Politecnico di Torino

Master's Degree Course in Computer Engineering

Master's Degree Thesis

Implementation of an automated repairing mechanism for GUI test scripts for mobile applications

Supervisor:
Prof. Luca Ardito

Candidate:
Marilyn Fulgione

Co-supervisor:
Prof. Maurizio Morisio

Academic year 2019/20
Torino

Abstract

***Context*:** In the field of Mobile Application Development, testing practices can be very expensive and time-consuming - especially in an industrial setting. One of the main causes of this phenomenon is the constant evolution of the GUIs aspect, leading to frequent breakages of layout-based test cases.

***Goal*:** The aim of this thesis' work is building a tool that repairs layout-based test cases that break in the transition from an application release to the immediately following one. In particular, the tool tries to find in the newer release the widget that is most similar to the one the test located in the previous release, and it consequently replaces the related method calls.

***Method*:** The tool works on Android applications developed in Java and it focuses on repairing Espresso test cases containing calls to some methods of the *ViewMatchers* class. The most similar widget in the newer release has been selected as the one with the highest matching rate with the one located in the previous version of the app. This value was computed considering the most characterizing attributes of the View class and some of its subclasses, and assigning a weight to each of them based on how much that attribute is prone to changes in the evolution of Android applications layouts.

If the matching rate is equal or above 65%, the widget is considered equivalent to the one located in the previous version and the user is asked whether they want to proceed with the test class refactoring using that widget attribute values.

***Results*:** We tested our tool on three different applications and a total of 30 test cases, including 40 different method calls. The tool succeeded in repairing 80% of them with the truly equivalent widget of the newer version of the app.

***Conclusions*:** The tool proved to be successful to fulfill the goal it was developed for. It can also be easily extended to Kotlin based applications and other test tools, to enlarge its field of applicability. Few changes could be made with regards to the precision of the computation of the matching rate between widgets – mainly considering a larger set of widget attributes.

Contents

List of Figures

List of Tables

Chapter 1

Background and related works

1.1 Introduction to mobile testing

The testing activity is a fundamental phase in software development, especially to guarantee its quality and to ensure that no crashes and no undesired behaviors happen during a typical execution (Coppola et al., 2017) since that would lead to poor user experience and would cause the users to feel dissatisfied and immediately uninstall the application.

The modern development practices and the continuous integration of the software in the existing systems brought to the need to have automatic test execution and therefore to write specific scripts for this purpose.

A test script is a sequence of actions, each of them characterized by a widget locator and an event trigger: the widget locator is used to retrieve a widget (element) inside an activity that satisfies the specified criteria, while the event trigger can be either the interaction of the user with the widget itself or come from external input. Each event generally involves two screens: the one currently shown and the target screen, that becomes active in response to the event (Chang et al., 2018).

As mobile applications are mainly focused on giving visual feedback to the user possibly after every interaction with them, GUI testing becomes a fundamental step in mobile application development, especially since mobile applications GUIs change so rapidly release after release, making them prone to GUI testing fragility.

Writing test scripts is effective both in the exploitation of time and resources but also requires to overcome some challenges for the maintenance of said scripts: the evolution of the System Under Test (SUT) brings to *test breakage* caused by the modifications done on the system. For this reason, we need the test suite to evolve together with the system, or at least that the developer can identify the reason that brought to test breakage - i.e. if that is due to system evolution or just to a bug in the code's implementation or algorithm logic.

The changes that can lead to test breakage can be logical or structural: logical changes deal with modifications in the logic of the code or in functionalities that can be added, removed or modified, while structural changes deal with the layout and structure of the application. (Imtiaz et al., 2019)

We can also consider refactoring as a source of possible test breakage, but we'll focus on it further on.

Discarding broken test cases highly affects the quality of the regression test suite and therefore it can be an expensive task, since we then need to work a lot on realigning previous tests script with the application GUI and possibly new functionalities, to avoid a drastic drop of the test suite quality - especially considering that, in terms of lines of code, GUI testing can be an important portion of the project during its lifecycle, if compared to the number of LOCs of program code.

1.1.1 Approaches for automated GUI testing of Android applications

As far as automated testing is concerned, Linares-Vásquez (2015) lists four different approaches for testing the GUIs of Android apps:

- *Random / fuzzy testing:* first we create a model of the user interface and then we apply random testers (e.g. Monkey)[1] to distribute the input given to the interface more cleverly.
- *Model-based testing:* this type of testing starts with the modeling of the GUI

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

as a finite state machine or an event-flow graph. Test cases are generated by navigating the GUI model.

- *Capture and replay testing:* this approach consists of recording sequences of actions performed on the GUI, that automatically generated the corresponding repeatable test code sequences.

- *Scripted and white box testing:* this kind of testing requires a deep knowledge of the code and some effort for writing down testing code sequences of operations to be performed on the Application Under Test (AUT).

1.1.2 Categories of mobile applications

Regarding mobile applications, we can distinguish between three different categories: native, web-based and hybrid applications.

- **Native apps** are developed using specific SDKs for the destination platform (Coppola et al., 2019), have easy access to the device hardware resources and support all user interfaces and interactions that the operating environment has available.

  Java is generally used to develop native applications for the Android market, while Objective C is used for iOS and the .NET framework for Windows Phone.

- **Web-based apps** - also called mobile web applications - are typically developed with HTML-like languages, CSS and JavaScript, and are engineered to be loaded in the browsers of mobile devices. They try to mimic the behavior of a native app but in reality, they execute their code in a web browser on the host platform. According to Jobe (2013), mobile web apps can be further categorized into:

  1. Hardware intensive apps, mainly used for content creation, and
  2. Apps that don't need access to hardware resources, mainly for content consumption

- **Hybrid apps** exploit some native components to embed content that is created through web development technologies.

The study by Jobe highlighted that performances of native apps are generally superior to mobile web apps, especially when access to hardware resources is required (e.g. GPS location).

On the other hand, mobile web applications need less time and investment to be developed and are also significantly less complex. The cited study also underlined the idea that hybrid apps could be a valuable alternative to achieve good performances and functionalities reducing the cost of development.

1.2 The challenges of mobile applications testing

Developing mobile applications is already a pretty challenging task by itself, for the following reasons:

- mobile applications are event-driven, so they get their input not only from the user but also from external contexts (e.g. position sensors, NFC devices, screen orientation changes, etc.);
- limited energy, memory and bandwidth of the host device;
- constant interruptions caused by system and communication events (notification, incoming messages or calls, etc.);
- the need for the interface to adapt to different screen sizes and resolutions;
- very high multitasking and interaction with other apps.

When it comes to testing mobile applications, other challenges arise, making developers prefer manual testing approaches instead of automated ones. Let's see them more in details, starting from the list made by Linares-Vásquez et al. (2017):

- **Fragmentation.** This phenomenon is mainly due to the presence of many different configurations obtained taking into account several devices, versions and operating systems. The high number of configurations makes it very difficult to test the application in each of them - also for the high cost that it would require.

  A possible solution to this challenge could be moving the testing phase on cloud/crowd-based services that provide developers with the ability to test applications on a huge number of virtual and physical devices, or with a multitude of users with different kinds of devices. Anyway, this kind of service

is generally expensive and requires more time than it would fit in Agile and DevOps practices.

- **Test flakiness.** Testing applications that rely also on back-end servers or services add up other challenges, related to lack of connection, availability of the AUT's back-end, timeout, delays, memory usage and data integrity. All of that brings to non-deterministic app behaviors and consequently to possible test failure.

  One of the possible ways to face this challenge is using the testing tool Espresso[2]: it solves the problems with delays, only executing the tests when the GUI is in idle state, reducing the risk of test failures.

- **Lack of history awareness in test cases.** Some attempts have been made to implement execution history awareness within test cases, for example with *event-flow graphs* (EFG) or finite state machines - so handling the event flow as if it were a language - in which the events correspond to the transitions among the different states.

  EFGs don't have a specific memory mechanism but navigating the graph we can derive a history aware execution. Language-based models, instead, explicitly implement memory using a conditional probability distribution, that is focused on generating the following event in the sequence based on previous ones.

  Furthermore, the possibility to generate history aware tests is negatively affected by the problem of test flakiness we mention before.

- **Difficulties in evolving and maintaining GUI scripts/models.** This problem comes from the fact that many test scripts are closely tied to widgets positions within the GUI layout - and that these positions can vary a lot also based on the device characteristics - and to the actions that can be performed on said widgets.

  The model should evolve together with the application and updated test cases should be continuously generated starting from the updated model: by the

[2] https://developer.android.com/training/testing/espresso/

way, this approach wastes the potentially useful information and knowledge coming from previously generated models.

- **Absence of mobile-specific testing oracles.** Several options have been explored for mobile application oracles implementation. Some of those are based on the status of the GUI or on the raising of exceptions and errors to establish if a test has failed or not. The absence of specific oracles is still an open issue: approaches that rely on exceptions raised by the app are useful to detect crashes and unexpected errors but cannot identify errors occurring on the GUI and vice versa.

