POLITECNICO DI TORINO
Repository ISTITUZIONALE

Scripted GUI Testing of Android Open-Source Apps: Evolution of Test Code and Fragility Causes

Original
Scripted GUI Testing of Android Open-Source Apps: Evolution of Test Code and Fragility Causes / Coppola, Riccardo; Morisio, Maurizio; Torchiano, Marco; Ardito, Luca. - In: EMPIRICAL SOFTWARE ENGINEERING. - ISSN 1382-3256. - ELETTRONICO. - 24:5(2019), pp. 3205-3248. [10.1007/s10664-019-09722-9]

Availability:
This version is available at: 11583/2732445 since: 2021-04-01T19:30:58Z

Publisher:
Springer

Published
DOI:10.1007/s10664-019-09722-9

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
Springer postprint/Author's Accepted Manuscript

This version of the article has been accepted for publication, after peer review (when applicable) and is subject to Springer Nature's AM terms of use, but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/10.1007/s10664-019-09722-9

(Article begins on next page)

20 April 2024

Noname manuscript No.
(will be inserted by the editor)

Scripted GUI Testing of Android Open-Source Apps: Evolution of Test Code and Fragility Causes

Riccardo Coppola · Maurizio Morisio · Marco Torchiano · Luca Ardito

Received: date / Accepted: date

Abstract Background. Evidence from empirical studies suggests that mobile applications are not thoroughly tested as their desktop counterparts. In particular, GUI testing is generally limited. Like web-based applications, mobile apps suffer from GUI testing fragility, i.e., GUI test classes failing or needing interventions because of modifications in the AUT or in its GUI arrangement and definition.

Aims. The objective of our study is to examine the diffusion of test classes created with a set of popular GUI Automation Frameworks for Android apps, the amount of changes required to keep test classes up to date, and the amount of code churn in existing test suites, along with the underlying modifications in the AUT that caused such modifications. We defined 12 metrics to characterize the evolution of test classes and test methods, and a taxonomy of 28 possible causes for changes to test code.

Method. To perform our experiments, we selected six widely used open-source GUI Automation Frameworks for Android apps. We evaluated the diffusion of the tools by mining the GitHub repositories featuring them, and computed our set of metrics on the projects. Applying the Grounded Theory technique, we then manually analyzed diff files of test classes written with the selected tools, to build from the bottom up a taxonomy of causes for modifications of test code.

Results. We found that none of the considered GUI automation frameworks achieved a major diffusion among open-source Android projects available on GitHub. For projects featuring tests created with the selected frameworks, we found that test suites had to be modified often – specifically, about 8% of developers' modified LOCs belonged to test code and that a relevant portion (around 50% on average) of those modifications were induced by modifications in GUI definition and arrangement.

Conclusions. Test code written with GUI automation fromeworks proved to need significant interventions during the lifespan of a typical Android open-source project. This can be seen as an obstacle for developers to adopt this kind of

Riccardo Coppola, Maurizio Morisio, Marco Torchiano, Luca Ardito
Department of Computer and Automation Engineering
Politecnico di Torino, Italy
E-mail: first.last@polito.it

test automation. The evaluations and measurements of the maintainance needed by test code wrtitten with GUI automation frameworks, and the taxonomy of modification causes, can serve as a benchmark for developers, and the basis for the formulation of actionable guidelines and the development of automated tools to help mitigating the issue.

Keywords Mobile Development · Automated Software Testing · GUI Testing · Software Evolution · Software Maintenance

1 Introduction

With its ninth release already announced, the Android OS has established itself as the preferred operating system among mobile users (featured by 87.7% of handheld devices in Q2 2017[1]) and as one of the most popular among all families of OSs.

Today's mobile applications (from now on referred to as *Apps*) have reached a very high complexity. This should encourage a thorough Verification and Validation phase in the development process, to make sure that the promised features are actually offered to the users without the occurrence of crashes or malfunctionings. A relevant focus should be posed on testing the GUI (Graphical User Interfaces) of apps, since most of the interaction with the user is carried through them, involving sets of different input modalities and gestures to be recognised.

However, in several studies in literature (e.g., the one conducted by Kochhar et al. [25]) it is given evidence that the automated testing culture among open-source Android developers is not so well established. This fact can be justified by several characteristics that are proper of Android apps, as pointed out by Muccini et al. [39], like the great quantity of different context events to which the apps have to react properly, the diversity of devices and configurations where apps will eventually be deployed, the very fast pace of evolution of the operating system, the lack of resources that has been intrinsic for a long time for mobile devices. In addition to that, Android tests that traverse the GUIs of the apps are particularly prone to fragilities, i.e., they may fail or need interventions if even small changes are operated in the GUI, without modifications of the functionalities of the app. In our previous works we detailed – as a case study – the maintenance of a small test suite that we developed for a popular Android app (along with a preliminary characterization of fragility causes), finding that modifications performed on its appearance could lead to a need for maintaining up to 75% of the test cases [7].

Several approaches exist for automated GUI testing of Android applications. In this work we focused on a set of GUI Automation Frameworks and APIs, defined by Linares-Vásquez et al. as interfaces for obtaining GUI-related information (such as the hierarchy of components on screen) and simulating user interactions with a device [32]. We searched for the use of those tools among Android open-source applications whose source code is hosted on GitHub, and quantified their diffusion and the amount of test code developed with them. This estimation of the penetration of GUI testing frameworks extends the context of previous studies based on the F-Droid repository [25], since it is based on a set of about 20 thousand

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

open-source projects, and track how many of them were due to changes in the AUT.

We then quantified the amount of maintenance (in terms of raw count of modified lines of code) that had to be performed in code associated to the six selected GUI automation frameworks between subsequent releases of open-source projects, and – with the aid of a novel set of metrics of our definition – estimated the occurrence of fragilities for both test classes and test methods. Finally, through the application of the Grounded Theory technique on Git diff files, we constructed from the bottom up a possible fine-grained taxonomy of the types of modifications – performed either on production code or graphical appearance of Android apps – that may trigger the necessity of maintenance on test methods. We applied the classification to a wide sample of modified test classes featuring the six considered GUI automation frameworks, so that we could extract and discuss statistics about the occurrence of modification causes.

The current manuscript has been conceived as an extension of one of our previous conference papers and a companion journal paper, in which we first presented our research questions and metrics [8][10]. The original mining procedure, and the selected testing frameworks considered for the analyses, are common to the three papers. The following changes have been introduced in the current work, with respect to the original papers:

- Modified definition of fragility, and changed set of metrics. Fragility occurrence is now computed quantitatively on a set of diff files and not estimated auomatically from the number of modifications performed in test methods, as opposed to our previous works;
- Running samples for the computation of metrics (with intermediate results for releases and test classes) on a real project, reported as an appendix of the current manuscript;
- Additional filtering of the projects of the context and on the Java classes to be identified as *test* classes. This additional filtering provides a more precise heuristic to extract GitHub repositories actually containing real Android apps provided with an user interface. The results of a manual validation of the precision of such heuristic, which was missing in the original conference paper, is now reported;
- More thorough discussion about the measures obtained for the considered projects;
- Presentation of a taxonomy of modification causes, built through the application of the grounded theory technique. The application of the grounded theory technique was given a preliminary presentation in another conference work of ours [9], but on a set of 945 diff files involving the Espresso GUI Automation framework only. In this manuscript, we present an extended Grounded Theory study performed on a set of 1724 diff files, involving four different automation frameworks. The application of the grounded theory technique provides an additional validation of the derived taxonomy, proving also its generalizability to different GUI Automation frameworks..

The remainder of the paper is organized as follows: section 2 provides some information about the ways of testing Android apps proposed in literature, the challenges that are proper of Android app testing, and the concept of test fragility; section 3 explains the Research Questions we aimed to answer with this study,

the set of metrics we defined and the theoretical basis of the construction of the taxonomy; section 4 gives details about the procedure through which we defined our context of open-source projects, computed our statistics and built the proposed taxonomy; section 5 presents statistics about diffusion, maintenance and fragility of open-source projects featuring the considered GUI Automation Frameworks; section 6 describes the taxonomy of modification causes we defined, and gives the results of the classification applied to the considered set of projects; section 7 discusses related work on the topic; section 8 exposes the threats to the internal and external validity of the present study; section 8 draws the conclusions and the possible expansions and prosecutions of the present work.

2 Background

This section provides an introduction to Android testing, and a high-level survey about existing testing techniques and the challenges they present. Information is also provided about the Grounded Theory technique, that has been used to derive the proposed taxonomy of modification causes for Android test code.

2.1 Testing Android apps

Android (and mobile, in general) applications showcase some differences from traditional desktop applications. *Mobile apps* are defined by Muccini et al. [39] as mobile software (i.e., applications that run on mobile devices) taking input from the context where they are executed (for instance, contextual sensing and adaptation, and context-triggered actions).

Mobile apps can be divided in three categories: native apps, if they are designed and programmed for a specific platform, according to its guidelines; web-based apps, if they are developed as normal web applications that are then loaded in a browser installed on the mobile device; hybrid apps, if they contain native components in their interface, that are then used for launching a core that is still web-based.

Mobile testing can be defined, as done by Gao et al., as "testing 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" [13].

The first and most immediate option for testing Android applications and their GUIs is the execution of manual test cases. In a study by Linares-Vásquez et al. in the field of performance testing [30], manual testing is identified as the option preferred by developers, along with an examination of reports and feedback from users. The technique, as discussed by Kropp et al. [26], is however not exhaustive, error prone and not easily reproducible.

Automated testing of mobile apps can be performed on a series of different levels: in addition to the traditional unit testing, integration testing, system testing and regression testing, scopes that are specific to the mobile scenario must be considered. Kaur et al. [23] highlight the importance of compatibility testing (i.e., to ensure that the application works on different handheld models and/or OS versions), performance testing (i.e., to ensure that the mobile devices do not

consume too many of the resources available) and security testing. GUI testing is identified as a very prominent testing need for all mobile applications. For Android applications, GUI testing is focused on testing the *Activities* (i.e., the components in charge of managing the appearance and the composition of each screen exposed to the user) and the transitions between them.

Various classifications have been provided in literature for testing techniques of mobile applications. As pointed out by Alegroth et al. [1], automated GUI testing techniques can be generally – and not only in the mobile domain – categorized under three different *generations*: *coordinate-based* testing techniques, that identify the elements to test according to their absolute position on screen, and hence provide very limited dependability; *widget/component-based* testing technique, that identify elements of the interface according to the definition and arrangement of layouts, and their properties; *image-recognition* testing techniques, that recognize the elements on screen by their visual appearance.

Linares-Vásquez et al. [32] classify the approaches to automated testing of Android applications in different families, according to the level of knowledge that the tools have of the code of the application, and to the way the inputs are generated and distributed. GUI-Automation Frameworks and APIs provide a basic interface to obtain GUI-related information, and to exercise the widgets and objects the Activities are made of; those frameworks can be used to manually write down testing code with sequences of operations to be performed on the AUT (Application Under Test), and are investigated in several other studies existing in literature, like the ones by Kropp et al. [26] and Singh et al. [45].

Capture & Replay testing tools (examples are presented in works by Gomez et al. [17], Kaasila et al. [22] and Liu et al. [33]) record the operations performed on the GUI to generate repeatable test sequences, thus providing a faster and cheaper alternative to the manual writing of test scripts. Event-sequence generation tools are based on the construction of test cases as streams of events, that then can be inserted in repeatable scripts: the works by Choi et al. [5] and Jensen et al. [21] provide examples of such paradigm.

