Automated Mobile UI Test Fragility: An Exploratory Assessment Study on Android

Automated Mobile UI Test Fragility: An Exploratory Assessment Study on Android

Riccardo Coppola, Emanuele Raffero, Marco Torchiano
Dipartimento di Automatica e Informatica
Politecnico di Torino, Torino, Italy
{first.last}@polito.it

ABSTRACT

Automated UI testing suffers from fragility due to continuous – although minor – changes in the UI of applications. Such fragility has been shown especially for the web domain, though no clear evidence is available for mobile applications.

Our goal is to perform an exploratory assessment of the extent and causes of the fragiliy of UI automated tests for mobile applications.

For this purpose, we analyzed a small test suite -that we developed using five different testing frameworks- for an Android application (K-9 Mail) and observed the changes induced in the tests by the evolution of the UI.

We found that up to 75% of code-based tests, and up to 100% of image recognition tests, had to be adapted because of the changes induced by the evolution of the application between two different versions. In addition we identified the main causes of such fragility: changes of identifiers, text or graphics, removal or relocation of elements, activity flow variation, execution time variation, and usage of physical buttons.

The preliminary assessment showed that the fragility of UI tests can be a relevant issue also for mobile applications. A few common causes were found that can be used as the basis for providing guidelines for fragility avoidance and repair.

CCS Concepts

•**Software and its engineering → Software testing and debugging;**

Keywords

Test; UI; Automated; Fragility; Empirical

1. INTRODUCTION

Heading towards its seventh release, the Android programming platform provides the developer with a very vast collection of functionalities, as well as fashionable alternatives for user interfaces. Current applications manage a vast array of sensible data about the user and perform very complex tasks, making mobile devices really close to universal computational machines like desktop computers.

Among the characteristics that have caused the success of Android, the large quantity of apps available on the market is one of the most prominent [2]. The openness of the developing platform, and the opportunity of rapidly reaching very huge crowds of users, lead to very pressing competitive forces for apps deployed on the Play Store. Thus, it is crucial for applications to be capable of guaranteeing the promised behavior to their users, assuring a sufficient level of reliability.

In addition to the typical unit testing, which makes use of the JUnit4 testing framework, the developer can take advantage of several APIs engineered for the execution of instrumentation tests. The main difference between the two categories of tests lies in the fact that, while the former can be run on the Java Virtual Machine on the development environment, the latter require an instance of the Android system, either on an AVD (Android Virtual Device) or on an actual physical device. Instrumentation tests are particularly fit for automatic UI testing, that is ensuring that the final user does not get improper behaviors when giving input to the app -classifications of typical Android bugs have been given in literature, for instance in [10]- without manually repeating sequences of inputs to the application.

The creation of test cases for Android applications comes with a set of domain-specific challenges that the developer has to face. Applications are engineered to run on a very large amount of different configurations of display sizes, pixel densities, layouts and arrangements of buttons. Users may use a set of different input channels (e.g., voice commands or accelerometer inputs) and devices may have to change their behaviour seamlessly if the battery status or the network to which they are connected change [7]. Moreover, the dynamic swapping of applications on actual Android devices may easily lead them to unexpected behaviors, even during the execution of test suites.

When the normal evolution of an application may lead to a test to fail, when this failure is not due to an alteration of the behavior but to other changes, e.g. in the UI, the test is said to be *fragile*. Fragile tests may trigger extra effort upon failure to: (i) verify no regression has occurred in the system and then (ii) modify the test to adapt it to the changed UI [5]. There are techniques to automate such modification [14] but they are not often applicable or used.

Therefore, care must be taken in the process of test creation, in order to make tests as robust and portable (i.e. executable on various device classes), and less fragile as pos-

INTUITEST'16, July 21, 2016, Saarbrücken, Germany

http://dx.doi.org/10.1145/2945404.2945406

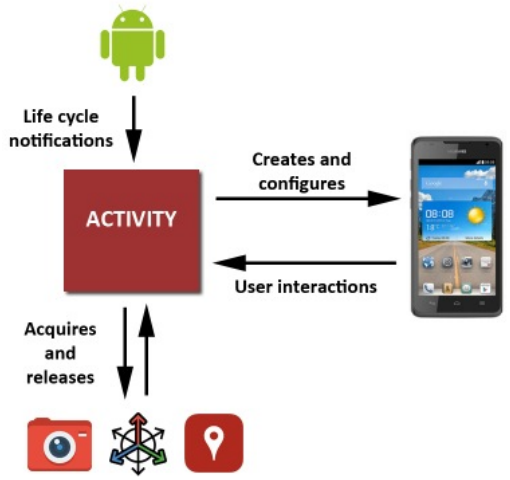

Figure 1: Relationship between activities and GUI in the Android OS.

sible, possibly running adapted versions of them on a set of different configurations [1].

The main goal of our work is to provide an exploratory study of the fragility of test suites in the context of mobile development. Taking in consideration a set of tools commonly used for the purpose of testing Android user interfaces, we also evaluate the advantages and disadvantages they present when applied to a popular open-source application.

Our aim is to obtain a basis for a taxonomy of the sources of fragility for mobile automated UI testing. The differences and the points of contact with other types of UI testing can be investigated starting from such a preliminary classification.

