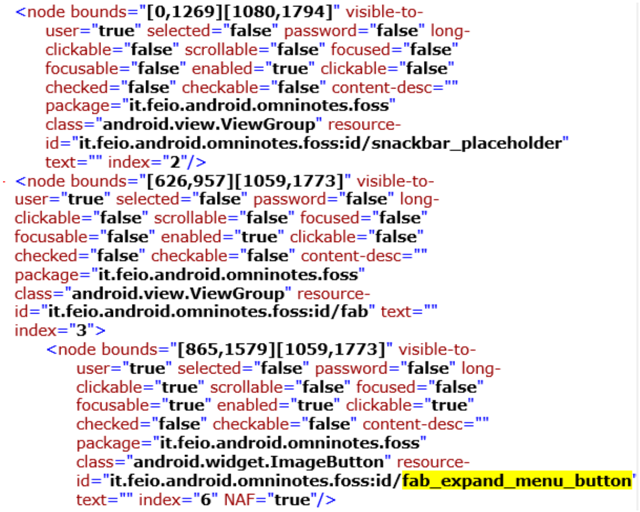
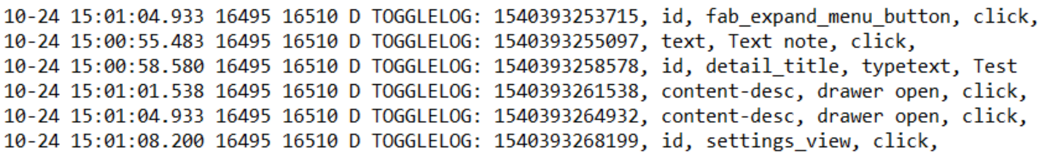


Fig. 9.9 TOGGLE - Executor: Screen Capture extracted for the Main Activity of the Omni Notes application



```
<node bounds="[0,1269][1080,1794]" visible-to-
    user="true" selected="false" password="false" long-
    clickable="false" scrollable="false" focused="false"
    focusable="false" enabled="true" clickable="false"
    checked="false" checkable="false" content-desc=""
    package="it.feio.android.omninotes.foss"
    class="android.view.ViewGroup" resource-
    id="it.feio.android.omninotes.foss:id/snackbar_placeholder"
    text="" index="2"/>
<node bounds="[626,957][1059,1773]" visible-to-
user="true" selected="false" password="false" long-
clickable="false" scrollable="false" focused="false"
focusable="false" enabled="true" clickable="false"
checked="false" checkable="false" content-desc=""
package="it.feio.android.omninotes.foss"
class="android.view.ViewGroup" resource-
id="it.feio.android.omninotes.foss:id/fab" text=""
index="3">
    <node bounds="[865,1579][1059,1773]" visible-to-
        user="true" selected="false" password="false" long-
        clickable="true" scrollable="false" focused="false"
        focusable="true" enabled="true" clickable="true"
        checked="false" checkable="false" content-desc=""
        package="it.feio.android.omninotes.foss"
        class="android.widget.ImageButton" resource-
        id="it.feio.android.omninotes.foss:id/fab_expand_menu_button"
        text="" index="6" NAF="true"/>
```

Fig. 9.10 TOGGLE - Executor: Screen Dump extracted for the Main Activity of the Omni Notes application (excerpt)



```
10-24 15:01:04.933 16495 16510 D TOGGLELOG: 1540393253715, id, fab_expand_menu_button, click,
10-24 15:00:55.483 16495 16510 D TOGGLELOG: 1540393255097, text, Text note, click,
10-24 15:00:58.580 16495 16510 D TOGGLELOG: 1540393258578, id, detail_title, typetext, Test
10-24 15:01:01.538 16495 16510 D TOGGLELOG: 1540393261538, content-desc, drawer open, click,
10-24 15:01:04.933 16495 16510 D TOGGLELOG: 1540393264932, content-desc, drawer open, click,
10-24 15:01:08.200 16495 16510 D TOGGLELOG: 1540393268199, id, settings_view, click,
```

Fig. 9.11 TOGGLE - Executor: Log extracted after the execution of a test script on the Omni Notes application

From the Log it is evident that six operations are performed on the application: a click on a button identified with the id *fab_expand_menu_button*, a click on a button identified by the contained text *Text note*, the insertion of the string *Test* in a textbox identified by the id *detail_title*, two consecutive clicks on a view identified by the content description *drawer open*, and a final click on a view identified by the id *settings_view*. It can also be noticed that the last field of the Log format (the *interaction_params* field) is empty for all interactions, except for the *typetext* interaction.

9.2.3 Log Parser

The Log parser module is in charge of extracting all the information related to the operations performed on the application. It is launched after the execution of the test cases is finished on the Android Virtual Device.

The first operation performed by the Logcat Parser is an access to the full log of the Android app through the Android Debug Bridge, using the command *adb logcat -d* to locally save the full log. Then, the full log is filtered for rows containing the "TOGGLELOG" keyword.

Hence, for each line in the filtered logcat, a ToggleInteraction object is created. As shown in figure 9.12, the class is characterized by the following attributes:

packagename : A string with the packagename of the tested application. It is used for searching ids or resources in the dump .xml files.

search_type : A string indicating the type of search that is performed to identify the interacted element in the visual hierarchy.

search_keyword : A string indicating the keyword used to identify the interacted element in the visual hierarchy, according to the type of attribute searched.

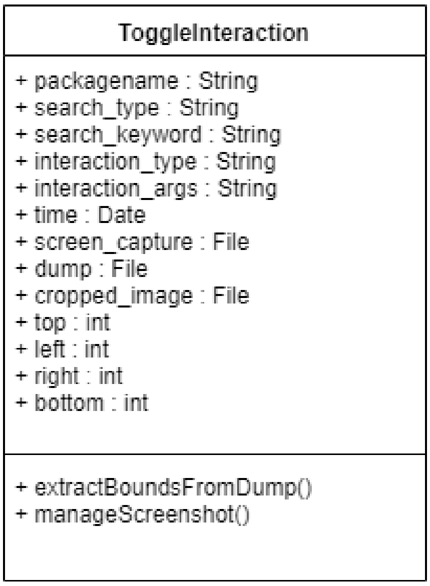


Fig. 9.12 TOGGLE - Log Parser: ToggleInteraction Class

time : A string containing the timestamp of the moment at which the interaction has been performed.

interaction_type : A string indicating the type of interaction performed on the view.

interaction_args : A string containing one or multiple arguments to describe the interaction that is performed on the view.

screen_capture : A File pointer to the Bitmap capture of the full screen, taken right before the interaction is performed.

dump : A File pointer to the .xml file containing the full dump of the current activity, taken right before the interaction is performed.

left, top, right, bottom : Coordinates for identifying the corners of the rectangle occupied by the view on the screen.

cropped_image : A bitmap of the actual appearance of the view that has been interacted, extracted from the full screen capture.

Most of the attributes of the ToggleInteraction object are populated by its constructor, by passing the string elements extracted from each filtered Logcat line. The private method *extractBoundsFromDump()* of the ToggleInteraction class allows obtaining the exact coordinates inside the screen of the interacted view. They are obtained through recursive inspection of the .xml dump file, searching the innermost attribute *search_type* with value *search_keyword* and returning the value of the *node-bounds* attribute.

Once the boundaries are obtained, the *manageScreenshot()* method cuts the rectangle with the given boundaries from the full screen capture of the current activity and saves it in an image which is named *timestamp_cropped.png*.

Several interactions do not require movements of the mouse pointer or click operations, and hence do not require a screenshot to be managed, cut and resized. For instance, a TypeTextIntoFocusedView interaction will be translated in just a set of key presses, without any click on the app GUI. In those cases, the manageScreenshot() method is not called and a screen capture is not created for the specific interaction.

The output of the Logcat Parser module is a sequence of interaction types, coupled with exact screen captures (when required) of the views that have to be found by the Visual Testing Tool for the execution of the test case, and with the required arguments by the specific interaction types.

9.2.4 3^{rd} generation script creator

The $3^{rd}GenerationScriptCreator$ module is dependent on the Visual testing tool towards which the test case is translated. It receives as input a sequence of ToggleInteractions, and translates each operation to the destination syntax, using the set of commands that are available with the destination tool.

For instance, the click operation on a given screen capture can be translated to the following line if the EyeAutomate Visual testing tool is used:

```
Click "image_folder\1540393264932_cropped.png"
```

or, instead, to the following line if the Sikuli Visual testing tool is used:

```
click("1540393255097_cropped.png").
```

In general, however, a 1-to-1 mapping between 2nd-generation layout-based interactions and 3rd-generation image recognition-based is not possible. Espresso, like all 2nd-generation tools, uses platform-specific information to identify the widgets

Table 9.6 Translation alternatives

Name	Meaning
EA	EyeStudio Text Script
S	SikuliX Ide Python Script
EAJ	EyeAutomate Java Method
SJ	SikuliX Java Method
CES	Combined Java Method, EyeAutomate First
CEJ	Combined Java Method, Sikuli First

on which to perform interactions, and several atomic operations may be included in a single Espresso interaction. Those interactions, hence, must be translated to a series of atomic mouse and keyboard operations when they have to be translated to a visual test script operated on an emulated device on a desktop pc.

Toggle supports translation to EyeAutomateì and Sikuli. The translated scripts are in the native formats of the two tools that can be run by the tools' respective IDEs.

However, since both the tools also have Java APIs, the creation of Java code calling the respective APIs has also been considered.

Finally, the Java APIs allow translations of the 2nd generation scripts into *combined* test cases that use both tools, such that if one tool's image recognition fails, the script will try to perform the interaction, or a check, with the other. Two different combined, Java-based, test script types can thereby be obtained, with EyeAutomate interactions first (followed by Sikuli if EyeAutomate fails) and with Sikuli interactions first (followed by EyeAutomate if Sikuli fails).

Table 9.6 summarizes the six possible translations for 2nd generation test cases that are offered by the 3rd generation script creator, along with the acronyms that are used in the continuation of the manuscript.

Section C.2 of appendix C reports the translated commands into the destination EyeAutomate or Sikuli syntax (respectively, in the plain text and python format). In the table the parameters of the commands are indicated, with *img* being the screen capture attached to the log, and *argN* the n-th argument in the log line.

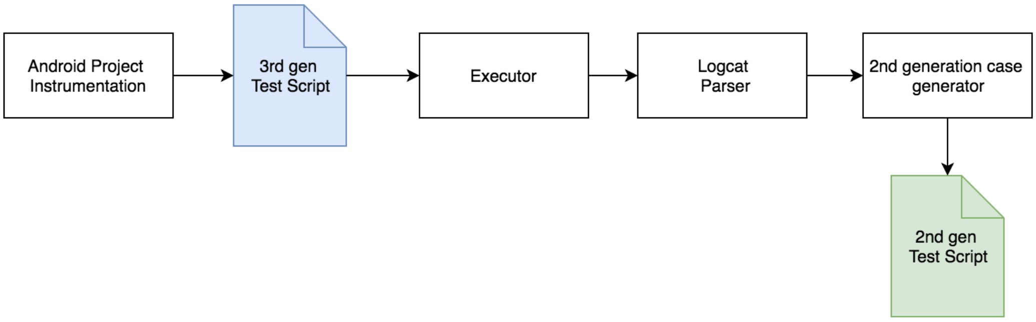

Android Project Instrumentation
3rd gen Test Script
Executor
Logcat Parser
2nd generation case generator
2nd gen Test Script

Fig. 9.13 TOGGLE: Architecture of the translator from Visual to Layout-based GUI testing tools (Proof of Concept)

9.3 Visual to Layout-based GUI test scripts translator (Proof of Concept)

The translation process from Visual to Layout-based GUI test scripts (Figure 9.13) starts from a single script or a suite of scripts created with a Visual GUI testing tool.

The tentative architecture for the not-yet-implemented translator can be described with the following set of separate logic blocks, as shown in figure 9.13.

- In the first phase, an *Instrumentation* of the Android Project is performed. This step is necessary for the translation from visual references to layout-based identifiers. The instrumentation of the Android Project enables (or modifies already existing) callbacks to any widget shown on the screen. The callbacks contain code that logs the interactions that have been performed on the GUI of the tested app during the execution of a visual test script. Future evaluations of the features offered by the available tools (like UI Automator viewer) may make the Instrumentation of the Android Project unnecessary if it is found that sufficient information about the user interface is obtainable without the need for adding callbacks in the app code.