A significant amount of research in the field of mobile testing is focused on techniques for Automated Test Input Generation. Without the need for in-depth information about the AUT, Random and Fuzzy input generation techniques feed random sequences of inputs to activities, in order to trigger potential defects and crashes. Monkey[2] is the random tester supported by Android. Random testers can be applied after a model of the user interface is created (like it is done in works by Machiry et al. [34], Moran et al. [38], Zhauniarovich et al. [53]) to distribute the input given to the interface in a more intelligent way.

Model-based input generation techniques leverage models (typically Finite State Machines or Event-Flow Graphs) of the GUI of the AUT, that can be created manually or extracted automatically with a process called GUI ripping. Such models are therefore used to generate systematic test cases traversing the GUI. The tools and studies by Amalfitano et al. [2, 3] and Yang et al. [50] can serve as examples of this approach.

[2] https://developer.android.com/studio/test/monkey.html

2.2 Challenges in testing Android Apps

Several studies (like the ones by Kaur et al. [23] and Muccini et al. [39]) are focused on the peculiarities of Android apps that make testing them properly a complex challenge: limited energy, memory and bandwidth; rapid changes of context and connectivity type; constant interruptions caused by system and communication events; the necessity to adapt the input interface to a wide set of different devices; very short time to market; very high multitasking and interaction with other apps.

Pinto et al. [40] found that time constraints, compatibility issues, complexity and lack of documentation of available testing tools are among the most relevant challenges experienced by the interviewed developers, that may therefore discourage them from testing their applications.

Test fragility (of which a general definition has been provided by Garousi et al. [15]) represents a problem for different kind of software, and has been largely explored in literature: Grechanik et al. [18], and Memon et al. [35] proposed approaches for automatically fix broken test cases for GUI-based applications. Gao et al. developed SITAR [14], a technique to automatically repair test suites, modeling and repairing test cases using Event-Flow Graphs (EFGs).: Leotta et al. [27, 28] report a study about fragility of web application GUI tests, in which they compared the fragility of different approaches used for the identification of GUI widgets.

For our purposes and in the remainder of this manuscript, we use the following definition:

> *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.*

Being system level tests, test cases developed with GUI automation frameworks may be affected by variations in the underlying functionalities of the app, but also from even small interventions in the appearance and presentation of the screens by which the GUI is composed. Our definition distinguishes 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.

With the definition provided, a certain amount of fragility is expected during the lifespan of any test case: testware, in fact, is supposed to follow the evolution of the Application Under Test, and hence it is likely that significant interventions in the AUT would require the related test cases to be updated accordingly. However, high values for our fragility measurements may indicate a tendency of test cases to be invalidated even by trivial (in terms of LOCs modified, or in relevance of the changed feature) variations applied to the AUT, and hence suggest the need for high effort in performing maintenance of testware during the normal evolution of the application.

2.3 Taxonomies and Grounded Theory

Objectives, quality criteria and methodological guidelines for obtaining process theories and taxonomies in Software Engineering have been provided by Ralph [41].

Taxonomies are identified as an important mean for software engineering studies, to provide nomenclature helping understanding and describing the entities of a specific domain. Taxonomies can be organized or not around mutually exclusive classes, and can provide hierarchies of classes for different layers of categorization of entities. According to Ralph, the following strategies can be followed to obtain a taxonomy: Grounded Theory, Interpretative Case Studies, Single-Source primary studies, or personal experience.

Stol et al. provide a critical review and a set of guidelines about the use of Grounded Theory in Software Engineering research, and a literature review of existing studies leveraging the method. [46]. Grounded Theory (GT), originally introduced by Glaser and Strauss [16], is a qualitative method to generate theory from data that is progressively and comparatively analyzed, instead of applying an existing theory to answer a Research Question. In its classic form, also called Glaserian grounded theory, GT does not contemplate the a priori definition of a Research Question: instead, the Research Question emerges from the research itself. Also, the process of reviewing existing literature should be delayed as much as possible, to avoid that existing concepts influence the emerging theory. A later revision of the method by Strauss and Corbin [47] (also called Straussian grounded theory) allows a prior definition of a research question, and the consultation and utilization of concepts from literature during the generation of the theory.

Both versions of Grounded Theory suggest different types of *coding* activities: *Open Coding* is based on text data, which is examined line by line to capture the main concepts of the theory, and the categories that can be based on them. During Open coding, researchers can leverage the practice of *Memoing*, to sketch down the knowledge emerging from data during the examinations [4]. *Axial Coding* is the process of understanding how codes, concepts and derived categories are linked one to another, to identify a structure in the theory that is being built. *Selective Coding* is the procedure of defining a central category to which all other macro-categories can relate.

Several examples of taxonomies created through Grounded Theory are available in literature. For instance, Scott et al. leverage Straussian GT to create a taxonomy of affect in online-chats [42]; Sedano et al. create a taxonomy of waste in software development applying GT to an observation study of software companies [43].

3 Study Design

The main objectives of the present work are: estimating the penetration of GUI testing frameworks, assessing the amount of modifications performed in test code, giving a categorization of the motivations that induce the need for modifications in test suites, and quantifying the fraction of them that are related to modifications in the underlying AUT and its GUI.

These goals entail answering the following research questions:

RQ1 Diffusion: how many open-source Android projects leverage GUI automation frameworks, and how much test code do they feature?

RQ2 Evolution: how much is test code associated to GUI automation frameworks modified over different releases of Android open-source projects?

Table 1 Metrics definition

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

RQ3 Modification Causes: what are the main causes behind the need for maintaining GUI test code in Android open-source projects?

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

The first step of our research was an estimation of the diffusion of Android GUI automation frameworks. We started from a repository of Android open-source applications – we selected GitHub for this purpose – and we performed a code search in order to detect the usage of a set of six testing GUI Automation Frameworks that are frequently cited in literature.

Then, we studied how applications (and their test code) were changed throughout their release history, by means of file-by-file comparisons. Finally, with the aid of an automated shell script, we tracked the modifications of individual test classes and methods to compute a set of change indicators, explained in detail in section 3.1. Test code that underwent modifications has been finally manually examined to construct a taxonomy of modification causes, and to compute the frequency of occurrence of individual causes.

3.1 Metrics Definition

We defined a set of metrics that can be divided into two groups: Diffusion and Size metrics, that characterize the amount of test code and the size of the tested projects, and Test Evolution metrics, that give insights about the evolution of the test code during the lifespan of the project. Table 1 reports the metrics together with the relative descriptions. The metrics are explained in detail in the following subsections.

The metrics we defined are normalized, to allow comparison across projects of different sizes. Most of them can be defined on top of lower-level metrics for the quantification of absolute changes in test classes and test cases. For instance, Tang et al. [49] report eighteen basic metrics for the description of bug-fixing change histories (e.g., number of added or removed files, classes, methods or dependencies).

In Appendix B we investigate a GitHub project to give a running sample of the computation of the full set of metrics.

3.1.1 Diffusion and size (RQ1)

To estimate the diffusion of Android automated GUI testing tools and of the size of test suites using them, we defined the following five metrics:

TD (Tool Diffusion) is defined as the percentage, among the set of Android projects in our context, of those featuring a given testing tool.

NTR (Number of Tagged Releases) is the number of tagged releases of an Android project (i.e., the ones that are listed by using the command *git tag* on the GIT repository). This metric can be used to understand what is the nature of the applications that are more likely tested using GUI Automation Frameworks.

NTC (Number of Test Classes) is the number of classes featured by a release of an Android project, featuring code associated to a specific tool.

TTL (Total Test LOCs) is the number of lines of code that can be attributed to a specific GUI automation framework in a release of an Android project.

TLR (Test LOCs Ratio) defined as

$$TLR_i = TTL_i / Plocs_i,$$

where $Plocs_i$ is the total amount of Project LOCs (including both program and test code) for release i. This metric, lying in the $[0, 1]$ interval, allows us to quantify the relevance of the testing code attributed to a given GUI automation framework.

3.1.2 Test suite evolution (RQ2)

The metrics addressing RQ2 aim to describe the evolution of Android projects and the relative test suites; they have been computed for each pair of consecutive tagged releases.

MTLR (Modified Test LOCs Ratio) defined as

$$MTLR_i = Tdiff_i / TTL_{i-1},$$

where $Tdiff_i$ is the amount of added, deleted or modified LOCs in classes associated to a given GUI testing framework between tagged releases $i-1$ and i, and TTL_{i-1} is the total amount of test LOCs in release $i-1$ (the metric is defined only when $TTL_{i-1} > 0$, i.e., the previous version is provided with test code). This quantifies the amount of changes performed on existing test LOCs for a specific release of a project.

MRTL (Modified Relative Test LOCs) defined as

$$MRTL_i = Tdiff_i / Pdiff_i,$$

where $Tdiff_i$ and $Pdiff_i$ are respectively the amount of modified (or added, or deleted) LOCs in code associated to a given GUI automation framework and in the whole project (including both program and test code), in the transition

between release $i-1$ and i. It is defined when $Pdiff_i > 0$ and computed only for releases with test code associated to the GUI automation framework under inspection (i.e., $TRL_i > 0$). This metric lies in the $[0, 1]$ range. Values close to 1 imply that a significant portion of the total effort in making the application evolve is needed to keep tests up to date.

MRR (Modified Releases Ratio), computed as the ratio between the number of tagged releases in which at least a test class associated to a given GUI automation framework has been modified, and the total amount of tagged releases. This metric lies in the range $[0, 1]$ and bigger values indicate a minor adaptability of the test suite – as a whole – to changes in the AUT.

TSV (Test Suite Volatility), is defined for each project as the ratio between the number of test classes associated to a given GUI automation framework that are modified at least once in their lifespan, and the total number of test classes of the project history.

MCR (Modified test Classes Ratio) defined as

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

where MC_i is the number of classes associated to a given GUI automation framework that are modified in the transition between release $i-1$ and i, and NTC_{i-1} the number of classes associated to the framework in release $i-1$ (the metric is not defined when $NTC_{i-1} = 0$). The metric lies in the $[0, 1]$ range: the larger the values of MCR, the less test classes are stable during the evolution of the app.

MMR (Modified test Methods Ratio) defined as

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

where MM_i is the number of test methods associated to a given GUI automation framework that are modified between releases $i-1$ and i, and TM_{i-1} is the total number of methods associated to the framework in release $i-1$ (the metric is not defined when $TM_{i-1} = 0$). The metric lies in the $[0, 1]$ range: the larger the values of MMR, the less test methods are stable during the evolution of the app they test.

MCMMR (Modified Classes with Modified Methods Ratio) defined as

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

where $MCMM_i$ is the number of classes associated to a given GUI automation framework that are modified, and that feature at least one modified method between releases $i-1$ and 1. The metric is not defined when $NTC_{i-1} = 0$. The metric is upper-bounded by MCR, since by its definition $MCR_i = MC_i / TC_i$, and $MCMM_i \leq MC_i$.

3.2 Selected GUI Automation Frameworks

We have chosen six different popular GUI Automation Frameworks for our investigations. We selected open-source tools that were already considered in similar explorations of the testing procedure of Android applications. All the tools give the possibility to write test scripts in Java: this was a fundamental inclusion criterion since the computation of some of the metrics requires comparisons with the production code of Android apps, written in Java. The selected set of GUI Automation Frameworks covers most of the possible peculiarities that can be attributed to open-source GUI Automation Frameworks [31].