The remainder of the paper is organized as follows: section 2 provides some information about the Android Platform and the testing APIs we are considering. Section 3 introduces the methodology we have used for our study, the application we have chosen, and the test cases we developed. Section 4 presents the outcomes of the experiment and the fragilities found. Finally, section 5 discusses the results and provides a classification of the main causes of fragility we observed.

2. BACKGROUND

Android is an operating system as well as an application development platform, based on the Linux Kernel. Its architecture is formed by four layers: the topmost one is made by Android apps, each running its own process in a dedicated user space and on a dedicated copy of the Dalvik virtual machine, for security and data protection purposes. The application layer is served by the libraries of the application framework, and by the Linux Kernel layer.

The underlying hardware is accessed through the APIs provided by the Android SDK, with Java used as the privileged programming language. To use specific functions offered by the operating system, an application must require the related permissions on its manifest XML file.

Android apps are composed by several components, each one having its own lifecycle driven by the operating system with the invocation of a set of methods (e.g., *onCreate*). The app is alive if at least one of its component has not yet been destroyed by the system.

Components can be divided in four classes: Activities, Services, Content Providers and Broadcast Receivers. *Services* allow the performance of lenghty operations in the background; *Content Providers* manipulate the data used by the application; *Broadcast Receivers* listen for messages created either by the Operating System (e.g., "low battery") or by other applications (as a form of interprocess communication). *Activities* are the main components of each application, and in particular they are the only ones providing a user interface with which the user can interact (the work of activities is schematized by figure 1).

User interfaces exposed by the Activities are composed by a set of Views arranged on the screen according to a particular layout. Such arrangement can be specified by the programmer either using a static XML file describing the layout, or instantiating the corresponding classes in the Java code of the activity class, or using an hybrid approach. Each view in the layout can be given a unique *id* to be distinguished from the others, and to have the possibility of getting a reference to it in the Java code. The user interaction can actually have effects on the application once callbacks are registered on the elements of the layout (for instance, the *setOnClickListener* method is used to associate a particular behaviour when the user presses a button of the interface).

From API 12 *Fragments* have been introduced as an intermediate component between Views and Activities, in order to manage more easily interfaces that must adapt in complex ways to different classes of screens.

2.1 Testing Tools

Among the various tools available for testing the GUI of Android Apps, we focused on five automated testing tools that let the programmer write test cases simulating the typical interaction patterns of the users, checking whether the applications provides the expected behavior without incurring in bugs or exceptions.

The Android Testing Support Library [4] offers a versatile testing framework for developers, that enables the test of single activities as well as multiple activities run on the same device. The framework is based on AndroidJUnitRunner, a Java class able to launch JUnit3 and JUnit4 tests on Android applications: Espresso and UIAutomator rely on it to execute their tests. Other libraries, like dumpsys [3] allow to check the performance of the device (in terms of frame latency and timing).

In addition to those frameworks directly supported by Android, in our works we have also taken into account a couple of adaptations of UI testing tools for web applications (Selendroid and Sikuli) and a commercial tool (Silk Mobile).

Testing tools capable of automatically creating test cases for interface -either randomly generated, or according to some model- are also available in literature. A classification of them is made in [6].

What follows is an overview on the testing tools that we evaluated.

2.1.1 *Espresso*

Espresso provides a series of interfaces for the creation of automatic UI tests for a single app, using a grey-box testing approach. In fact, the internal arrangement of elements inside the view tree of the application needs to be known to actuate the most relevant input simulations of the framework.

The programmer has to specify the name of the class that is istantiated by the first activity of the application. With his *onView()* method, Espresso allows to locate specific components of the application UI. The method receives as parameter a *Matcher*, that permits to select the correct view according to some criteria. For instance, views can be selected according to their ids or class names, their actual state, or their textual content. The *onData()* method provides similar features, but is designed to work on AdapterViews. Once a view has been selected, Espresso allows the execution of operations on it (e.g., clicking, typing text, hitting buttons, swiping) by means of *ViewInteraction.perform()* and *DataInteraction.perform()*.

The framework uses synchronization mechanisms, and is capable to check whether activities are actually in a stable state before performing operations on views [11].

2.1.2 UIAutomator

The tool is designed for black-box and grey-box testing, i.e. it allows the interaction only with the elements of the interface actually displayed on the device display, using content or unique identifiers to retrieve elements of the interface.

In addition to the functionalities provided by Espresso, UIAutomator enables the access to the device status. It is possible to obtain information about the orientation of the device, the screen dimensions and resolution, and to perform operations like changing orientation and pressing physical buttons.

To perform testing operations, UIAutomator offers the classes *UiDevice*, to access the handheld device under examination, and *UiObject*, to refer specifically to a component of the user interface actually shown on the screen. Objects of interest can be identified using the function *findObject()* or the class *UiSelector*. Actions are finally performed on the UiObjects previously identified.

UIAutomator allows the creation of tests that span over multiple applications, either user or system ones, and perform operations on the system UI (e.g., the launcher, from which applications are selected). This feature is of unquestionable importance, since the architecture of Android encourages the frequent switch between different applications to handle particular user requests (for instance, an application may ask for the control of the system camera application to take a picture, instead of accessing the sensors by itself).

2.1.3 Selendroid

Selendroid is an open source testing framework based on Selenium2 API and the JSON Wire Protocol, instruments originally conceived for the test of web applications. It allows the creation of automated test cases for native Android applications, or for responsive websites tailored to handheld devices.