  For this reason, oracles are still implemented and updated manually, meaning they still are a very expensive voice in the testing activities.

- **Missing support for multi-goal automated testing.** The most part of testing activities and the currently used approaches are focused on *destructive testing*, so aimed at eliciting failures. Other types of important testing categories - such as functional, regression, performance, energy and security testing - lack of a stable support tool.

Other challenging aspects of mobile application testing include (Joorabchi et al., 2013):

- limited unit testing support for mobile-specific features (e.g. sensors, rotation, navigation, etc.);
- missing tools to monitor, measure and visualize application metrics such as memory management, battery consumption, CPU usage and network performances;
- emulators and simulators lack appropriate support to mimic real environments, as well as access to features outside the application which could be part of a test case;
- usability testing: usability can make or break the success of an app - as the most common type of feedback on usability is the application download and rates on the app store - so there is a significant need for tools allowing this kind of testing.

1.2.1 The cost of application testing

Testing, seen as test scripts development and their maintenance is - as previously stated - a very expensive activity.

For applications having a big number of screens and possibly thousands of total widgets and visual elements, test automation has a fundamental role in cost reduction for testing itself. At the same time though, putting more effort in test script writing is key when tests need to be run repeatedly to determine if the application followed the correct or expected behavior.

To have an idea about the costs we are talking about, just think that only in Accenture the cost due to manual maintenance and test script evolution varies from 50 to 120 million US dollars. About the tools, we can, for example, talk about *Quick Test Pro (QTP)*, a very widely used tool which license has a price of over ten thousand US dollars.

For this reason, and to investigate the real need to buy and use tools to support the testing activities, Grechanik et al. (2009) have conducted a study observing a group of 45 employees with different studies background and also different experiences with the world of application testing.

They observed that test engineers obtained almost the same level of productivity both using QTP and a manual approach. Furthermore, QTP has shown to be very useful in supporting test scripts writing and repairing for those that didn't have prior knowledge on testing, while the experienced employees still obtained better performances with the manual approach.

The authors then recommend organizations to supply their programmers with testing tools to make test repairing faster but not to provide testers with expensive tool licenses since they still are more efficient if they fix things manually.

Maintenance cost is mainly affected by design choices and the strategies used to implement the tests themselves, missing documentation and the too few guidelines available to create tests that can be easily maintainable and reusable (Berner et al., 2005). Alégroth et al. (2016) also demonstrated with a case study that in company environments where there is a considerable need for test scripts maintenance, the use of automated testing brings to a *Return Of Investment (ROI)*, even if it's on the

long run. In that case study, developed in Saab, there would be a ROI in a range from 180 (best case) to 532 weeks (worst case) since the adoption of test repairing automation mechanisms.

The value of the ROI and even the chance to get to it are strongly influenced by the weight the company gives to the *Verification and Validation (V&V)* phase of the development process.

1.3 Test breakage avoidance and prevention

For what concerns widget location inside activities, some tips can be helpful to prevent test case breakage such as:

- Having elements IDs that are descriptive and not automatically generated, so that it's less likely that they are frequently changing with the application evolution;
- Using resource names to define widget properties like text, color, margins and others, since modifying files in the values folder won't affect the test cases.

1.4 Goals

The aim of this thesis work is to provide an automated solution to repair test scripts that are broken due to refactoring or widget properties modifications.

In particular, an example of use case can be described as following: a test case involves a widget on the screen that is located using its text; in the following release, the test case breaks because the text contained in the widget is changed, so the locator doesn't return any valid widget. We want to provide a way to select the correct widget on which we need to perform the action - i.e. the widget that in the current release corresponding screen is most similar to the one from the previous release.

1.4.1 Notes on refactoring and fragility of GUIs

Daniel et al. (2011) states that *"GUI refactoring only changes the aspects of a GUI* (and correspondingly the view and the controller parts of the code that implement the GUI) *but not the way it behaves, so not the underlying model"*.

There are two possibilities when it comes to refactoring: either it has no effects on the interface - because it doesn't change the output of GUI related methods, or some methods or parts of them are simply moved - or it does. In the latter case, refactoring alters the state of relevant elements, causing the test not being able to find them when needed.

The main ways in which we can identify a view inside an activity - and therefore its properties that are most likely to cause test breakage if changed - are its ID and text. Also choosing a different property to identify the views may cause the same problem in layout-based test scripts.

Visual-based test scripts, based on graphic recognition, can also fail due to the modification of colors, animations, transparency, sizes and position of widgets. Coppola et al. (2017)

Table 1.1 shows the main types of frailties that can affect different testing approaches, where the 'x' means that an approach is affected by a given fragility issue (Coppola et al., 2019).

Type of fragility	**Layout-based**	**Visual-based**
Text changes	x	x
Graphic changes		x
Widget substitution	x	
Application behavior change		x
Widget addition / removal	x	x

Table 1.1. GUI frailties and how they affect different testing approaches

It's very clear at this point that these kinds of changes can affect the written test cases, especially if visual-based; so the cited study by Daniel et al. came up with the idea of a tool that could automate the repair of GUI test scripts, refactoring them based on the changes done on the GUI. This is based on the assumption that the refactoring tool knows how to map each GUI method to the corresponding one after the code has been refactored.

Table 1.2 shows the refactoring categories affecting the interface according to Gao et al. (2015).

Refactoring category	Subcategory	Affects interface?
Adding element	Adding parameter	yes
	Adding verification condition	no
	Adding method	yes
Removing element	Removing parameter	yes
	Removing method	yes
Adjusting element	Reorganizing method	no
	Simplifying code	no
	Moving element	yes
	Changing access purview	yes
	Changing access path	yes
	Replacing element	yes
	Disassembling element	yes
	Renaming element	yes
	Combining elements	yes

Table 1.2. Refactoring categories affecting the application interface

1.5 State of the art and existing tools

The study by Imtiaz et al. (2019) highlights the main trends in the field of studying test scripts repairing automation. Table 1.3 presents the results of the 41 papers selected for that study. Please note that one paper can be inserted in one or more categories for some of the selected *research questions*.

In the following, we give an overview on the existing tools for test repairing for web, desktop and mobile applications.

1.5.1 SAFIRA

When performing changes on an application GUI, it can be really difficult to understand if said changes are going to affect the application's behavior, and therefore if and how these modifications will cause tests to break. The challenge is even harder considering Java not so trivial semantics, especially with regards to inheritance.

While compilation errors introduced by the refactorings are easily detected in IDEs, behavioral changes may pass unnoticed. That's why Mongiovi (2011) proposes

Research question	Category	Papers
Type of research method	Solution proposal	12
	Validation research	23
	Evaluation research	6
Test framework	ROBOT	2
	JUnit	14
	QTP	2
	Selenium	6
	Others	17
Target platform	Desktop	29
	Web	8
	Mobile	2
	Other	2
Automation level	Manual	10
	Semi-automated	19
	Automated	12
Type of approach for test repair	Model-based	15
	Search-based (metaheuristic)	5
	Heuristic-based	8
	Computer vision-based	1
	Symbolic and concolic-based	14

Table 1.3. Data from previous studies on test script repair automation

SAFIRA as a practical approach to automatically generate test cases focused only on the entities that are affected by the changes.

More in detail, SAFIRA generates a series of methods to exercise directly or indirectly the impacted entities (e.g. widgets). The set of all impacted entities is defined as the union of the sets impacted by each change. If the generated test cases reveal a result that's different from the expected one (namely, the behavior before the changes), SAFIRA reports the transformation as not behavior preserving.

In this way, SAFIRA builds a set of tests that pass successfully on the base version of the app (V1) but fail on the target version (V2). If this set is empty, the developer is more confident about the fact that the transformation is behavior preserving; otherwise, the test cases will show behavior changes.

SAFIRA vs SAFEREFACTOR

In the cited study, Mongiovi compares SAFIRA's results with those found with another similar tool called SAFEREFACTOR.

They discovered that SAFEREFACTOR, developed by Soares et al. (2009) requires a greater time limit to detect the behavioral changes than SAFIRA, and it also takes more time to analyze the transformation because it generates a larger test suite: SAFEREFACTOR generates many unnecessary test cases while SAFIRA focuses on test cases for the impacted entities only.

1.5.2 ReAssert

ReAssert has been presented by Daniel et al. (2009, 2010) as a tool that automatically generates repair suggestions for test cases in Java, allowing the user to accept them, and therefore immediately repair the test case, with just a single mouse click, reducing the effort required to fix broken test cases. It works with both manually written and automatically generated test cases.

The main difference between ReAssert and refactoring tools is that the first one requires to analyze also runtime execution and it suggests repairs that can change the behavior of the test code making the failing test cases pass, while refactoring tools preserve code behavior making structural code changes.

To determine the possible repairs, ReAssert records the variables' actual values during test execution and replaces the wrong expected ones. The steps followed by the tool are shown in Figure 1.1 - taken from Daniel et al. (2009).

This kind of approach works quite well with assertions involving simple-typed variables - such as Integer and String - but it can get a lot slower when using *assertEquals* on objects of complex nested types/classes.