The first two tools we have searched for are part of the official *Android Instrumentation Framework*[3]. *Espresso*[4] [24] is an open-source automation framework that allows to test the GUI of a single application, leveraging a gray-box approach (i.e., the developer has to know the internal disposition of elements inside the view tree of the app, to write scripts exercising them). Espresso allows to test one activity at a time. Recent extensions of the Espresso GUI automation frameworks allow the development of test scripts using the Capture & Replay approach, with the use of the Espresso Test Recorder tool[5]. *UI Automator*[6] [6, 29] adds some functionalities to those provided by Espresso: it allows to check the device status and performance, to perform testing on multiple applications at the same time, and operations on the system UI. Both tools give only partial support to the test of non-native apps.

Selendroid[7] [48] is a testing framework based on Selenium, that enables GUI black-box testing of native, hybrid and web-based application; the tool allows to retrieve elements of the application and to inspect the current state of the GUI without having access to its source code, and to execute the test methods on multiple devices at the same time.

Robotium [19, 52] is an open-source extension of JUnit for testing Android apps, that has been one of the most used testing tools since the beginning of the diffusion of Android programming; it can be used to write black-box test scripts or function tests (if the source code is available) of both native and web-based apps.

Robolectric[8] [2, 36, 37] is a tool that can be used to perform white-box testing directly on the Java Virtual Machine, without the use of a real device or an emulator; usable for other purposes than GUI testing (especially for the execution of Unit tests of the internal logics of the application), it can be considered as an enabler of Test-Driven Development for Android applications, since the instrumentation of Android emulators is significantly slower than the direct execution on the JVM.

Appium [44, 45] leverages WebDriver and Selendroid for the creation of black-box test methods that can run on multiple platforms (e.g., Android and iOS); test methods can be created through an inspector that enables basic functions of recording and playback, image recognition, or manual coding. It can be used to

[3] https://developer.android.com/studio/test/index.html
[4] https://developer.android.com/training/testing/ui-testing/espresso-testing.html
[5] https://developer.android.com/studio/test/espresso-test-recorder
[6] https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html
[7] https://github.com/selendroid/selendroid
[8] http://robolectric.org/

test both native and web-based applications. Test scripts can be data-driven, and can be created with the Capture & Replay technique using the *inspector* module.

3.3 Procedure

The following paragraphs describe in more detail the steps performed to conduct this study.

3.3.1 Context Definition and Test Code Search (RQ1)

The approach we adopted for the selection of the context (i.e., the set of projects that we used for the subsequent study) is a sequence of different steps, the first one being a search for the word "Android" in descriptions, readme files and names of projects. The Repository Search API of GitHub has been leveraged to this purpose. The data mining procedure has been performed between September and December 2016. We leveraged the cURL bash function, inside a Linux bash script, to access the GitHub Repository Search API, and we then examined the output – which is in given as a json file – using the jsawk tool. Since the API returns at most 1000 projects, we performed multiple searches, limiting the amount of possible results by cycling over different disjoints date ranges (the *created* parameter of the API is used for this purpose), starting from before 2013 to the end of 2016.

All the projects that have no tagged releases were cut out from the context. This has been done because the aim of the experiment was to track the evolution of the projects, by means of computing differences between tagged releases (as it is explained later). That considered, projects without at least a single tagged release (which allows for a single comparison, made between it and the master release) were not of interest. To know how many releases were featured by each repository, we leveraged the GitHub Tags API, which outputs the names of all the tagged releases of a given GitHub repository.

Searching for the keyword "Android" alone would have included in the context libraries, utilities, and applications intended to interface with Android counterparts. Hence, a third filtering has been applied, in order to cut out these spurious results. Since it is mandatory for any Android app to have a Manifest file in its root directory, we searched for the "Manifest" keyword using the GitHub Code Search API in all the remaining repositories. We cut out from our context all the projects which returned an empty result for this query.

To limit the investigations on the test code only to applications that had an actual GUI, we performed an additional filtering phase, cutting out from our context all the projects that did not feature either any call to the "setContentView" method or any declaration of a FragmentTransaction object. The "setContentView" method is typically called as the first one in the onCreate() method, which is itself called as the beginning of the lifecycle of any Activity of the application, and is used to load a layout file to populate the GUI of the current screen of the app. The FragmentTransaction object is used to construct the GUI of the current Activity dynamically through the instantiation of Fragments.

Examining the resulting projects, it could be noticed that many of the repositories were clones of the Android sdk or sets of applications/frameworks that were not of interest for our study. We removed the projects named this way from the

sets of projects that we considered for our further investigations. All the projects that were named after the following list were hence automatically removed from our context, along with the ones which featured combinations of the keywords "android", "platform", "frameworks" and "base" in their full name.

Once a filtered list of Android projects was obtained, the GitHub Code Search API has been leveraged to search in the repositories for the name of the frameworks and include statements that can be attributed to them. We considered any ".java" file featuring the name of a testing technique in its code as a test class (for instance, a class featuring the statement "import static android.support.test. espresso.Espresso. onView;" is considered as a class featuring Espresso). To avoid having false negatives when identifying Android projects featuring a given tool (e.g., improper usage of functions related to the GUI Automation Frameworks in files that were not test files, or presence of keywords related to the GUI automation frameworks inside insignificant sections of Java code) we also set the condition that a .Java class had to contain the "test" keyword in its absolute path to be considered as a test class, and we removed line and block comments before searching for keywords related to the six selected GUI Automation Frameworks. Six subsets of the context were thus obtained, each containing all the projects featuring code that can be associated to one of the selected GUI Automation frameworks. All the repositories of these six subsets were cloned locally. Sets of projects featuring different testing tools are not necessarily disjoint: it is possible that a repository features more than just one scripted testing tool.

For each tool its adoption has been estimated by means of the TD metric. For each test class the lines of test code have been counted using the cloc[9] tool, so that TTL and NTC could be computed for each project, on the master release. The use of the *git tag* command allowed to obtain the NTR metric.

3.3.2 Test LOCs analysis (RQ2)

To answer RQ2, for each pair of consecutive tagged releases of any project, the total amount of modified LOCs was computed.

Then, the total amount of LOCs added, removed or modified in classes associated to the featured testing frameworks was computed. Throughout all our study, we have considered moved or renamed files as different test files. The *git diff* command was used to perform all computations of changed lines for the whole projects or for individual files associated to the testing tools

Those measurements allowed to compute TLR (that we also tracked throughout the history of the projects in addition to measuring it statically on the master release), $MTLR$, and $MRTL$ for each tagged release of the project.

Finally, when the exploration of the project history was complete, global averages were computed: $\overline{TLR} = Avg_i\{TLR_i\}$, $\overline{MTLR} = Avg_i\{MTLR_i\}$, $\overline{MRTL} = Avg_i\{MRTL_i\}$, with $i \in [2, NTR]$, being NTR the number of tagged releases featured by the project.

Volatile classes (i.e., classes featuring modifications throughout their lifespan) have been identified inside each project, in order to compute the TSV value.

We have then tracked the evolution of single test classes and methods, taking into account the tagged releases in which each test class has been added, modified

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

or deleted. To enumerate the methods in each test class, and compare the methods to identify the added, deleted or modified ones, we leveraged an open-source tool, JavaParser[10], that we integrated in our bash script.

For each tagged release we have obtained the number of modified classes and methods, i.e., MCR, MMR, and $MCMMR$. Also in this case, at the end of the exploration averages have been computed as $\overline{MCR} = Avg_i\{MCR_i\}$, $\overline{MMR} = Avg_i\{MMR_i\}$, $\overline{MCMMR} = Avg_i\{MCMMR_i\}$, with $i \in [1, NTR]$.

3.3.3 Diff files analysis (RQ3, RQ4)

To analyze the causes behind the modifications in test code, we collected all the diff files pertaining to classes with code associated to the selected GUI Testing Frameworks, containing modifications in test methods.

According to Ralph's guidelines for the construction of taxonomies [41], we followed the Grounded Theory approach. In accordance with the Straussian definition of Grounded Theory [11], our Research Question (RQ3) was defined upfront, as a follow-up of the previous ones already answered, and did not emerge from the research.

As the *site* for our taxonomy construction, we selected the repository of Android open-source projects mined in the first step of the study, whereas the *Data Collection* is performed through *technical observation* of the extracted diff files. Starting from the 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. When individual widgets were involved in the modified test LOCs of the examined diff files, the layout files where such widgets are defined were identified and inspected for the presence of modifications in their definition, properties or widgets arrangement; also, the activities where the identified layouts are inflated were inspected, to understand whether the modifications in the test code derived from changes in the application behaviour. When the modified LOCs in test classes were not evidently linked to individual widgets of the screen, the modifications were considered as pertaining test code only, and no examinations on associated source code were performed.

Based on those inspections, the categories of the taxonomy were generated through *Open Coding*, and each modification has been categorized under one or more classes of the taxonomy.

Causes of modifications of test methods have not been considered as mutually exclusive, i.e., two different causes could concur to a single modification operated on a test method. The taxonomy has been built incrementally, with new categories of modification causes added every time a modification appeared to be unclassifiable under the available categories. Categories of modification causes were linked one to another, leading to the construction of a set macro-categories. Comparisons have been performed constantly over already labeled modifications, any time a new category was added to the taxonomy.

The open coding procedure was performed by one of the authors of the manuscript, involving two iterations over the collected set of diff files. All the diff files containing modifications in test methods for the Android open-source projects associated to the GUI Automation Frameworks Espresso (819 diff files), Robotium (424 diff

[10] http://javaparser.org/

Table 2 Metrics for RQ1

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

files) and UI Automator (59 diff files) were manually examined. Sets of diff files with modifications on projects featuring Appium and Selendroid were not considered, due to their size, way smaller with respect to the other ones. On the contrary, the sets of diff files extracted for Robolectric were subsampled, due to the size of the set of modified methods featuring Robolectric, that did not allow to perform manual open coding on the entirety of the set. This selection led to 422 examined diff files (out of the full set of 4221) of test classes associated to Robolectric. To sum up, the open coding procedure involved the manual examination of a total of 1724 diff files.

The application of the taxonomy over the four sets of diff files also served as a conceptual evaluation of its transferability, dependability and confirmability when different GUI Automation Frameworks are taken into account.

The percentage of modifications of test classes that can be justified by modifications in the AUT or in its GUI has been finally computed, in order to obtain an estimation of the fragility of test suites developed using GUI Automation Frameworks.

We have made available a replication package containing all the test scripts used for the execution of this study on our website[11].

4 Measures for Diffusion and Evolution

In the following paragraphs, we report the results we obtained by applying the procedure described in section 3.4. The results measured for the metrics defined in section 3.1 are detailed, along with the conclusions we can base on them. The full set of raw intermediate data measured about classes and releases of each project has been made available online[12].

4.1 Diffusion and Size (RQ1)

We initially gathered a total of 280,447 GitHub repositories featuring the term *Android* in their names, descriptions or readmes. Then, a significant amount of projects were pruned because of their lack of tagged releases (so they had no history to be investigated), or Manifest files, obtaining a set of 18,930 Android projects (6.75% of the initial number of projects). After the last filtering phase, in which we cut out from the context all the projects which were not likely to offer any GUI to

[11] http://softeng.polito.it/coppola/replication_package.zip
[12] https://figshare.com/articles/Testing_Fragility_Data_-_ESEM/7149104

Table 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 user, we obtained a final set of 15,326 Android projects (5.46% of the original number of projects retrieved by just searching ''Android"). We graphically report in Appendix C all the projects that remained in the considered context after each filtering phase we applied.

