Evolution and Fragilities in Scripted GUI Testing of Android applications

# Evolution and Fragilities in Scripted GUI Testing of Android applications

Riccardo Coppola, Maurizio Morisio, Marco Torchiano
Dipartimento di Informatica e Automatica
Politecnico di Torino
Turin, Italy
first.last@polito.it

No Institute Given

**Abstract.** In literature there is evidence that Android applications are not rigorously tested as their desktop counterparts. However – especially for what concerns the Graphical User Interface of mobile apps – a thorough testing should be advisable for developers.
Some peculiarities of Android applications discourage developers from performing automated testing. Among them, we recognize fragility, i.e. test classes failing because of modifications in the GUI only, without the application functionalities being modified.
The aim of this study is to provide a preliminary characterization of the fragility issue for Android apps, identifying some of its causes and estimating its frequency among Android open-source projects. We defined a set of metrics to quantify the amount of fragility of any testing suite, and measured them automatically for a set of repositories hosted on GitHub. We found that, for projects featuring GUI tests, the incidence of fragility is around 10% for test classes, and around 5% for test methods. This means that a significant effort has to be put by developers in fixing their test suites because of the occurrence of fragilities.

## 1 Introduction

Recent years have witnessed a significant diffusion of mobile devices, which have largely overtaken desktop computers in terms of shipped units [1]. The most widespread operating system available for mobile devices is Android, adopted by around 85% of them in 2016 [2].

Thanks to the growth of the capabilities of handheld devices, nowadays Android applications can perform tasks that a few years ago were exclusive to high-end desktop computers. Several marketplaces are available for developers to release their software, and Android users can choose among a vast quantity of competitive Applications to satisfy their needs.

In such a scenario, testing becomes crucial, to ensure that the user experience is not harmed by undesired behaviours and crashes. Studies in literature have highlighted the need for performing different kinds of testing for Android apps [3]: GUI (i.e., Graphical User Interface), unit, integration, system, regression,

compatibility, performance, security testing. Among those, GUI testing is a very prominent need, since the interfaces offered to the users can be very complex and expose various risks of undesired behaviours.

However, several studies have examined the test culture among mobile developers, finding out that most of them do not practice any kind of GUI testing. A high percentage of developers just rely on the execution of manual tests on the user interface, or do not perform testing at all, leaving the recognition of bugs and malfunctions to the final users. Evidence about this lack of testing is given by Kochar et al. [4], who found that under 15% of the set of applications they considered featured any kind of test classes.

The very short time to market of Android apps is not a sufficient justification for such a lack of automated testing. Intrinsic characteristics of Android applications can discourage developers from testing them: for instance, the necessity of deploying the app on different versions of the OS and different device models (which may come with different screen characteristics and customized interfaces); the pace of evolution of the operating system; the diversity of contexts and unpredictable events to which the application may have to respond to; the possible variability (or lack) of resources available on the device.

We believe that the fragility of GUI scripted tests -a problem already explored for different kinds of software, e.g. for web applications [5, 6]- may be an important deterrent to the large-scale adoption of GUI testing for Android applications: developers may decide to discard completely the testing phase of their products, if even small changes in their GUI can break entire test suites. In short, we define a GUI test as fragile if it has to be modified when changes are made in the GUI, and not in the functionalities of the app.

With this work, we aimed at gathering information about the diffusion and incidence of the fragility problem. We gave our definition of fragility for mobile GUI tests, and defined a set of metrics that may be helpful to evaluate its incidence on the maintainance of an Android app. We evaluated our metrics and based our findings on six different sets of applications, that we extracted from the GitHub repository searching for popular scripted GUI testing tools.

## 2 Background and Related Work

This section provides an introduction to Android programming and testing, and a survey of existing papers about Android testing and its challenges.

### 2.1 Overview of Android Applications

A definition of mobile apps can be given [7] as mobile software (i.e., applications that run on mobile devices like smartphones or tablets) taking input from the context of execution (i.e., that performs actions in response of characteristics or events in their environment).

In the Android environment, apps can be built starting from four base components. *Activities* build the user interface, by defining their elements and the

responses that will be triggered by different classes of user input (e.g., tactile or vocal). *Services* handle background operations that can take longer than user interactions, like the management of network connections. *Broadcast Receivers* listen to events that are launched by the system and allow the app to respond to them (e.g., low battery signal). *Content providers* are in charge of managing the application data. Components are characterized by a specific life cycle, that should be properly tested in order to avoid crashes in the app. GUI testing is strictly tied to the testing of Activities, since the user interaction and the graphic elements of the app are managed by them.

Mobile Apps are typically classified into three categories [8]. *Native* mobile apps are designed to run on a specific platform, with elements and interfaces specifically designed for it. *Web-based* apps are developed in web languages, and are loaded by a browser on mobile devices. *Hybrid apps* use native code to provide an interface to their users, but the core of the application logic is still a web application that is loaded at run-time.

### 2.2 Alternatives for Android GUI testing

In general, automated mobile testing can be defined as "testing activities for native and Web applications on mobile devices, using well defined software test methods and tools to ensure quality in functions, behaviours performance, and quality of service" [9].

As briefly discussed in section 1, there are different levels at which Android testing can be performed. In our study, we focus on scripted GUI testing.

The most immediate option to perform GUI testing is the execution of manual test cases. Manual testing is preferred by a large percentage of developers. While the technique is easy and fast to actuate and is not a prerogative of specialized workforce, it is not exhaustive, error prone and hardly reproducible [10]. Automated GUI testing techniques are thus recommendable, since they can produce sets of scripts that allow to exercise -in a repeatable and exhaustive way- all the functionalities of a given user interface. Some techniques do not need knowledge about the source code of the apps to test (i.e., only the .apk packages are necessary) and can automatically produce test scripts, that can be refined and parametrized afterwards. If the source code is available, it is possible to write scripted white-box test methods, like it can be done for typical JUnit testing of Android applications.

Different approaches are proposed in literature for the GUI testing of Android Applications [11]. *Random* and *Fuzzy* testing techniques [12–14] need no information about the SUT (i.e., Software Under Test) and give random inputs to its activities, evaluating if any of the inputs triggers undesired behaviours or exceptions.