A fundamental component of the architecture is the *Selendroid Server*, installed and running on the target device. The *Selendroid Standalone Driver* interacts with it and the application under test is shown on the Inspector webpage (see figure 2), accessing the *HTTP Server*. *WebDriver Api* is leveraged for the purpose of creating Android tests.

The tool [18] has several interesting capabilities, like the possibility of sending inputs to multiple devices at the same time, or the ability of removing the current device and plugging a new one with no interruptions for the test. Complex

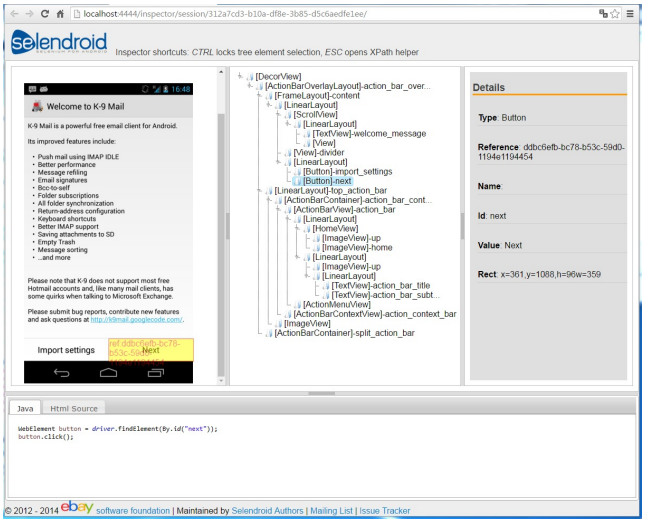

Figure 2: Testing an application with Selendroid.

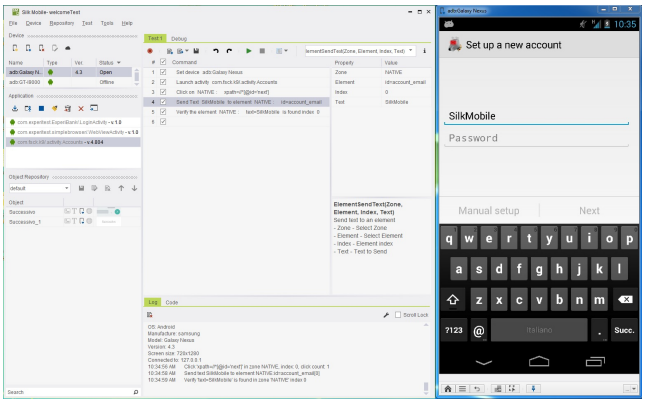

Figure 3: Silk Mobile User Interface.

gestures are also supported. Applications under test do not need to be modified.

2.1.4 Silk Mobile

Silk Mobile is a commercial instrument developed by Micro Focus and released by Borland, designed for the test of applications on multiple devices. The interaction with the user is recreated simulating the inputs on a mirrored interface on the desktop pc. Commands like multi-touch, drag and drop, zoom and swiping are allowed. Test results, alongside all the interactions between user and device, are reported in HTML format and are thus viewable by means of a web browser. The Silk Mobile interface is shown in figure 3: the proprietary emulator where the user can perform the inputs is on the right, whereas the central panel lists the recorded inputs, that are subsequently exportable and repeatable.

The tool supports several programming languages for the export and adaptation of tests in different contexts. The programmer can modify in any moment the registered test, inserting new commands in any point of it. As pointed out in [17], Silk Mobile is capable of testing apps in parallel on multiple devices, tracking memory and CPU usage on the device during the execution of tests, supporting complex ges-

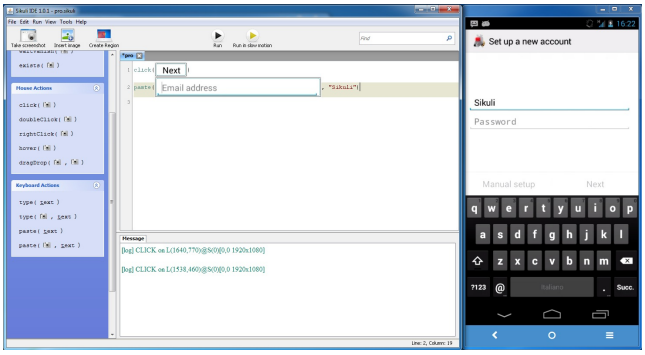

Figure 4: Sikuli IDE.

tures, aiding the inclusion of mobile testing in the Continuous Integration process.

2.1.5 *Sikuli GUI Automation Tool*

Sikuli is an open source image recognition tool that uses screenshots to identify specific elements of the user interface under test. Presented in [21], it was originally intended for the testing of web apps and applications written in Flash. Visual references to elements allow the developer to put screenshots straight into the test code (see figure 4), and then to specify what kind of operations must be performed on the corresponding GUI elements. Despite an intrinsic fragility due to the constant variation of user interfaces, the use of screenshots, in some cases, can make the creation of tests faster and more intuitive.

Although the specific developing environment for Sikuli uses Python, a Java library is also available, with methods capable of automating mouse and keyboard inputs given to a pc. To use Sikuli for the creation of mobile tests, it is necessary to run an emulator like androidscreencast [9] or Vysor [19], capable of executing remote operations on the device under test, by means of mouse and keyboard.

3. METHODOLOGY

The goal of this work can be formulated using the GQM (Goal, Question, Metric) template [20] as: analyze a set of automated UI tests for the purpose of characterizing the evolution with respect to their fragility from the point of view of the testers in the context of an Android mobile application.