The main limitations of this tool are of course the time needed to repair all broken tests (a time limit should be set by the user) and the limited number of strategies it can apply to try to repair a test case: if none of them can be applied, the test has to be manually checked and repaired.

Also, since this tool tries to change as little as possible the application code, of course, it will not be able to repair all broken test cases.

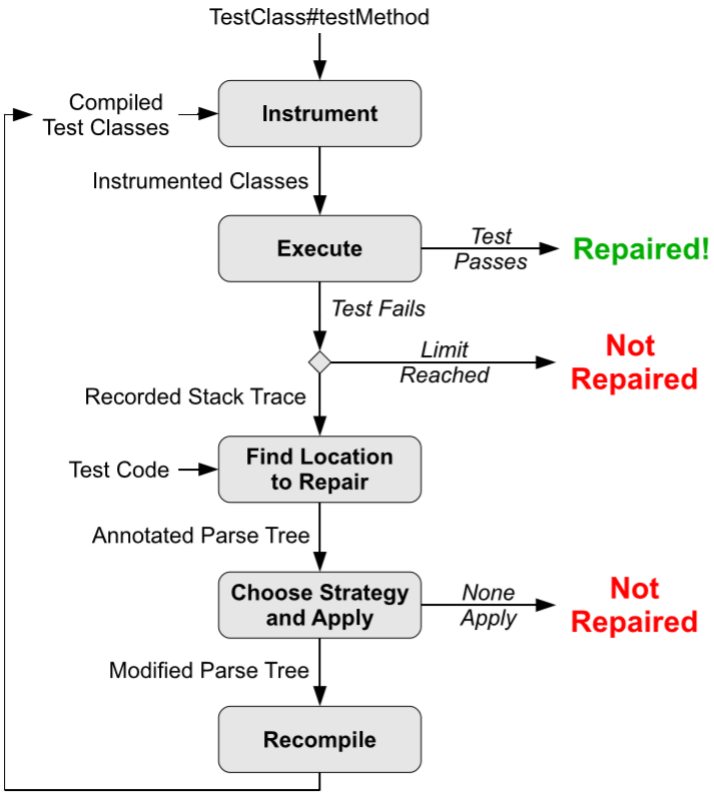


Figure 1.1. ReAssert's repair process.

Some of the strategies ReAssert tries to apply are briefly described in the following list.

- *Replace assertion method:* an assertion that failed is replaced with one that passes or that can be repaired using a different strategy - so it's used as a preprocessing phase;
- *Invert relational operator:* works well with comparable objects, it inverts the operator in the argument of an assertion;
- *Replace literal in an assertion:* takes the literal value of a simple-typed variable, computed at runtime, and replaces the expected value for it with the computed one - making it the expected side of an assertion;
- *Replace with related method:* this strategy applies when the argument to *assertTrue* or *assertFalse* is a call to a common library method that is closely

related to another;

- *Trace declaration-use path:* used when we have helper methods containing several assertions that are called many times in the code. This technique needs to track back the call chain and the arguments of the method that caused the test to fail;

- *Accessor expansion:* replaces a single *assertEquals* on a complex object with as many assertions as are the fields of the complex class the object is an instance of;

- *Surround with try-catch:* used when we need to check that a specific exception was thrown.

ReAssert also provides an extended API that can be used to build other repair strategies, custom made by the developers based on their application requirements, allowing to repair specific failures or tests written in a custom test framework.

1.5.3 VISTA

The Graphical User Interface of an application is the one that allows user interaction and that is, therefore, the main focus of testers inspections oriented to test script writing. VISTA is proposed by Stocco et al. (2018a, 2018b) as a tool that keeps track of the visual elements of the interface, instead of just focusing on the code as other previously in use tools did.

The main fragility of web applications is locators (Hammoudi et al., 2016), that let the elements on the screen be identified based on their properties - like ID, path, text or other DOM properties. For example, test breakage can occur simply moving an element inside the page or changing its text: the repairing phase has then to be triggered on the test and not directly on the application because these trivial modifications don't result in bugs while building and executing the source code.

To find the root cause of test breakage and a possible solution, we need to inspect the test case until we find the GUI element involved in the statement that caused the test to stop: this is the first big challenge of web applications test repair.

Another challenge, as we can expect, is the needed time: this is mainly because the used tools don't offer adequate support to quickly discover the root cause of test breakage and in particular, they don't provide information on the modifications done on the app that brought to it.

The idea behind VISTA is to check the actions executed on the GUI and to validate them runtime, checking possible deviation from the expected or correct behavior. In this way, VISTA supports automation of test repairing in case of locator breakage using the visual information obtained from test execution.

All of this is based on the assumption that the GUI doesn't undergo major changes in its visual structure between two consecutive releases; the DOM, on the other hand, can be updated much more frequently.

The architecture of VISTA

VISTA is written in Java and executes Selenium test cases using Eclipse as an IDE: it follows test cases execution from a visual point of view and it's able to find out at runtime the presence of any locator breakage together with possible solutions, reported to the user for further inspection.

VISTA is composed of two modules that we analyze in the following.

Visual Execution Tracker. In the first phase, the statements contained in the test we are executing are mapped to the corresponding involved DOM locators and the visual aspect of the associated GUI components. All of this is done on a correctly working version of the application (V1). The *Visual Execution Tracker* integrates another tool that is able to intercept calls to methods contained in the Selenium WebDriver library and that automatically creates visual representations of locators, that uniquely identify each element on the screen that is involved in the current test case.

At the end of test execution, the complete mapping is saved in a json file that will be the input of the next VISTA component.

Visual-Augmented Test Runner. The second phase is executed in the updated version of the application (V2).

First, it looks for a corresponding DOM level locator in the new version of the application (V2): if this search has a result, VISTA checks that the returned locator is equivalent to the original one, and in that case, the examined statement is also validated visually.

By the way, there may be some discrepancies between the DOM locator and the visual one: in this case, we call it *mis-selection*, and the tester will have to manually select the correct locator if any.

Another cause of test breakage can be that no locator is found neither following the DOM criteria nor the visual one. VISTA then tries to find equivalent locators in different screens that are at least one level far from the current one and it adds a new transition and a new statement in the test case, if needed, to reach the other screen. If all of this fails, VISTA assumes that the element we're searching has been deleted and suggests the user to remove the corresponding statement in the test case.

1.5.4 ATOM

In the area of mobile applications, the GUI is for sure the component that's most frequently modified and updated to give the user the best possible experience: therefore many test cases involving the GUI become obsolete very quickly and their maintenance is a really expensive task, worth a lot of effort. Furthermore, the development cycle of mobile applications is short and feedback oriented, leaving little time to dedicate to testing practices and test scripts maintenance.

Introduced by Li et al. (2017), ATOM is a tool for automated maintenance of test scripts for mobile applications. To perform this task, ATOM uses two different models: an *Event Sequence Model* (ESM) and a *Delta ESM* (DESM), that respectively represent a possible event sequence and the possible changes done on the GUI transitioning from a version of the application to the next one.

The ESM and the DESM can be represented as finite state machines but they lack explicit start and final states, as shown in Figure 1.2 (Li et al., 2017).

Receiving the ESM and the DESM as inputs, ATOM automatically updates the tests that were written for the base version of the application (represented by the ESM), trying to find alternative mappings in case of absence of references to the searched elements in the updated model.

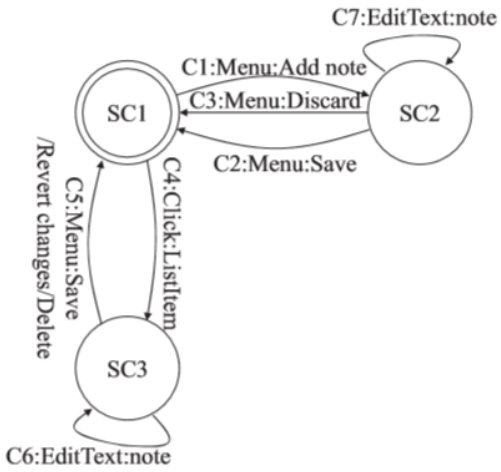


Figure 1.2. Example of ESM as a finite state machine.

In the maintenance phase, ATOM merges the ESM and the DESM, producing the ESM of the updated version of the application. This will be used ad input for the following cycle of execution of ATOM.

Even though building the two models may be an expensive task, the authors consider the overhead to be acceptable since the ESM is built only the first time that ATOM is executed on the app, and each DESM can be simply built from the current ESM because they just represent small variations of it.

How ATOM works

To make the building phase easier, in the ESM we only take into account the status in which the GUI elements are and which are the events that can be performed on each of them; a DESM, on the other hand, specifies the changes related to the application screens in their visual appearance and to the connections between them, triggered by a specific event. In particular:

- the modification of a connection in a DESM is associated with two connections, the removed one and the added one;
- the screens associated with at least one modified transaction are going to be the states of the DESM.

The ESM of the updated version of the app can be computed as $ESM \oplus DESM$.