Capture & Replay testing tools [15–17] allow to start recording sessions of operations on the GUI of Android apps, obtaining repeatable test scripts that can be re-executed in a second moment (e.g., for performing regression testing).

Model-based testing techniques [18–24] take advantage of models of the SUT, that can be extracted automatically by a GUI ripping component, or created

manually by the developer. Provided the model, these techniques generate a systematic traversing of its transitions, to explore the application GUI as thoroughly as possible. Event-sequence generation tools [25, 26] adopt a similar approach that considers the test cases as streams of events.

Minor coverage is given in literature [10, 27] for scripted and white-box testing techniques, which require the testers to manually select operations to be performed on the SUT, and write down test classes and methods based on them.

Several characteristics of Android applications, that may make testing them a complex task, have been investigated in literature [3,7,8]: limited energy, memory and bandwith of the devices; constant interruptions and interactions with other applications running and with the system; wide sets of different devices to which the interface must be adapted; very short time to market; very frequent changes in user interfaces; In addition to those causes related to the nature of the Android OS and mobile devices, some studies [4] highlight that developers may also neglect testing because of complexity and missing documentation of the available testing tools.

### 2.3 GUI testing Fragility approximations

For our purposes, which is an evaluation of GUI testing we adopt the following definition for fragile GUI tests.

> We consider a GUI test class as fragile when two conditions hold:
> - it needs modifications when the application evolves;
> - the need is not due to a change in any functionality of the application, but to changes in the user interface arrangement and/or definition.

We believe that fragility can be one of the main factors discouraging developers from adopting GUI testing, because trivial changes in the GUI may break entire test suites. Our preliminary study on a test suite developed for an open-source Android application, K-9 mail [28], found that up to the 75% of scripted tests had to be modified because of modifications in the GUI, during their lifespan. Among the causes of such modifications, we found: identifier and text changes inside the visual hierarchy of activities; deletion or relocation of GUI elements; usage of physical buttons; layout and graphics change; adaptation to different hardware and device models; activity flow variations; execution time variability.

Modifications performed to test code can be divided in 4 categories [29]: perfective maintenance, when test code is refactored to increase its quality; adaptive maintenance, to adapt the test code to the evolution of the product code; preventive maintenance, to avoid the possible need for intervention in test code in future releases; corrective maintenance and bug fixes. According to our definition, a test method is fragile when the modifications performed on it are adaptive, and related to aspects of the application GUI.

To implement an automated classification of test classes without semantically discriminating any modification between these four categories, we adopt a

simple heuristic: we consider any modified method of a GUI test class as fragile. When a test class is modified, we consider it as non fragile if there are no modified methods inside it. Hence, we do not consider as fragile the classes whose modifications are only among import statements and constructors, or caused by the addition or removal of test methods.

We suppose, in fact, that the addition of a new method should reflect the introduction of new functionalities or new use cases to be tested in the application, and not the modification of existing elements of the already tested activities. On the other hand, if some lines of code inside a single test method had to be modified, it is more likely that tests had to be rewritten due to minor changes in the application and possibly in its user interface (e.g. modifications in the screen hierarchy and in the transitions between activities).

## 3 Study Design

The aim of this work was identifying how a set of well-known set of GUI testing tools were used among Android open-source applications, and to quantify how many changes are needed by test classes. Finally, we wanted to give evidences of the presence of GUI testing fragility.

Thus, we organized the paper around the following research questions:

- **RQ1** *Evolution: how much Android GUI test code is modified over different releases?*
- **RQ2** *Fragility: how many fragilities occur in Android GUI test suites?*

As the first step of our research, we extracted a set of open source projects from GitHub that presented multiple releases and that could be considered as Android applications.

Then, we studied how applications were changed through their release history, and we did the same for their test classes. We tracked the evolutions of individual test classes and methods, so that we could compute fragility estimations.

In the following, we give details about the set of metrics we defined to track modifications and fragilities, and the tools we selected for our investigations.

### 3.1 Metrics definition

Table 1 reports the metrics we defined, and the ranges of value they can belong to. The metrics are explained in detail in the following subsections.

Some metrics have been presented in literature, for the quantification of modifications in test classes and test methods. For instance, Tang et al. [30] define eighteen metrics for the description of bug-fixing change histories. They are divided into three different categories: size (e.g., lines of code added or removed, number of changed classes, files or methods), atomic (e.g., boolean values indicating whether a class has added methods), or semantic (e.g., number of added or removed dependencies). The metrics defined in the following are at a higher level than the ones provided in the mentioned paper, and can also be defined on top of them.

**Table 1.** Metrics definition

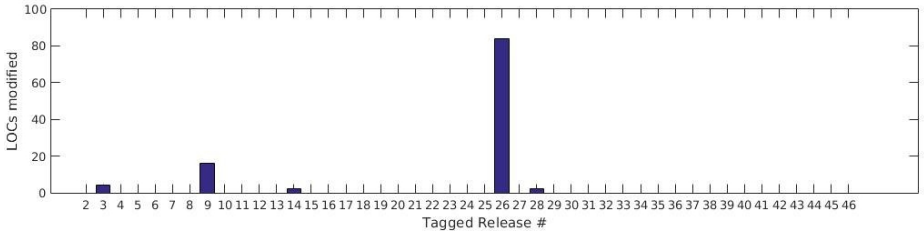
| Group | Name | Description | Range |
|---|---|---|---|
| Test evolution (RQ1) | TLR | Test LOCs Ratio | [0, 1] |
| | MTLR | Modified Test LOCs Ratio | [0, $\infty$] |
| | MRTL | Modified Relative Test LOCs | [0, 1] |
| | MRR | Modified Releases Ratio | [0, 1] |
| | TCV | Test Class Volatility | [0, 1] |
| Fragility (RQ2) | MCR | Modified Test Classes Ratio | [0, 1] |
| | MMR | Modified Test Methods Ratio | [0, 1] |
| | FCR | Fragile Classes Ratio | [0, 1] |
| | FRR | Fragile Releases Ratio | [0, 1] |
| | TCFF | Test Class Fragility Frequence | [0, 1] |

**Test suite evolution (RQ1)** The metrics answering RQ1 aim to describe the evolution of Android projects and relative test suites; they have been computed for each couple of consecutive tagged releases, and then averaged over the whole history of each project.