- The *Executor* module runs the Visual GUI tests on the emulated device on the desktop screen. At each step of the test case, as in a typical execution of a Visual GUI test script, the position of the image on the screen and its coordinates are identified. Together with height and width information from the expected image, additional information can be acquired about the interacted element by cross-referencing available information with either Layout-based data or other meta-data. For instance, this can be done leveraging the debug connection with the instrumented device, e.g. using ADB (i.e., Android Debug Bridge) and the Android UI Automator Viewer, which allows navigation of an XML-description of a dump of the current interface shown on the emulated device screen. The extracted data can thereby be used to correlate an element represented by the Visual GUI testing tool as only an image with a Layout-based element represented by a set of properties.

Since the operation of dumping the current screen may be long and requires the UI to remain still, the Executor may need to insert sleep instructions between consecutive operations in the Visual GUI test scripts. Additionally, the transition of certain UI elements might require additional steps to be inserted into the test scripts. For instance, for drop-down lists, Layout-based tools generally access the elements directly without expanding the lists. In contrast, Visual GUI testing tools must first expand the list to make the elements visible to be able to interact with them.

Due to the high abstraction of Visual GUI testing scripts, a proposed technical solution to ease and speed up the translation to Layout-based scripts is to store additional meta-data about existing objects from previously run test scripts (e.g., coordinates, properties, actual appearance). This may enable the association of images in new test scripts with already interacted objects (it is the case, for instance, of buttons that are interacted in two different test cases). Additionally, the meta-data must be aligned with the Layout-based test data to ensure a 1-to-1 association between Layout-based and Visual elements used in the test cases.

- The output of the Executor logic block is a trace of the operations that will compose the translated test script: a log of tuples with the *properties* associated with an identified Visual element, and the *action* performed on the element. This information is given as output through the built-in Logcat tool of Android. The *Logcat Parser* logic block of the translator is in charge of parsing such a trace, in order to obtain a language and technology-independent sequence of operations and widget descriptions that can be then used for the generation of test scripts.

- Finally, the *Test Case Generator*, based on the output of the previous module, creates Layout-based test script in the syntax desired by the user. The generated test script is then merged with existing test cases to be replayed as part of the Layout-based generation test suite counterpart.

9.4 Experimental Validation

An experiment was designed to apply TOGGLE to a set of test cases, in order to verify the dependability and the performance of the generated test cases. In this section, the experiment design and results are detailed. The experiment allowed to answer the high-level research question ***RQ5*** - *What is the dependability and performance of visual test cases generated by translation?*

9.4.1 Experiment Design

To perform the evaluation, TOGGLE was applied on two test suites that were developed on Android open-source applications, available both on GitHub and on the PlayStore: Omni-Notes v6.0.0[2] and PassAndroid v2.5.0[3]. The applications were chosen because of the differences they exhibited in the way their GUIs were built, and in the different operations to perform on the activities to go through their principal usage scenarios.

Each of the test suites was made of 30 independent test cases (i.e., a failure in one test case does not influence the result in test cases that are executed later). Test cases were built based on the Espresso commands recognized by the Enhancer (i.e., all the Espresso ViewActions except for ScrollTo and PressIMEActionButton), and were composed by a number of interactions comprised between 4 and 18, including checks.

Each test case was translated with TOGGLE, to the six destination syntaxes detailed in the previous section. All the generated visual test cases were executed ten times, to evaluate their robustness. The machine on which the executions were performed is an Intel i7-8550U 1.80GHZ clock, with 16GB RAM and Windows

[2] https://github.com/federicoiosue/Omni-Notes

[3] https://github.com/PassAndroid

10 operating system. The emulated AVD for the execution of the apps was a Nexus 5X with API 25 installed, with enabled device frame and enabled keyboard input. Executions of visual test scripts (or Java code embedding image recognition API calls) were performed on a solid black background, to minimize the possible interference of other visual elements appearing on-screen at the same time of the AUT.

RQ5 can be split into two sub-questions, each related to a different non-functional property measured for generated 3rd-generation test cases. First, to understand the dependability and the robustness of generated test suites, we gathered insights about the percentage of failing and passing executions of generated test cases. Hence, RQ5.1 could be formulated as:

RQ5.1 : What are the differences in reliability between the six combinations of visual test script techniques?

To answer RQ5.1, we relied on the Success Rate (SR) metric, that can be computed for each test case as

$$SR_t = N_s / N_{ex}, \tag{9.1}$$

being N_s the number of executions ending with success, and N_{ex} the total number of executions of test case t, in the experiment fixed to 10 for all the generated test cases.

Based on the SR metric, test cases were labeled in three different classes:

- *Passing*, when all 10 executions of the test case ended with success (SR = 1);
- *Failing*, when all 10 executions of the test case ended with failure (SR = 0);
- *Flaky*, when some of the 10 executions of the test case ended with success, and some other with failure (0 < SR < 1).

It is assumed that flakiness is due to imprecisions of the image recognition algorithm, while failing test cases are considered the consequence in translation errors or intrinsic limitations of the visual testing tools. This unpredictability was

Reason	Sleep time
Long-click	600ms
Swipe	200ms
Multiple key press (e.g., Ctrl + M)	20ms
Replace text	50ms
Post-interaction sleep	1000ms
EyeAutomate failure	5000ms
Sikuli failure	5000ms

Table 9.7 TOGGLE: sleep times introduced in generated test scripts

expected, as several studies have reported the inherent uncertainty of the outcomes of Visual GUI test executions, especially for those produced with Sikuli [6].

A Fisher's Exact Test for success (pass or fail) of the test scripts vs. the tool used for the 3rd generation translation was applied, to assess the difference between the alternative tools in terms of the correctness of the execution of the generated test cases.

In addition to the success rate of the generated test cases, the performance of the 3rd generation testing tools was measured and compared to that of Espresso. RQ5.2 can be formulated as:

RQ5.2 : What are the differences in performance between the six combinations of the visual test scripts and the original 2nd generation test scripts?

To answer RQ5.2, the average execution time (T_x) of all the passing test executions was measured. The execution time was normalized by the number of interactions performed inside the test case, in order to make the measures for different test cases comparable. It must also be considered that, by construction of the translated test scripts, the execution time is not comparable with that of Espresso, because of the static sleep instructions that were introduced in the translated interactions, and between each couple of interaction. Table 9.7 reports the added sleep instructions. Sleeps between interactions were added to minimize synchronization challenges, to avoid failures of visual test cases if the visual element on which to interact is not displayed on screen immediately after the execution of the previous interaction. Those sleeps are not needed by Espresso test scripts since the tool automatically waits for the required widgets to be loaded on-screen by the Activity code.

An ANOVA test was applied for the execution time (normalized by the number of interactions). First, the effect of the generation (2nd vs. 3rd) was tested; then the effect of the specific 3rd generation tool combined with the app was tested.

9.4.2 Threats to Validity

Threats to Conclusion Validity

To check the statistically significant difference among different target tools standard statistical tests were applied. The results are clear cut and consistent with the visual representations that report standard (95%) confidence intervals or complete distributions.

Threats to External Validity

The results of this evaluation are not generalizable to any Espresso test suite. Additionally, since the objective of the evaluation is primarily to evaluate the precision of the generated 3rd generation test cases, it did not make sense to use generic Espresso test cases with interactions not supported by the tool.

The conclusions about the reliability and performance of 3rd generation test suites are limited to the considered tools for the evaluation, namely Sikuli and EyeAutomate. The same limited generalizability of the results also applies to the AUTs that were selected. Apps with a very different graphical appearance may induce significantly different results.

Threats to Internal Validity

The results about the performance of the generated 3rd generation test scripts are influenced by the static sleeps added during the translation of 2nd generation test scripts, which by converse need no explicit sleep instructions. In future versions, sleeps may be dynamic, utilizing GUI-state information to determine that components have loaded properly before proceeding. Dynamic sleeps are perceived to help the performance by mitigating unnecessary waiting time between interactions.

The evaluation of the robustness of generated test cases is based on the assumption that all the operations have been performed correctly if the final state of the application is verified. This assumption does not take into account the possibility – albeit unlikely – that multiple wrong operations on the widgets, during a single test case, may compensate each other leading the test case to success at the final visual check.

9.4.3 Experiment Results

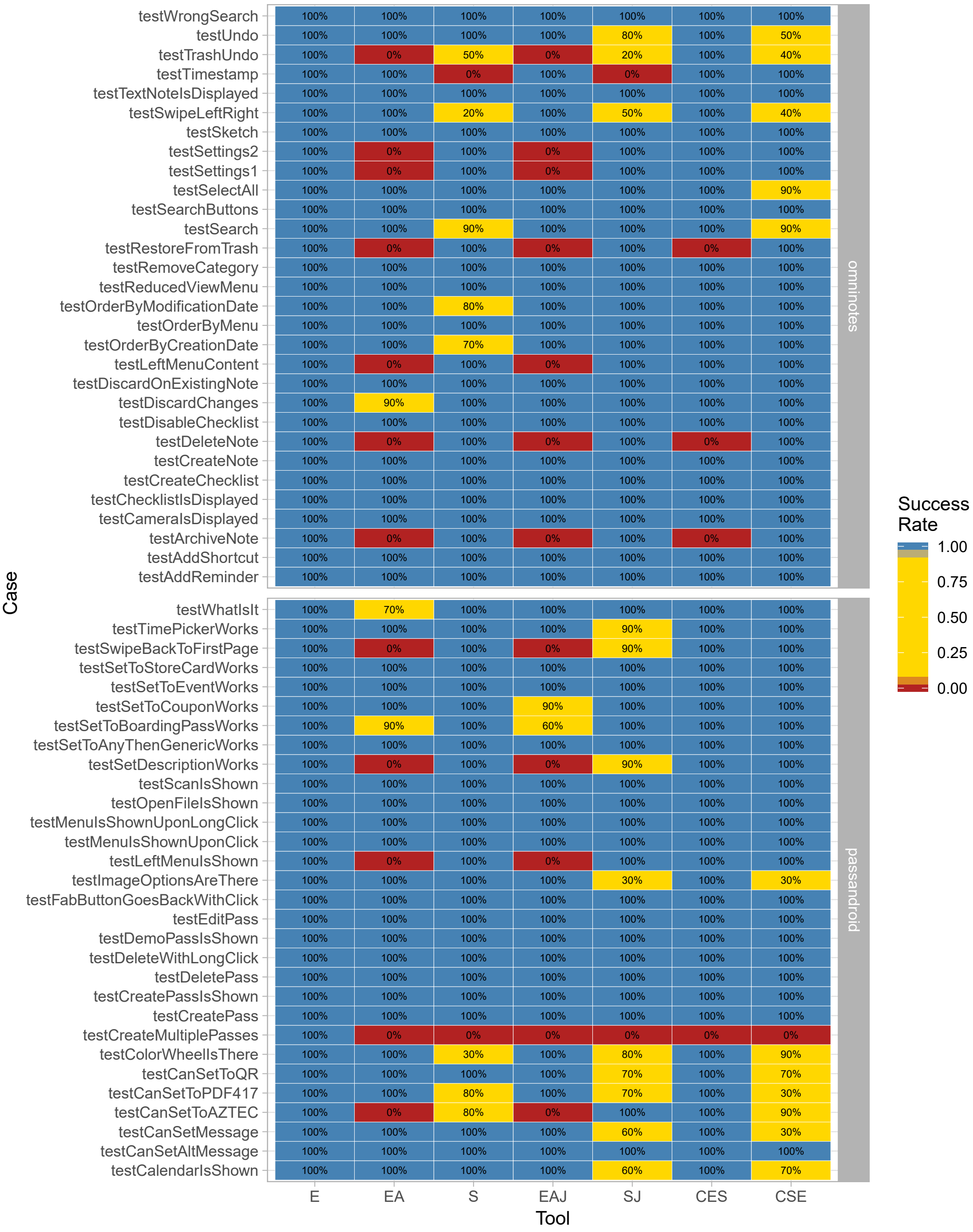
This section reports the values measured to answer the two Research Questions detailed in the previous sections.

Success Rate

Figure 9.14 reports the list of all the test cases developed for the two considered Android apps (PassAndroid and OmniNotes) along with the success rate obtained with each of the six different syntaxes obtained by translation. For each test case, the success rate of the original Espresso test execution is reported. All original Espresso test cases had 100% success rate, meaning that original test cases were not flaky and the app was in a stable and predictable state for the execution of the entire test suites.

All the translated test cases were checked manually, in order to verify that the Enhancer or the Log Parser failed and provided wrong screen captures to the 3rd generation testing tools. The absence of wrong screen captures leaves the responsibility for errors in the test cases only to issues in the image recognition libraries used or to the inapplicability of the visual paradigm because of too small visual locators. Moreover, the addition of a final check of the whole screen at the end of each test case reduces the possibility that, when a test is considered as passing, some of its operations have been performed on the wrong widgets. If so, in fact, the final state of the application would likely be different.