The goal of our work gives rise to the following research questions:

RQ1 *What are the main advantages and disadvantages of common tools for automated Android UI testing?*

RQ2 *How many tests have to be modified to adapt to subsequent releases of the same app?*

RQ3 *Which are the intrinsic causes of the fragilities of UI tests?*

As it has been briefly introduced in previous sections, UI testing is a fundamental practice for the development of reliable Android apps. Testing can be performed either manually (with testers physically executing sequences of inputs on devices running the app under test) or through automated testing tools. However, since writing valid automated UI tests may be complex and time-consuming, it is important to investigate, on a case by case basis, whether it is worth investing in their development instead of doing manual tests. For this reason, we evaluated some testing tools, underlining their strenghts and weaknesses (RQ1).

Since Android apps are rapidly changed and kept at pace with the evolution of the operating system, it is important to assess whether the tests are reusable for subsequent releases, or are invalidated (RQ2). In the latter case, it may be useful to estimate how much effort is required on testers side to adapt the implementation of test to the new version of the application. Similar evaluations have been done in the field of Web application testing [12, 13]. The metric we use to answer RQ2 is the number of tests failed when updating the application to the subsequent version.

Finally, a study and a classification of the typical causes of fragility in user interfaces (RQ3) may lead to a set of best practices that can be adopted a priori in the development of Android apps, in order to obtain more robust and portable testing later. The metric we use to answer RQ3 is the number of occurrences of each of the kinds of fragilities we identify.

3.1 Selected Application and Tools

The development and execution of test is typically conducted within an Integrated Development Environment (IDE). We have used the Android Studio 1.1 IDE for the creation of Espresso and UIAutomator test suites, and the Eclipse Mars IDE for Selendroid and Sikuli test suites.

Our tests have been executed on two different handheld devices, using Android API 23 and Android API 19 (since Selendroid 0.10.0, the version we have used, supports only API prior to 20). We have used Vysor to simulate inputs to the handheld device using mouse and keyboard.

We selected K-9 Mail as the mobile application to be tested. We have based our tests on the localized version (in Italian) of the application. The app, whose original release dates back to 2009, has recently reached its 5.010 release [8]. The open-source nature of the application is fundamental since instruments like Espresso and UI Automator have a strong connection to the application code (for instance, for the retrieval of identifiers used to pick out elements of the user interface). The application was selected after a search on the GitHub platform for an open-source application with the suitable characteristics, i.e. a long enough life and a significant code base. K-9 Mail, quite a large software project with more than 120 thousand lines of Java code, is a stable and widespread application with a long release history (6306 commits and 336 releases until today), and presents an established graphic interface which is not significantly subject to changes among subsequent releases.

K-9 Mail is an e-mail client supporting multiple accounts at the same time, relieving the user of the trouble of managing different mailboxes. Typical e-mail client functionalities are hence provided for each account. In addition to them, the application allows to save the account data (i.e., the e-mails in the various folder) and preferences, and to retrieve them when the application is used on a different device (those features, however, are not available in earlier releases).

As we detail later, a test suite of ten tests has been written by one of the author of this paper and tested on the latest stable version of the application. All the main functionalities offered by the application have been considered by the test cases. The tests have also been applied, when possible, to

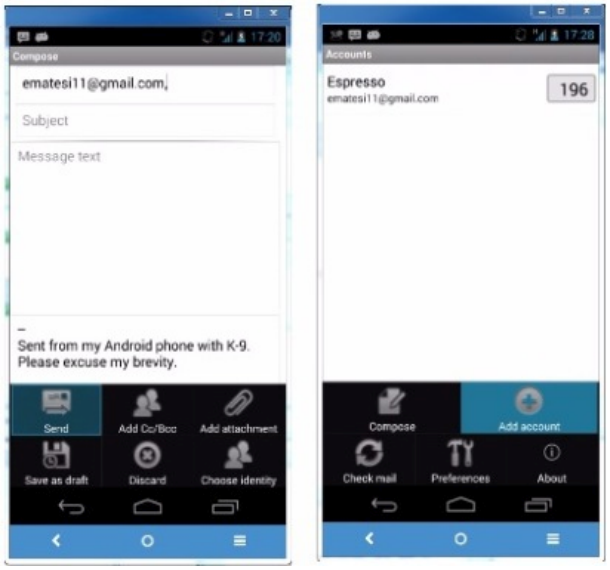

Figure 5: K-9 Mail v2.995 User Interface.

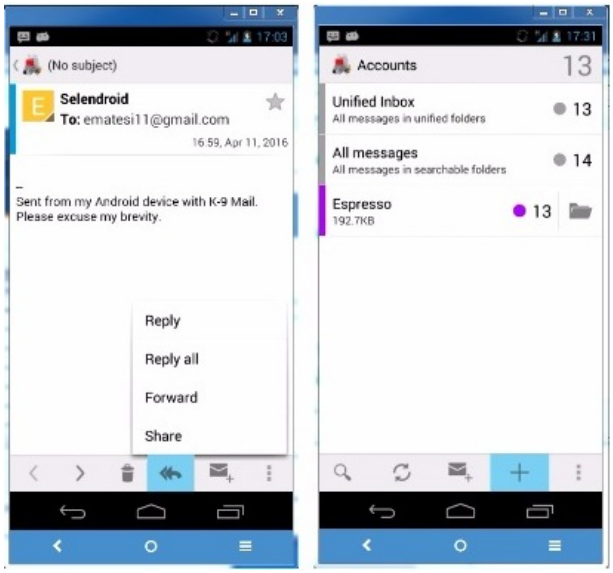

Figure 6: K-9 Mail v5.010 User Interface.

five previous versions of the application. We have selected the last stable versions belonging to three major releases: v2.995, released in April 2010; v 3.993, released in December 2011; v4.804, released in June 2014. In addition to them, we have chosen randomly other two versions of the application: v2.102, released in November 2009; v3.309, released in November 2010.