- **TLR** (i.e., Test LOCs Ratio) defined as $TLR_i = \frac{Tlocs_i}{Plocs_i}$ where $Tlocs_i$ is the absolute number of lines of code in test classes for release $i$, and $Plocs_i$ is the total amount of Program LOCs for release $i$. This metric, lying in the $[0, 1]$ interval, allows us to quantify the relevance of the testing code inside the release.
- **MTLR** (i.e., Modified Test LOCs Ratio) defined as $MTLR_i = \frac{Tdiff_i}{Tlocs_{i-1}}$, where $Tdiff_i$ is the amount of added, deleted or modified test LOCs between tagged releases $i-1$ and $i$, and $Tlocs_{i-1}$ is the total amount of test LOCs in release $i-1$. This quantifies the amount of changes performed on existing test LOCs for a specific release of a project.
- **MRTL** (i.e., Modified Relative Test LOCs) defined as $MRTL_i = \frac{Tdiff_i}{Pdiff_i}$ where $Tdiff_i$ and $Pdiff_i$ are respectively the amount of added, deleted or modified test and production LOCs, in the transition between release $i-1$ and $i$. It is computed only for releases featuring test code (i.e., $TRL_i > 0$). This metric lies in the $[0, 1]$ range, and values close to 1 imply that a significant portion of the total effort in making the application evolve is needed to keep the test classes up to date.
- **MRR** (i.e., Modified Releases Ratio), computed as the ratio between the number of tagged releases in which at least a test class has been modified, and the total amount of tagged releases featuring test classes. 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 SUT.
- **TCV** (i.e., Test Class Volatility), can be computed for each test class as $TCV_j = \frac{MR_j}{Lifespan_j}$ where $MR_j$ is the amount of releases in which test class $j$ is modified, and $Lifespan_j$ is the number of releases of the application featuring the test class $j$.

As an example of the computation of TCV, figure 1 shows the number of LOCs modified in test class NotesScreenTest.java of the project nhaarman/Triad

**Fig. 1.** Modified test LOCs for class NotesScreenTest.java of project nhaarman/Triad




[31] (featuring Espresso), for each tagged release in which it is present. The test class is introduced in release 0.2.0 (number 2 in the history of the project) and is present until the master release (number 46 in the history). Hence, the lifespan of the class is 45. From the bar plot is evident that the test class has been modified five times during its lifespan. Thus, the volatility of the test class can be computed as $TCV = \frac{5}{45} = 0.11$.

**Fragility of test classes and methods (RQ2)** The following metrics aim to give an approximated characterization of the fragility of test suites.

- **MCR** (i.e., Modified test Classes Ratio) defined as $MCR_i = \frac{MC_i}{TC_{i-1}}$ where $MC_i$ is the number of modified test classes in the transition between release $i-1$ and $i$, and $TC_{i-1}$ the total number of test classes in release $i-1$ (the metric is not defined when $TC_{i-1} = 0$). 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 they test.
- **MMR** (i.e., Modified test Methods Ratio) defined as $MMR_i = \frac{MM_i}{TM_{i-1}}$ where $MM_i$ is the number of modified test methods between releases $i-1$ and $i$, and $TM_{i-1}$ is the total number of test methods 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.
- **FCR** (i.e. Fragile Classes Ratio) defined as $FCR_i = \frac{MCMM_i}{TC_{i-1}}$ where $MCMM_i$ is the number of test classes that are modified, and that feature at least one modified method between releases $i-1$ and 1. $TC_{i-1}$ is the number of test classes featured by release $i-1$ (the metric is not defined when $TC_{i-1} = 0$). This metric represents the percentage of fragile classes (according to our definition), upon the entire set of test classes featured by a tagged release of the project. The metric is upper-bounded by $MCR$, since by its definition $MCR_i = MC_i/TC_i$, and $MCMM_i \leq MC_i$.
- **FRR** (i.e., Fragile Releases Ratio), computed as the ratio between the number of tagged releases featuring at least a fragile class, and the total amount of tagged releases featuring test classes. This metric lies in the range $[0, 1]$ and is upper bounded by $MRR$.

**Fig. 2.** Diff for class TheFullScreenBarcodeActivity.java of ligipassandroid (releases 3.2.0 - 3.2.1)

```
@@ -2,7 +2,7 @@ package org.ligi.passandroid;

 import android.graphics.Bitmap;
 import android.graphics.drawable.BitmapDrawable;
-import android.test.sitebuilder.annotation.MediumTest;
+import android.suport.test.filters.MediumTest;
 import android.widget.ImageView;
 import butterknife.ButterKnife;
 import com.squareup.spoon.Spoon;
```

**Fig. 3.** Diff for class ThePassEditActivity.java of ligipassandroid (releases 3.2.0 - 3.2.1)

```
@@ -1,7 +1,7 @@
-import android.test.suitebuilder.annotation.MediumTest;
+import android.support.test.filters.MediumTest;
 import com.squareup.spoon.Spoon;
 import javax.inject.Inject;
 import org.ligi.passandroid.model.PassStore;

@@ -43,6 +43,7 @@ public class The PassEditActivity extends BaseIntegration<
     PassEditActivity> {
  onView(withId(R.id.categoryView)).perform(click());
+onView(wihText(R.string.select_category_dialog_title)).
perform(click());

@@ -53,6 +54,7 @@ public class The PassEditActivity extends BaseIntegration<
     PassEditActivity> {
  public void testSetToCouponWorks() {
   onView(withId(R.id.categoryView)).perform(click());
+onView(withText(R.string.select_category_dialog_title))
.perform(click());
   onView(withText(R.string.category_coupon)).perform(click());
   assertThat(passStore.getCurrentPass().getType()).isEqualTo(PassType.COUPON)
       ;

@@ -73,7 +75,7 @@ public class The PassEditActivity extends BaseIntegration<
     PassEditActivity> {
  public void testColorWheelIsThere() {
   onView(withId(R.id.categoryView)).perform(click());
-onView(withText(R.string.button_text_change_color))
.perform(click());
+onView(withText(R.string.change_color_dialog_title))
.perform(click());
```