Table 3 summarizes the metrics gathered to answer RQ1. The columns show: the total number of projects featuring the GUI automation frameworks considered; the Tool Diffusion (TD) metric; the average and median values for Number of Tagged Releases (NTR), Number of Test Classes (NTC), Total Test LOCs (TTL) and Test LOCs Ratio (TLR), computed on the master releases of the projects. Last line in table 3 shows an average for all the projects considered, weighted by the number of projects for each set. The acronyms used in the subsection, and their meanings, are summarized in table 2.

Even though an overestimation may be possible due to overlaps (since the sets for the individual tools are not necessarily disjoint) slightly less than 8% of the filtered set of Android projects feature tests belonging to at least one of the five selected tools. None of the testing frameworks reached by itself a significant level of diffusion. To provide a comparison value to those extracted for the GUI Automation Frameworks, the JUnit unit testing tool was present in 20% of the projects of our context. The absolute number of projects featuring Selendroid and Appium test classes is practically irrelevant. A higher number (the 4.11% of the total) of projects featuring Robolectric has been found, but the tool has been available for a longer time than the others (especially Espresso and UI Automator) and is often used solely for Unit Testing. A single project (namely, moneymanagerex/android-money-manager) featured the Selendroid tool after the filtering phases that were performed. Hence, statistics about such tool have not been included in table 3.

Although the total number of Android projects extracted can take into account some projects that are not likely to feature test code (e.g. experiments, duplicates, exercises, prototypes, projects that are abandoned at very early stages) the statistics extracted about the metric TD give evidence of the lack of an extensive usage of scripted automated GUI testing on Android. However, it must be taken into account that the study we performed is limited to the GUI automation frameworks we considered, hence it is possible that different scripted testing tools are used by some other projects of the context.

The average and median number of classes featuring a GUI testing framework can be quite small (e.g., just 3 and 2, respectively, in the case of Espresso) possibly due to the typical coding patterns for Android applications, according to which – usually – one test class is written specifically for each Activity. Most apps – this is particularly evident for small or experimental open-source projects – do

not feature many screens to show to their users, and therefore they do not feature many activities to be tested.

The Number of Tagged Releases (NTR) of the examined project was between 13 (for projects featuring Espresso) and 37 (for projects featuring Appium). Those extreme values can give indications about the nature of the projects that are more likely adopting the studied GUI Automation frameworks. The smaller average number of tagged releases for projects featuring Espresso can suggest a preference of the use of the Espresso testing framework for smaller applications, with projects having shorter lifespans. On the other hand, the higher value for projects featuring Appium suggests the adoption of such tool in longer-lived projects. However, in the case of Appium the result can be heavily influenced by the small size of the set that is considered (just 12 projects).

Average TTL values are very large for the sets of open-source Android projects featuring code associated to Robolectric and Appium; however, the very distant TLR metrics for the two tools suggests that the amount of test code written with Robolectric is typically more relevant if compared to the amount of project LOCs. On the other hand, Appium appears to be used in bigger projects, hence having very small ratios of test code upon project code.

The fact that the set of projects featuring Espresso has the lowest average TTL can be due to the little effort required for exercising functionalities of the app using such tool, and for the accessibility of the framework to even non-experienced developers. This can lead it to be used also in very small projects, in tryouts, and even for experimental and partial coverage of the use cases of the app.

Answer to RQ1: The considered GUI testing tools reach a diffusion that is always lower than 4.11%. Projects that have their GUI tested on average have 6 test classes, with a total of 908 LOCs (10.49% of the whole project code).

4.2 Evolution (RQ2)

Table 5 shows the statistics collected about the average evolution of test code for the sets of projects featuring Espresso, UI Automator, Robotium and Robolectric. Since from the diffusion analyses a marginal presence of Appium and Selendroid among Android projects emerged, we did not take into account the sets of projects featuring them for the remaining phases of our study. For every set, Test LOCs Ratio ($\overline{TLR}$), Modified Test LOCs Ratio ($\overline{MTLR}$), Modified Relative Test LOCs ($\overline{MRTL}$), Modified Releases Ratio (MMR), Test Suite Volatility (TSV), Modified Test Classes Ratio ($\overline{MCR}$), Modified Test Methods Ratio ($\overline{MMR}$) and Modified Classes with Modified Methods Ratio ($\overline{MCMMR}$) have been averaged on all the projects. The values in last row are obtained as averages of the four values above, weighted by the respective sizes of the four sets. The acronyms used in the subsection, and their meanings, are summarized in table 4.

The values reported for average Test LOCs Ratio ($\overline{TLR}$) show that – when present – GUI testing can be an important portion of the project during its lifecycle, if compared to the number of LOCs of program code. The average values range from about 5.11% (for the set of projects featuring Robotium) to 11.23% (for the set of projects featuring Robolectric).

Table 4 Metrics for RQ2

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

Tool	$\overline{TLR}$	$\overline{MTLR}$	$\overline{MRTL}$	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%

Average Modified Test LOCs Ratio ($\overline{MTLR}$) measures show that typically around 5% of code associated to the selected testing frameworks is modified between consecutive releases. The smallest values were obtained for the projects featuring UI Automator. In general, this should be a consequence of bigger test suites, in terms of absolute LOCs, with respect to the ones written with other testing frameworks. Hence, the influence of a similar amount of absolute modified LOCs would result in a lower $\overline{MTLR}$ value.

The measures about Modified Relative Test LOCs ($\overline{MRTL}$) show that, on average, when GUI automation frameworks are used, the 4.54% of the modified LOCs belong to test classes. With this metric, however, we are still unable to discriminate what is the reason behind the modifications to be performed on test classes. Highest $\overline{MRTL}$ values were found for projects featuring Robolectric: this can be justified by the higher value of $\overline{TLR}$ for these projects, that suggests an higher relevance of test code, which is hence more likely to need modifications during the normal evolution of the projects.

The Modified Releases Ratio (MRR) metric gives an indication about how often the developers had to modify any of their test classes when they published new releases of their projects. On average, 18.37% of releases needed modifications in the test suite (with a maximum of 20.39% for the set of projects featuring Robolectric). Since releases may be frequent and numerous for GitHub projects, this result explains that the need for updating test classes is a common issue for Android developers that are leveraging scripted testing. The average 19.43% value for the Test Suite Volatility (TSV) metric, which characterizes the phenomenon from the point of view of whole test suites, highlights that, on the lifespan of a project, about one fifth of test classes require at least one modification.

The column about the Modified Classes Ratio ($\overline{MCR}$) metric shows that, on average, 15.43% of test classes are modified between consecutive tagged releases in our set of Android projects. The 3.83% average value found for the Modified Methods Ratio ($\overline{MMR}$) metric highlights that the percentage of modified methods

is – as expected – smaller than the percentage of modified classes: this is obviously due to the fact that multiple test methods are contained in single test classes.

Not all modified test classes contained significant modifications. The Modified Classes with Modified Methods Ratio ($\overline{MCMMR}$) metric gives a statistic about the possibility of a modified class to contain modified methods, and not having changes limited to irrelevant sections of code (e.g., import statements). The results collected show that more than half of the classes having modified lines featured modifications inside the code of test methods as well, and hence were worth to undergo the following phase of the study, about the identification of reasons for modifications in test methods.

Answer to RQ2: On average, near 5% of testing code is modified between consecutive tagged releases. 4.54% of the overall LOCs modified between consecutive tagged releases belong to testing code. On average, around 20% of tagged releases require modifications in the test suite, and 20% of any test suite needs modifications during the project history. On each new release, on average, 15.43% of test classes and 3.83% of test methods are modified.

4.3 Precision of the heuristics

We resorted on heuristic methods to automatically identify test classes related to GUI Automation Frameworks inside Android projects hosted on GitHub. It is not assured whether the Java classes extracted through the described procedure are actual test classes, and in case they are so, whether they are actually used for testing features of the AUT related to the GUI. We hence extracted a random sample of classes after we performed our measurements about diffusion and evolution of test code, on which to perform a manual inspection in order to compute the precision of the heuristics we leveraged[13]. The manual inspection was performed by one of the authors of the paper. We first computed the precision in intercepting actual test classes. On the sample set of 100 test classes we used, just 9 of them could not be considered as actual test classes (e.g., they were collections of helper methods for other test classes, or empty examples of the structure of a given GUI Automation Framework), and hence we computed a precision of 91% for the identification of test classes.

Among the 91 Java classes of the sample that could be considered test classes, we also discriminated the ones that were exercising elements of the GUI and those that were not. We did so again with a manual inspection of test code, searching for method calls related to the instantiation or the interaction with Widgets of the user interface, method calls related to the Activity life cycle, or references to textual and/or graphical resources of the application. Of the 91 considered test classes, we found that 63 of them actually tested features related to the GUI of the projects (and hence were true positives of our heuristic) and 28 did not (and hence were false positives of our heuristic). Thus, we measured a precision of 69.3% for the extraction of test classes that were actually testing the GUI of the AUT. Among the test classes that were considered, most of the ones that did not

[13] The sample and the evaluations based on the examined Java classes is available at the following URL: http://softeng.polito.it/coppola/precision_evaluations.csv

interact with the GUI of the AUT were associated to the Robolectric testing tool. This was an expected finding, since – even though it is frequently categorized as a GUI Automation Framework [32] – Robolectric can be leveraged simply as an automation engine for unit or integration testing of Android apps, without even instantiating the widgets composing the GUI.

5 Taxonomy of Modification Causes in Test Methods

The present section illustrates the taxonomy that has been derived, and the frequency of occurrence of each category of modifications among the examined sets of diff files.

A previous effort towards the classification of modifications performed on generic test code has been provided by Yusifoglu et al. in their systematic mapping of test-code engineering, where they identified four types of maintenance activities for test code [51]: *Perfective maintenance*, when modifications are performed only to improve the quality of test code (e.g., refactoring); *Adaptive maintenance*, when modifications to test code are performed to follow the evolution of the production code; *Preventive maintenance*, when test code is modified after the detection of defects (e.g., test code smells or redundancies); *Corrective maintenance*, when bugs are found in test code and fixed. The novelty of the taxonomy we derived is assured by the lack of prior effort in defining a detailed taxonomy of modification causes of test code specific to Android (and mobile, in general) development.

5.1 Modification Causes (RQ3)

In this section, the taxonomy of all the causes of test modifications that have been found is shown. The individual causes have been divided in nine macro-categories. The first, *Test Logic Change*, is not related to the GUI of the AUT; the macro-categories *Application Logic Change*, *Execution Time Variability*, *Compatibility Issues* are linked to the AUT, but not specifically to its GUI. The remaining five categories are related to the GUI of the app, on its elements and their definition, on the access to graphic widgets, and on the transitions between them.

The graphic taxonomy is shown in figure 1 (the three groups of macro-categories described before have been depicted using different colors), whilst detailed descriptions of all causes are given in the following.