It can be seen that, in the first four releases considered, several features are delegated to a Menu reachable by pressing a physical button on the device (see figure 5). These operations are therefore no longer executable with the majority of current devices, since Android 3.0 Honeycomb interrupted the use of physical buttons and started to rely on *Action Bars* instead. With version v4.804, the UI starts to look similar to the one of the latest versions (see figure 6), with the addition of two bars at the top and at the bottom of the screen, and the dismissal of the use of physical buttons. The versions of the app also differ in how the views can be identified (in the older ones, unique identifiers are not provided for the elements of the visual hierarchy).

3.2 Procedure

All the main features of the app, discussed in the previous section, have been exercised in ten different test cases that are listed below. Since the application requires an authentication phase, tests are intended to run sequentially, so just one authentication has to be performed.

The application starts from a known state, then the tests produce a series of operations in some cases dependent from each other.

The tests are:

- Successful authentication;
- Send a message;
- Reply to a message;
- Delete a message;
- Add user account;
- Delete user account;
- Delete account data;
- Restore account data;
- Export account settings;
- Import account settings.

Each test case has been implemented in an automated test script using all the five testing frameworks under consideration. For the latest version of the application, we implemented 10 tests for each framework, thus obtaining a total of 50 distinct test methods. We adapted them to different versions of the app when needed.

Since earlier versions of the application do not offer all the listed functionalities, only the available tests have been performed on them. In particular only seven cases were applicable to version 2.102, and eight test cases to versions 2.995, 3.309 and 3.993.

Although Silk Mobile allows the export of the generated test cases for the use with several frameworks (e.g., JUnit) we did not take advantage of this facility and we just used the proprietary IDE of the application. We had not written the complete test suite with Silk Mobile, due to the absence of the possibility of simulating physical buttons of the device. The same limitation applies also to Sikuli, but we managed to get around it by using Mobizen [15], a tool for controlling an Android smartphone (simulating even the physical buttons) from a desktop PC.

4. RESULTS

In this section, we discuss how the tests have been written for the individual versions of the application, and the fragilities we have found for each one of them. We highlight which test cases applied to subsequent versions and, on the converse, what changes have had to be made in order to adapt them.

We have limited our discussion to the tools with which we were able to write the full set of tests, thus excluding Silk

Mobile. However, we noticed that the weakness exhibited by tests written in Silk Mobile are similar to those of Espresso, UIAutomator and Selendroid.

4.1 K-9 Mail v2.102

Seven tests out of ten can be written for this version of the app, according to the functionalities it provides.

For the *Successful authentication* test, unique identifiers can be used for the detection of the specific buttons and textboxes. For the test case *Send a message*, it is necessary to pass through a menu that can be opened pressing the physical Menu button of the device. Buttons that have to be pressed have no unique identifier, and are detected through their textual description. Text boxes used to compose the message are instead detected using the unique identifier they are given in the application layout.

The *Reply to a message* test case opens an incoming message from the inbox. A click in a specific portion of the screen is done in order to obtain the first message in the inbox folder. The reply functionality is retrieved by its unique identifier. The same is done for the *Delete a message* test case, that leverages the unique identifier given to the delete button. Also *Add user account* needs the pressing of the physical menu button, and the access to a menu in which elements have no unique identifier. *Delete user account*, *Delete Account Data* require the access to a contextual menu with a long click on an account name. Unique identifiers are not present even in these cases.

As it can be seen, in this release various actions are available through the physical Menu button of the device. These actions are not distinguished by an identifier. This defect makes tests weaker because it forces the developer to find elements through textual description. This kind of description is not so safe, because some test instruments cannot access the textual language resources of the application. Therefore, the text to be searched cannot be dynamically linked to the text actually appearing in the elements of the user interface, according to the device language. Moreover, textual descriptions of the elements can be slightly different according to screen sizes and between subsequent releases of the application.

In general, the need for a device button instead of a GUI button represents a weakness point for testing instruments which do not support the automation of this kind of inputs.

We have considered, as the starting point of our tests, the initial activity shown once the authentication has been performed. If this starting activity is changed in future releases, tests cannot pass anymore.

4.2 K-9 Mail v2.995

Eight tests out of ten can be written for this version of the app.

Authentication, *Send a message*, *Reply to a message*, *Delete a message* test cases are completely compatible with the ones written for release v2.102.

The functionalities *Delete account data* and *Delete user account* have been moved to an advanced options menu, hence the related tests had to be rewritten.

The feature *Restore account data*, that allows the restoration of an user's messages, is available in this version of the application and therefore has been tested as well. It is also placed in the advanced options menu, and -since the button to perform it has no unique identifier- its textual description has to be used to test it.