– **TCFF** (i.e., Test Class Fragility Frequence) defined as $TCFF_j = \frac{FR_j}{Lifespan_j}$ where $FR_j$ is the the amount of releases in which the test class $j$ contains modified methods, and $Lifespan_j$ is the number of releases of the application featuring the test class $j$. This metric is upper bounded by $TCV$, since by construction $MR_j$ (the number of releases in which the class is modified) is higher or equal to $FR_j$

For instance, in figure 2 the output of the Git Diff command for the class TheFullScreenBarcodeActivity.java of the repository ligi/passandroid ( [32], featuring Espresso), between releases 3.2.0 and 3.2.1, is shown. The class is modified, but there are no modifications inside test methods. In fact, the only two lines modified are import statements. Hence, this class counts as a modified class (thus counting for $MC$) but, since $MM = 0$, it is not considered as a fragile class. Thus, it does not count for $MCMM$.

The second sample in figure 3 is the diff for the class ThePassEditActivity.java. In the class there are four rows modified inside three different test methods. In this case, the class counts for $MC$ and also for $MCMM$, since $MM = 3$.

To validate the metrics defined for fragile classes and fragile methods (since we consider as fragile also tests that are modified for reasons different from GUI modifications) we compute the *Precision* metric as $P = \frac{TP}{TP+FP}$, where $TP$ is the number of True Positives, in our case the test methods whose changes actually reflect modifications in the GUI (or, if we consider test classes, the ones with modified methods and actually fragile); $FP$ is the number of False Positives, i.e., the methods that feature changed LOCs, but due to different reasons (or the test classes with modified methods, but not fragile). $P$ is defined in the range $[0, 1]$: values closer to 1 are an evidence that the presence of modified lines in test methods is a dependable proxy to identify modifications in test classes due to changes related to the user interface of the application.

### 3.2 Instruments

This section describes the tools and scripts that have been used to extract identify and filter projects, and to extract the measures we used for our analysis on evolution and fragility.

**GitHub Repository Search** GitHub Repository Search API allows to extract all projects containing a certain keyword in their names, readmes or descriptions. To access the GitHub API we have used the cURL Linux program inside a bash script.

Since the output of the GitHub Repository Search is limited to the first 1000 occurrences found, we have taken advantage of the "created" parameter to limit consecutive searches over a set of time ranges. The "language" parameter can be used to limit the search repositories according to the language they are written in. The output of the API call is in the form of a Json file: we used jsawk to inspect its content.

The following query returns a set of up to 1000 repositories -written in java- featuring a given keyword in their description, readme or name:

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

**Git Code Search** To search for particular code inside files belonging to a given project, the GitHub Code Search API can be leveraged. The search can be parametrized with the "filename" parameter, which makes the search to be performed only in files named as the provided keyword (if the parameter is omitted, the keyword is searched in all files of the repository). The "repo" parameter allows to specify the repository in which to perform the search.

Some limitations have to be considered when GitHub Code Search API is used, as it is explained in the Git documentation [33]: (i) only the default branch (in most cases the master branch) is considered for the code search; thus, if the

searched keyword is present in older releases but is removed in the master branch, the project will not be part of the results; (ii) for performance reasons, only files smaller than 384kb are searchable; (iii) only repositories with fewer than 500,000 files are searchable. Latest two issues may not be very relevant in our context, since the size of the projects and files considered is typically not so big - with the exceptions of whole firmwares and clones of the entire Android Operative System. The following sample query returns the names of the files, containing a provided keyword, found inside a GitHub repository:

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

**Count of (modified) lines of code** To count the total lines of code inside a repository (or a set of files) we leveraged the open-source Cloc tool [34].

The number of modifications performed to files of a GitHub repository can be obtained using the git diff command. Giving just the two releases as parameters of the command, the modifications performed to the whole repository are shown; as an alternative, it is possible to specify the full paths of a file to be found in each release, to obtain the modifications that were performed on that specific file. Specifying the -M parameter allows to identify files that have been renamed or moved, without considering such operation as the combination of a deletion and an addition of a file.

**Java class parser** We based our Java class examiner on JavaParser [35], an open source tool (available on GitHub) that can be used to explore the structure of any Java application. We developed a console tool which, given two releases of the same test class:(i) extracts all methods that are declared in both the releases; (ii) lists all methods that have been removed from the class; (iii) lists all the methods that have been added to the class; (iv) inspects the diff file for common methods, and for each modified line checks whether it has been performed inside an existing method. The output of the script gives the count of total, added, modified, and removed methods.

### 3.3 Selected Testing Tools

The first two tools we have searched for are part of the official *Android Instrumentation Framework* [36]. *Espresso* [37] 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). *UI Automator* [38], adds some functionalities to those provided by Espresso: it allows to check the device performance, to perform testing on multiple applications at the

same time, and operations on the system GUI. Both tools can be used only to test native apps.

*Selendroid* [39] is a testing framework based on Selenium, that allows to test the GUI of native, hybrid and web-based applications; the tool allows to retrieve elements of the application and to inspect the current state of the app's GUI without having access to its source code, and to execute the test cases on multiple devices at the same time.

*Robotium* [40] is an open-source extension of JUnit for the testing of 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* [41] is a tool that can be used to perform black-box testing directly on the Java Virtual Machine, without the use of a real device or an emulator; 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* [42] leverages WebDriver and Selendroid for the creation of black-box test cases that can be run on multiple platforms (e.g., Android and iOS); test cases can be created via an inspector that enables basic functions of recording and playback, via image recognition, or via code. It can be used to test both native and web-based applications. Test scripts can be data-driven.

### 3.4 Procedure

**Context definition** The first step we performed for the definition of our context was the search for the word "Android" in the descriptions, readmes and names of projects of GitHub, using the GitHub Repository Search API. All the projects that had no tagged releases (and hence, no evolution to investigate) were excluded from the context.