5.1.1 Test Code Change

This category covers all the changes in test code that are not due to modifications in the user interface, nor to changes in the actual behaviour and functionalities of the app, but only to the way tests are actually set up and executed. Those modifications occur on test code only, and have no evident connection with production code. According to the classification of test code maintenance provided by Yusifoglu et al. [51], this macro-category covers the modifications related to Perfective, Preventive and Corrective maintenance.

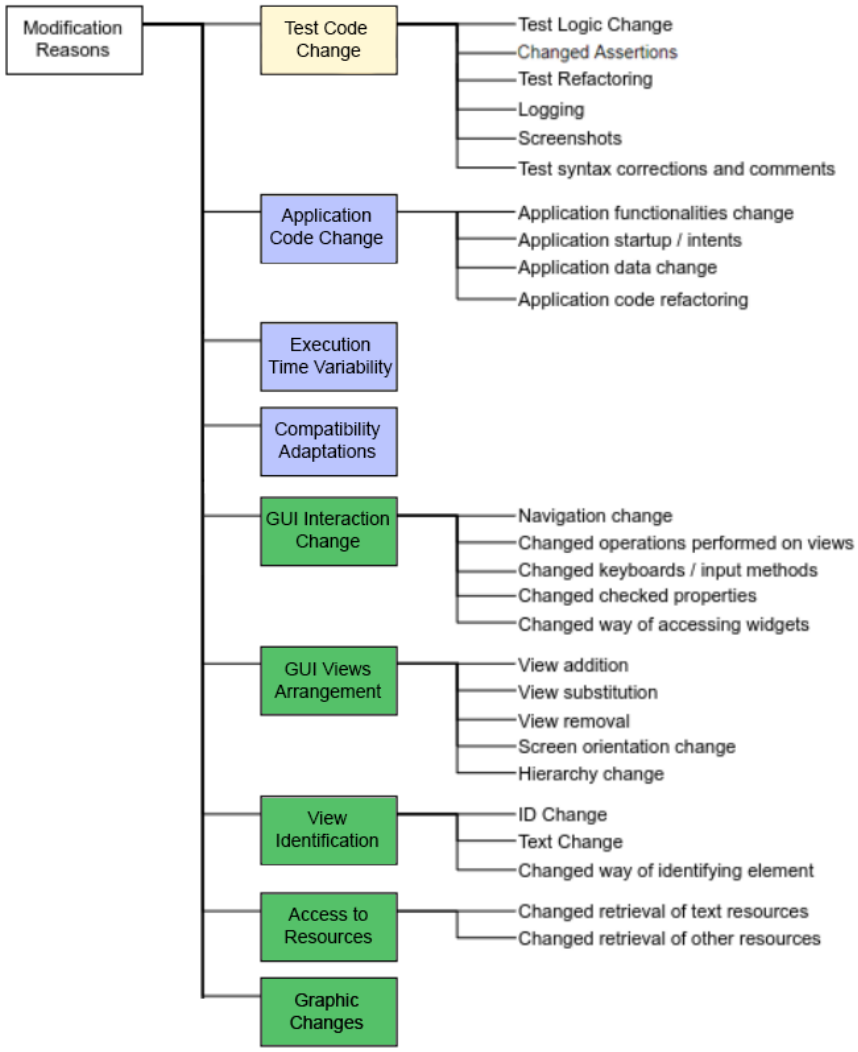

Fig. 1 Graphic taxonomy of modification causes

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 (e.g., for coping with API changes, breakages or deprecations).

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).

5.1.2 Application Code Change

This macrocategory contains all modifications that are due to changes in the production code, that are not related to the graphic appearance of the app. Under this category are considered all the functionality and behaviour changes of the app, e.g. the addition or modification of methods defined inside the activities, or changes in the data model used by the app. Changes in the application code are expected to have effects on all levels of testing performed, from unit tests – expected to verify the compliance to requirements of small, individual components – to system level tests exercising the application through its user interface. According to the classification of test code maintenance given by Yusifoglu et al. [51], this macrocategory covers the modifications operated on test code to perform Adaptive maintenance, when the adaptations are to the functionalities of the AUT only and not to its GUI appearance and/or definition.

Application Functionalities Change. Changes in the functionalities provided by the application, in classes and methods that do not pertain 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).

5.1.3 Execution Time Variability

In some cases an application may require a few seconds to perform a given operation; for instance, while performing an authentication that has to establish a secure connection with a server. Network and system resources may significantly affect the latency. However, also changes in graphic animations can sensibly alter the amount of time to wait for a view to be fully shown to the user.

Moreover, if the tests are performed on real handheld devices, other applications running concurrently may cause additional delays to the loading time.

Typically, the way the execution time variability is reflected by test methods is the need for adding, removing or changing sleep instructions between operations and checks on the views, like in the following diff file excerpt:

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

5.1.4 Compatibility Adaptations

Behavioural adaptations of test classes to guarantee compatibility with different versions of the Android operating system. Compatibility issues may also pertain the visual appearance or the navigation inside existing activities, and often translate to the adoption of new classes for the same widget, due to deprecations of the 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);
+ }
```

5.1.5 GUI Interaction Change

Test code may need modifications because of changes in the operations that can be performed over existing views and widgets that compose the user interface. For instance, changes in the order of operations to perform on widgets, in the available operations on existing widgets, and in the operations that must be performed to access some views, are classified under this category.

Navigation Change. It is possible that the order of the interactions with the widgets is changed between consecutive releases, even though the views featured by the tested activity are not modified. Test code must reflect the changed navigation inside activities.
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. Even though the navigation between different widgets in the user interface does not change, it is possible that the way to execute the same operations requires different gestures to be performed on the same widgets (e.g., long clicks instead of normal clicks). In these cases, the operations must be replicated by test cases accordingly.
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. Properties – not only graphic – that are checked on the tested widgets may need to change between subsequent versions of the application.
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 screen:

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

Changed Way of Accessing Widgets. According to the definition of the application, the same widgets can be accessed in different ways, e.g. from the context menu instead of the normal placement inside layouts inflated by the activity. In some cases, to comply with modifications of libraries or the deprecation of API functions, it is possible that the way to perform the same actions on graphic elements of the interface changes between a version of the application and another.
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());
```

5.1.6 GUI Views Arrangement

All the modifications in the number and type of elements in the visual hierarchy belong to this macro-category of modification causes.

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 modifications 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 which 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.

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 are linkable 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)),
    hasText("    ")));
```

5.1.7 View Identification

The chosen way to identify an individual view in test code, or the actual identifier used, may change between subsequent releases, thus invalidating otherwise working test methods.

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 semantical 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 elements 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. Also screen name changes are 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());
```