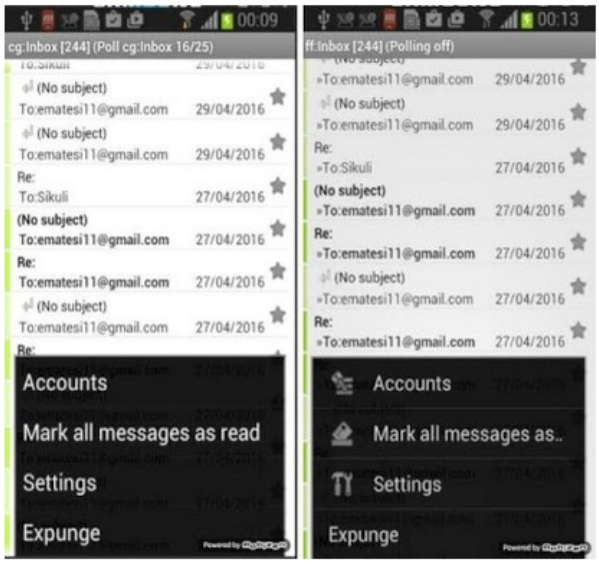

Figure 7: User Interface differences between v2.995 and v3.309.

The textual description of the feature *Add user account* has slightly different colour and dimensions with respect to the previous version. Even though the difference is little, the related test case in Sikuli has required the re-creation of the screenshot used for the button identification. The test did not show any fragility with Espresso, UIAutomator and Selendroid.

4.3 K-9 Mail v3.309

Eight tests out of ten can be written for this version of the app.

All tests written for v2.995 are compatible with this version, except the *Add user account* realized with Sikuli: once again, a slight difference in the appearance of the commands creates the need for new captures (see figure 7).

4.4 K-9 Mail v3.993

Eight tests out of ten can be written for this version of the app.

Authentication, *Send a message*, *Reply to a message*, *Delete a message*, *Add user account*, *Delete user account* tests are completely compatible with the ones written for release v3.309.

The buttons to perform the operations of the tests *Delete account data* and *Restore account data* have had their text description changed: these descriptions are mantained in English in v3.309, even though the application is localized to Italian; in v3.993 such inaccuracy has been fixed. Therefore, the previous tests are invalidated since the identification phase for the involved views must be updated. Obviously it is so also for Sikuli: new captures have to be taken to enclose the new text descriptions.

The problems related to the physical button are still present in this version. In many menus of the application there are still no unique identifiers for the options.

4.5 K-9 Mail v4.804

All the ten tests can be written for this version of the application. Most tests needed to be rewritten -the entire test suite, with Sikuli- since the organization of menu changed significantly with respect to the previous considered version.

Authentication and *Delete a message* tests, done with Espresso, UIAutomator and Selendroid, are the only one completely compatible with the previous release.

For the test *Send a message*, the commands to compose a new message and to send it are moved with respect to the previously analyzed version. In fact, they are available in two toolbars on the lower and on the upper border of the application, respectively. The toolbar has unique identifiers for each of its commands, thus making the writing of test easier. The commands to perform the test case *Add user account* are also located in the new toolbar. To perform the operation *Reply to a message*, the view which shows a received e-mail is different too. In fact, a new pop-up menu with various replying and forwarding options is provided. Once again, fragilities due to the absence of unique identifiers are present in some menus. Since the options menu varies with respect to the previous version that has been tested and there is no longer an "advanced options" menu, the test cases *Delete account data*, *Restore account data* and *Delete user account* had to be updated too.

Two new test cases have been written for the new functionalities *Export account settings* and *Import account settings*.

4.6 K-9 Mail v5.010

For this version, all functional tests analyzed for release 4.804 are still valid, as all the graphics of the user interface are the same, and the activity flow that starts once the authentication has been performed remains the same as well.

Identifiers and textual descriptions are the same of the previous versions of the application. Once again, some menus lack unique identifiers for all of their options.

5. DISCUSSION

Based on the test cases explained in the previous paragraphs, we now discuss the process of creating test suites with each of the tools under study. Starting from the fragilities found when adapting the test cases to different versions of the same application, we give some indications about practices leading to fragilities, that may be useful for the creation of guidelines for programmers aimed at avoiding such problems for the testing phase.

5.1 Evaluation of Testing Tools (RQ1)

Upsides and downsides of the five considered testing techniques are explained hereafter.

5.1.1 Espresso

Espresso is integrated in the Android development environment, so it can use direct references to the GUI elements and to the textual resources of the application.

A major advantage of Espresso is the possibility of automatically recognizing the loading times of the application: thanks to this feature, it is possible to reduce slightly the number of instructions to write for each test case, since waiting instructions can be omitted. Moreover, few operations are needed, in general, to pick specific elements of the user interface. The framework offers a rather simple instrument (the *onView()* method) that receives a single identification method as parameter. The execution of user inputs on the views is simply described as well. In general, Espresso has a well organized API, easier to use with respect to the other testing tools considered.

The framework offers a specific object for the definition of the entry point of the test (it may be unclear, since an Android application may have multiple ones). Methods are available for the simulation of physical buttons, hence testing is made possible also for rather old applications.

However, the possibility to go to arbitrary views of the application may be considered a drawback of the framework, since it makes the execution flow differ from the typical human usage of the application and thus limits the validity of tests.