A second step of filtering has been applied, in order to try to avoid including in the context spurious results (e.g., libraries, utilities, applications developed for other systems intended to interface with Android counterparts). In fact, as it is explained in the Android developer's guide [43], it is mandatory for any Android app to have a Manifest file in its root directory. Therefore we used GitHub Code Search to search for files named "AndroidManifest.xml" and cointaining the keyword "manifest", and we cut out from the context the projects not giving any positive occurrence to this search.

To search for any of the testing tools considered, we used again GitHub Code Search on the Android projects that are included in the context. We considered Java files featuring the name of a testing technique as test classes.

**Test LOCs analysis (RQ1)** To answer RQ1, for each pair of consecutive tagged releases of every project, the total amount of modified LOCs is computed. Then, the total amount of LOCs added, removed or modified in the test files

previously identified is computed. Those values allow to compute *TLR*, *MTLR* and *MRTL* for each tagged release.

Finally, when the exploration of the project history is completed, global averaged values are 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. At the end of the exploration of the tagged releases of each project, *MRR* is computed to quantify the percentage of them featuring modifications in test classes.

Lifespan and volatility ($TCV_j$) are computed for each test class, and then an overall average is computed as $\overline{TCV} = Avg_j\{TCV_j\}$ with $j \in [1, NTC]$.

**Test classes history tracking, Fragility (RQ2)** We have tracked the evolution of single test classes and methods, taking into account the tagged releases in which each test class has been added, modified or deleted.

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

At the end of the exploration of the tagged releases of each project, *FRR* and has been computed to quantify the percentage of them featuring modifications due to fragility.

Test class Fragility Frequence ($TCFF_j$) has been computed for each test class, and then an overall average has been computed for each project, as $\overline{TCFF} =$ $Avg_j\{TCFF_j\}$ with $j \in [1, NTC]$.

To understand how the frequency of fragility depends on the amount of tagged releases and testing code ratio of Android projects, we also extracted a subset of our original context, composed by projects having more than 5 tagged releases and 10% Testing LOCs Ratio. We computed again the fragility metrics on this subset, and compared the results to the ones computed over the whole context.

A manual inspection of a set of modified test classes with modified methods has been conducted, in order to verify the dependability of the metrics defined to identify fragile methods and fragile classes (i.e., $MMR$ and $FCR$). 30 couples of releases of different classes have been selected randomly, and manually examined before and after they were modified. The modifications performed were characterized under four categories: (i) bug fixing and refactoring (e.g. enhancement of test methods, adaptations to new APIs and removal of deprecated method calls, etc.); (ii) syntactical correction and formatting; (iii) adaptation to changes in program code not related to GUI; (iv) adaptation to changes in program code related to GUI. Only the modifications belonging to the last class are considered as true positives for our analysis; all the others are considered as false positives. Based on that subdivision, precision is estimated for two metrics: the percentage of fragile classes, and the percentage of fragile methods.

**Table 2.** Sets of projects considered

| Tool | Projects | Mean $Tlocs$ | Mean $TLR$ |
|---|---|---|---|
| Espresso | 398 | 558 | 8.8% |
| UIAutomator | 105 | 3,155 | 8.6% |
| Selendroid | 6 | 8,627 | 19.4% |
| Robotium | 145 | 873 | 8.7% |
| Robolectric | 826 | 1,448 | 16.4% |
| Appium | 11 | 4,469 | 37.3% |

## 4 Results and Discussion

Applying the search procedure and the filtering to cut out spurious projects, a set of 18,930 Android projects was mined from GitHub, and used for the computation of the metrics defined before.

The sizes of the sets featuring each of the six chosen testing tools are shown in table 2. The table reports also information about the size of the testing suites: the average total amount of testing LOCs ($Tlocs$) and the average relative amount of testing LOCs ($TLR$) computed for master releases.

### 4.1 Test suite evolution (RQ1)

Table 3 shows the statistics collected about the average evolution of test code, for the six selected testing tools. For every set, $\overline{TLR}$, $\overline{MTLR}$, $\overline{MRTL}$, $MMR$ and $\overline{TCV}$ have been averaged on all the projects. The values in last row are obtained as averages of the six values above, weighted by the size of the six sets.

The values reported for average Test LOCs Ratio ($\overline{TLR}$) show that – when present – GUI testing can be a relevant portion of the project during its lifecycle, if compared to the total LOCs of the application. The average values range from about 7.3% (for the set of Espresso projects) to 31.9% (for the set of Appium projects). For the largest set of projects considered (the ones featuring Robolectric) the mean $\overline{TLR}$ is 13.4%. The $\overline{TLR}$ averaged over the releases of applications is typically smaller than the $TLR$ computed for master releases (see table II): this is due to the graduality of the construction of test suites, which may be very small or absent in intial releases. The boxplots in figure 4 show the distribution of $\overline{TLR}$ values for the six sets of projects.

The average Modified Test LOCs Ratio ($\overline{MTLR}$) shows that typically around 2.8% of test code is modified between consecutive releases. Very small values were obtained for the projects featuring UIAutomator; this should be a consequence of bigger test suites, in terms of absolute total test LOCs, with respect to the ones written with Espresso, Robolectric and Robotium (the sets featuring a significant number of projects).

The measures about the Modified Relative Test LOCs ($\overline{MRTL}$) show that, on average, when GUI testing tools are used, the 7.4% 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 performed on test classes.