A test script consists of a series of actions, each of them characterized by a type and a target descriptor that indicates the widget on which the action is executed. Each event in the ESM is mapped to the corresponding action through a relationship described in the configuration file.

The maintenance of the test script described in Li et al. (2017) (Algorithm for maintaining an ESM path) can be easily divided into two phases:

1. Identification of changes and development of a new simulation: this is done by updating, in order, the elements present in the previous simulation;

2. Mapping the events that caused the changes to a test script action. In particular, if an event p is eliminated, ATOM looks for an intermediate state to be able to connect the previous and subsequent path to p that would otherwise be disconnected.

Results and weaknesses

The ATOM implementation presented by Li et al. (2017) relates to scripts based on the Robot Framework and uses Appium for test automation: this can be seen as a limitation of the tool itself. However, the authors indicate the possibility of extending support to other frameworks by simply defining the mapping between the actions performed in the tests and the connections present in the ESM.

Performance-wise, ATOM is proved to be fast, managing to update the tests of each of the applications in less than a second.

Experimental results demonstrated that *"ATOM is effective in achieving high screen and connection coverage* [...] and *high test action preservation when maintaining test scripts"*.

Some weaknesses or suggestions for improvement of ATOM as a tool are listed below.

- automatic test generation techniques could be integrated into ATOM to improve the code coverage provided by the tests;

- the possible errors in implementing the tool are for sure ATOM's biggest weak point;
- the applications chosen by the authors to experiment with the tool may have introduced bias in the study, in particular relating to the appearance and characteristics of the GUI, as they have all been selected from the Chinese Android market.

1.5.5 CHATEM

CHATEM has been developed by Chang et al. (2018) and takes some hints from ATOM to implement change-based testing. In practice, taking two different versions of the same application (e.g. two consecutive releases) it's able to extract the changes between the two GUIs and to generate maintenance actions for each change, combining them to create repair actions for the broken test scripts.

How CHATEM works

Differently from ATOM, CHATEM takes as input the two ESMs of the base and updated version (V1 and V2, using the same notation as before) and a set of test scripts that passed successfully on V1.

From there, it automatically extracts the changes and identifies their impact on the test cases, generates and directly applies the repair actions to update affected test scripts.

The main difference concerning ATOM's approach is that the ESMs are built using automated model extraction combined with manual confirmation from the developer. By the way, the cited study shows that as far as performances are concerned, the two considered tools are almost equivalent in finding and repairing broken test cases considering a trade-off between time and number of repaired test cases.

Equivalence and derivation relations

CHATEM provides us with a useful hint that we also used in this thesis work to implement our automated test repairing tool. CHATEM defines two relations that are used to locate the widgets involved in a test case (Chang et al., 2018).

- *Equivalence relation:* two widgets w1 and w2 (where w1 belongs to a screen s1 and w2 belongs to screen s2 *derived* [see below] from s1) are considered

equivalent if and only if all their properties have the same values, they trigger the same set of events, and their event handlers for each event transit the app to screens with matching IDs

- *Derivation relation:* given two widgets w1 and w2 (where w1 belongs to a screen s1 and w2 belongs to screen s2 derived from s1), CHATEM considers w2 to be the derivation of w1 if:

  1. w1 and w2 are not equivalent and
  2. the majority of their locator properties have comparable values [3]

We used the concept of *derivation* as a "majority criterion" to identify corresponding locators in different versions of the applications GUI if an equivalent widget - namely the same one without modifications - is not found.

The remainder of this thesis is structured as follows:

- Chapter 2 will cover the development process of the tool, together with the description of some used tools and patterns;
- Chapter 3 presents the results of some needed preliminary analysis done on the evolution of Android applications layouts (Section 3.1) and the results of the tool's usage in a few real applications (Section 3.2);
- Chapter 4 covers the threats to the validity of the developed tool;
- Chapter 5 shows the conclusions and hints for further improvement of the tool.

[3]Two String-typed values v1 and v2 are comparable if and only if they have at least half of the words in common, while two values of other types are comparable if and only if they are equivalent

Chapter 2

Architecture and design

Our tool has been developed to support refactoring in case of test breakage due to the impossibility to locate the widget involved in the test case because the attribute used to locate it, for example, the id of the widget or the text inside it, has changed. The tool is written in Java and takes as its main input, files from two different folders:

- one containing the last release of the project, including the test class where all test cases pass successfully (later referred to as **version A**)
- the other containing the release that is currently under development (later referred to as **version B**): this folder also contains the same test class present in version A, but some of its tests don't pass in version B due to layout changes.

The last input of the tool is the log file of the test class of version A executed on version B, listing the errors that caused test breakages.

Figure 2.1 shows the modular structure of the tool. In particular, we have:

1. **Error Parser and Evaluator.** The *Error Parser* takes as input the execution error log of the test class and it produces 2 outputs: the test class path and a list of pairs (in blue in the figure) - one for each broken test case - indicating the *type of error* - i.e. the possible method(s) - and the *argument* of the call causing the error itself.

   The *Evaluator* part internally analyses each pair provided by the parser and produces all possible method calls (in red) that may have caused the breakage. Each method call will be then searched in the test class for possible refactoring.

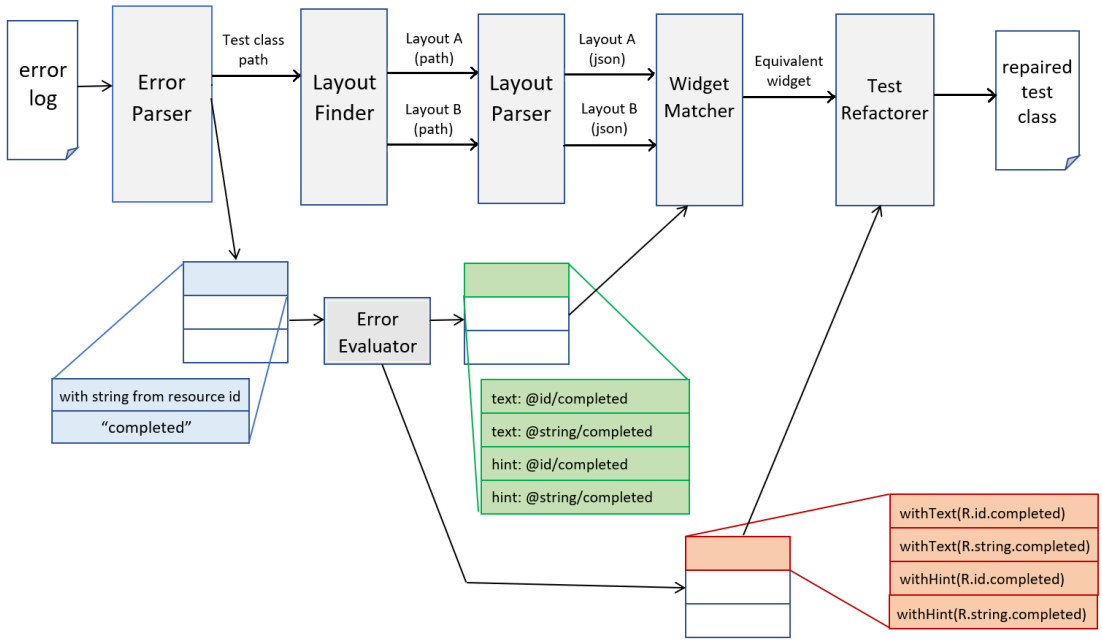


Figure 2.1. Modular representation of the developed tool.

Another output of the error evaluation phase is the possible attribute-value pairs (in green) we have to look for in the layout file of version A to find the referenced widget in that release (so it serves also as an input of the *Widget Matcher* module).

2. **Layout Finder.** It looks for the Activity Under Test and finds its layout in the /res/layout folder of the current release (as described in Section 2.2.2): this is done using *JavaParser*, a tool that can analyze Java code to retrieve its Abstract Syntax Tree, compute statistics on it and also provides methods to refactor code (see Section 2.2.1);

3. The third phase compares versions A and B of the activity layout, to find the most similar widget:

   (a) The **Layout Parser** creates a simplified version of the layout files, assuming that they have the same name in the two versions (Section 2.3.2). For this part we took some concepts from the *Page Object Pattern* (Section 2.3.1);

   (b) Then, the **Widget Matcher** searches, in version B of the layout, for the

widget with the highest match with the one that was involved in the test case - located in version A (Section 2.4).

The widget with the highest match is being selected comparing the widgets attribute values, looking for equivalences and similarities. To do this, we computed some metrics to understand which widget attributes are more subject to modifications throughout the application releases, and consequently assigned a weight to each attribute: the ones that change more often have a lower weight since they are less reliable for this purpose.

Finally, a widget can be selected as equivalent to the one of version A if and only if its matching rate, computed following the previously described metrics, is above a threshold value set, in our case, to 65% (see Section 2.4.1).

4. Lastly, the **Test Refactorer** module performs the repairing/refactoring of the test file according to the results provided by the previous steps (Section 2.5): this is done only after having asked the user/developer whether they want the refactoring to be performed automatically or not.