According to the results of the Fisher Exact Test, it has been observed that in general there is a statistically significant correlation between the success of test scripts and the used 3rd generation tool for the translation ($p < 10^{-5}$).



Case	E	EA	S	EAJ	SJ	CES	CSE
testWrongSearch	100%	100%	100%	100%	100%	100%	100%
testUndo	100%	100%	100%	100%	80%	100%	50%
testTrashUndo	100%	0%	50%	0%	20%	100%	40%
testTimestamp	100%	100%	0%	100%	0%	100%	100%
testTextNoteIsDisplayed	100%	100%	100%	100%	100%	100%	100%
testSwipeLeftRight	100%	100%	20%	100%	50%	100%	40%
testSketch	100%	100%	100%	100%	100%	100%	100%
testSettings2	100%	0%	100%	0%	100%	100%	100%
testSettings1	100%	0%	100%	0%	100%	100%	100%
testSelectAll	100%	100%	100%	100%	100%	100%	90%
testSearchButtons	100%	100%	100%	100%	100%	100%	100%
testSearch	100%	100%	90%	100%	100%	100%	90%
testRestoreFromTrash	100%	0%	100%	0%	100%	0%	100%
testRemoveCategory	100%	100%	100%	100%	100%	100%	100%
testReducedViewMenu	100%	100%	100%	100%	100%	100%	100%
testOrderByModificationDate	100%	100%	80%	100%	100%	100%	100%
testOrderByMenu	100%	100%	100%	100%	100%	100%	100%
testOrderByCreationDate	100%	100%	70%	100%	100%	100%	100%
testLeftMenuContent	100%	0%	100%	0%	100%	100%	100%
testDiscardOnExistingNote	100%	100%	100%	100%	100%	100%	100%
testDiscardChanges	100%	90%	100%	100%	100%	100%	100%
testDisableChecklist	100%	100%	100%	100%	100%	100%	100%
testDeleteNote	100%	0%	100%	0%	100%	0%	100%
testCreateNote	100%	100%	100%	100%	100%	100%	100%
testCreateChecklist	100%	100%	100%	100%	100%	100%	100%
testChecklistIsDisplayed	100%	100%	100%	100%	100%	100%	100%
testCameraIsDisplayed	100%	100%	100%	100%	100%	100%	100%
testArchiveNote	100%	0%	100%	0%	100%	0%	100%
testAddShortcut	100%	100%	100%	100%	100%	100%	100%
testAddReminder	100%	100%	100%	100%	100%	100%	100%

Case	E	EA	S	EAJ	SJ	CES	CSE
testWhatIsIt	100%	70%	100%	100%	100%	100%	100%
testTimePickerWorks	100%	100%	100%	100%	90%	100%	100%
testSwipeBackToFirstPage	100%	0%	100%	0%	90%	100%	100%
testSetToStoreCardWorks	100%	100%	100%	100%	100%	100%	100%
testSetToEventWorks	100%	100%	100%	100%	100%	100%	100%
testSetToCouponWorks	100%	100%	100%	90%	100%	100%	100%
testSetToBoardingPassWorks	100%	90%	100%	60%	100%	100%	100%
testSetToAnyThenGenericWorks	100%	100%	100%	100%	100%	100%	100%
testSetDescriptionWorks	100%	0%	100%	0%	90%	100%	100%
testScanIsShown	100%	100%	100%	100%	100%	100%	100%
testOpenFileIsShown	100%	100%	100%	100%	100%	100%	100%
testMenuIsShownUponLongClick	100%	100%	100%	100%	100%	100%	100%
testMenuIsShownUponClick	100%	100%	100%	100%	100%	100%	100%
testLeftMenuIsShown	100%	0%	100%	0%	100%	100%	100%
testImageOptionsAreThere	100%	100%	100%	100%	30%	100%	30%
testFabButtonGoesBackWithClick	100%	100%	100%	100%	100%	100%	100%
testEditPass	100%	100%	100%	100%	100%	100%	100%
testDemoPassIsShown	100%	100%	100%	100%	100%	100%	100%
testDeleteWithLongClick	100%	100%	100%	100%	100%	100%	100%
testDeletePass	100%	100%	100%	100%	100%	100%	100%
testCreatePassIsShown	100%	100%	100%	100%	100%	100%	100%
testCreatePass	100%	100%	100%	100%	100%	100%	100%
testCreateMultiplePasses	100%	0%	0%	0%	0%	0%	0%
testColorWheelIsThere	100%	100%	30%	100%	80%	100%	90%
testCanSetToQR	100%	100%	100%	100%	70%	100%	70%
testCanSetToPDF417	100%	100%	80%	100%	70%	100%	30%
testCanSetToAZTEC	100%	0%	80%	0%	100%	100%	90%
testCanSetMessage	100%	100%	100%	100%	60%	100%	30%
testCanSetAltMessage	100%	100%	100%	100%	100%	100%	100%
testCalendarIsShown	100%	100%	100%	100%	60%	100%	70%

Fig. 9.14 TOGGLE: Graphical summary of individual test success rate

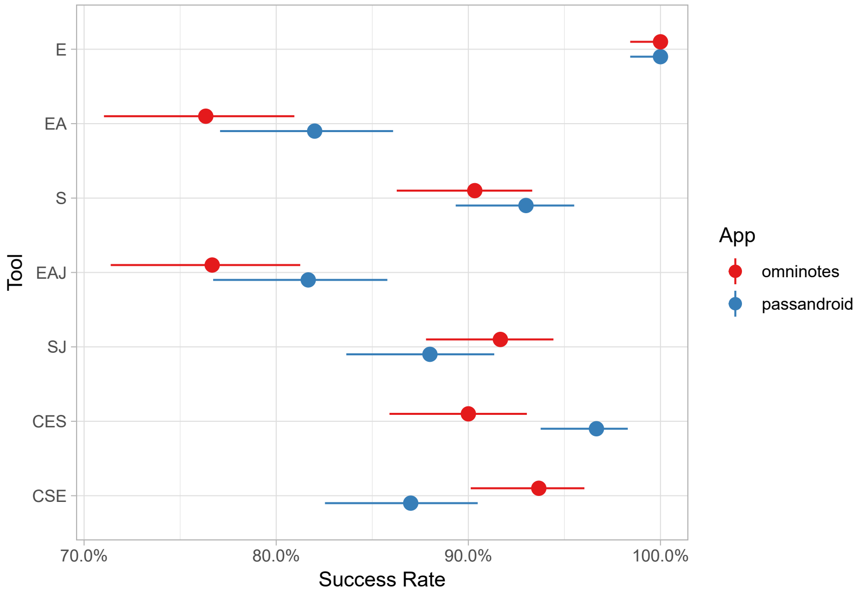


Fig. 9.15 TOGGLE: Average success rate by tool and app with 95% CI

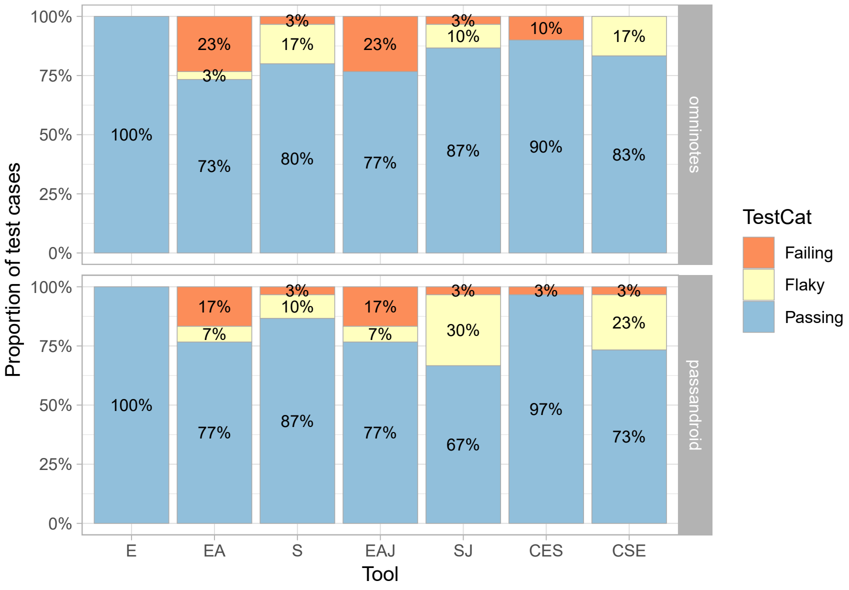


Fig. 9.16 TOGGLE: Proportion of passing, flaky and failing translated test cases

Figure 9.15 shows the Average Success Rate for the six sets of 3rd generation scripts obtained by applying Toggle, compared – also in this case – to the 100% success rate of the original Espresso test suite. The average success rate is plotted with 95% confidence interval. From the graph, it is evident that there was no single (or combination of) 3rd generation test technique that was the best for both OmniNotes and PassAndroid. For OmniNotes, the combination of Sikuli first and EyeAutomate second (CSE) guaranteed the highest success rate, for PassAndroid it was the reverse combination (CES).

Figure 9.16 shows the distributions of passing, flaky and failing test cases for all the six generated sets of tests. From the figure is evident that most of the flaky solutions included Sikuli. This high proportion of flaky Sikuli tests was mainly due to the need of performing swipe operations, that were less precisely reproduced by the Sikuli atomic mouse commands. A relevant number of tests in both OmniNotes and PassAndroid involved such swipe operations. EyeAutomate was the least successful visual testing tool, both when test scripts were created in the tool-specific plain-text syntax or through calls to the EyeAutomate Java API. Most of the test failures for EyeAutomate were caused by the inability of the visual recognition tool to find elements with a low amount of details (as the NavigationDrawer button in figure 6.3 of chapter 6).

From both figure 9.15 and 9.16 it can be deduced that the combination of the tools helps, by leveraging the two against each other improving overall success rate on OmniNotes. With PassAndroid, the combined version with Sikuli First (CSE) performed just slightly better than the Java versions of EyeAutomate and Sikuli separately. This is justified again by the lesser robustness of Sikuli when swipe operations are involved.

Answer to RQ5.1: None of the 3rd generation test suites achieved the same average success rate as Espresso. The best solution overall, in terms of Success rate, was to use EyeAutomate first, supported by Sikuli (CES). If only one image recognition algorithm can be used, Sikuli is the most suitable solution.

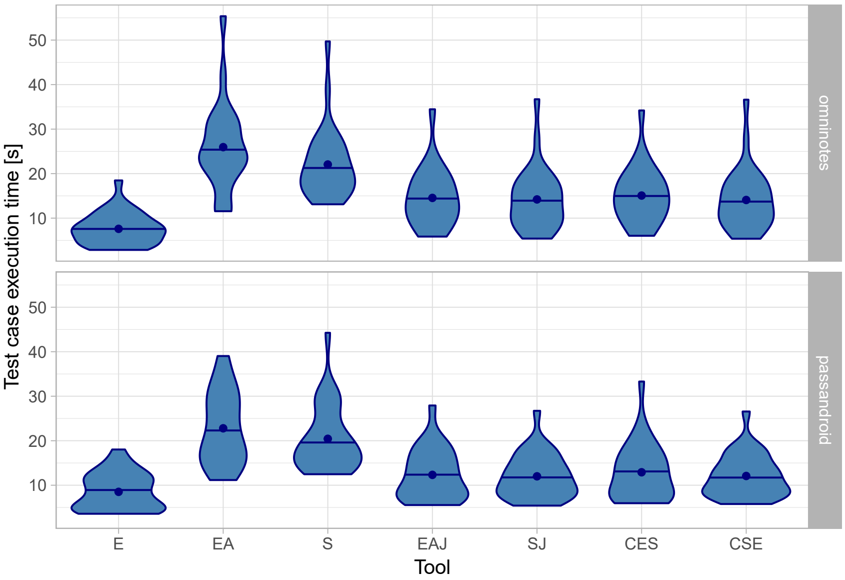


Fig. 9.17 TOGGLE: Average Execution time, by tool and app

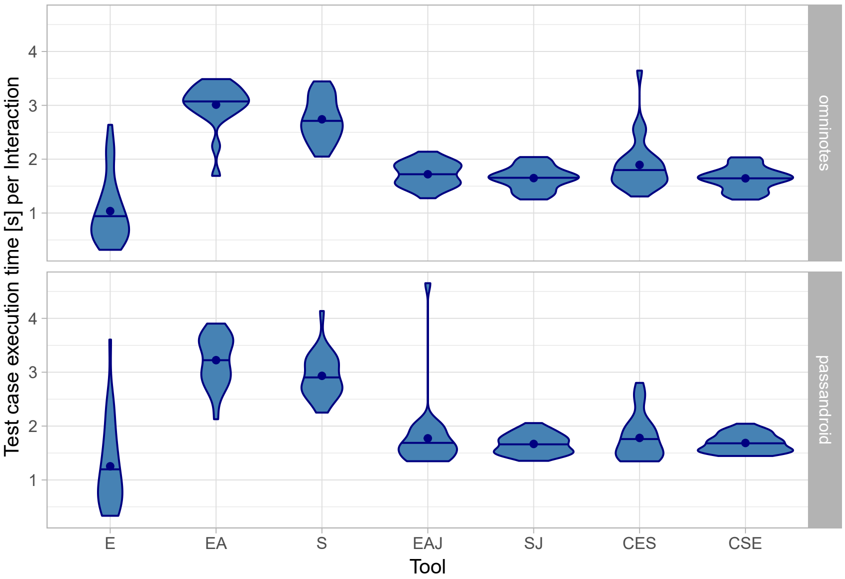


Fig. 9.18 TOGGLE: Average Execution time normalized by interaction, by tool and app

Execution Time

Figure 9.17 shows the average execution times of the passing executions of the translated test cases. Figure 9.18 shows the average execution times normalized by the number of interactions of test cases, hence a measure that is comparable with test cases written for other usage scenarios and other applications. Espresso, once more, has been added as a benchmark for the considered visual testing techniques. The number of interactions for Espresso test cases was the same as the number of interactions of created visual test cases, minus one (for the final full check, that is not performed in original cases).