5.1.8 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 different 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. 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 consequence on an Espresso method due to the 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));
```

5.1.9 Graphic Changes

Even though the widgets are not entirely modified, small modifications in their appearence (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 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 RQ3: Examining a set of 1724 diff files related to Espresso, UI Automator, Robotium and Robolectric, we identified 28 different possible causes for modifications of test methods developed for Android apps with the use of GUI Automation frameworks. We found nine different macrocategories of change reasons: 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.

5.2 Diffusion of Modification Causes and Fragility Occurrences (RQ4)

Table 5 shows the percentage of occurrence for any category of modification causes, among the four considered sets of diff files, each pertaining a different GUI Automation Framework. Non-normalized numbers of occurrences of modification causes are given in Appendix A.

Since the causes of modifications have not been considered as mutually exclusive (i.e., multiple causes, either related to GUI changes or not, can concur to the modification of the same test method) the columns of frequencies do not mandatorily sum up to 100%.

The *Test Logic Change* category showed a high percentage of occurrence for all the four sets considered. This highlights that there is a relevant amount of situations in which modifications only pertain to test code, without any evident connection to the production code. In the case of Espresso test methods, this category of modifications is often represented by changes in the way the library functions are accessed, or in the kind of assertions that are checked inside test methods. Another common example, in the set of diff files containing code written with Robolectric, is the change of the way the interface is mocked by the GUI Automation Framework (i.e., using the Mockito mocking framework or not).

Table 6 Frequencies of occurrence of modification causes

		Espresso	UI Automator	Robotium	Robolectric
Test Code Change	Test logic change	15.85%	17.95%	20.52%	29.86%
	Assertions Change	2.56%	6.78%	4.48%	5.42%
	Test refactoring	5.24%	5.08%	17.92%	18.01%
	Logging	2.44%	1.69%	0.48%	1.18%
	Screenshots	2.44%	0.00%	0.94%	0.00%
	Test syntax corrections and comments	5.98%	15.25%	12.26%	12.00%
Application Code Change	Application functionalities change	17.80%	0.00%	9.90%	13.44%
	Application startup / intents	6.83%	3.38%	3.53%	4.48%
	Application data change	0.49%	1.69%	0.23%	3.30%
	Application code refactoring	2.93%	5.08%	11.56%	8.25%
Execution Time Variability	Sleeps add	3.54%	13.56%	7.78%	0.00%
	Sleeps change	3.41%	1.69%	2.83%	0.00%
	Sleeps removal	2.68%	3.38%	2.12%	0.23%
Compatibility Adaptations		0.98%	5.08%	0.0%	2.12%
GUI Interaction Change	Navigation change	9.27%	20.34%	10.14%	2.59%
	Changed operations performed on views	1.71%	0.00%	0.94%	0.47%
	Changed keyboards / input methods	1.46%	1.69%	0.23%	0.00%
	Changed checked properties	1.83%	0.00%	0.71%	0.47%
	Changed way of accessing widgets	7.68%	0.00%	3.77%	0.00%
GUI Views Arrangement	View Addition	1.71%	3.38%	0.94%	0.23%
	View substitution	0.73%	3.38%	0.94%	1.18%
	View removal	0.37%	1.69%	0.0%	0.00%
	Screen orientation change	0.37%	0.0%	0.23%	0.00%
	Hierarchy change	0.24%	1.69%	1.65%	0.00%
View Identification	ID Change	7.56%	0.0%	0.48%	2.83%
	Text Change	5.00%	15.25%	4.24%	0.94%
	Changed way of identifying elements	3.78%	6.78%	3.54%	0.00%
Access to Resources	Changed retrieval of text resources	4.27%	1.69%	2.83%	1.41%
	Changed retrieval of other resources	1.22%	0.0%	1.41%	1.18%
Graphic Changes		1.71%	1.69%	0.48%	2.59%

Modification reasons related to the design of the test cases, that we filed under the *Assertions Change* category, happened more rarely than the changes in the functions called in test classes. This suggests that the use cases on which the tests are based were rather stable during the evolution of the considered Android apps. *Test Refactoring* operations, that we interpret as a form of perfective maintenance of working test code, were quite common for all the considered GUI Automation Frameworks. A higher occurrence of refactoring operations occurred in the set of diff files associated to Robotium and Robolectric. This may be a consequence of more rigorous and complex test cases, hence needing more frequent fixes without corresponding modifications in tested functionalities.

A relevant amount of modifications in test cases (more than 11% in the case of Robotium test classes) were due only to added or modified documentation (i.e., commenting) or to syntax fixing of existing test code.

The modifications in test methods that are due to *Application Code Change* had a minor frequency of occurrence than the ones related to *Test Logic Change*, for all the four sets considered. *Application Refactoring* and *Application Data Change* proved to be relevant in terms of modifications triggered in test code, and another modification cause that proved to be frequent (especially for Espresso test classes, which have a tigther coupling with the activities of the tested applications) is the change in the way activities and components are called and instantiated, and in the way the information to the newly created screens are bundled and passed between Activities (*Application Startup/Intents*). A possible overlap between *Application Logic Change* and *Test Logic Change* must also be kept in account. We performed our manual inspection of diff files without executing the tests, hence it is possible

that a modification in a test method may be due to modifications in the application logic that span multiple releases of the app, with tests adapted only once. In addition to that, our inspection only considered the last cause that created the need for maintenance of test methods: hence, modifications that are classified as pure test logic changes, could be sometimes backtracked to previous modifications in the application logic.

Modifications linked to *Execution Time Variability*, that are reflected by the addition, removal or changes in the parameters passed to Sleep functions, occurred rather rarely in the set of modified test methods examined. It must be considered that, for what concerns the animations of the GUI, Espresso and Robotium wait for the views to appear in their final state, before performing any kind of interactions or check on them: hence, the sleep instructions added or removed in test methods written with them are most of the times related to long-running tasks (e.g., logging to a service and awaiting for the response) and not to the waiting times needed for the GUI widgets to appear. The same reasoning does not apply to UI Automator: for test methods written with this GUI Automation Framework, explicit sleep instructions have to be inserted in test code, because no automated recognition of the complete rendering of the widgets is provided. This characteristic of UI Automator is reflected by the higher occurrence of modifications caused by Execution Time Variability for the relative set of diff files. Robolectric test methods, instead, are run on the Java Virtual Machine without the need to wait for an actual rendering of the GUI elements: this peculiarity is reflected by the lacking occurrences of modifications linked to Execution Time Variability in the relative set of diff files.

Another rare category among causes of modifications to test code is the one linked to *Compatibility Adaptations*: this is mainly justifiable with the fact that, in general, Android apps guarantee retrocompatibility to previous releases of the OS, and the Android Development Framework does not need many mandatory modifications in code (either belonging to the AUT or to the test suites) for the app to remain working. New features offered by the new OS releases, furthermore, can be considered as new functionalities offered by the application, or added/changed widgets in the GUI, and not as mere modifications for guaranteeing compatibility.

GUI Interaction changes proved to be the most frequent modification cause among those that are GUI-related. The most relevant among them was the change in the navigation inside the current tested activity, i.e., the order of the operations performed on the existing widgets. The overall percentages of occurrence of causes collected under the *GUI Interaction Change* macrocategory were high for all the sets of tools, except for the set of diff files pertaining Robolectric test methods, that in general exposed very small percentages of occurrence for all GUI-related modification causes. Often, especially for the set of diff files regarding projects featuring Espresso, tests had to be modified because of changes in the way the individual graphic elements were accessed. For instance, a high number of methods had to be modified due to the different ways Action Bars and Drawer Menus had to be interacted with, in order to access the elements that they contained.

Modifications in *GUI Views Arrangement* were significantly less common than the ones pertaining changed operations performed on unmodified views and widgets. The lower occurrence of such causes can be justified with the fact that we focused on the examination of modified existing test methods. Instead, the addition or removal of individual widget may be more frequently linked to added or

deleted test methods, reflecting changes in the set of possible use cases traversing the GUI of the app.

Among the changes in the way views are identified, *ID Change* is the most relevant modification cause for the set of diff files of test classes written with Espresso: this is due to the fact that most of the times Espresso test methods identify widgets directly based on IDs, as opposed, for instance, to what is shown by UI Automator test methods. In general, the problem of ID changes is less relevant for Android (or mobile) applications than it is for web applications, where the IDs are often derived programmatically and/or randomly, and possibly changed in every release. In addition to that, the development of test suites in the same IDE of the mobile app (as it is typically done with Espresso, UI Automator, and Robotium) allows to leverage the automated refactoring tools provided by the IDE for the adaptations of IDs used in test methods. The same cannot apply for modifications of plain text used for populating (and later identifying, in the test methods) the elements of the user interfaces. Even though the use of string resources for any textual content is a recognised best practice for Android development, the cases in which plain text is modified and requires parallel intervention in test code proved to be still frequent (up to 15.25% of frequency in diff files of UI Automator test classes).

The changes in *Access to Resources* (e.g., modifying the application in order to use shared preferences or String resources, instead of plain text, for the content of textual widgets), instead, were not so common as causes of modifications in test methods: the relative frequency of occurrence ranged from 1.41% for Robolectric to 4.27% for Espresso for what concerns the retrieval of text resources, and from none for UI Automator to 1.41% for Robotium for what concerns the retrieval of other resources. Also in this case, a good practice for developers to avoid this type of fragility to changes in the AUT would be the adoption from early releases of the application of stable identifiers of the text or media elements in the "res" Xml files designated in the Android project hierarchy.

Pure *Graphic Changes* were a significantly rare macrocategory of modification causes. Individual graphic modifications that led to modified methods in test classes varied from changes in the dimensions and exact coordinates of the drawables used for the GUI composition, to details like the padding, color, transparency and animations applied on individual widgets. The frequency of occurrence of modifications due to Graphic Changes in the GUI of the AUT ranged from 0.48% (for the set of diff files of Robotium test classes) to 2.59% (for the set of diff files of Robolectric test classes). This very low occurrence was expected, and is linked to the fact that all the considered GUI Testing Frameworks are – according to the classification of testing tools given by Alegroth et al. [1] – second-generation or component-based GUI testing tool, that strongly focus on properties (e.g., ids, text, and positions in the layout hierarchies) to identify elements of the GUI, and that best work when having full access to the production code. Obviously, tools that are more focused on graphic characteristics of the interface or that perform any kind of visual recognition of elements (namely, image recognition based testing tools) would have suffered more from pure graphic changes.

Table 7 gives a higher-level result for the sets of diff files considered, showing the percentages of diff files that featured modifications that were related or not to changes in the AUT, and among the latter ones the fraction that was related to changes influencing the graphical appearance of the app. The last column in the table reports the average value for the three fractions, obtained as an average of the

Table 7 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%

values for the individual GUI Automation Frameworks weighted by the number of diff files examined for each of them. Also in this table, the values reported in each column do not necessarily sum to 100%, because the causes of modifications are not mutually exclusive in modified test cases (i.e., modifications that are AUT-related and others that are not AUT-related may be spotted in the examination of the same diff file).

We have considered only the modifications filed under the category *Test Code Change* as not related to changes in the AUT. Non-AUT related changes were rather frequent for all the sets of projects considered, with an average 46.36% fraction measured, the 32.80% frequency for Espresso being the lowest measured value, and 65.80% for Robolectric the highest. Those modifications, according to the definition we provided, are not related to fragility, since there is no evident connection between them and prior changes in the AUT; hence, they are related to the test code only and do not respond to maintenance needs triggered by the evolution of the underlying app. The lower value for Espresso and UI Automator test cases may suggest that those GUI Automation Framework are typically used for higher level test cases strictly tied to activities or widgets of the app, that are less prone to changes in their logic during the evolution of the application.

At the same time, in fact, around 70% of diff files of classes linked to Espresso and UI Automator contained modifications linked to changes in the AUT of the application – and hence, by our definition, pertaining to fragile test code – with around 70% of them due to changes related to the GUI of the app or its arrangement and definition. Those results suggest that the Espresso GUI Automation Framework is the most strictly tied to the AUT appearance of those considered for this evaluation, closely followed by UI Automator.

The values measured for Espresso and UI Automator are very far from those measured for Robolectric and Robotium, for which we measured, respectively, a frequency of modifications related to changes in the AUT of near 40% and 56.37% (just 31.35% and 45.44% of them concerning the GUI). This rather low frequency of occurrence in test classes featuring such tool was indeed expected, being Robolectric used often for traditional unit testing and not for system tests traversing the GUI. Respectively 65.80% and 53.53% of modifications in tests associated to Robolectric and Robotium were not related to the AUT, and hence – according to our definition – were not induced by test fragility.

It must also be considered that, according to the manual validation described in section 4.3, only 70% of a sample of mined test classes were actually performing operations on the GUI of the app. This result suggests that many of the diff files considered were of test classes that were not interacting with the GUI in the first place, and that hence would never need maintenance for modifications to the GUI of the app or to its definition and description.

In general, however, high values for non GUI-related causes were expected, even for testing frameworks specialized for GUI testing: GUI tests are by nature affected not only by changes in the user interface itself, but by changes performed on all levels of abstraction of the AUT.

Answer to RQ4: Examining diff files of test classes containing changes in test methods, we measured a percentage of about 60% of modifications due to changes in the AUT, and hence of classes that are categorized as fragile according to our definition. 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 production code.

6 Threats to Validity

Threats to internal validity. We have identified the following threats to the findings exposed in the present paper:

- The test class identification process is 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 amount of classes extracted using the heuristic) of about 90%, while slightly less than 70% of those test classes contained code that exercised the GUI of the AUT. This suboptimal heuristic may lead to misleading values for what regards the ratio of modifications due to changes in the AUT and specifically in the GUI of the AUT (table 7), with lower values that those that would have been obtained by taking into consideration exclusively test classes traversing the GUI. Better procedures for 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* function, or similar ones, for the Espresso framework). Also, the code of identified test classes can be parsed in order to find calls to methods that are proper of the Activity or Fragment graphical lifecycle (e.g., setContentView()) or that contain references to layout or graphical resources, to exclude test classes that do not interact in any way with the GUI of the AUT. 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 versions 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 our investigations; it is not assured that this check is the most dependable one for pruning negligible projects.

- The metrics we defined have not been tested outside the scope of this study, hence we cannot ensure the correctness of the assumptions we based on them.
- Our 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 we performed. 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 of the considered testing frameworks can be used to perform not only GUI testing. A manual validation of our 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 in the discussion we provide. The study was also conducted in a static way, 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.
- Currently, the taxonomy of modification causes is based on the examination of diff files of test classes only, and hence it takes into account the most superficial modifications that are ultimately performed to test methods. This can cause some overlaps between different categories of the taxonomy, especially if the objective is to identify the real root cause of each modification: e.g., changes in the test logic may be backtracked to prior modifications in the AUT.
- Researcher bias is a possible threat to the validity of this study. The different steps of the adopted procedure that required manual inspections of testing code (i.e., the derivation of the qualitative findings, the validation of the heuristic for the identification of test cases, and the open coding phase of the grounded theory technique) were performed by an author of the paper only. This may add bias to the reported results. However, no author of the manuscript is involved in the development or support of any of the investigated tools, hence there is no inclination to support any particular tool nor to demonstrate any specific result.

Threats to conclusion validity. These threats concern the derivation of the correct conclusion based on the results.

- In the result section, analyses were performed based on the NTR (i.e., the Number of Tagged Releases of a given GitHub Repository), and TD (i.e., Tool Diffusion) metrics. Those results may be significantly impacted by the time frame in which the study has been performed (until the end of December 2016), because of the evolution of the considered frameworks and possible

migrations from one framework to another after the last mining of repositories from GitHub. Even though the relative adoption of testing frameworks is confirmed by other studies in literature (e.g., by those performed by Kochhar et al. [25] and Linares-Vasquez et al. [32]), measures collected in more recent times may exhibit different values of penetration of the considered testing tools among open-source repositories.

Threats to external validity. We identify the following threats to the generalizability of our work:

- 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 our findings, based on a very large repository of open-source projects, can be applicable to the development of commercial or closed-source projects.
- We have collected measures 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. The same reasoning applies to the causes of fragility that we defined and the percentages of occurrence we measured for them: other testing frameworks and techniques may exhibit different motivations for maintenance of test code, or the prominence of different causes with respect to the six considered ones.
- The metrics we defined 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, neither the results of the application of similar metrics on them can be compared to the results we provide in the paper.

7 Related Work

In this section we discuss related work investigating the topic of mobile testing and its difficulties. We compare the outcomes of our study with them, and highlight the differences in the approach and methodology that we followed.