**Table 3.** Measures of the evolution of test code (averages on the sets of repositories)

| Tool | $\overline{TLR}$ | $\overline{MTLR}$ | $\overline{MRTL}$ | $MRR$ | $\overline{TCV}$ |
|---|---|---|---|---|---|
| Espresso | 7.3% | 2.6% | 4.7% | 22.2% | 8.6% |
| UI Automator | 9.6% | 1.4% | 3.5% | 16.5% | 6.4% |
| Selendroid | 19.4% | 4.3% | 11.5% | 39.6% | 11.5% |
| Robotium | 7.8% | 3.8% | 5.3% | 22.1% | 9.9% |
| Robolectric | 13.4% | 2.9% | 9.5% | 28.2% | 8.6% |
| Appium | 31.9% | 1.8% | 16.6% | 27.3% | 10.3% |
| Average | 11.1% | 2.8% | 7.4% | 25.2% | 8.6% |

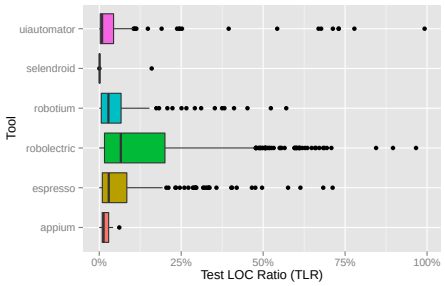



**Fig. 4.** Distribution of $\overline{TLR}$

The higher $\overline{MRTL}$ values for the sets of projects featuring Appium and Selendroid can be justified by the small size of the two sets, and by the nature of the projects examined. For instance, the whole Selendroid framework, on GitHub as selendroid/selendroid, is subject to heavy modifications.

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, about 25% of releases needed modifications in the test suite (with a maximum of 39.6% for the set of projects featuring Selendroid). 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 8.4% value for the Test Class Volatility ($\overline{TCV}$) metric, which characterizes the phenomenon from the point of view of the individual test classes, highlights the fact that each test class has to be modified, on average, every ten tagged releases in which it appears.

As an example of evolution of GUI testing code, we considered the project ligi/PassAndroid, that feature both unit tests written in jUnit, and GUI tests built with Espresso. Figure 5 shows the amount of Espresso and JUnit code throughout the history of the app. The graph highlights that the need for GUI testing can be even more stringent than the need for Unit testing checking the application logic. Figure 6 shows the history of modifications of Espresso test classes of Passandroid. A portion of the set of test classes had to be modified for each release of the application. In an occasion (v2.4.5) the entire test suite

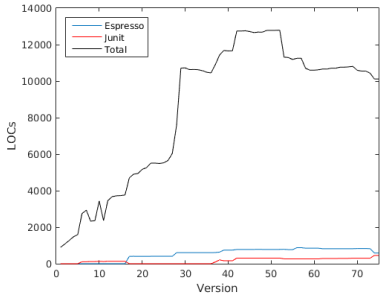



**Fig. 5.** PassAndroid: GUI test LOCs compared to JUnit and total LOCs

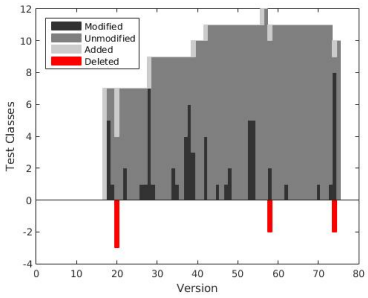



**Fig. 6.** PassAndroid: history of the set of Espresso test classes.

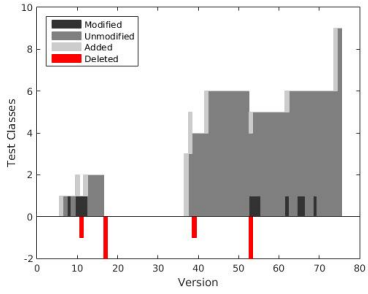



**Fig. 7.** PassAndroid: history of the set of JUnit test classes.

had to be rewritten. The absolute number of interventions performed on the test classes is significantly higher for Espresso than for JUnit (see figure 7).

### 4.2 Fragility of test classes and methods (RQ2)

Table 4 shows the fragility estimations that we have computed for each project, and then averaged over the six sets: $\overline{MCR}$, $\overline{MMR}$, $\overline{FCR}$. Based on them, we computed two additional derived metrics: $FRR$ and $\overline{TCFF}$ The values in last row are obtained as averages of the six values above, weighted by the size of the six sets.

The first column about the Modified Classes Ratio ($\overline{MCR}$) metric shows that, on average, 14.8% of test classes are modified between consecutive tagged releases in our set of Android projects. The exception is represented by the set of projects adopting UIAutomator, we need to investigate further the cause of such difference.

The 3.5% 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.

**Table 4.** Measures for fragility (averages on the sets of repositories)

| **Tool** | $\overline{MCR}$ | $\overline{MMR}$ | $\overline{FCR}$ | $FRR$ | $\overline{TCFF}$ |
|---|---|---|---|---|---|
| Espresso | 15.2% | 3.5% | 8.3% | 14.4% | 4.6% |
| UI Automator | 9.0% | 1.8% | 4.6% | 10.2% | 3.1% |
| Selendroid | 16.5% | 2.7% | 4.9% | 28.2% | 3.4% |
| Robotium | 16.4% | 3.5% | 9.3% | 15.2% | 5.7% |
| Robolectric | 15.1% | 3.8% | 8.5% | 20.6% | 4.9% |
| Appium | 15.2% | 4.6% | 7.7% | 17.1% | 5.1% |
| *Average* | 14.8% | 3.6% | 8.2% | 17.7% | 4.8% |

**Table 5.** Percentage of projects without modifications in test suites, classes and methods

| Tool | Unmodified suites | Unmodified classes | Unmodified methods |
|---|---|---|---|
| Espresso | 24.6% | 57.0% | 65.8% |
| UIAutomator | 16.0% | 40.0% | 55.0% |
| Selendroid | 60.0% | 60.0% | 80.0% |
| Robotium | 16.6% | 44.1% | 60.0% |
| Robolectric | 15.8% | 45.3% | 53.3% |
| Appium | 27.3% | 54.5% | 72.7% |

The Fragile Classes Ratio ($\overline{FCR}$) metric gives the ratio between the classes that we define fragile upon all the classes contained by each project. On average, about 8% of the classes were fragile in the transition between consecutive releases.

The Fragile Releases Ratio ($FRR$) metric gives an indication of how many releases of the considered project contained test classes that we identify as fragile. The value is upper-bounded by $MRR$, which is the frequence of releases featuring any kind of modification. A value of $\overline{FRR} = 17.5\%$ is relevant, because it means that about one in five releases require a change in test methods.