2.1 Analysing the evolution of Android projects layouts

The first step towards this thesis goal was to understand how much the layouts of the applications change with the evolution of the project. To do this we analyzed 705 GitHub projects and searched for the differences in layout files found in the /res/layout folder in each couple of consecutive releases from the first one to the last one available. It was all done through a bash script, run on a Windows subsystem for Linux that ran Ubuntu 16.04 LTS.

All the projects involved in this phase have at least two tagged releases.

To start, we used the command *git tag* to obtain the tags of all releases of the project from the git log file. The command *git checkout* was then used to realign the project to a given tagged release.

For the sake of simplicity, from now on we will consider just two consecutive versions of a given project: let's call them r1 and r2, where r2 chronologically follows r1.

We looked for layout files in the /res/layout folder of each of the two releases, and first of all compared layout files having the same name in both, using the command *cmp*. If the files were different (meaning *cmp* had produced some output), then the analysis phase started[1].

Then, we executed the *diff* command passing as arguments each pair of files with the same name (f1 and f2, where f1 belongs to r1 and f2 belongs to r2) and in particular, we computed the number of lines in the *diff* command output related to some relevant attributes of the widgets contained in the layout files.

Said attributes are those of the View class - which is extended by all widgets - and of its main subclasses such as textView, button and others.

Since the complete list of attributes of the View class[2] and all its subclasses is really long, we considered just a subset of them, selected through a visual analysis of the XML layout files of some of the applications involved in this first phase. An extensive list of the considered attributes is shown in Table 2.1.

In this table, the strings in the left column are the ones we searched for in the diff and layout files, however, looking for that word also brought up results for other attributes having a similar name, such as the ones we listed in the right column.

Next, we took into account the chance that, in the evolution of the project from r1 to r2, one of the layout files has just been renamed: this case has been evaluated according to the algorithm described in the following.

Let s1 be the set of layout files which (names) are present in r1 but not in r2 (files that may have been removed) and s2 be the set of layout files which are present in r2 and were not present in r1 (files that may have been added).

[1] *cmp* produces output even if one of the two files passed as arguments doesn't exist, so we had to check if both the requested files existed before calling *cmp*.

[2] https://developer.android.com/reference/android/view/View.html

Searched string	Corresponding widget attributes found
android:background	android:background, android:backgroundTint, android:backgroundTintMode
android:clickable	android:clickable
android:focusable	android:focusable, android:focusableInTouchMode
android:id	android:id
android:layout_height	android:layout_height
android:layout_margin	android:layout_margin, android:layout_marginBottom, android:layout_marginEnd, android:layout_marginLeft, android:layout_marginRight, android:layout_marginStart, android:layout_marginTop
android:layout_width	android:layout_width
android:longClickable	android:longClickable
android:onClick	android:onClick
android:padding	android:padding, android:paddingBottom, android:paddingEnd, android:paddingHorizontal, android:paddingLeft, android:paddingRight, android:paddingStart, android:paddingTop, android:paddingVertical
android:scroll	android:scrollIndicators, android:scrollX, android:scrollY, android:scrollbarAlwaysDrawHorizontalTrack, android:scrollbarAlwaysDrawVerticalTrack, android:scrollbarDefaultDelayBeforeFade, android:scrollbarFadeDuration, android:scrollbarSize, android:scrollbarStyle, android:scrollbarThumbHorizontal, android:scrollbarThumbVertical, android:scrollbarTrackHorizontal, android:scrollbarTrackVertical, android:scrollbars
android:src	android:src
android:text	android:text, android:textAlignment, android:textDirection

Table 2.1. Some of the attributes of the View class and of its subclasses considered in this work.

We computed the *diff* between each file in s1 with every single file in s2 to find out if there is the chance than they are just the same file - maybe with minor changes - and the one in s2 simply has a different name. In that case, we considered the two files when computing the global differences between r1 and r2, otherwise, we

discarded them.

In case the two files are identical and the newer version just has a different name, we don't consider it as changed, while we do if up to 10% of the lines appear in the diff command output.

To make it simple, if a file has just been renamed and no other modifications were made on it, it is not considered as one added and one removed file from r1, and not even as changed, as it doesn't comply to our analysis criteria. While if some minor modifications were made on it, we increment by one the counter of modified files in the transition from r1 to r2.

To scale the measured values, the script also computes the total number of lines containing each attribute in both files, so we can easily see the percentage of lines involved in changes, additions or deletions for each considered widget attribute.

The collected data have been stored in a .csv file[3] with the following column headers:

- package [4]
- tag of release r1
- tag of release r2
- number of layout files in release r1
- number of added files from r1 to r2
- number of removed files from r1 to r2
- number of modified files from r1 to r2
- number of lines containing a given attribute (one column for each searched string from Table 2.1) summed over all layouts (ALL id, ALL text, etc.)
- number of lines containing a modification for a given attribute (one column for each searched string from Table 2.1) summed over all layouts (DIFF id, DIFF text, etc.)

[3] https://figshare.com/articles/Evolution_of_Android_applications_layouts_Thesis_Appendix_1_/11777727

[4] Due to the parsing strategy applied to the Manifest file to retrieve the package name, we printed the package name in the csv file only when the code line containing its name matched the specific pattern we were searching for (i.e. the keyword "package" and its value are on the same line and the information about the package name is the only one in said line).

To make the analysis faster, we searched for attributes in the layout files using *rg* (ripgrep)[5] instead of *grep*, since it's been proved that it can be up to 10 times faster than grep itself.

2.1.1 Proposed metrics

Our study revealed that around the 37,3% of releases have at least one change on the layout for one of the attributes we examined with respect to the immediately previous release. We also computed some other metrics to have a better understanding of the collected data. To better visualize the results we found, we computed four different metrics and we displayed them in a couple of histograms. Each column represents the computed value for each of the widget attributes we considered in our analysis, namely the strings in the first column of Table 2.1, without the "android:" prefix. The metrics we computed are described in the following.

1. **Absolute number of modified lines over all releases of all projects (DIFF)**. This metric has been computed as the sum of all lines containing modifications over all releases of all considered projects.

   In symbols:

   $$DIFF_{attr} = \sum_{project=p_1}^{p_n} \sum_{release=r_1}^{master} diff\_lines_{attr} \tag{2.1}$$

   where n in our case is equal to 705 and *attr* is any of the attributes listed before.

2. **Attribute Change Quantity (ACQ)**. This value has been computed, for each attribute, as the percentage of lines changed over all transition of each project, averaged on the number of releases of each project and then averaged over the total number of projects. The column 'any' shows that on average we have changes on 6,8% of lines in layout files when transitioning from a release to the immediately following one.

[5] https://github.com/BurntSushi/ripgrep

In symbols:

$$ACQ_{attr} = \frac{\sum_{project=p_1}^{p_n} \frac{\sum_{release=r_1}^{master} \frac{diff\_lines_{attr}}{all\_lines_{attr}}}{\#releases_p}}{\#projects} \tag{2.2}$$

where $\#releases_p$ is the total number of releases available for project p and $\#projects$ is 705 in our case.

3. **Attribute Change Rate (ACR)**. For this metric, we computed for each project and each attribute, the percentage of transitions between two consecutive releases in which there is *at least one* line that has been modified (meaning changed, added or deleted) involving that specific attribute. This value has been averaged on the number of releases of each project and then on the total number of analyzed projects.

   In symbols:

$$ACR_{attr} = \frac{\sum_{project=p_1}^{p_n} \frac{\sum_{release=r_1}^{master-1} x_{attr}}{\#releases_p}}{\#projects} \tag{2.3}$$

   where x_{attr} is equal to 1 if there is at least one line in the diff between the layouts of release r and the consecutive one, involving attribute *attr*, 0 otherwise.

   master-1 simply indicates the release that immediately precedes the master (last release available).

4. **Attribute Change Presence (ACP)**. This value represents, for each attribute, the percentage of projects having *at least one* changed line for that attribute over all releases. The column 'any' represents the number of projects having at least one change in the layout for any of the considered attributes, in their evolution from the first to the lats release: in our case, this value is around 37,3%.

   In symbols:

$$ACP_{attr} = \frac{\sum_{project=p_1}^{p_n} y_{attr}}{\#projects} \tag{2.4}$$

   where y_{attr} for a project is equal to 1 if there is at least a couple of consecutive releases having x_{attr} (computed as previously defined for ACR) equal to 1, 0 otherwise.

The data collected in this phase have been reported in Chapter 3 (Section 3.1) since they are already a preliminary result of this thesis work and in the field of Android applications layouts evolution. The concept behind this analysis was determining those attributes that change most frequently in the layout evolution: an attribute that is changing its value very frequently from a release to another would be of course less reliable as a reference when trying to identify a widget inside an activity layout.

2.1.2 About this preliminary analysis

This preliminary analysis was done to discover which of the most common attributes of layout widgets are most subject to changes during the evolution of the project.

This will be very useful for the following phases of this thesis work since it will help us build a criterion to associate each widget of a release to its corresponding widget of the immediately following one using a majority rule: the corresponding widget will be the one that has the most similarities with the one from the previous release.