The testing tools we searched for in Android projects hosted on GitHub reflect the selection we made for another previous study [7], in which we developed manually a test suite for an open-source Android application, and explored the need for modifications in such test suite transitioning between selected releases of the app. With respect to this former study, the current work extends the sample to the whole population of Android applications hosted on GitHub that are provided with a release history and with manifest files, and evaluates the needed modifications in all tagged releases of a given project.

The present manuscript is conceived as a complement and an extension of our previous work presented to PROMISE '17 [8]. The work introduced our set of metrics, that we have now complemented with a more in-depth discussion, with some modifications to the way they are measured and with running examples of their computation. The work has also been complemented with a taxonomy of which we have presented a preliminary version in a previous paper. With respect to the first presentation of the taxonomy [9], in this work we detailed in a more rigorous way the way the taxonomy was built – using the grounded theory principles –

and extended our manual examinations to all the diff files of the studied projects, while our previous efforts only considered diff files of Espresso test classes.

Linares-Vasquez et al. performed several studies that are related to our current work. In an empirical studies aimed at identifying how Android developers test their applications [31], they surveyed developers and testers to understand what automated testing practices they follow, and the perception they have of quality metrics that can be used to evaluate their test suites. The results provided in the study confirm the ones that we gathered from the raw inspection of the presence of source code attributable to automated GUI testing tools: the authors find that the participants mostly rely on manual testing for their applications, giving several motivations for such lack, from fastly changing requirements to limited time for performing testing. The authors also found that mobile developers heavily rely on usage models for designing and documenting test cases, that automatically generated test cases expressed in natural language are preferred by the surveyed developers, and that the most adopted tools by developers for mobile application testing are JUnit, Robolectric and Robotium. The authors found the respondents for their survey among the contributors of 102 different Android open-source projects hosted on GitHub, hence their findings can be considered applicable to the developers of the applications of our context.

Linares-Vasquez et al. also conducted a survey about automated mobile app testing [32], highlighting again that the state of the art of automating testing tools has some limitations, inducing developers to grow a preference towards manual testing in practice. In this work, the scripted testing tools that we considered in our paper are considered among a set of *GUI Automation Frameworks*, along with UI Automation (for iOS), Ranorex, Calabash, Quantum and Qmetry. The authors found that GUI level tests utilizing those frameworks are very expensive to maintain as the application evolves (the concept that has been described as *fragility* throughout this manuscript), and thus are rarely adopted by developers.

Kochhar et al. performed a study to understand the Test Automation Culture of Android Developers [25]. In addition to a survey for Android developers, they also performed a quantitative analysis on a set of 600 open-source Android applications, mined from the F-Droid repository. The study classifies the application in terms of number of lines of code, contributors, test suites and test coverage, and concludes – from the results of the survey – that most of the developers resort to manual testing or completely neglect testing. In particular, the authors found that 14% of the applications they examined contained test cases, and that only 9% of them had executable test cases with a coverage over 40%. The mostly cited testing tools by the surveyed developers were JUnit, Monkeyrunner, Robotium and Robolectric. Our study also finds a lack adoption of GUI testing frameworks (or, at least, the six ones that we selected) among Android applications, extending the context to a larger (the largest up to now) set of studied open-source projects.

8 Conclusion and Future Work

We analyzed the use of some of the most widely adopted GUI Automation Frameworks for GUI testing of Android applications – namely, Espresso, UI Automator, Selendroid, Robotium, Robolectric, and Appium – among Android open-source projects hosted on the GitHub portal.

Overall, automated GUI testing frameworks do not appear to be widely adopted by Android projects hosted on GitHub: only 9% of the examined apps had files that were identified as test classes associated to any of the six GUI Automation Frameworks that we considered. On average, when present, the GUI testing code represent about 11% of the whole project code.

Test code proved to be rather important during the lifespan of the considered apps, weighing for about 9% of the total project code. Concerning its evolution in each release, on average, about 5% of the changed lines are in the GUI test code and about 4.5% of test code is modified. Also classes that were associated to the considered GUI Automation Frameworks were subject to frequent modifications, with almost one fifth of test classes being modified in each release transition. In general, this amount of modification in test code was expected, especially for code developed with GUI Automation Frameworks, which – since they work at system level – are influenced by changes on any level of the AUT that is tested.

Through Grounded Theory and Open Coding, based on a set of diff files containing fragilities in test methods, we identified a set of 28 causes of modifications, subdivided in 9 different macrocategories. We also identified macrocategories that were related to the test only (*Test Code Change*), to the AUT in general (*Application Code Change*, *Execution Time Variability*, *Compatibility Adaptations*) or more directly to its GUI (*GUI Interaction Change*, *GUI Views Arrangement*, *View Identification*, *Access to Resources*, *Graphic Changes*).

We also evaluated the frequencies of those modification causes, finding that when GUI Automation Frameworks are used for testing Android applications, up to the 50% of the modifications to test methods can be due to changes in the GUI arrangement or definition only. At the same time, many of the modifications in test methods developed with the selected GUI Automation Framework proved to be due to modifications performed to the non-graphical functionalities of the AUT, with a frequency of occurrence of non GUI-related modification causes that reached the 66% in one of the sets of diff files that we considered. Testware developed with the studied GUI Automation Frameworks proved, hence, to be rather fragile to changes in the AUT. During the normal evolution of an Android open-source app, the amount of changes needed to cope with the changes in the AUT code is comparable to the changes performed to test code with no specific connection to the AUT features or appearance.

8.1 Contributions for practitioners and researchers

The taxonomy of test case modification reasons can be used, as it is, as a valuable source of information for predicting a fragility of a test case, and in general for the effort needed by its maintainance. In particolar, the frequency of occurrence of modification causes reported in the current manuscript can be paired with static analysis of test code already developed, to identify what are the test commands and methods that involve characteristics that may be prone to fragility. Developers may also leverage the statistics about the average maintenance effort required by the inspected test suites to decide whether they can afford the average effort for maintaining a test suite. The averages reported in this paper can finally be used as a benchmark to understand whether the amount of maintenance needed by

already developed test suites is in line with typical values for a considerably big amount of open-source tested projects.

The results of this paper also provide hints for additional studies and more empirical understanding of test suite maintenance. A first possible addition may take into consideration evolutionary analytics of the considered repositories and the measured metrics, to investigate how the usage and maintenance of testing code produced with Automated GUI testing frameworks varies with time. As done in recent studies [12], applications can be automatically categorized – with the use of static analysis – under different categories, to investigate how the testing maintenance varies with the type of AUT. The type of investigated open-source app can be considered as a factor in future investigation, to correlate the nature of the AUT with the measured metrics and with the evolution of testware.

The evolution metrics defined in this paper may be used by researchers to find correlations with other measurements on test code, e.g. coverage metrics of test suites or measures of their effectiveness in finding bugs or defects. Diffusion and Size metrics hereby defined are fit for being correlated with measures about the perceived quality of open-source applications, e.g., ratings on the stores where the apps are eventually released, sentiment analysis of the users' comments (a practice already explored in literature [20]), or repository activity and/or popularity on GitHub.

Fragility and Maintenance metrics can be object of additional comparisons with the required maintenance by other techniques of testing tools for Android apps (e.g., model based test suites, or visual test suites). A more quantitative measurement of the maintenance effort (e.g., in terms of person hours) evaluated for different testing techniques would provide significant added value for practitioners in finding the right testing technique for the type of applications that they are developing.

8.2 Future work

As our immediate future work, we plan to provide a set of guidelines to help developers in reducing the impact that modifications in production code have on testing maintenance, and to categorize also modifications to production code, to understand how they can be traced to maintenance in test code and to find to what changes test code proves to be fragile the most. Regarding the fragility issue, we plan to monitor also the distribution of fragility occurrences for a project during time, and to take into consideration also the concept of fragility *frequence* for the same test class or method in the evolution of the AUT.

We plan to develop automated tools, capable of recognizing patterns that are related to fragility when modifications are performed in production code (e.g., changes of GUI arrangement, renaming of widgets, changes in text presented to the user), and signal potential fragile classes/methods in the test suite at development time; such tools can be also implemented as plug-ins for popular IDEs (e.g., Android Studio). We also aim at applying the evolution and fragility measurements to different testing tools, frameworks, and platforms, and to take into consideration the testing of Android applications developed with hybrid frameworks (e.g.,

Flutter or Cordova). Finally, we plan to generalize the taxonomy of test maintenance causes to any kind of GUI-based software, and not only for Android (or mobile, in general) apps.

References

1. Alégroth E, Feldt R, Ryrholm L (2015) Visual gui testing in practice: challenges, problemsand limitations. Empirical Software Engineering 20(3):694–744
2. Amalfitano D, Fasolino AR, Tramontana P, De Carmine S, Imparato G (2012) A toolset for gui testing of android applications. In: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, IEEE, pp 650–653
3. Amalfitano D, Fasolino AR, Tramontana P, Ta BD, Memon AM (2015) Mobiguitar: Automated model-based testing of mobile apps. IEEE software 32(5):53–59
4. Charmaz K (2014) Constructing grounded theory. Sage
5. Choi W, Necula G, Sen K (2013) Guided gui testing of android apps with minimal restart and approximate learning. In: Acm Sigplan Notices, ACM, vol 48, pp 623–640
6. Choudhary SR, Gorla A, Orso A (2015) Automated test input generation for android: Are we there yet?(e). In: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, IEEE, pp 429–440
7. Coppola R, Raffero E, 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, ACM, pp 11–20
8. Coppola R, Morisio M, Torchiano M (2017) 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, ACM, pp 22–32
9. Coppola R, Morisio M, Torchiano M (2018) Maintenance of android widget-based gui testing: A taxonomy of test case modification causes. In: Proceedings of the 1st IEEE Workshop on NEXt level of Test Automation 2018, IEEE
10. Coppola R, Morisio M, Torchiano M (2018) Mobile gui testing fragility: A study on open-source android applications. IEEE Transactions on Reliability (Early Access) pp 1–24
11. Corbin JM, Strauss A (1990) Grounded theory research: Procedures, canons, and evaluative criteria. Qualitative sociology 13(1):3–21
12. Cruz L, Abreu R, Lo D (2018) To the attention of mobile software developers: Guess what, test your app! ArXiv
13. Gao J, Bai X, Tsai WT, Uehara T (2014) Mobile application testing: a tutorial. Computer 47(2):46–55
14. Gao Z, Chen Z, Zou Y, Memon AM (2016) Sitar: Gui test script repair. Ieee transactions on software engineering (2):170–186
15. Garousi V, Felderer M (2016) Developing, verifying, and maintaining high-quality automated test scripts. IEEE Software 33(3):68–75
16. Glaser BG, Strauss AL, Strutzel E (1968) The discovery of grounded theory; strategies for qualitative research. Nursing research 17(4):364

17. Gomez L, Neamtiu I, Azim T, Millstein T (2013) Reran: Timing-and touch-sensitive record and replay for android. In: Software Engineering (ICSE), 2013 35th International Conference on, IEEE, pp 72–81
18. Grechanik M, Xie Q, Fu C (2009) Maintaining and evolving gui-directed test scripts. In: Proceedings of the 31st international conference on software engineering, IEEE Computer Society, pp 408–418
19. Grgurina R, Brestovac G, Grbac TG (2011) Development environment for android application development: An experience report. In: MIPRO, 2011 Proceedings of the 34th International Convention, IEEE, pp 1693–1698
20. Islam MR (2014) Numeric rating of apps on google play store by sentiment analysis on user reviews. In: 2014 International Conference on Electrical Engineering and Information Communication Technology, pp 1–4, DOI 10.1109/ICEEICT.2014.6919058
21. Jensen CS, Prasad MR, Møller A (2013) Automated testing with targeted event sequence generation. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ACM, pp 67–77
22. Kaasila J, Ferreira D, Kostakos V, Ojala T (2012) Testdroid: automated remote ui testing on android. In: Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia, ACM, p 28
23. Kaur A (2015) Review of mobile applications testing with automated techniques. International Journal of Advanced Research in Computer and Communication Engineering 4(10):503–507
24. Knych TW, Baliga A (2014) Android application development and testability. In: Proceedings of the 1st International Conference on Mobile Software Engineering and Systems, ACM, pp 37–40
25. Kochhar PS, Thung F, Nagappan N, Zimmermann T, Lo D (2015) Understanding the test automation culture of app developers. In: Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on, IEEE, pp 1–10
26. Kropp M, Morales P (2010) Automated gui testing on the android platform. Testing Software and Systems p 67
27. Leotta M, Clerissi D, Ricca F, Tonella P (2013) Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In: Reverse Engineering (WCRE), 2013 20th Working Conference on, IEEE, pp 272–281
28. Leotta M, Clerissi D, Ricca F, Tonella P (2014) Visual vs. dom-based web locators: An empirical study. In: International Conference on Web Engineering, Springer, pp 322–340
29. Linares-Vásquez M (2015) Enabling testing of android apps. In: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, IEEE, vol 2, pp 763–765
30. Linares-Vasquez M, Vendome C, Luo Q, Poshyvanyk D (2015) How developers detect and fix performance bottlenecks in android apps. In: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, IEEE, pp 352–361
31. Linares-Vásquez M, Bernal-Cárdenas C, Moran K, Poshyvanyk D (2017) How do developers test android applications? In: Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on, IEEE, pp 613–622

32. Linares-Vásquez M, Moran K, Poshyvanyk D (2017) Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In: Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on, IEEE, pp 399–410
33. Liu CH, Lu CY, Cheng SJ, Chang KY, Hsiao YC, Chu WM (2014) Capture-replay testing for android applications. In: Computer, Consumer and Control (IS3C), 2014 International Symposium on, IEEE, pp 1129–1132
34. Machiry A, Tahiliani R, Naik M (2013) Dynodroid: An input generation system for android apps. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, pp 224–234
35. Memon AM (2008) Automatically repairing event sequence-based gui test suites for regression testing. ACM Transactions on Software Engineering and Methodology (TOSEM) 18(2):4
36. Milano DT (2011) Android application testing guide. Packt Publishing Ltd
37. Mirzaei N, Malek S, Păsăreanu CS, Esfahani N, Mahmood R (2012) Testing android apps through symbolic execution. ACM SIGSOFT Software Engineering Notes 37(6):1–5
38. Moran K, Linares-Vásquez M, Bernal-Cárdenas C, Vendome C, Poshyvanyk D (2017) Crashscope: A practical tool for automated testing of android applications. In: Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on, IEEE, pp 15–18
39. Muccini H, Di Francesco A, 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, IEEE Press, pp 29–35
40. Pinto LS, Sinha S, Orso A (2012) Understanding myths and realities of test-suite evolution. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, p 33
41. Ralph P (2018) Toward methodological guidelines for process theories and taxonomies in software engineering. IEEE Transactions on Software Engineering
42. Scott TJ, Kuksenok K, Perry D, Brooks M, Anicello O, Aragon C (2012) Adapting grounded theory to construct a taxonomy of affect in collaborative online chat. In: Proceedings of the 30th ACM international conference on Design of communication, ACM, pp 197–204
43. Sedano T, Ralph P, Péraire C (2017) Software development waste. In: Proceedings of the 39th International Conference on Software Engineering, IEEE Press, pp 130–140
44. Shah G, Shah P, Muchhala R (2014) Software testing automation using appium. International Journal of Current Engi neering and Technology 4(5):3528–3531
45. Singh S, Gadgil R, Chudgor A (2014) Automated testing of mobile applications using scripting technique: A study on appium. International Journal of Current Engineering and Technology (IJCET) 4(5):3627–3630
46. Stol KJ, Ralph P, 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, IEEE, pp 120–131
47. Strauss A (8) Corbin, j.(1998) basics of qualitative research. techniques and procedures for developing grounded theory. Thousand Oaks, CA: Sage

48. TAN M, CHENG P (2016) Research and implementation of automated testing framework based on android. Information Technology 5:035
49. Tang X, Wang S, Mao K (2015) Will this bug-fixing change break regression testing? In: Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on, IEEE, pp 1–10
50. Yang W, Prasad MR, Xie T (2013) A grey-box approach for automated gui-model generation of mobile applications. In: International Conference on Fundamental Approaches to Software Engineering, Springer, pp 250–265
51. Yusifoğlu VG, Amannejad Y, Can AB (2015) Software test-code engineering: A systematic mapping. Information and Software Technology 58:123–147
52. Zadgaonkar H (2013) Robotium Automated Testing for Android. Packt Publishing Ltd
53. Zhauniarovich Y, Philippov A, Gadyatskaya O, Crispo B, Massacci F (2015) Towards black box testing of android apps. In: Availability, Reliability and Security (ARES), 2015 10th International Conference on, IEEE, pp 501–510

A Absolute number of modification causes

Table A shows the absolute number of occurrences of the categories of modification causes among the examined diff files; each column shows the number of occurrences for the set of diff files that are associated to a given GUI Automation Frameworks.

Table 8 Absolute number of occurrences of modification causes

		Espresso	UI Automator	Robotium	Robolectric
Total Classes		948	237	465	550
Test Code Change	Test logic change	130	10	87	126
	Assertions Change	21	4	19	23
	Test refactoring	41	3	76	76
	Logging	20	1	2	5
	Screenshots	20	0	4	0
	Test syntax corrections and comments	49	9	52	53
Application Code Change	Application functionalities change	146	0	42	57
	Application startup / intents	56	2	15	19
	Application data change	4	1	1	14
	Application code refactoring	24	3	49	35
Execution Time Variability	Sleeps add	29	8	33	0
	Sleeps change	28	1	12	0
	Sleeps removal	22	2	9	1
Compatibility Adaptations		8	3	0	9
GUI Interaction Change	Navigation change	76	12	43	11
	Changed operations performed on views	14	0	4	2
	Changed keyboards / input methods	12	1	1	0
	Changed checked properties	15	0	3	2
	Changed way of accessing widgets	63	0	16	0
GUI Views Arrangement	View Addition	14	2	4	1
	View substitution	6	2	4	5
	View removal	3	1	0	0
	Screen orientation change	3	0	1	0
	Hierarchy change	2	1	7	0
View Identification	ID Change	62	0	2	12
	Text Change	41	9	18	4
	Changed way of identifying elements	31	4	15	0
Access to Resources	Changed retrieval of text resources	35	1	12	6
	Changed retrieval of other resources	10	0	6	5
Graphic Changes		14	1	2	11

B Running Sample of Metric Computations

Table 9 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 10 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

To provide samples of metric computations, we resort on reporting all the intermediate and final measures for a small projects of the sample that we considered, namely WheresMyBus/android[14]. 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 9 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.

[14] http://github.com/WheresMyBus/android