5.1.2 UIAutomator

The test code is integrated in the Android development environment and therefore it can use references to the application resources. Writing tests using this tool is rather simple. The object representing the Android device offers also methods for the simulation of the pressure of physical buttons. References to the textual resources of the application are no longer direct like in Espresso, but are instead obtained through a *Context* object, opportunely initialized for the application under test.

The most important feature provided by UI Automator is the possibility to access system applications (like Settings or Contacts) and perform operations on them.

UIAutomator does not wait automatically for the loading times of the application, so *sleep()* and *wait()* functions are needed. Another downside of the tool is the limited support for older versions of Android: in fact, UI Automator works only on versions of the API higher or equal to Android 4.3 (API 18) [16].

5.1.3 Selendroid

As an evolution of Selenium, Selendroid takes advantage of an already established framework originally thought for web application testing. The functionalities offered are slightly more comprehensive with respect to Espresso and UIAutomator.

Selendroid does not need any modification of the application under test, since the only requirement is to have its binary file. An advantage of this testing tool lies in the fact that the data produced by the application is deleted after each test session. This allows to have the application immediately ready for a new test session.

After a set-up procedure, that is more complex to manage with respect to the other techniques, the amount of methods necessary to perform operations is quite small.

The web interface provided by the framework allows a really easy retrieval of information and identifiers about each element of the visual hierarchy, by just clicking on it.

The tool allows an easy management of physical buttons (it is sufficient to import the *AndroidKeys library*), and elements of ListViews.

Selendroid does not allow the performance testing of the devices running the applications, like other testing tools can do.

5.1.4 Silk Mobile

The principal strength of Silk Mobile is its rapidity in the generation of commands to be executed in a test. The

Table 1: Test suite implementation on various versions of K-9 Mail, with Espresso, UIAutomator and Selendroid.

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

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

Table 2: Tests compatible with previous versions, with Espresso, UIAutomator and Selendroid.

	v2.995	v3.309	v3.993	v4.804	v5.010
Number of unmodified tests still working	5/7	8/8	6/8	2/8	10/10
Percentage of unmodified tests still working	71%	100%	75%	25%	100%

Table 3: Causes of fragilities in broken test cases.

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

commands given in input by an user can be easily registered and repeated, and the test cases can be furtherly enriched by inserting additional operations. The tool allows to perform a complete monitoring of the device performance (e.g., memory and CPU usage, frame latency).

Silk Mobile does not automatically wait for the loading time of the applications.

Since the development environment for the test is not integrated with the one used to write the code of the application, it is not possible to access the textual resources for the language used by the application.

Silk Mobile does not allow the automatization of physical buttons of the device (like Menu and Back), hence it may be impossible to test old applications using this tool.

Without exporting the suite to a test framework, the Silk Mobile IDE exhibits a few limitations: it is not possible to launch a series of test cases but the programmer has to execute them one at a time; it is not possible to manage exceptions, like it can be done with tests written in Java, therefore it is not possible to handle undesired behaviors generated by the components of the interface; finally, it is not possible to generate routines that can be used in different test cases to perform recurrent operations.

5.1.5 Sikuli

An important advantage of Sikuli is the possibility of operating on elements of the GUI based on Flash, since they have no identifiers and are therefore hardly spottable by other testing tools.

Sikuli is not affected by fragilities caused by changing identifiers, since it does not use them to identify elements of the application. However, it is really vulnerable to text change and graphics change.

The correspondence between the reference image specified by the programmer, and the actual element of the interface, may be not precise for matters of resolutions and color tuning of the device. In these cases, the recognition procedure may fail.

The tests may take a big amount of memory if the number of images used is high. Naturally, in order to make the test succeed, the absence of multiple correspondence to the same screenshot is needed; otherwise, a wrong element may be selected, or the test may end up in failure.

Sikuli does not allow performance testing of the running applications.

5.2 Changes in Test Suite (RQ2)

In table 1 we show the compatibility of individual test cases between subsequent versions of K-9 Mail, with three testing tools that showed the same fragilities and hence had the same test cases broken (Espresso, UIAutomator and Selendroid). We do the same for Sikuli in table 4.

In table 2 we show the number (and the percentage with respect to the whole test suite) of tests that, for each version

Table 4: Test suite implementation on various versions of K-9 Mail, with Sikuli.

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

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

Table 5: Tests compatible with previous versions, with Sikuli.

	v2.995	v3.309	v3.993	v4.804	v5.010
Number of unmodified tests still working	4/7	7/8	6/8	0/8	10/10
Percentage of unmodified tests still working	57%	87%	75%	0%	100%

of the application, could be maintained as they were written for the previous version (with Espresso, UIAutomator and Selendroid). We do the same for Sikuli in table 5.

As it is expected, in correspondence with tangible interface modifications, as it happens between the third and fourth major releases, the majority of test cases has to be rewritten.

On the other hand, in correspondence with releases that have mainly corrected bugs and added support for additional languages and protocols, no test case was broken.

5.3 Fragilities Found (RQ3)

On the basis of the analysis of test fragilities induced by the evolution, as discussed in § 4, we classified the causes of the fragilities that were observed in our study.

- *Identifier change*: a test that detects elements by their identifier is invalidated if one of these attributes is changed.
- *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 description. This strategy is not safe for tests, because the textual attribute depends from the device language, so the tests must use the language corresponding to the device settings.

  Moreover, the textual description of an element is more likely to be changed in future releases than identifiers. In this case, obviously, test cases based on textual recognition are invalidated.