Upper-bounded by $\overline{TCV}$ (the overall volatility for test classes), the average Test Class Fragility Frequence ($\overline{TCFF}$) provides information about the frequency of modifications that test classes must undergo because of fragilities. The average value of 4.8% tells us that a typical test class must be modified because of fragilities every 20 releases in which it appears.

It must also be considered that the averages reported above are heavily lowered by those projects in which test classes and methods are inserted – at the beginning or at some point in their history – but are never modified later. In table 5 we show: the percentage of projects whose test suites are never modified; the percentage of projects with no modifications in test classes (i.e., only additions and modifications of test classes are performed); the percentage of projects with no modifications in test methods (i.e., only additions and modifications of methods are performed, and no fragility is detected).

In table 6 we show the results that have been gathered only for those projects that feature a relevant percentage of test code among product code (more than 10%) and more than five tagged releases. We show our results for the four biggest sets of projects considered, being the ones featuring Selendroid and Appium not

**Table 6.** Measures for fragility for long-lived and tested projects

| Tool | $\overline{MCR}$ | $\overline{MMR}$ | $\overline{FCR}$ | $FRR$ | $\overline{TCFF}$ |
|---|---|---|---|---|---|
| Espresso | 21.2% | 4.5% | 12.5% | 26.6% | 9.0% |
| UI Automator | 10.0% | 0.9% | 4.9% | 11.7% | 8.1% |
| Robotium | 25.7% | 5.2% | 13.8% | 29.0% | 18.3% |
| Robolectric | 30.0% | 5.3% | 12.5% | 36.0% | 14.7% |

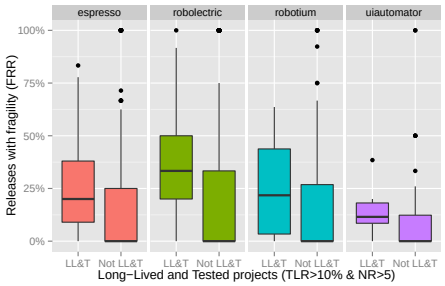



**Fig. 8.** Distribution of $FRR$ metric for long-lived and tested projects

relevant for such a comparison because they contain a few units. Our hypothesis is that projects that have a longer history and bigger test suites, are more inclined to have test classes and methods modified than shorter-lived projects in which the amount of testing is negligible. The metrics we computed confirm our supposition: with the only exception of the set of projects featuring UIautomator, average $FRR$ is nearly doubled for all sets (comparative boxplots are shown in figure 8). Also the average $\overline{TCFF}$ is largely increased if only applications with relevant test code percentage and release history are considered.

### 4.3 Metrics validation

Table 7 shows the results of the validation procedure for RQ2. We found that about 70% of the modifications of methods are true positives if we consider them as proxies of modifications limited to the GUI and not affecting the functionality of the application. Hence, we can consider that modifications in the GUI of the application are involved in the majority of the modifications to test methods and classes.

We can also compute an estimate of the precision if we consider classes and not individual methods. We consider a class found as fragile a true positive if it contains at least a method whose modifications are connected to the GUI of the SUT. In this case we found 21 classes with modifications related to the GUI of the SUT, and hence 21 true positives among 30 samples (70%).

**Table 7.** Precision for Fragile Methods and Classes

| Metric | Measured | TP | FP | P |
|---|---|---|---|---|
| Fragile Methods | 65 | 45 | 20 | 69.2% |
| Fragile Classes | 30 | 21 | 9 | 70.0% |

## 5 Threats to Validity

*Threats to internal validity.* The test class identification process is based on the search of the name of the tool as keyword: any file containing one of such keyword is considered as a test file without further inspection; this procedure may miss some test classes, or consider a file as a test file mistakenly. The number of tagged releases is used as a criterion to identify a project as worth to be investigated; it is not assured that this check is the most dependable one for pruning negligible projects. The scripts and tools we used assume that no syntactic errors are present inside the test classes on which they operate, and that the names of those files are properly spelled (e.g., without the presence of special characters or blank spaces); the correctness of the metric extraction technique is not assured in different circumstances.

*Threats to external validity.* Our findings are based only on the GitHub open-source project repository. Even though it is a very large repository, it is not assured that such findings can be generalized to closed-source Android applications, neither to ones taken from different repositories. The applications we extracted are not necessarily released to final users. Nevertheless, we selected a subset of projects that were released on the Play Store, and the average metrics computed on them were not significantly different from the ones computed on the whole sample. We have collected measures for six scripted GUI automated testing tools. It is not certain that such selection is representative of other categories of tools or different tools of the same category, which may exhibit different trends of fragilities throughout the history of the projects featuring them.

*Threats to construct validity.* We link the GUI test fragility to any change in the interface that requires an adaptation of the test. The proxy we used - a change in any test method - is not perfectly linked to a change in the GUI. The magnitude of this threat has been evaluated with a Precision measure equal to 70%. This might reduce our fragility estimate but not change its order of magnitude.

## 6 Conclusion and Future Work

The objective of this paper was characterizing the amount of GUI testing fragilities that have been experienced by open-source Android projects during their history. We analyzed the transitions between consecutive tagged releases for projects featuring some relevant GUI automated testing tools -Espresso, UIAutomator, Selendroid, Robotium, Robolecric, Appium- among the repositories hosted by the GitHub portal.

We found that, when present, the GUI testing code can be up to 13% of the whole program code. Concerning the evolution of test code, on average in every release about 7.5% of the changed lines are in the UI test code, and about 3% of test code has to be modified.

The fragility of the tests can be estimated with two metrics based on the raw count of classes and methods modified. Overall we can estimate fragility of the analyzed test classes around 8% (meaning that there is such probability that a test class may include a modified test method). The association between modified test methods and modifications in the GUI has been proved dependable in 70% of the cases examined.

On average, around 25% of releases featuring test classes need intervention in test classes to keep them aligned with the production code. About 18% of test classes need intervention because of the presence of classes that are marked as fragile by our definition, and for each transition, there is 5% possibility for any class to need modifications due to fragility. These results show that developers need to rather frequently adapt their GUI scripted testing suites to the evolution of the application. Each individual test class, on average, has to be modified one every ten tagged releases of the project, and one every twenty if we consider just modification due to fragilities.