```diff
@@ -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. 2 Diff file for test class TestAlertForumActivity.java of WheresMyBus/android, between releases 1.3.0 and 1.4.0.

Table 10, 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 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 to 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 2 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).

C Filtering Procedure

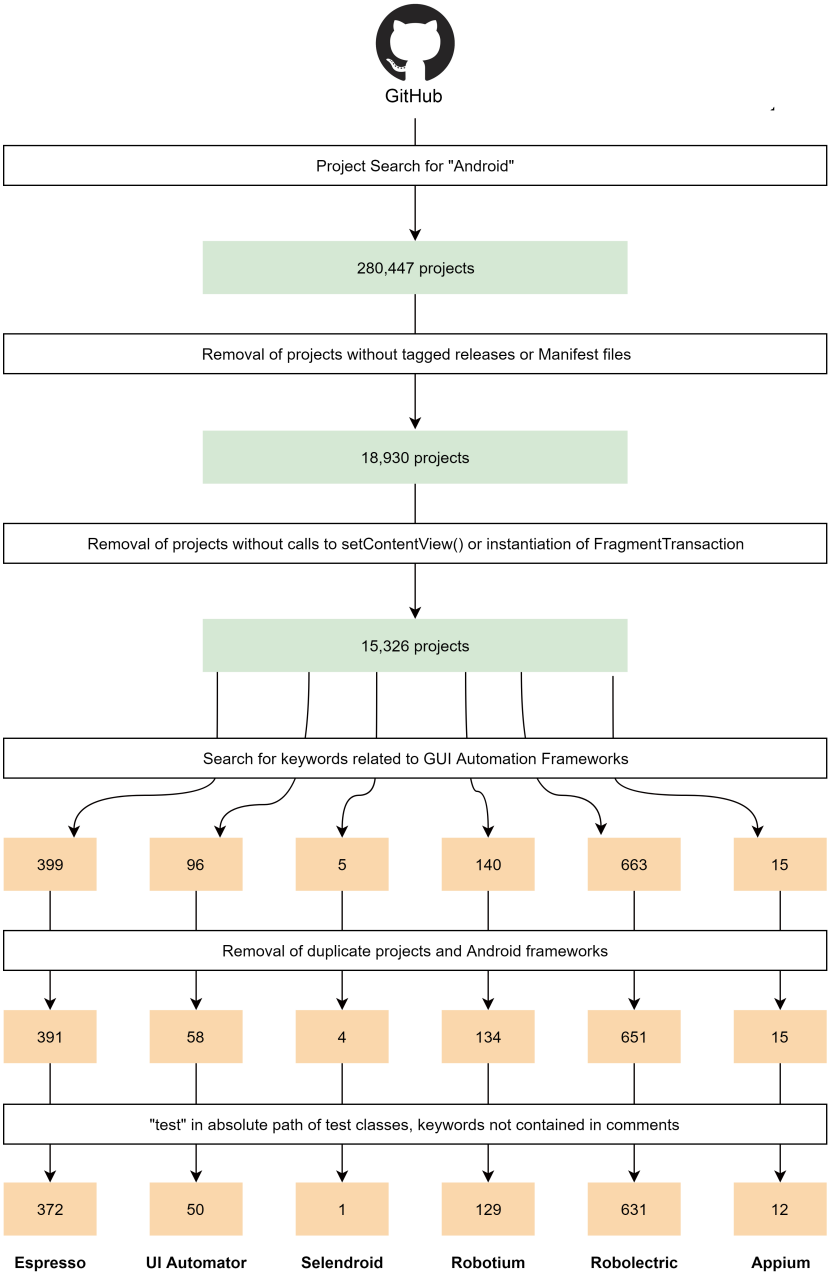

Fig. 3 Number of projects after each step of the filtering procedure