The permutation test ANOVA confirmed a statistically significant difference between 2nd and 3rd generation tools ($p < 10^{-15}$), as well as an effect of the interaction between tool generation and application ($p < 10^{-15}$). Applied to the distribution of time per interaction measured for the six 3rd generation testing techniques and the two apps, the ANOVA parametric test shows statistically significant difference on measured average time per interaction depending on tool ($p < 10^{-15}$), no clear effect of the application ($p = 0.1$), and statistically significant effect of the interaction of the two factors ($p < 10^{-15}$).

The difference in average time per interaction between the two applications can be explained by the nature of the interactions performed during the test scripts. PassAndroid usage scenarios involved a higher number of swipe operations, that require a longer execution time than simpler click and type operations.

Espresso had a better average execution time per interaction. The main reason for this result is the tool's support for automatic wait instructions for widgets to appear on-screen, instead of the discussed heuristic sleep time added to created visual test scripts. The second best tool was the Java version of Sikuli, closely followed by the Java version of EyeAutomate. Scripts written in the native syntaxes of the two tools had significantly worse average times per interaction than the respective Java versions. This can be explained by the fact that those test cases were run inside a Java environment, instantiating script runners provided by the respective APIs. This embedding can create additional delays and explain such a higher execution time. Finally, the combined versions of the tests had a bit worse performance than any of the testing tools based in Java individually. The reason is TOGGLE's approach of generating tests that always try with one tool first and only if it fails (and hence, after the 5s sleep delay for triggered failure) the second tool is used. This waiting time before the transition adds considerable overhead to each test script. However, both the combined versions had better performance than the scripts developed in tool-specific syntax.

Answer to RQ5.2: 2nd generation approach had significantly better performance (in terms of average execution time normalized by the number of instructions) than any of the created 3rd generation test scripts. There is a significant difference between average time per interaction measured for the six considered 3rd generation testing tools, with the Java versions of Sikuli test cases (SJ) performing the faster.

Chapter 10

Revisit of Study Findings

In this section, a final revisit of the findings of the presented studies is provided, along with a comparison with relevant related work from literature, and a summary of the contributions that this work provides to both practitioners and researchers.

10.1 Study 1 - Survey with mobile developers from the industry

In Study 1 of this research, insights about the perception of automated testing (and especially GUI testing) were gathered among professional developers. The findings of the experiments are reported in table 10.1. The seven interviews conducted with relevant players of the Italian software industries mainly proved that a test automation culture for mobile application is still lacking in developers. The results gathered from the structured interviews were in line with those of the investigations performed by Kochhar et al. [57] and Linares-Vasquez et al. [67], who highlighted a still rooted preference for manual testing for Android applications, in industrial contexts. The interviewed mobile and web developers generally did not perceive the ROI guaranteed by adopting automated testing techniques; studies in the literature have highlighted that positive ROI can be obtained with such techniques in more than 180 weeks [5], quite a long time frame for small to medium-sized companies. Many works in the literature have underlined the developers' perception of automated GUI testing as a very expensive activity in terms of setup costs and required training. The

Table 10.1 Study 1: Answers to the Research Questions

RQ1.1: Are mobile applications tested by the interviewed sample of industry practitioners? How? To what extent?	All the respondents to the survey performed manual testing on their mobile applications. Among the automated testing tools used, the interviewed developers mostly relied on Capture & Replay test, with a rare adoption of scripted testing tools.
RQ1.2: What are the most peculiar properties to test in mobile applications according to the interviewed sample of industry practitioners? What aspects of mobile apps discourage them from adopting automated testing?	The respondents to the interview underlined several aspects that are peculiar to mobile vs. desktop or web development. The two main characteristics that required specific forms of testing – according to the interviewed developers – were resource and battery saving and adaptation to different devices and display sizes.
RQ1.3: What are the main challenges felt by developers from the industry performing automated testing, and the directions research should take according to them?	All the interviewed mobile developers found that the maintenance of test cases for mobile applications is one of the most relevant challenges that prevent the adoption of automated testing techniques. Almost all respondents expressed the desire for better ways to manage fragilities and to reduce the effort for making the test suites evolve. Little enthusiasm was instead shown towards new paradigms of testing explored by literature, like model-based testing and visual recognition testing.

responses of the sample interviewed in this study are in line with the findings by Rafi et al., who interviewed developers and testers of desktop applications: 88% of their respondents agreed that automated testing requires extra effort than manual for designing test scripts, whilst only 6% of them considered automated software testing as capable of finding complex bugs that are instead spottable by manual testing practices [86].

At the time of the interviews (end of 2017) the need for automated testing of Android apps was not felt by the interviewed developers as urgent as that felt for applications of other domains. Most papers in the literature involving interviews with the developers agree with finding the most important deterrent from systematic (GUI) testing of mobile applications in the difficulties in maintaining test code, and in the scarce usability (and poor documentation) of available testing tools [59].

The needed maintenance by automated test cases was felt like one of the most important deterrents for the adoption of such techniques. Some of the estimated maintenance efforts reported by the respondents of the survey are in line with related work in the literature that has investigated the effort in maintaining test cases for desktop applications. For instance, respondent E identified the cost of maintenance of existing test scripts as 60% of total testing effort; this result pairs with a case study conducted at Siemens by Alegroth et al., where upward of 60% of time spent on test automation each week was devoted to maintenance [5]. The same study reports measures performed at Saab, with an average and maximum time for repairing individual test scripts of respectively 23 and 110 minutes, and 7% of total project

Table 10.2 Study 2: Answers to the Research Questions

RQ2.1: Productivity - What is the productivity of inexperienced developers when approaching to Layout-based and Visual GUI testing tools?	No statistically significant difference has been found between the respective productivity obtained using EyeAutomate or Espresso. It can be deduced that the learnability of the two tools is similar, for non-professional developers approaching them.
RQ2.2: Quality - What is the percentage of working test scripts produced by undergraduate programmers using Layout-based and Visual GUI testing tools?	A statistically significant difference has been found between the respective quality obtained using EyeAutomate or Espresso. In particular, test suites developed with EyeAutomate had a higher quality than the ones that the participants developed with Espresso.
RQ2.3: Obstacles - Which are the perceived difficulties in approaching visual and layout-based GUI testing techniques?	The respondents to the experiment found slightly easier the development of test suites using the EyeAutomate library, in the context of the EyeStudio companion IDE, with respect to developing scripted Espresso test cases in the Android Studio IDE. The respondents identified the imprecision of the image recognition library, and the difficulty in finding individual ids for the widgets, the most problematic aspects of, respectively, the proposed Visual and Layout-based testing tools.

time dedicated to test maintenance: a result that relates to the estimates of two-person day for test-fixing at each release, provided by Respondent D. The lower maintenance effort reported by other respondents can be justified by their admittedly very low adoption of automated testing. Several studies in the literature measured the test maintenance effort at bigger companies: Grechanik et al. report the annual cost at Accenture for that purpose, estimated to be between $50 to $120 million [46].

Albeit all the interviewed developers from the industry were quite skeptical about the possibility of adopting advanced automated testing techniques for their applications, recent works in the literature testify the adoption of different forms of GUI testing (from model-based to visual) for large Android projects [38][4]. The set of interviews with developers from the industry is, however, representative of the Italian development scenario and can be seen as a valuable source of hints for developers of automated testing tools (open-source or not) to try to meet the needs of the industry.

10.2 Study 2 - Controlled experiment with Graduate students

In Study 2 of this research, insights about the perception of automated GUI testing of mobile apps were gathered among graduate students. The findings of the experiments are reported in table 10.2, and are based on a comparison of the results obtained by the students in developing the same test suite, for the same open-source application

(Omni-Notes), with the aid of two tools belonging to different GUI testing generations (EyeAutomate for visual testing, and Espresso for Layout-based testing). The analysis of the submitted test suites, along with the answers to the interviews that were subministrated to the students at the end of the experiment session, suggest that EyeAutomate (and hence, the Visual approach to GUI testing) is considered slightly easier than Espresso (and hence, of the Layout-based approach), and that test suites created with EyeAutomate were of higher quality than those generated with Espresso.

Empirical studies with students are often used in Software Engineering research, even though their external validity is typically questioned [24]. The main use of studies with students is to gain insights in techniques, methods, and approaches to a given problem. Many studies with students are designed to compare two different techniques of solving the same problem: examples exist in the literature about the comparison of Extreme Programming vs. traditional software construction [71], different methods for software requirements inspection [89], domain-specific or general purpose programming language [58].

No prior controlled experiment with students dealing with different GUI testing techniques (and, more specifically, with GUI testing tools for Android apps) has been found at the time of the conduction of the study. A comparison between tools of the two generations has been performed by Min et al. in the field of mobile applications [78]. The findings of the study are in line with the results of the experiment: tools using 3rd generation approach are said to be more efficient in terms of implementing test cases, but had more false positives than 2nd generation approach, and needed more maintenance effort during the evolution of a typical mobile application. Two other similar comparative studies, albeit not involving mobile applications, were conducted by Leotta et al., who investigated Capture & Replay vs. Programmable (i.e., Scripted) web testing [61] and DOM (i.e., Layout-Based) vs. Visual locators [62]. They find that the development of test suites is more expensive in terms of required effort (up to 30% more time needed) when the Layout-based approach is adopted; on the contrary, the needed maintenance is higher (up to 50% more) when the visual approach is used. The results about the required effort can be considered as a confirmation of the results of Study 2, about the higher amount of test cases delivered when using EyeAutomate instead of Espresso, or leveraging Espresso TestRecorder instead of manually inspecting layout and widget properties.

Table 10.3 Study 3: Answers to the Research Questions

RQ3.1: Adoption and size - What is the level of adoption of a set of automated testing tools among open-source Android projects?	The considered GUI testing tools reached a diffusion that is always lower than 4.11% individually, and a combined adoption of about 8% by the considered set of 15 thousand Android repositories hosted on GitHub. The projects that are tested with the considered tools are typically rather short-lived, with an average of 15 releases, and feature on average few very few test classes for around 10% of total production code devoted to testing.
RQ3.2: Evolution - How much are GUI test classes associated with the analyzed sets of tools modified through consecutive releases of an open-source Android project?	An average 5% of the testing code is modified between consecutive tagged releases of Android repositories hosted on GitHub featuring tests associated with the six selected testing tools. 4.54% of the whole maintenance effort on production code is limited to changes in classes that are identified as tests developed with the studied testing tools. On average, one every five release required efforts of maintenance on test classes, and one every five classes had to be modified at least once during the lifespan of a project. On each new release, an average 15.43% of test classes (3.83% of test methods) feature modifications.

10.3 Study 3 - Measures of Diffusion and Evolution of Testware in OS projects

In Study 3 of this research, a data mining experiment was conducted on open-source Android apps hosted on GitHub, to quantify the adoption of popular automated GUI testing tools for Android, and to measure the average size and the needed amount of maintenance of developed automated test suites. The findings of the experiment are reported in table 10.3.