These evaluations show that fragility can be a relevant issue for automated scripted testing for Android applications, and that it may be one of the factors discouraging developers for a thorough testing of their software. We have also found that the fragility problem is significantly more relevant when test code is not a negligible portion of Program code, and when the applications feature a sufficient number of tagged released to be considered long-lived.

In the future it might be possible to define a taxonomy of causes of fragilities, guidelines to help developers to avoid it, and finally automated tools capable of adapting the test cases to modificatons made in the user interfaces. An extension of the study to other databases of open-source projects, to compare different testing frameworks or typologies, or to other sotware platforms (like iOS) is also possible.

**Acknowledgement.** This work was supported by a fellowship from TIM.

## References

1. "Worldwide device shipments to grow 1.9 percent in 2016, while end-user spending to decline for the first time," http://www.gartner.com/newsroom/id/3187134, 2016, accessed: 2017-01-18.
2. "Global mobile os market shares in sales to end users from 1st quarter 2009 to 1st quarter 2016," https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/, 2016, accessed: 2017-01-18.
3. A. Kaur, "Review of mobile applications testing with automated techniques," *interface*, vol. 4, no. 10, 2015.

4. P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, April 2015, pp. 1–10.
5. M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution." in *WCRE*, 2013, pp. 272–281.
6. ——, "Visual vs. dom-based web locators: An empirical study," in *International Conference on Web Engineering*. Springer, 2014, pp. 322–340.
7. H. Muccini, A. D. Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *2012 7th International Workshop on Automation of Software Test (AST)*, June 2012, pp. 29–35.
8. B. Kirubakaran and V. Karthikeyani, "Mobile application testing #x2014; challenges and solution approach through automation," in *2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering*, Feb 2013, pp. 79–84.
9. J. Gao, X. Bai, W. T. Tsai, and T. Uehara, "Mobile application testing," *Computer*, vol. 47, no. 2, pp. 46–55, 2014.
10. M. Kropp and P. Morales, "Automated gui testing on the android platform," *on Testing Software and Systems: Short Papers*, p. 67, 2010.
11. M. Linares-Vásquez, "Enabling testing of android apps," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 763–765.
12. "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey.html, 2016, accessed: 2017-01-18.
13. A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234.
14. Y. Zhauniarovich, A. Philippov, O. Gadyatskaya, B. Crispo, and F. Massacci, "Towards black box testing of android apps," in *Availability, Reliability and Security (ARES), 2015 10th International Conference on*. IEEE, 2015, pp. 501–510.
15. J. Kaasila, D. Ferreira, V. Kostakos, and T. Ojala, "Testdroid: automated remote ui testing on android," in *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*. ACM, 2012, p. 28.
16. L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 72–81.
17. C. H. Liu, C. Y. Lu, S. J. Cheng, K. Y. Chang, Y. C. Hsiao, and W. M. Chu, "Capture-replay testing for android applications," in *Computer, Consumer and Control (IS3C), 2014 International Symposium on*. IEEE, 2014, pp. 1129–1132.
18. M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "Mining android app usages for generating actionable gui-based execution scenarios," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 111–122.
19. D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, 2015.
20. D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and G. Imparato, "A toolset for gui testing of android applications," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 650–653.

21. D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 252–261.
22. D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 258–261.
23. T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based gui testing of an android application," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 377–386.
24. W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.
25. W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *ACM SIGPLAN Notices*, vol. 48, no. 10. ACM, 2013, pp. 623–640.
26. C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 67–77.
27. S. Singh, R. Gadgil, and A. Chudgor, "Automated testing of mobile applications using scripting technique: A study on appium," *International Journal of Current Engineering and Technology (IJCET)*, vol. 4, no. 5, pp. 3627–3630, 2014.
28. R. Coppola, E. Raffero, and M. Torchiano, "Automated mobile ui test fragility: an exploratory assessment study on android," in *Proceedings of the 2nd International Workshop on User Interface Test Automation*. ACM, 2016, pp. 11–20.
29. V. G. Yusifoğlu, Y. Amannejad, and A. B. Can, "Software test-code engineering: A systematic mapping," *Information and Software Technology*, vol. 58, pp. 123–147, 2015.
30. X. Tang, S. Wang, and K. Mao, "Will this bug-fixing change break regression testing?" in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015, pp. 1–10.
31. "Triad: Custom view based screen navigation in android," https://github.com/nhaarman/triad/, 2016, accessed: 2017-01-18.
32. "Passandroid: Android app to view passes," https://github.com/ligi/PassAndroid/, 2016, accessed: 2017-01-18.
33. "Search - github developer guide," https://developer.github.com/v3/search/, 2016, accessed: 2017-01-18.
34. "Cloc - count lines of code," https://github.com/AlDanial/cloc, 2016, accessed: 2017-01-18.
35. "Java parser and abstract syntax tree," https://github.com/javaparser/javaparser, 2016, accessed: 2017-01-18.
36. "Android instrumentation framework," https://developer.android.com/studio/test/index.html, 2016, accessed: 2017-01-18.
37. "Testing ui for a single app," https://developer.android.com/training/testing/ui-testing/espresso-testing.html, 2016, accessed: 2017-01-18.
38. "Testing ui for multiple apps," https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html, 2016, accessed: 2017-01-18.
39. "Github: selendroid/selendroid," https://GitHub.com/selendroid/selendroid, 2016, accessed: 2017-01-18.

40. H. Zadgaonkar, *Robotium Automated Testing for Android*. Packt Publishing Ltd, 2013.
41. "Robolectric - test-drive your android code," http://robolectric.org/, 2016, accessed: 2017-01-18.
42. G. Shah, P. Shah, and R. Muchhala, "Software testing automation using appium," *International Journal of Current Engineering and Technology (IJCET)*, vol. 4, no. 5, pp. 3528–3531, 2014.
43. "App manifest," https://developer.android.com/guide/topics/manifest/manifest-intro.html, 2016, accessed: 2017-01-18.