Similarities will be weighed using data collected in this first phase and, in particular, the more an attribute is prone to frequent changes during the evolution of the application, the less weight it will have when computing the similarity rate between a widget and the corresponding one in the following release.

2.2 Finding the root cause of test breakage

As previously stated, in this work we are focusing on test breakage caused by the impossibility to find a widget inside an activity layout because the attribute used to locate the widget involved in the test case has changed its value with respect to the one that is used in the test code (e.g. using methods such as *withId()* and *withText()*).

To pursue our aim of repairing a broken test case, starting from the test file itself, we need to find out which widget caused the test to break, but before that we should find the layout file containing it. To do this, we used *JavaParser*[6].

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

Note: In this section, we won't explain the step-by-step procedure followed to build the tool, but the top-down logical approach that was lying under the building process itself. This is done to make it simpler to understand the thought process behind it and how the tool works.

The procedure follows the steps described at the beginning of this chapter.

2.2.1 *JavaParser*

JavaParser is a set of tools to parse, analyze, transform and generate Java code[7], and it's what we used to perform all the steps that will be described in this section, following the guidelines found on the presentation provided by Tomassetti et al. (2017).

It can parse Java code to get the Abstract Syntax Tree (AST) of it, but it also can generate code starting from a given AST: combining these two features, we can perform refactoring on existing code.

It also includes a library called *JavaSymbolSolver*, that is used to resolve symbols in the AST to retrieve their type and declaration starting from a reference, calculating the result type of an expression, and many other tasks.

Let's see the main features more in detail.

- **Support for code generation:** *JavaParser* can be used to avoid writing boilerplate code, to build transpilers, DSLs and so on. This is done very easily thanks to some specific methods that can be applied to a *CompilationUnit* object that will later be transformed to code simply calling the method *toString()* on the *CompilationUnit*;

- **Support for code analysis:** in particular, support for code metrics computation, for checking if quality standards are met, and even for performing some simple queries to familiarize with the code we are working with;

- **Support for code refactoring:** these features can be used when in need to modernize a huge codebase, to update dependencies, or to change usage patterns.

[7] https://javaparser.org/

2.2.2 From the test file to the layout file

We considered two different ways to locate the corresponding layout file from the test file:

1. From the test class constructor: in this case, the needed class name is found as one of the arguments of the call to the constructor of the super class (namely as *super([. . . ,] SomeActivity.class [, . . .])*;

2. From the declaration of an *ActivityTestRule<T>* object, annotated with *@Rule*: the needed class name in this case is simply *T*;

In both cases, after this step, we should have found the name of the class to which the test case applies.

Analyzing the imports of the test class and its containing package, we can find the complete path of the class of the Activity Under Test, starting from the main app code folder.

The Activity Under Test is then analysed to find its *onCreate()* method and, in its body, the call to *setContentView()* which has a layout resource as parameter.

The layout resource is generally specified using the syntax *R.layout.some_ name.xml*: this shows that the layout file we are looking for can be found under the /res/layout folder of the project.

Once we've found the XML file used as activity layout in the current version of the project, we look for its respective version in the last working release. Knowing which method call(s) could have caused the breakage, we look for the reference widget in the previous release (i.e. the one that was correctly located by the test case then). When we find it, we start searching for the widget in the current version of the project with the highest matching percentage.

To search more easily for a matching widget (one with the most similarities to the one on which the test passed in the previous working release), we built a JSON file containing a simplified version of the layout tree, and this has been done as described in Section 2.3.

2.3 Building a layout object representation

In the second phase, we parsed the layout file to identify each widget and map it to a simplified version of itself. To do this we took some principles from the *Page Object Pattern*.

2.3.1 The *Page Object Pattern*

The Page Object Pattern has been developed in the field of web applications to lower the level of coupling between the test code and the application code. Specifically, building a page object for each page of the application provides a new level of abstraction to the developer, except when writing the page object code itself.

A page object is a representation of the web page, and it provides some methods to access the page structure and data. Page objects are generally written in the same programming language used to write test code.

A study by Leotta et al. (2013) showed that using the Page Object Pattern reduces by a factor of three the amount of time needed to repair web test cases and by a factor of ten the number of lines of code that need to be modified when repairing said test cases: these results are a consequence of the fact that, with this pattern, we don't have to add explicit references to the actual page implementation inside test case because all the details describing the page are encapsulated inside the page object that represents it.

In this way, when we need to repair test cases, changes are concentrated within the page object rather than the test code. Changes in the actual test cases may be needed as well if the web application under test undergoes major evolution or needs radical maintenance actions.

It has also been observed that this pattern doesn't reduce the test cases fragility, it just makes repairing easier - and faster.

2.3.2 Layout parser

Based on the Page Object Patter approach, we tried to develop something similar for mobile applications, building an object representing each activity and its widgets in a way that would make test cases easier to be repaired in case a valid widget locator was not found for a given test action.

The activity layout file found in the /res/layout folder of the analyzed project was read using a java application that used the method *parse* of the java class *DocumentBuilder*.

This XML file has been parsed to create a corresponding JSON file containing all the widgets with just the attributes we considered in phase one of this work (Table 2.1). Also, we kept the hierarchical structure of the layout adding the attribute "children" to nodes that were also containers for other widgets (e.g. a button containing both an image and some text), and the attribute "container" in each child node: its value has been set to the container id - if present - or to the container class name (e.g. RelativeLayout) if the id was not specified.

Figure 2.2 shows the class diagram of JSON file as it has been built.

In practice, we used the class *DocumentBuilder* found in the package *javax.xml.parsers* and in particular its method *parse*. Then we also wrote a custom method that recursively visited the layout tree in pre-order, and setting the "container" attribute in each child as a simple string, while the children have been represented in the container as a list of JSON objects.

Figures 2.3 and 2.4 show the main activity of a note-taking application called OmniNotes[8] and its class diagram / UML representation based on the previously described criteria.

The associations indicate the container-child node relationship: the child node has its container id or class as the value of the "container" attribute, while the root of the layout tree has "root" as an arbitrary value for the "container" attribute.

Using the JSON format and keeping only selected attributes for each widget has made it easier to build the layout objects again when needed in the following steps.

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

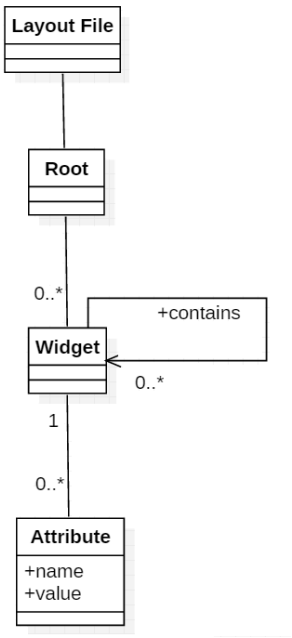


Figure 2.2. Class diagram of a JSON file built from the analysis of a layout file found in the /res/layout folder.

Figure 2.3. Screenshot of the main activity of the last release of OmniNotes.