The metrics about adoption and size reflect what has been found by other studies in the literature, pertaining to mobile applications: Linares-Vasquez et al. [68] conducted a survey about automated mobile app testing, identifying the testing tools of which we studied the diffusion as the most used by developers, with the addition of UI Automation (for testing iOS apps), Ranorex, Calabash, Quantum, and Qmetry. The latter tools were not considered in this study because they were either closed-source or based on languages different from Java (and hence not comparable with production code of the mined apps).

Kochhar et al. [57], in addition to their interviews with open-source developers, performed a quantitative analysis on 600 open-source Android apps mined from F-Droid. The authors found that 14% of mined apps contained test cases (with 9% of the apps having executable test cases) with a coverage of 40%. The most widespread testing tools cited by the authors were JUnit, Monkeyrunner, Robotium, and Robolectric. Those results are in line with the findings reported in this thesis

(8% of applications having test cases with the considered testing frameworks) for several reasons: JUnit and Monkeyrunner were not considered among the studied frameworks (since they were not GUI-level testing frameworks), and the Espresso framework had just been published at the time of publication of the considered study.

Cruz et al. [35] performed another mining of applications from F-Droid, to measure the amount of test code they feature and some correlation between the presence of test code and quality indicators (e.g., ratings and downloads from the PlayStore, Repository Activity and popularity of the related GitHub repositories). It is found that the presence of test code correlates with the number of contributors and the number of commits of a given GitHub repository, but not with the ratings on the PlayStore (and hence, the perceived quality of the app by its users). Regarding the adoption of testing tools, they found that Appium, Espresso, and Robotium were the most used GUI testing, hence compatible findings with those of the present study.

With respect to all the studies cited in this subsection, the study documented in this thesis has the element of novelty of mining projects from GitHub directly, instead of mining Android apps from F-Droid and then pairing them with the relative GitHub projects. This way Android projects that are on GitHub only, and not on F-Droid, are also taken into consideration. With the adoption of such a mining procedure, the described metrics have been computed on the largest set of open-source Android application packages documented in the literature.

Literature about software testing typically identifies the amount of Verification and Validation for a software project as spanning between 20% and 50% of the total effort for the project [37]. The metrics measured in this work found that on average 10% of the total production code of analyzed open-source apps is produced with testing tools and that 5% of the total maintenance effort on production code is localized in test code. Those values are slightly lower than the average effort values identified by the literature, and can be motivated by supposing a co-existence of manual testing activities with a rather limited amount of automated scripted test code for Android open-source projects.

The evolution metrics gathered in this study can be compared with evidence from papers about the maintenance of various testing techniques. The considered tools featured 15% modified test classes, on average, at each new release; 20% of test classes had to be modified at least once during the lifespan of the project. The results can be compared to those measured for automated test cases of web-applications:

Cristophe et al., analyzing the evolution of test classes developed with Selenium, found out that up to 75% of test classes needed modifications during the lifespan of the application [27]. This higher value can be justified by taking into consideration the smaller sample of applications considered in their study, and by the pre-filtering operated by the authors, who only considered projects that performed extensive test with Selenium. Alegroth et al. measured the effort for co-evolving a Visual GUI test suite in an industrial project as 25.8% of the total evolution effort of the project [6]: this value is significantly higher than that measured for the average MRTL metrics in this study. A reason for this difference may be in intrinsic lower robustness of Visual test cases to changes in the AUT, with respect to scripted testing techniques analyzed in Study 3.

The diffusion and size metrics defined in this study can be leveraged by practitioners to know which are the most widespread testing tools for Android apps, and are fit for further investigations by researchers, to study possible correlations with other measurements on test code or about the perceived quality of published apps. The evolution metrics can be used directly by developers, to understand whether they would be able to tackle the effort required by a typical test suite written with a given scripted GUI Automation Framework. They can also be used as a benchmark for already developed test cases, to understand if the maintenance effort needed is in line with the average maintenance effort needed by open-source projects mined from GitHub.

10.4 Study 4 - Taxonomy of Fragility causes

Built upon the mining section that was already used by Study 3, Study 4 of this thesis detailed a taxonomy of possible modification causes for testing code of Android applications. Based on the taxonomy (and according to our definition, i.e. changes in test code that are not isolated but are induced by changes in the logic, behaviour or appearance of the AUT) a quantification of the average frequency of occurrence of modifications that are related to fragility was also obtained. The findings of the experiment are reported in table 10.4.

The goal of the work was to provide insights about the causes that make the maintenance for GUI test cases necessary, in the domain of Android apps.

Table 10.4 Study 4: Answers to the Research Questions

RQ4.1: Modification Causes - what are the main causes behind the need for maintaining GUI test code in Android open-source projects?	Examining a set of 1724 diff files related to Espresso, UI Automator, Robotium and Robolectric, 28 different possible causes were identified for modifications of test methods developed for Android apps with the use of GUI Automation frameworks. Nine different macro-categories of change reasons were identified: changes in the functions and logic of test code, changes in the application functionalities, changes in the interaction with the GUI, varied arrangements of the widgets of the layout, changed identification of views, changed retrieval of resources, pure graphic changes, execution time variations, and adaptations to provide compatibility with different OS versions.
RQ4.2: Fragility - how fragile are test methods and classes to modifications in the AUT or in its appearance?	A percentage of about 60% of modifications due to changes in the AUT, and hence of fragile classes, was measured. On average upon all the diff files examined, more than 50% of the modifications on test classes triggered by changes in the AUT were connected to the GUI of the app or to its appearance. However, test suites were modified often for reasons that were not connected to changes in the AUT: 46.36% of the modified diff files that were examined featured changes that were local to test code and that could not be backtracked to variations in the AUT.

Among the modifications related to the GUI and its definition, the changes to textual properties of Layout-based tests (e.g., IDs or plain string content) proved to be relevant in terms of induced fragilities in test cases. This finding is in line with the work by Linares-Vasquez et al. [68], where the authors highlight that GUI automation frameworks create test scripts that are coupled to change-prone component ids, and that as of today there is no approach for automatically evolving test scripts written with Automation APIs.

The taxonomy of modification causes is a novel contribution to the field of mobile application testing since no prior effort has been made in such direction specifically for Android applications.

Hammoudi et al. derived a taxonomy of reasons for test case breakages for Record and Replay testing of web applications [50]. Even though the categories of the taxonomy imply the usage of specific properties of web applications, there are several commonalities with the taxonomy discussed in this thesis. For instance, the most common breakage reasons (50.42%) were the changes in attribute-based locators and text; attribute-based locators can be considered equivalent to the id locators in the layouts of Android applications. Other elements of the web-based taxonomy, like the addition or removal of JavaScript Popup Boxes, can be considered as similar to Widget Addition/Removal or Navigation change. On the other hand, several elements of the web-based taxonomy, like the need for page reload or the length of the user session, had no equivalent in the taxonomy for Android tests. Conversely, modification causes like Screen orientation change or changed keyboard or input methods are specific to Android test cases only. While the commonalities with the work by Hammoudi et al. can be deemed as a confirmation of the findings of the study, the exclusive elements of both taxonomies can be considered as a demonstration of the need for building a specific test breakage taxonomy for the Android application domain.

Other characterizations of the reasons for maintenance are available in the literature, like that provided by Yusifoglu et al. [101], that is however defined for generic test code, and lacks the level of detail provided in the proposed taxonomy.

The inferred taxonomy of modification causes, along with the measured occurrences for the considered testing tools, can be used by practitioners as a source of information for predicting the effort needed by a test case. The frequency of occurrence can be paired with static analysis of already developed test code, to identify which of the used commands involve characteristics that are prone to fragility.

For developers and testers, a few guidelines can be deduced by the frequency of occurrence of modification causes that was measured, in order to minimize the fragility occurrence:

- Always provide unique and stable IDs for the widgets inside the layout definition, since the textual content (or other properties like hints and content descriptions) is likely to be changed often during the evolution of an app;
- Always define the names of the locators semantically, to decrease the likelihood of IDs or text hints being changed frequently between subsequent releases of the same app;
- *Design for Testability*, clearly identifying independent and mockable parts of the AUT [21];
- Avoid small yet frequent changes on locators, that may have a high impact on the maintenance of test suites;
- When writing test cases, if possible, avoid relying on volatile elements of the user interface (e.g., the textual content of a TextView) as locators;
- Adopt change-resilient patterns for the definition of test cases, e.g. the Page Object Pattern [60];

Table 10.5 Study 5: Answers to the Research Questions

RQ5.1: What are the differences in reliability between the six combinations of visual test script techniques?	None of the 3rd generation test suites achieved the same average success rate as Espresso. The best solution overall, in terms of Success rate, was to use EyeAutomate first, supported by Sikuli (CES). If only one image recognition algorithm can be used, Sikuli is the most suitable solution.
RQ5.2: What are the differences in performance between the six combinations of the visual test scripts and the original 2nd generation test scripts?	2nd generation approach had significantly better performance (in terms of average execution time normalized by the number of instructions) than any of the created 3rd generation test scripts. There is a significant difference between average time per interaction measured for the six considered 3rd generation testing tools, with the Java versions of Sikuli test cases (SJ) performing the faster.

- Keep the test cases as independent as possible, without creating – even implicit – sequences of tests, in order to avoid that a single fragility in one of the first test cases invalidates many following ones;

- Reduce the number of transitions between the different app screens in individual test cases, to avoid fragilities due to changes in the navigation between existing activities.

10.5 Study 5 - Layout-based vs Generated visual test cases: An experiment with TOGGLE

Study 5 of this thesis detailed a proposal of a combined approach for Automated GUI testing, that provides the prototype of a translation from Layout-based to Visual test cases (and backward) with an empirical comparison of the precision and performance of written layout-based vs generated visual test cases. The findings of this empirical experiment are reported in table 10.5.

The developed tool, and the results of the related experiment show that it is possible to translate 2nd generation test cases, written in Espresso, to 3rd generation test cases, written with either Sikuli or EyeAutomate. The precision of the translation and the dependability of the generated test scripts proved to be rather high for both the applications considered for the study, except for few test cases that were invalidated either by the use of a particular image, or by the use of multiple swipes in the same user scenario.

Although the 3rd to 2nd generation translator is still not implemented, and the 2nd to 3rd generation translator still lacks support for some Espresso commands, the results demonstrate that the users of the tool can get benefits from combined usage of the two testing approaches with the proposed translational approach. Little effort is in fact needed for generating 3rd generation test suites from 2nd generation ones, and hence the users can get the benefits of Visual testing without the costs of developing or maintaining multiple GUI-based test suites.

The proposed approach adds to studies, available in the literature, that conceptualized the possible benefits of a combined approach of 2nd and 3rd generation testing tools [9]. A similar translational approach has been already proposed in the field of Web-Application testing, where DOM-based 2nd generation test cases (developed with Selenium WebDriver) were translated to 3rd generation test cases (written with Sikuli) [92][63]. The authors of this tool also highlighted the enhanced maintainability and ease of re-creation of 3rd generation test cases, with respect to the original 2nd generation ones from which the first translation is obtained.

Chapter 11

Conclusion and Future Work

The present dissertation investigated the principal tools and techniques for testing Android applications, their main advantages and drawbacks, and the issue of fragility, i.e. higher need for maintenance of test scripts when the Application Under Test evolves.

The main contribution of the thesis revolved around four principal research goals, each related to specific characteristics of the considered tools or techniques: Perception and Usability; Adoption and Size; Evolution and Fragility; General Android testing issues.

For what concerns the *Perception and Usability* of testing frameworks for Android apps, evidence collected from interviews with practitioners from the Italian industrial landscape (as detailed in Chapter 5) proved that GUI testing is not an established practice for Android developers. Many of the interviewed developers, on the other hand, relied only on manual executions of test cases on finished applications. From a controlled experiment with graduate students (detailed in Chapter 6), that had as subjects representative of junior developers that are frequently assigned to software testing in IT companies, it was deduced that GUI testing frameworks for Android applications are seen as somehow imprecise and characterized by a steep learning curve. The respondents from the interviewed sample of students, however, expressed a moderate preference towards 3rd generation (or Visual) testing frameworks, with which the development of test cases was considered much easier, at the price of lesser precision of generated test scripts. The respondents from the industry pointed out several challenges that, albeit generalizable to many testing domains, are exacerbated by the rapidly varying nature of Android applications, and by the emphasis that is posed in the interaction through their pictorial User Interface.

Regarding the *Adoption and Size* of GUI automation frameworks, the mining study performed on open-source Android apps (detailed in Chapter 7) proved that the most commonly adopted frameworks in open-source projects were Robolectric and Espresso, with about 8% of all Android applications available on GitHub featuring code attributable to GUI Automation frameworks. When apps are tested, test code constitutes a relevant portion (almost 10% of LOCs) of the total project code.