  It is worth highlighting that image recognition testing tools -like Sikuli-, which cannot rely on identifiers to discrimine between the elements of the user interface, are particularly subject to this kind of vulnerability.
- *Deletion or relocation*: between different releases of the same app, it may occur that an element is removed or moved to another activity. Consequently, a test which has to use it is invalidated.
- *Physical buttons*: older Android applications made use of physical buttons, deprecated since Android 4.0 and replaced with the use of Action Bars. The corresponding features are no longer testable with specific testing tools that are unable to simulate the input given through such buttons.
- *Graphics change*: as stated before, image recognition testing techniques discriminate the elements of the GUI by their graphic appearance. A new arrangement of the elements in the graphical layout -or even just a simple modification in the style of the application- invalidates all tests written with such tools.

Table 3 summarizes the main causes of the fragilities we have found for our Espresso, UIAutomator and Selendroid test suites. The same test case can be weakened by multiple causes.

The test writing process has also made us notice two issues that, albeit not happening in our test suite, may harm test cases in general.

- *Activity flow change*: operations performed in tests are designed to be applied starting from specific views. If the flow between the activities is changed inside the application, tests are invalidated consequently.
- *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. Moreover, if the tests are performed on real handheld devices, other applications running concurrently may cause additional delays to the loading time.

In general, if testing instruments are not able to wait for loading times implicitly, explicit pause commands have to be used, to wait for the availability of the user interface elements. The duration of such pauses can be estimated empirically, performing repeated executions, or a safe upper bound can be set. Tests are invalidated when network and system resources are not enough and latency overtakes pauses in the test, and so the requested elements are still not available when the pause inside the test runs out.

6. CONCLUSIONS

The goal of our work was to conduct a preliminary assessment of the test fragility phenomenon in the mobile application context.

We analyzed five distinct mobile UI testing frameworks, each showing distinctive strengths and weaknesses.

The analysis of the test suite changes induced by the application evolution showed that, depending on the specific modification applied in each release, up to 75% of the tests for code-based testing tools, and up to 100% for image recognition testing tools, had to be adapted. This fact supports the idea that test fragility is a relevant problem also in the context of mobile UI testing.

The main objective of our exploration was to assess the importance of the problem of test fragility in a mobile environment and identify the relative main causes and characteristics. The classification resulting from our observations, although preliminary, can be the basis for a more extensive taxonomy of the causes of fragility.

As future work we plan to apply our analysis to other large applications, in particular focusing on large test suites and long-lived projects. We also plan to run the same test suites on different devices and OS versions, and to study the types of fragility exposed by hybrid applications compared to native mobile ones. The goal is to extend our taxonomy of causes of fragility. A sound taxonomy is a fundamental asset to enable automatic detection of test fragilities in existing testing code.

Moreover we aim at providing guidelines for test definition to avoid or at least minimize the fragility of the UI tests for Android applications.

7. ACKNOWLEDGMENTS

This work was supported by a fellowship from TIM.

8. REFERENCES

[1] 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*, pages 252–261, 2011.

[2] D. Amalfitano, A. R. Fasolino, and P. Tramontana. Mobiguitar: Automated model-based testing of mobile apps. *Software, IEEE*, pages 53–59, 2015.

[3] Android Developer's Guide. Testing display performance. http://developer.android.com/training/testing/performance.html/.

[4] Android Developer's Guide. Testing support library. http://developer.android.com/tools/testing-support-library/index.html/.

[5] S. Berner, R. Weber, and R. K. Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 571–579, New York, NY, USA, 2005. ACM.

[6] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? In *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 429–440, 2015.

[7] J. Gao, X. Bai, W.-T. Tsai, and T. Uehara. Mobile application testing: A tutorial. *IEEE Computer*, 47(2):46–55, 2014.

[8] GitHub. K-9 mail - advanced email for android. https://github.com/k9mail/k-9/.

[9] Google Code Archive. androidscreencast. https://code.google.com/archive/p/androidscreencast/.

[10] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 77–83, 2011.

[11] T. W. Knych and A. Baliga. Android application development and testability. In *MOBILESoft' 14*, pages 37–40, 2014.

[12] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Proc. of 20th Working Conference on Reverse Engineering (WCRE)*, 2013.

[13] M. Leotta et al. Visual vs. dom-based web locators: An empirical study. *Web Engineering. Springer International Publishing*, pages 322–340, 2014.

[14] A. M. Memon. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):4:1–4:36, Nov. 2008.

[15] Mobizen Homepage. http://www.mobizen.com/.

[16] S. Gunasekaran and V. Bargavi. Survey on automation testing tools for mobile applications. *International journal of Advanced Engineering Research and Science (IJIAERS)*, 2(11), 2015.

[17] N. Saas, A. A. Bakar Husna, and N. Sham. Automated testing tools for mobile applications. In *Information and Communication Technology for The Muslim World (ICT4M), 2014 The 5th International Conference on.*, 2014.

[18] Selendroid Homepage. Quickstart. http://selendroid.io/quickStart.html/.

[19] Vysor Homepage. http://www.vysor.io/.

[20] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction.* Kluwer Academic Publishers, 2000.

[21] T. Yeh, C. Tsung-Hsiang, and R. C. Miller. Sikuli: using gui screenshots for search and automation. In *Proc. 22nd annual ACM symposium on User interface software and technology.*, 2009.