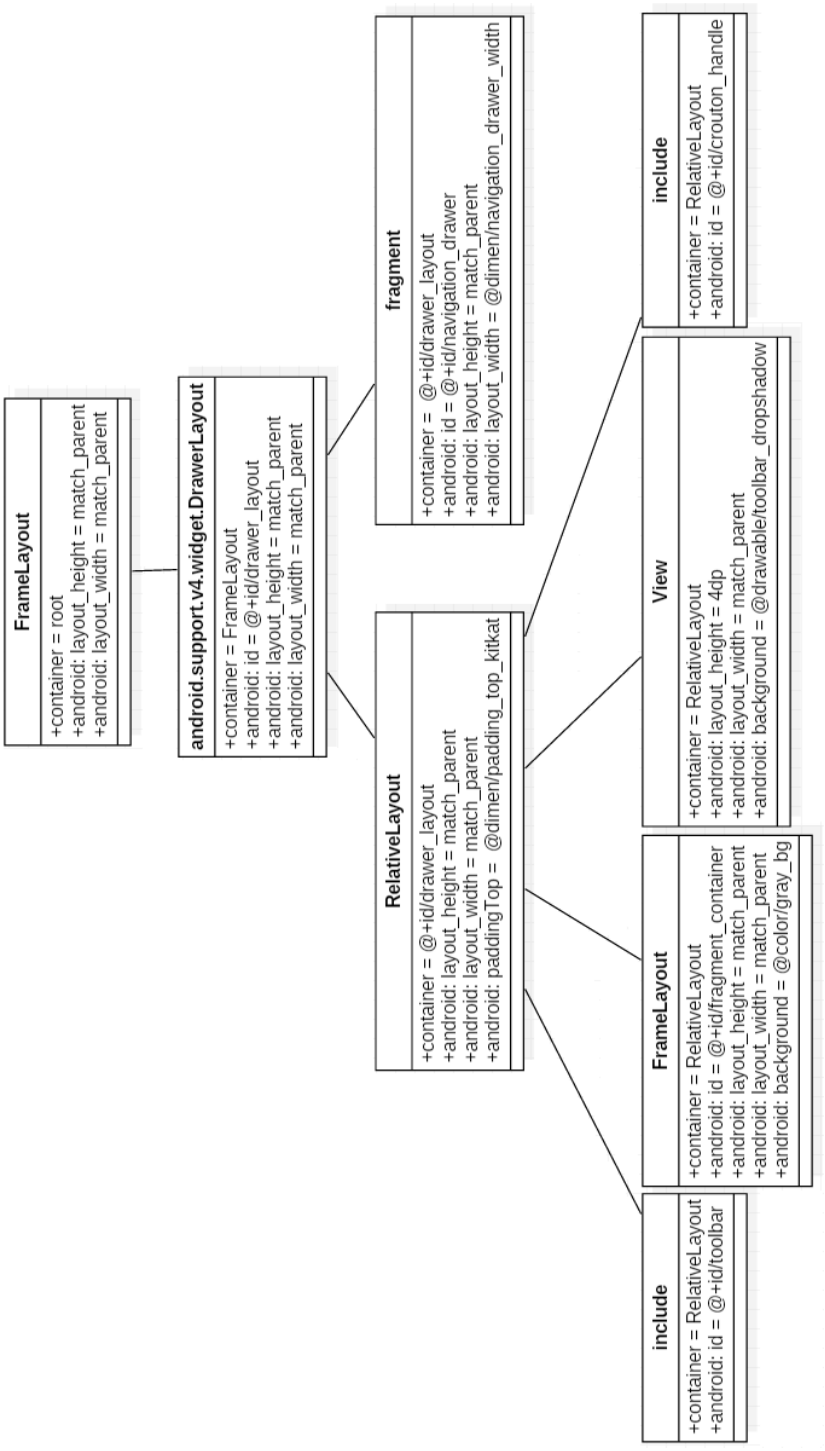


Figure 2.4. Object diagram of the main activity of the last release of OmniNotes.

2.4 Finding the most similar widget in the current version of the project

Let's consider two consecutive releases, we will call them version A and B: A is the last working release, on which the test case passed, and B is the current release. Let's also imagine that the activity layout file is called *some_ activity.xml*. As previously described in this chapter, both versions A and B of the layout file have been used to build the corresponding simplified JSON files representing them.

Let j_A be the JSON representation of *some_ activity.xml* in version A and j_B the JSON representation of the layout file having the same name in version B.

Each widget in j_A can be mapped to the one of j_B that is most similar to it with respect to the values of the considered attributes, weighed on the data previously collected about how often these attributes change values over releases and projects (shown in Figures 3.1 and 3.2).

The weight given to each attribute (or group of attributes) i in this step is shown in Table 3.1 and it has been computed as:

$$Weight_i = \frac{\frac{1}{Avg_i}}{\sum_i \frac{1}{Avg_i}} \times 100 \tag{2.5}$$

where

$$Avg_i = \frac{ACP_i + ACQ_i + ACR_i}{3} \tag{2.6}$$

In this way, the more frequently an attribute value changes - i.e. the higher its values for ACQ, ACP and ACR are - the less weight it has, because a frequently changing attribute is, of course, less reliable for our purpose.

Finally, the matching rate between the reference widget in version A and each of those present in the corresponding layout of version B has been computed as

$$match = \frac{\sum_{attribute=a_1}^{a_n}(weight_a \times x_a)}{\sum_{attribute=a_1}^{a_n} weight_a} \times 100 \tag{2.7}$$

where a represents each attribute in Table 2.1 that has a value for both widgets and x_a is equal to 1 if the two widgets have the same value for attribute a, 0 otherwise.

Weights for each attribute have been computed as shown in Equation 2.5, so

taking into account the values of the metrics that we proposed for this purpose.

The output of this mapping step is the id of the widget having the highest match with the one that was located in version A and made the test pass then. In a real case, version A would be the last available release while version B would be the one currently being developed.

In case the widget we were looking for didn't have an id, we would identify it using its contained text or content description.

2.4.1 Minimum matching threshold

Notice that if a matching percentage is too low, we can imagine that the widget we are trying to map to a widget of the newest release has been removed so it is no longer present, or that the navigation dynamics of the application changed, hence it could be possibly found in another layout file - and not necessarily in the one with the same name we are searching in.

For this reason, we analyzed a few Android applications and their layout evolution to establish a minimum matching threshold value to consider the widget found in version B worthy of being used to refactor the test case.

The complete analysis, explained in details in Section 3.1.2, brought the value for this threshold to be set to 65%

2.5 Repairing a broken test case

Up to this point, we covered the first 3 steps described at the beginning of this chapter, therefore we found the widget in the current version that's most similar to the one we were looking for. So now it's time to repair the test case and possibly the other ones that would be affected by the same kind of breakage because they involved the same widget in the last working release.

What we will do is an advanced version of refactoring: in particular, we will be replacing, inside the broken test case, the old content description or text with the one of the most similar widget in the current release.

For example, if the widget contained text is what changed (from version A to version

B) and caused the test to break, we will look for the string *withText("Text in version A")* inside the test class and replace it with *withText("Text in version B")*.

Using a regular expression, we also have to consider the possible spaces before and after the brackets, when searching these strings in the test code.

2.5.1 Analysing the broken test case error log

The last step in the development of this tool has been the analysis of the error log when a test case fails because we can't locate the widget on which we a test action has to be performed.

In particular, we considered 4 of the ViewMatchers class methods - *withId*, *withText*, *withContentDescription* and *withHint* - and 3 possible types of arguments: an id resource, a string resource and a quoted string[9]. Examples of method calls are shown in Table 2.2: the reference case number is a way to link this table with Table 2.3, where the obtained outputs are listed for each possible method-argument combination.

Method	Argument	Call example	Reference case number
withId	id resource	withId(R.id.button)	1a
	string resource	withId(R.string.send)	1b
withText	id resource	withText(R.id.button)	2a
	string resource	withText(R.string.send)	2b
	quoted string	withText("Hello")	3
withContentDescription	id resource	withContentDescription(R.id.button)	4a
	string resource	withContentDescription(R.string.send)	4b
	quoted string	withContentDescription("Hello")	5
withHint	id resource	withHint(R.id.button)	2c
	string resource	withHint(R.string.send)	2d
	quoted string	withHint("Hello")	6

Table 2.2. Possible method calls of class ViewMatchers causing errors in Espresso test files

[9]with the exception of the method *withId*, that doesn't have an implementation accepting a quoted string.

Reference case number	**Output in case of error (No views in hierarchy found matching: ...)**
1a	with id: [file path]:id/[resource name]
1b	with id: [file path]:string/[resource name]
2 (a, b, c, d)	with string from resource id: <[resource number]>[resource name]
3	with text: is [quoted string]
4 (a, b)	with content description from resource id: <[resource number]>[resource name]
5	with content description: is [quoted string]
6	with hint: is [quoted string]

Table 2.3. Possible error outputs caused by errors in method calls shown in Table 2.2

As we can see in the tables, outputs having the same format can be obtained with different method-argument type combinations: this means that we have to possibly replace each possible call that caused the error with another one using, in the best case, the same method.

Let's take an example: let's assume that the method call that caused the error is of type [4], so with a call to *withContentDescription*, but in the current version of the project, the most similar widget doesn't have a value for the attribute *android:contentDescription*. In this case, we looked for values of one of the other attributes (*android:id*, *android:text* and *android:hint*): in case none of these is present, the method call causing the error is not going to be refactored.

Chapter 3

Experiment and results

3.1 Preliminary research results

The starting point of this thesis work was, as previously mentioned, analyzing the evolution of Android applications layouts and, in particular, how much a widget attribute changes its value across all the application releases.

We studied a set of 705 GitHub projects and computed the values of each proposed metric on this set. To do this, we considered some of the most common attributes of the View class, since all widget elements are extensions of it. In particular, we selected the attributes listed in Table 2.1, based on a visual analysis of the XML layout files.

3.1.1 Collected data on Android applications layouts evolution

Proposed metrics

For what concerns the DIFF metric, computed as defined in (2.1), the values are shown in Figure 3.1: the sum of the values of each column adds up to 118.623 for 705 analysed projects.

The values for ACQ, ACR and ACP have been calculated as defined in (2.2), (2.3) and (2.4) respectively and can be seen in Figure 3.2

.