At the same time, test classes associated to the examined GUI automation frameworks proved to need relevant and continued maintenance effort by developers, that was quantified as occurring, on average, every 5 releases of the open-source projects examined, and amounting at 5% of the total development effort of the projects. An investigation about the main reasons for maintenance of test code, conducted as a Grounded Theory study for the construction of a taxonomy of modification reasons (detailed in Chapter 8), proved that the interventions on Android app's testware are caused by both changes on the test logic (and hence, unrelated to the Application Under Test) but also – with a frequency of near 60% of all modifications to test code – due to changes in the Application Under Test. This issue, that was defined throughout this study as *Fragility* of test code, is considered one of the principal deterrent for a systematic adoption of automated GUI testing of Android apps, since rapidly changing applications (like most Android apps are) may trigger the need for frequent and costly maintenance on related test code.

From interviews, investigations on open-source projects and reviews from literature, it was evident that several issues and challenges can be considered as characteristic of a given GUI test generation. More specifically, 2nd generation tests suffer from changes in the description of the user interface, and 3rd generation tests suffer from changes in the pictorial GUI of the apps. A prototype tool, based on the creation of 3rd generation test scripts via translation of existing 2nd generation ones, has been designed and implemented, with an experimental validation that proved that generated Visual tests can be considered of quality comparable to that of Layout-based ones. Such translational approach can enhance the bug-finding power of available test suites, since 2nd generation test scripts are not sensitive to errors in the actual rendering of the user interface, while 3rd generation test scripts cannot detect code-level bugs in the arrangement and definition of layouts. At the same time, being the two generations of scripts fragile to complementary changes applied to the AUT GUI, having two parallel test suites can enable repairing a fragile test suite by leveraging the other one. To sum up, a combined approach can relieve the testers/developers from (part of) the effort in maintaining test suites, and in general, mitigate the drawbacks of the two considered GUI test generations by leveraging the respective benefits.

The natural prosecution of the work detailed in this thesis will be the implementation of the second part of the TOGGLE tool, the 3rd to 2nd generation translator of which by now just a proof of concept is provided. After that, a thorough validation of the tool, possibly by applying it in industrial contexts, is forecasted. The frequency of occurrence of fragility causes, and the analysis on diff files of open-source projects can be used together for the definition of a maintenance cost predictor for test code. Finally, the taxonomy of modification reasons can be the basis for guidelines to testers/developers for writing more robust test scripts, and for tools (possibly as plug-ins for popular IDEs) for automated repair of fragilities, still not available to the community as highlighted by several research papers.

References

[1] Ahmad, A., Li, K., Feng, C., Asim, S. M., Yousif, A., and Ge, S. (2018). An empirical study of investigating mobile applications development challenges. *IEEE Access*, 6:17711–17728.

[2] Alamri, H. S. and Mustafa, B. A. (2014). Software engineering challenges in multi platform mobile application development. *Advanced Science Letters*, 20(10-11):2115–2118.

[3] Alégroth, E. (2013). *On the industrial applicability of visual gui testing*. PhD thesis, Chalmers University of Technology.

[4] Alégroth, E. and Feldt, R. (2017). On the long-term use of visual gui testing in industrial practice: a case study. *Empirical Software Engineering*, 22(6):2937–2971.

[5] Alégroth, E., Feldt, R., and Kolström, P. (2016). Maintenance of automated test suites in industry: An empirical study on visual gui testing. *Information and Software Technology*, 73:66–80.

[6] Alegroth, E., Feldt, R., and Olsson, H. H. (2013). Transitioning manual system test suites to automated testing: An industrial case study. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 56–65. IEEE.

[7] Alégroth, E., Feldt, R., and Ryrholm, L. (2015). Visual gui testing in practice: challenges, problems and limitations. *Empirical Software Engineering*, 20(3):694–744.

[8] Alegroth, E., Gao, Z., Oliveira, R., and Memon, A. (2015). Conceptualization and evaluation of component-based testing unified with visual gui testing: An empirical study. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10.

[9] Alégroth, E., Gao, Z., Oliveira, R., and Memon, A. (2015). Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10. IEEE.

[10] Alégroth, E., Karlsson, A., and Radway, A. (2018). Continuous integration and visual gui testing: Benefits and drawbacks in industrial practice. In *Software Testing, Verification and Validation (ICST), 2018 IEEE 11th International Conference on*, pages 172–181. IEEE.

[11] Amalfitano, D., Fasolino, A. R., and Tramontana, P. (2011). A gui crawling-based technique for android mobile application testing. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*, pages 252–261. IEEE.

[12] Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Imparato, G. (2012a). A toolset for gui testing of android applications. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 650–653. IEEE.

[13] Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Memon, A. M. (2012b). Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM.

[14] Amalfitano, D., Fasolino, A. R., Tramontana, P., Ta, B. D., and Memon, A. M. (2015). Mobiguitar: Automated model-based testing of mobile apps. *IEEE software*, 32(5):53–59.

[15] Andrews, A. A., Offutt, J., and Alexander, R. T. (2005). Testing web applications by modeling with fsms. *Software & Systems Modeling*, 4(3):326–345.

[16] Anureet, K. (2015). Review of mobile applications testing with automated techniques. *interface*, 4.

[17] Ardito, L., Coppola, R., Morisio, M., and Torchiano, M. (2019). Espresso vs. eyeautomate: An experiment for the comparison of two generations of android gui testing. In *Proceedings of the Evaluation and Assessment on Software Engineering*, pages 13–22. ACM.

[18] Ardito, L., Coppola, R., Torchiano, M., and Alegroth, E. (2018). Towards automated translation between generations of gui-based tests for mobile devices. In *Proceedings of INTUITESTBEDS 2018, joint Workshop of the 4th International Workshop on User Interface Test Automation, and 8th Workshop on TESting Techniques for event BasED Software*. ACM.

[19] Banerjee, I., Nguyen, B., Garousi, V., and Memon, A. (2013). Graphical user interface (gui) testing: Systematic mapping and repository. *Information and Software Technology*, 55(10):1679–1694.

[20] Barmi, Z. A., Ebrahimi, A. H., and Feldt, R. (2011). Alignment of requirements specification and testing: A systematic mapping study. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 476–485. IEEE.

[21] Berner, S., Weber, R., and Keller, R. K. (2005a). Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering*, pages 571–579. ACM.

[22] Berner, S., Weber, R., and Keller, R. K. (2005b). Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering*, pages 571–579. ACM.

[23] Borjesson, E. and Feldt, R. (2012). Automated system testing using visual gui testing tools: A comparative study in industry. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 350–359. IEEE.

[24] Carver, J. C., Jaccheri, L., Morasca, S., and Shull, F. (2010). A checklist for integrating student empirical studies with research and teaching goals. *Empirical Software Engineering*, 15(1):35–59.

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

[26] Choudhary, S. R., Gorla, A., and Orso, A. (2015). Automated test input generation for android: Are we there yet? *arXiv preprint arXiv:1503.07217*.

[27] Christophe, L., Stevens, R., De Roover, C., and De Meuter, W. (2014). Prevalence and maintenance of automated functional tests for web applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 141–150. IEEE.

[28] Coppola, R. (2017). Fragility and evolution of android test suites. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 405–408. IEEE.

[29] Coppola, R., Morisio, M., and Torchiano, M. (2017a). Evolution and fragilities in scripted gui testing of android applications. In *Proceedings of the 3rd International Workshop on User Interface Test Automation*, pages 83–104. Springer.

[30] Coppola, R., Morisio, M., and Torchiano, M. (2017b). Scripted gui testing of android apps: A study on diffusion, evolution and fragility. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 22–32. ACM.

[31] Coppola, R., Morisio, M., and Torchiano, M. (2018a). Maintenance of android widget-based gui testing: A taxonomy of test case modification causes. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 151–158.

[32] Coppola, R., Morisio, M., and Torchiano, M. (2018b). Mobile gui testing fragility: A study on open-source android applications. *IEEE Transactions on Reliability*, pages 1–24.

[33] Coppola, R., Morisio, M., Torchiano, M., and Ardito, L. (2019). Scripted gui testing of android open-source apps: evolution of test code and fragility causes. *Empirical Software Engineering*, pages 1–44.

[34] Coppola, R., Raffero, E., and Torchiano, M. (2016). Automated mobile ui test fragility: an exploratory assessment study on android. In *Proceedings of the 2nd International Workshop on User Interface Test Automation*, pages 11–20. ACM.

[35] Cruz, L. and Abreu, R. (2019). To the attention of mobile software developers: Guess what, test your app! *arXiv preprint arXiv:1902.02610*.

[36] Das, T., Di Penta, M., and Malavolta, I. (2016). A quantitative and qualitative investigation of performance-related commits in android apps. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 443–447. IEEE.

[37] Ellims, M., Bridges, J., and Ince, D. C. (2006). The economics of unit testing. *Empirical Software Engineering*, 11(1):5–31.

[38] Espada, A. R., Gallardo, M. d. M., Salmerón, A., and Merino, P. (2017). Performance analysis of spotify® for android with model-based testing. *Mobile Information Systems*, 2017.

[39] Fazzini, M., Freitas, E. N. D. A., Choudhary, S. R., and Orso, A. (2017). Barista: A technique for recording, encoding, and running platform independent android tests. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 149–160. IEEE.

[40] Fewster, M. et al. (2001). Common mistakes in test automation. In *Proceedings of Fall Test Automation Conference*.

[41] Gao, J., Bai, X., Tsai, W.-T., and Uehara, T. (2014). Mobile application testing: a tutorial. *Computer*, (2):46–55.

[42] Gao, Z., Chen, Z., Zou, Y., and Memon, A. M. (2016). Sitar: Gui test script repair. *Ieee transactions on software engineering*, (2):170–186.

[43] Garousi, V. and Felderer, M. (2016). Developing, verifying, and maintaining high-quality automated test scripts. *IEEE Software*, (3):68–75.

[44] Glaser, B. G., Strauss, A. L., and Strutzel, E. (1968). The discovery of grounded theory; strategies for qualitative research. *Nursing research*, 17(4):364.

[45] Gomez, L., Neamtiu, I., Azim, T., and Millstein, T. (2013). Reran: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 72–81. IEEE Press.

[46] Grechanik, M., Xie, Q., and Fu, C. (2009a). Experimental assessment of manual versus tool-based maintenance of gui-directed test scripts. In *2009 IEEE International Conference on Software Maintenance*, pages 9–18. IEEE.

[47] Grechanik, M., Xie, Q., and Fu, C. (2009b). Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st international conference on software engineering*, pages 408–418. IEEE Computer Society.

[48] Halpern, M., Zhu, Y., Peri, R., and Reddi, V. J. (2015). Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 215–224. IEEE.

[49] Ham, H. K. and Park, Y. B. (2011). Mobile application compatibility test system design for android fragmentation. In *International Conference on Advanced Software Engineering and Its Applications*, pages 314–320. Springer.

[50] Hammoudi, M., Rothermel, G., and Tonella, P. (2016). Why do record/replay tests of web applications break? In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 180–190. IEEE.

[51] Han, D., Zhang, C., Fan, X., Hindle, A., Wong, K., and Stroulia, E. (2012). Understanding android fragmentation with topic analysis of vendor-specific bugs. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 83–92. IEEE.

[52] Hesenius, M., Griebe, T., Gries, S., and Gruhn, V. (2014). Automating ui tests for mobile applications with formal gesture descriptions. In *Proceedings of the 16th international conference on Human-computer interaction with mobile devices & services*, pages 213–222. ACM.

[53] Hu, Y., Azim, T., and Neamtiu, I. (2015). Versatile yet lightweight record-and-replay for android. In *ACM SIGPLAN Notices*, volume 50, pages 349–366. ACM.

[54] Jabbarvand, R., Sadeghi, A., Bagheri, H., and Malek, S. (2016). Energy-aware test-suite minimization for android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 425–436. ACM.

[55] Joorabchi, M. E., Mesbah, A., and Kruchten, P. (2013). Real challenges in mobile app development. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 15–24. IEEE.

[56] Kirubakaran, B. and Karthikeyani, V. (2013). Mobile application testing—challenges and solution approach through automation. In *Pattern Recognition, Informatics and Mobile Engineering (PRIME), 2013 International Conference on*, pages 79–84. IEEE.

[57] Kochhar, P. S., Thung, F., Nagappan, N., Zimmermann, T., and Lo, D. (2015). Understanding the test automation culture of app developers.

[58] Kosar, T., Mernik, M., and Carver, J. C. (2012). Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical software engineering*, 17(3):276–304.

[59] Kropp, M. and Morales, P. (2010). Automated gui testing on the android platform. *on Testing Software and Systems: Short Papers*, page 67.

[60] Leotta, M., Clerissi, D., Ricca, F., and Spadaro, C. (2013a). Improving test suites maintainability with the page object pattern: An industrial case study. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 108–113. IEEE.

[61] Leotta, M., Clerissi, D., Ricca, F., and Tonella, P. (2013b). Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 272–281. IEEE.

[62] Leotta, M., Clerissi, D., Ricca, F., and Tonella, P. (2014). Visual vs. dom-based web locators: An empirical study. In *International Conference on Web Engineering*, pages 322–340. Springer.

[63] Leotta, M., Stocco, A., Ricca, F., and Tonella, P. (2018). Pesto: Automated migration of dom-based web tests towards the visual approach. *Software Testing, Verification And Reliability*, 28(4):e1665.

[64] Lin, Y.-D., Rojas, J. F., Chu, E. T.-H., and Lai, Y.-C. (2014). On the accuracy, efficiency, and reusability of automated test oracles for android devices. *IEEE Transactions on Software Engineering*, 40(10):957–970.

[65] Linares-Vásquez, M. (2015a). Enabling testing of android apps. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 763–765. IEEE.

[66] Linares-Vásquez, M. (2015b). Enabling testing of android apps. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 763–765. IEEE.

[67] Linares-Vásquez, M., Bernal-Cárdenas, C., Moran, K., and Poshyvanyk, D. (2017a). How do developers test android applications? In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 613–622. IEEE.

[68] Linares-Vásquez, M., Moran, K., and Poshyvanyk, D. (2017b). Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 399–410. IEEE.

[69] Liu, C. (2000). Platform-independent and tool-neutral test descriptions for automated software testing. In *Proceedings of the 22nd international conference on Software engineering*, pages 713–715. ACM.

[70] Machiry, A., Tahiliani, R., and Naik, M. (2013). Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM.

[71] Macias, F., Holcombe, M., and Gheorghe, M. (2003). A formal experiment comparing extreme programming with traditional software construction. In *Proceedings of the Fourth Mexican International Conference on Computer Science, 2003. ENC 2003.*, pages 73–80. IEEE.

[72] Mahmood, R., Mirzaei, N., and Malek, S. (2014). Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599–609. ACM.

[73] Mao, K., Harman, M., and Jia, Y. (2016). Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105. ACM.

[74] McMaster, S. and Memon, A. M. (2009). An extensible heuristic-based framework for gui test case maintenance. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 251–254. IEEE.

[75] McMinn, P. (2011). Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163.

[76] Memon, A. M. (2008). Automatically repairing event sequence-based gui test suites for regression testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(2):4.

[77] Memon, A. M. and Soffa, M. L. (2003). Regression testing of guis. *ACM SIGSOFT Software Engineering Notes*, 28(5):118–127.

[78] Min, Y. and Cai, S. (2018). Comparing different approaches of gui testing for mobile applications on android platform.

[79] Moran, K., Bonett, R., Bernal-Cárdenas, C., Otten, B., Park, D., and Poshyvanyk, D. (2017a). On-device bug reporting for android applications. In *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*, pages 215–216. IEEE.

[80] Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., and Poshyvanyk, D. (2015). Auto-completing bug reports for android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 673–686. ACM.

[81] Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., and Poshyvanyk, D. (2016). Fusion: A tool for facilitating and augmenting android bug reporting. In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pages 609–612. IEEE.

[82] Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., Vendome, C., and Poshyvanyk, D. (2017b). Crashscope: A practical tool for automated testing of android applications. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 15–18. IEEE.

[83] Muccini, H., Di Francesco, A., and Esposito, P. (2012). Software testing of mobile applications: Challenges and future research directions. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 29–35. IEEE Press.

[84] Neto, N. M. L., Vilain, P., and Mello, R. d. S. (2016). Segen: generation of test cases for selenium and selendroid. In *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services*, pages 433–442. ACM.

[85] Nguyen, B. N., Robbins, B., Banerjee, I., and Memon, A. (2014). Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, 21(1):65–105.

[86] Rafi, D. M., Moses, K. R. K., Petersen, K., and Mäntylä, M. V. (2012). Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 36–42. IEEE Press.

[87] Ralph, P. (2018). Toward methodological guidelines for process theories and taxonomies in software engineering. *IEEE Transactions on Software Engineering*.

[88] Sadeh, B., Ørbekk, K., Eide, M. M., Gjerde, N. C., Tønnesland, T. A., and Gopalakrishnan, S. (2011). Towards unit testing of user interface code for android mobile applications. In *International Conference on Software Engineering and Computer Systems*, pages 163–175. Springer.

[89] Sandahl, K., Blomkvist, O., Karlsson, J., Krysander, C., Lindvall, M., and Ohlsson, N. (1998). An extended replication of an experiment for assessing methods for software requirements inspections. *Empirical Software Engineering*, 3(4):327–354.

[90] Sjösten-Andersson, E. and Pareto, L. (2006). Costs and benefits of structure-aware capture/replay tools. *SERPS'06*, page 3.

[91] Skoglund, M. and Runeson, P. (2004). A case study on regression test suite maintenance in system evolution. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 438–442. IEEE.

[92] Stocco, A., Leotta, M., Ricca, F., and Tonella, P. (2014). Pesto: A tool for migrating dom-based to visual web tests. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 65–70. IEEE.

[93] Stol, K.-J., Ralph, P., and Fitzgerald, B. (2016). Grounded theory in software engineering research: a critical review and guidelines. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 120–131. IEEE.

[94] Strauss, A. (8). Corbin, j.(1998) basics of qualitative research. techniques and procedures for developing grounded theory. *Thousand Oaks, CA: Sage*.

[95] Tang, H., Wu, G., Wei, J., and Zhong, H. (2016). Generating test cases to expose concurrency bugs in android applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 648–653. ACM.

[96] Tang, X., Wang, S., and Mao, K. (2015). Will this bug-fixing change break regression testing? In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*, pages 1–10. IEEE.

[97] Tramontana, P., Amalfitano, D., Amatucci, N., and Fasolino, A. R. (2018). Automated functional testing of mobile applications: a systematic mapping study. *Software Quality Journal*, pages 1–53.

[98] Utting, M., Pretschner, A., and Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312.

[99] Wei, L., Liu, Y., and Cheung, S.-C. (2016). Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 226–237. ACM.

[100] Yeh, T., Chang, T.-H., and Miller, R. C. (2009). Sikuli: using gui screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 183–192. ACM.

[101] Yusifoğlu, V. G., Amannejad, Y., and Can, A. B. (2015). Software test-code engineering: A systematic mapping. *Information and Software Technology*, 58:123–147.

[102] Zadgaonkar, H. (2013). *Robotium Automated Testing for Android*. Packt Publishing Ltd.

[103] Zhang, S., Lü, H., and Ernst, M. D. (2013). Automatically repairing broken workflows for evolving gui applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 45–55. ACM.

[104] Zhou, X., Lee, Y., Zhang, N., Naveed, M., and Wang, X. (2014). The peril of fragmentation: Security hazards in android device driver customizations. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 409–423. IEEE.

Appendix A

Summary of all Research Questions and Sub-questions

Table A.1 Summary of Research Questions and sub-questions

Question	Subquestion	Description	Goal(s)
RQ1	RQ1.1	Are mobile applications tested by the interviewed sample of industry practitioners? How? To what extent?	G1
	RQ1.2	What are the most peculiar properties to test in mobile applications according to the interviewed sample of industry practitioners? What aspects of mobile apps discourage them from adopting automated testing?	G3, G4
	RQ1.3	What are the main challenges felt by developers from the industry performing automated testing, and the directions research should take according to them?	G4
RQ2	RQ2.1	What is the productivity of inexperienced developers when approaching to Layout-based and Visual GUI testing tools?	G1

Continued on next page

Table A.1 – continued from previous page

Question	Subquestion	Description	Goal(s)
	RQ2.2	What is the percentage of working test scripts produced by undergraduate programmers using Layout-based and Visual GUI testing tools?	G1, G4
	RQ2.3	What are the perceived difficulties in approaching visual and layout-based GUI testing techniques?	G1, G4
RQ3	RQ3.1	What is the level of adoption of a set of automated testing tools among open-source Android projects?	G2
	RQ3.2	How much are GUI test classes associated with the analyzed sets of tools modified through consecutive releases of an open-source Android project?	G3
RQ4	RQ4.1	What are the main causes behind the need for maintaining GUI test code in Android open-source projects?	G3
	RQ4.2	How fragile are test methods and classes to modifications in the AUT or in its appearance?	G3
RQ5	RQ5.1	How do translations of layout-based test cases to visual testing frameworks compare in terms of dependability?	G3, G4
	RQ5.2	How do translations of layout-based test cases to visual testing frameworks compare in terms of performance?	G4

Appendix B

Running Sample of Metrics Computation

To provide samples of metric computations, we resort to reporting all the intermediate and final measures for a small project of the sample that we considered, namely WheresMyBus/android[1]. The project features test classes that are attributable to the Espresso GUI Automation Framework. During the lifespan of the app, four different test classes are identified. The GitHub repository has a history of six distinct tagged releases, including the Master.

Table B.1 shows all the measures computed for the six distinct releases of the project. As detailed in the later Procedure section, all those metrics are obtained through (i) searches in the .java source files that are associated to the considered GUI Automation Framework (in this case, all .java files containing the keyword ”Espresso”); (2) examinations of the differences between the same files in consecutive releases of the project; (3) examination of the methods that are featured by each test class in all releases of the project. In the table, when a metric is not defined for a given release, the symbol ”-” is used. This happens, for instance, in the transition between release 1.4.0 and master, where no modifications are performed in the whole project (hence, $P_{diff} = 0$). In this case, the $MRTL$ metric is not defined. All the derived metrics which require a comparison with the amount of code, classes or methods of the previous release are not defined for the first tagged release of the project.

[1] http://github.com/WheresMyBus/android

Table B.1 Intermediate measures for project WheresMyBus/android

Metric	1.0.0	1.1.0	1.2.0	1.3.0	1.4.0	master
P_{locs}	981	4254	8417	8516	9031	9031
T_{locs}	0	0	485	647	699	699
TLR	0	0	0.58	0.76	0.78	0.78
P_{diff}	-	3599	5907	1531	733	0
T_{diff}	-	0	0	224	74	0
$MTLR$	-	-	-	0.46	0.11	0
$MRTL$	-	-	0	0.15	0.10	-
NTC	0	0	4	4	4	4
AC	-	0	4	0	0	0
DC	-	0	0	0	0	0
MC	-	0	0	3	3	0
NTM	0	0	19	25	25	25
AM	-	0	19	7	0	0
DM	-	0	0	1	0	0
MM	-	0	0	4	10	0
$MCMM$	-	0	0	3	3	0
MCR	-	-	-	0.75	0.75	0
MMR	-	-	-	0.21	0.4	0
$MCMMR$	-	-	-	1.0	1.0	-

Table B.2 Test class statistics for project WheresMyBus/android

1.2.0	app/src/androidTest/java/UITests/TestAlertForumActivity.java	80	-	3	-	-	-
1.2.0	app/src/androidTest/java/UITests/TestCatalogPage.java	272	-	8	-	-	-
1.2.0	app/src/androidTest/java/UITests/TestHomePage.java	68	-	5	-	-	-
1.2.0	app/src/androidTest/java/UITests/TestSubmitAlert.java	65	-	3	-	-	-
1.3.0	app/src/androidTest/java/UITests/TestAlertForumActivity.java	80	0	3	0	0	0
1.3.0	app/src/androidTest/java/UITests/TestCatalogPage.java	273	31	8	0	0	2
1.3.0	app/src/androidTest/java/UITests/TestHomePage.java	67	3	5	0	0	1
1.3.0	app/src/androidTest/java/UITests/TestSubmitAlert.java	227	190	9	7	1	1
1.4.0	app/src/androidTest/java/UITests/TestAlertForumActivity.java	85	7	3	0	0	1
1.4.0	app/src/androidTest/java/UITests/TestCatalogPage.java	274	5	8	0	0	3
1.4.0	app/src/androidTest/java/UITests/TestHomePage.java	67	0	5	0	0	0
1.4.0	app/src/androidTest/java/UITests/TestSubmitAlert.java	273	62	9	0	0	6
master	app/src/androidTest/java/UITests/TestAlertForumActivity.java	85	0	3	0	0	0
master	app/src/androidTest/java/UITests/TestCatalogPage.java	274	0	8	0	0	0
master	app/src/androidTest/java/UITests/TestHomePage.java	67	0	5	0	0	0
master	app/src/androidTest/java/UITests/TestSubmitAlert.java	273	0	9	0	0	0

Table B.2, shows statistics about the test classes that are featured by the examined project, during its lifespan. The table columns show, for each class, the absolute paths, the versions in which the class is present, the contained methods, and the total and modified LOCs, and the total, added, modified and deleted methods. The project features four distinct test classes during its lifespan. The statistics collected for the classes are finally used to compute the Test Suite Volatility, i.e., the percentage of

```
@@ -18,6 +18,7 @@ import com.wheresmybus.SubmitAlertActivity;
 import java.io.IOException;

 import controllers.WMBController;
+import modules.Route;
 import okhttp3.mockwebserver.MockResponse;
 import okhttp3.mockwebserver.MockWebServer;

@@ -46,7 +47,7 @@ public class TestAlertForumActivity {

     @Rule
     public ActivityTestRule<AlertForumActivity> rule =
-            new ActivityTestRule<>(AlertForumActivity.class);
+            new ActivityTestRule<>(AlertForumActivity.class, true, fals

     @Test
     public void testAlertDisplay() throws IOException {
@@ -68,6 +69,10 @@ public class TestAlertForumActivity {
         server.start();
         controller.useMockURL(server.url("/").toString());
         Intent startIntent = new Intent();
+        startIntent.putExtra("IS_ROUTE", true);
+        Route route = new Route("123", "some route", "1_100224");
+        startIntent.putExtra("ROUTE", route);
+        startIntent.putExtra("ROUTE_ID", "1_100224");
         startIntent.putExtra("TAB_INDEX", 1);
         rule.launchActivity(startIntent);
     }
```

Fig. B.1 Diff file for test class TestAlertForumActivity.java of WheresMyBus/android, between releases 1.3.0 and 1.4.0.

classes with at least a modification during their lifespan upon the total number of classes (in the case of this project, the 100%).

The metrics NTC, AC, DC, and MC, respectively the total, added, deleted and modified test classes, are computed by a raw count of the number of .java files that are associated with the testing tool under examination. The metrics NTM, AM, DM, and MM, respectively the total, added, deleted and modified test methods, are computed (i) in the case of AM and DM only, by counting the methods in added or deleted test classes; (ii) by applying the JavaParser tool on the individual test classes before and after the release transition, and examining the differences in the lists of methods. Diff files are also examined to identify the position of modified lines in test classes, in order to compute MCMM (i.e., the number of Modified Classes with Modified Methods). As an example, we report in figure B.1 the modifications in the test class TestAlertForumActivity.java between release 1.3.0 and release 1.4.0. It is evident from the diff file that a single test method is modified in the release transition, and that of the 7 modified test LOCs are outside test methods. Having a method modified, the class counts for the computation of the MCMM metric (i.e., the number of modified test classes with modified methods).

Appendix C

Translated Espresso Commands

C.1 Espresso Commands

Operations over the identified Widgets are performed in Espresso combining ViewAction objects, that are passed to the ViewInteraction.perform() method.

This section reports details about the operations performed by each ViewAction that is currently translated by TOGGLE.

C.1.1 Click actions

Simple actions that perform clicks or taps on the identified views in the current hierarchy.

click(int inputDevice, int buttonState) : Returns an action that clicks the view for a specific input device and button state. The default values for inputDevice is SOURCE_UNKNOWN and the default inputState is BUTTON_PRIMARY.

doubleClick() : Returns an action that double clicks the view.

longClick() : Returns an action that long clicks the view.

C.1.2 Keyboard actions

Actions that perform operations on the TextViews or EditTextViews of the current layout hierarchy.

clearText() : Returns an action that clears text on the view.

replaceText(String stringToBeSet) : Returns an action that updates the text attribute of a view.

typeText(String stringToBeTyped) : Returns an action that selects the view (by clicking on it) and types the provided string into the view.

typeTextIntoFocusedView(String stringToBeTyped) : Returns an action that types the provided string into the view. The view must be clicked before.

pressKey(int keyCode) : Returns an action that presses the key specified by the keyCode (eg. Keyevent.KEYCODE_BACK).

pressKey(EspressoKey key) : Returns an action that presses the specified key with the specified modifiers.

C.1.3 Swipe actions

The Swipe actions are applied to any kind of view, and perform a swipe operation on the entirety of a specific axis of the view.

swipeDown() : Returns an action that performs a swipe top-to-bottom across the horizontal center of the view. The swipe doesn't start at the very edge of the view, but has a bit of offset.

swipeUp() : Returns an action that performs a swipe bottom-to-top across the horizontal center of the view. The swipe doesn't start at the very edge of the view, but has a bit of offset.

swipeLeft() : Returns an action that performs a swipe right-to-left across the vertical center of the view. The swipe doesn't start at the very edge of the view, but is a bit offset.

swipeRight() : Returns an action that performs a swipe left-to-right across the vertical center of the view. The swipe doesn't start at the very edge of the view, but is a bit offset.

C.1.4 Special actions

Actions that are specific to the Android GUI, and that perform operations on elements that are proper of Android applications.

closeSoftKeyboard() : Returns an action that closes soft keyboard.

pressBack() : Returns an action that clicks the back button.

pressBackUnconditionally() : Similar to pressBack() but will not throw an exception when Espresso navigates outside the application or process under test.

pressMenuKey() : Returns an action that presses the hardware menu key (deprecated since Android 3.0 Honeycomb).

C.2 Translation to 3rd-generation specific syntax

Table C.1 TOGGLE - 3rd generation test script creator: Translation from Tool-agnostic instructions to Tool-specific commands

Logged interaction	**EyeAutomate commands**	**Sikuli commands**
clearText	i. Click *img* ii. Type [BACKSPACE] (*arg1* times)	i. click(*img*) ii. type(Key.BACKSPACE) (*arg1* times)
click	i. Click *img*	i. click(*img*)
closesoftkeyboard	i. Type [CTRL_PRESS] ii. Sleep 10 iii. Type [BACKSPACE] iv. Sleep 10 v. Type [CTRL_RELEASE]	i. keyDown(Key.CTRL) ii. sleep(0.01) iii. type(Key.BACKSPACE) iv. sleep(0.01) v. keyUp(Key.CTRL)
doubleclick	i. MouseDoubleClick *img* i. Click *img* ii. Type *arg1*	i. hover(*img*) ii. mouseDown(Button.LEFT) iii. sleep(0.001) iv. mouseUp(Button.LEFT)

Continued on next page

Table C.1 – continued from previous page

Logged interaction	EyeAutomate commands	Sikuli commands
		v. sleep(0.001) vi. mouseDown(Button.LEFT) vii. sleep(0.001) viii. mouseUp(Button.LEFT)
longclick	i. Move *img* ii. MouseLeftPress iii. Sleep 500 iv. MouseLeftRelease	i. hover(*img*) ii. mouseDown(Button.LEFT) iii. sleep(0.5) iv. mouseUp(Button.LEFT)
typetext	i. Click *img* ii. Type *arg1*	i. click(*img*) ii. type(*arg2*)
openactionbarmenu	i. Type [CTRL_PRESS] ii. Sleep 10 iii. Type m iv. Sleep 10 v. Type [CTRL_RELEASE]	i. keyDown(Key.CTRL) ii. sleep(0.01) iii. type(m) iv. sleep(0.01) v. keyUp(Key.CTRL)
pressback	i. Type [CTRL_PRESS] ii. Sleep 10 iii. Type [BACKSPACE] iv. Sleep 10 v. Type [CTRL_RELEASE]	i. keyDown(Key.CTRL) ii. sleep(0.01) iii. type(Key.BACKSPACE) iv. sleep(0.01) v. keyUp(Key.CTRL)
presskey	i. Type *arg1*	i. type(*arg1*)
pressmenukey	i. Type [CTRL_PRESS] ii. Sleep 10 iii. Type h iv. Sleep 10 v. Type [CTRL_RELEASE]	i. keyDown(Key.CTRL) ii. sleep(0.01) iii. type(h) iv. sleep(0.01) v. keyUp(Key.CTRL)
replacetext	i. Click *img* ii. Type [BACKSPACE] (*arg1* times) iii. Type *arg2*	i. click(*img*) ii. type(Key.BACKSPACE) (*arg1* times) iii. type(*arg2*)
swipedown	i. Move *img* ii. Sleep 10 iii. MouseLeftPress iv. MoveRelative "0" "250" v. MouseLeftRelease	i. r = find(*img*) ii. start = r.getCenter() iii. stepY = 250 iv. run = start v. mouseMove(start); wait(0.2) vi. mouseDown(Button.LEFT); wait (0.2) vii. run = run.below(stepY) viii. mouseMove(run) ix. mouseUp() xi. wait(0.2)
swipeleft	i. Move *img* ii. Sleep 10 iii. MouseLeftPress iv. MoveRelative "-250" "0" v. MouseLeftRelease	i. r = find(*img*) ii. start = r.getCenter() iii. stepX = 250 iv. run = start v. mouseMove(start); wait(0.2) vi. mouseDown(Button.LEFT); wait (0.2)

Continued on next page

Table C.1 – continued from previous page

Logged interaction	EyeAutomate commands	Sikuli commands
		vii. run = run.left(stepX) viii. mouseMove(run) ix. mouseUp() xi. wait(0.2)
swiperight	i. Move *img* ii. Sleep 10 iii. MouseLeftPress iv. MoveRelative "250" "0" v. MouseLeftRelease	i. r = find(*img*) ii. start = r.getCenter() iii. stepX = 250 iv. run = start v. mouseMove(start); wait(0.2) vi. mouseDown(Button.LEFT); wait (0.2) vii. run = run.right(stepX) viii. mouseMove(run) ix. mouseUp() xi. wait(0.2)
swipeup	i. Move *img* ii. Sleep 10 iii. MouseLeftPress iv. MoveRelative "0" "-250" v. MouseLeftRelease	i. r = find(*img*) ii. start = r.getCenter() iii. stepY = 250 iv. run = start v. mouseMove(start); wait(0.2) vi. mouseDown(Button.LEFT); wait (0.2) vii. run = run.up(stepX) viii. mouseMove(run) ix. mouseUp() xi. wait(0.2)

Appendix D

Publication List

The works presented in this thesis have been published in the following conference and journal conference proceedings, in which I have either been an author or a co-author.

Conference and Workshop Proceedings

- Coppola, Riccardo, Emanuele Raffero, and Marco Torchiano. "Automated mobile UI test fragility: an exploratory assessment study on Android." Proceedings of the 2nd International Workshop on User Interface Test Automation (INTUITEST). ACM, 2016 [34].

- Coppola, Riccardo. "Fragility and evolution of android test suites." Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on. IEEE, 2017 [28].

- Coppola, Riccardo, Maurizio Morisio, and Marco Torchiano. "Evolution and Fragilities in Scripted GUI Testing of Android applications." Proceedings of the 3rd International Workshop on User Interface Test Automation (INTUITEST). Springer, 2017 [29].

- Coppola, Riccardo, Maurizio Morisio, and Marco Torchiano. "Scripted GUI Testing of Android Apps: A Study on Diffusion, Evolution and Fragility." Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE). ACM, 2017 [30].

- Coppola, Riccardo, Maurizio Morisio, and Marco Torchiano. "Maintenance of Android Widget-based GUI Testing: A Taxonomy of test case modification causes." 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, 2018 [31]

- Ardito, Luca, Emil Alégroth, Riccardo Coppola, and Marco Torchiano. "Towards Automated Translation between Generations of GUI-based Tests for Mobile Devices." Proceedings of INTUITESTBEDS 2018, joint Workshop of the 4th International Workshop on User Interface Test Automation, and 8th Workshop on TESting Techniques for event BasED Software (INTUITESTBEDS). ACM, 2018 [18]

- Ardito, Luca, Riccardo Coppola, Maurizio Morisio and Marco Torchiano. "Espresso vs. EyeAutomate: An Experiment for the Comparison of Two Generations of Android GUI Testing." Proceedings of 23rd International Conference on Evaluation and Assessment in Software Engineering (EASE 2019). ACM, 2019. [17]

Journal articles

- Coppola, Riccardo, Maurizio Morisio, and Marco Torchiano. "Mobile GUI Testing Fragility: A Study on Open-Source Android Applications." IEEE Transactions on Reliability (2018). [32]

- Coppola, Riccardo, Luca Ardito, Maurizio Morisio, and Marco Torchiano. "Scripted GUI Testing of Android Open-Source Apps: Evolution of Test Code and Fragility Causes.", Empirical Software Engineering Journal (2019). [33]