We would expect ACP for 'any' attribute to be close to 100% but that is not

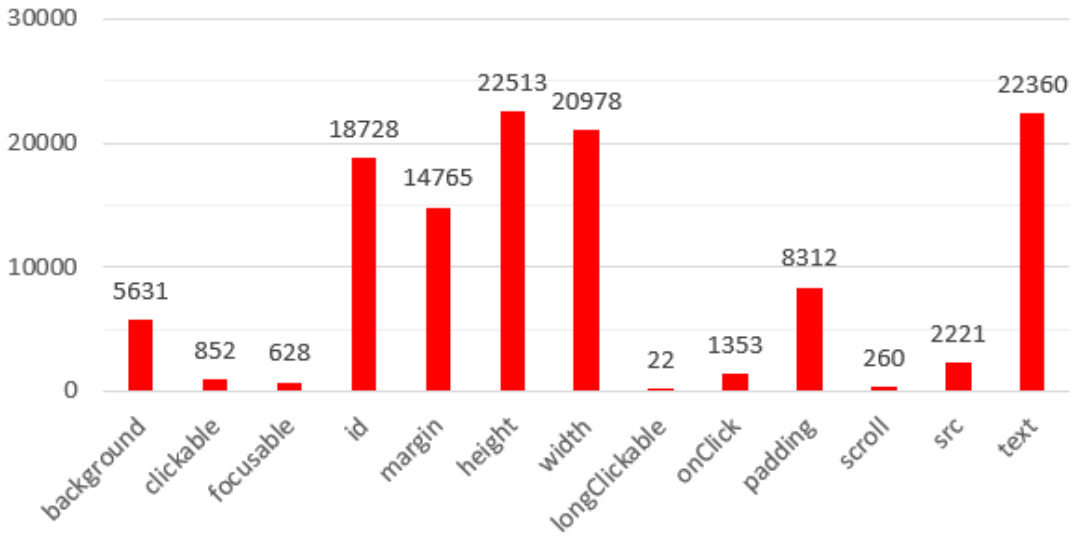
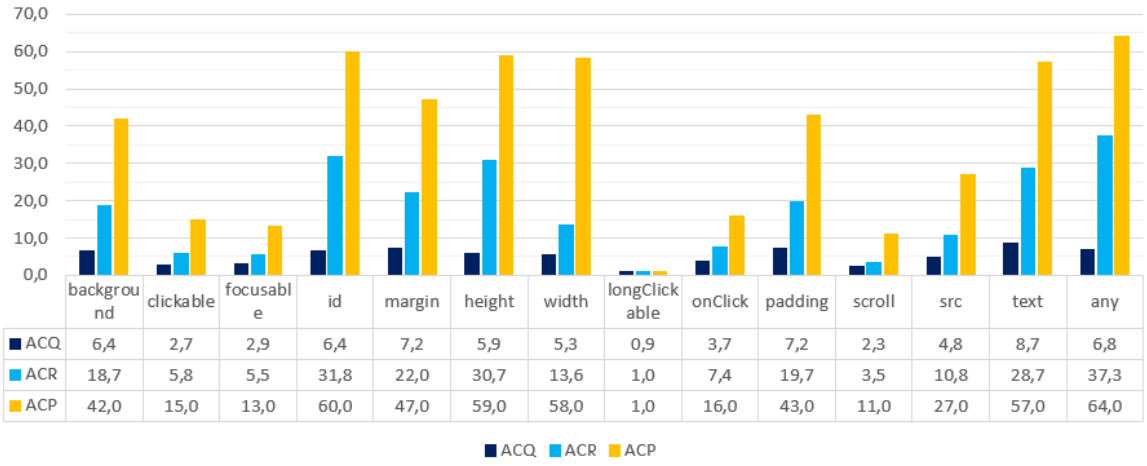


Figure 3.1. DIFF distribution over all considered attributes.



	background	clickable	focusable	id	margin	height	width	longClickable	onClick	padding	scroll	src	text	any
ACQ	6,4	2,7	2,9	6,4	7,2	5,9	5,3	0,9	3,7	7,2	2,3	4,8	8,7	6,8
ACR	18,7	5,8	5,5	31,8	22,0	30,7	13,6	1,0	7,4	19,7	3,5	10,8	28,7	37,3
ACP	42,0	15,0	13,0	60,0	47,0	59,0	58,0	1,0	16,0	43,0	11,0	27,0	57,0	64,0

Figure 3.2. ACQ, ACR and ACP distribution over all considered attributes.

the case: this may be caused by the fact that many of the considered projects have a few or just two available releases and therefore no actual changes on the layout files, but presumably, there should be changes regarding functionalities.

As we expected, the widget attributes involving text are often subject to changes over all the releases of a project.

Our analysis also highlights the fact that aside attributes regarding the size of widgets and their spacing from other widgets and the whole screen layout itself (namely height, width, margin and padding related attributes), widget ids and background have a high value for each of the proposed metrics.

Attribute weights for matching widget search

The values computed for the proposed metrics have then been reported also in Table 3.1 and used to compute the weight each considered attribute has when looking for the matching widget in the current release - as defined in (2.5).

The *Average* of attribute i (Avg_i) column is just the average of the ACQ, ACR and ACP values (2.6); *Inverse average* for the attribute i is computed as $\frac{1}{Avg_i}$. The *Weight* represents the percentage of that *Inverse average* value over the sum of the *Inverse average* column.

Attribute(s)	ACQ	ACR	ACP	Average	Inverse average	Weight (%)
android:background	6,4	18,7	42,0	22,38	4,47	2,31
android:clickable	2,7	5,8	15,0	7,85	12,74	6,59
android:focusable	2,9	5,5	13,0	7,12	14,04	7,26
android:id	6,4	31,8	60,0	32,73	3,06	1,58
android:margin	7,2	22,0	47,0	25,38	3,94	2,04
android:height	5,9	30,7	59,0	31,86	3,14	1,62
android:width	5,3	13,6	58,0	25,64	3,90	2,02
android:longClickable	0,9	1,0	1,0	0,98	101,88	52,71
android:onClick	3,7	7,4	16,0	9,05	11,04	5,71
android:padding	7,2	19,7	43,0	23,30	4,29	2,22
android:scroll	2,3	3,5	11,0	5,62	17,80	9,21
android:src	4,8	10,8	27,0	14,21	7,04	3,64
android:text	8,7	28,7	57,0	31,47	3,18	1,64

Table 3.1. Attribute change metrics and attribute weights to compute widget similarity rate

As previously stated, the more frequently an attribute value changes the less weight it has when computing the matching percentage between two widgets.

3.1.2 Computation of the minimum matching threshold for test refactoring

We analysed a few github projects (Glucosio[1], K9-MailClient[2], MiMangaNu[3], Omni-Notes[4] and PassAndroid[5]) having between 9 and 128 tagged releases to find the matching threshold under which a match would not be considered relevant for our purpose.

For all couples of consecutive releases, and for all their layouts, we applied the algorithm previously described in Section 2.4, to find the widget with the highest match in the following release.

All the collected data have been stored into a .csv file with the following format:

- App name
- Tag of release A
- Tag of release B
- Layout file name
- Widget id in version A
- Widget id in version B
- Widget text in version A
- Widget text in version B
- Widget contentDescription in version A
- Widget contentDescription in version B
- Match percentage

For the sake of this analysis, we discarded all the widgets having the three attributes *android:id*, *android:text* and *android:contentDescription* not set or null in version A: this was done because it's impossible to identify a widget inside a layout if it doesn't have a value for any of these attributes.

[1] https://github.com/Glucosio/glucosio-android

[2] https://github.com/k9mail/k-9

[3] https://github.com/raulhaag/MiMangaNu

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

[5] https://github.com/ligi/PassAndroid

The output of this step has been stored in a .csv file [6]

Out of 81223 lines (all widget of all layouts in all releases of all the previously listed 5 apps):

- 77914 lines (95,9%) have a 100% match;
- 1379 lines (1,7%) have a match between 90% (included) and 100% (excluded);
- 213 lines (0,26%) have a match lower than 10% (out of which, 205 have a 0% match, so we can assume those cases, the widget has been removed - or moved to another activity - within a transition between two consecutive releases);
- The average match is of 98,8%;
- The median value of matches percentages is 100%.

The distribution of matches percentages from 10 to 100 (excluding the upperpl extreme of the interval) is shown in Figure 3.3, while Table 3.2 shows the mean and median values of matches computed on different subsets.

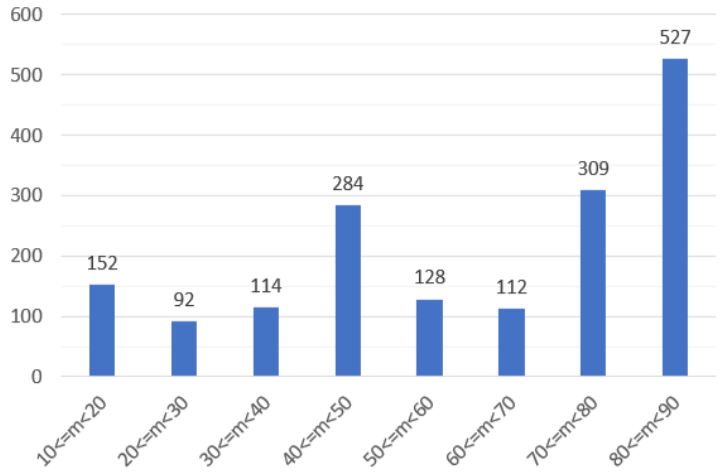


Figure 3.3. Matching percentage of widgets across consecutive releases

The rightmost column of Table 3.2 contains the values we used as a starting point for deciding the threshold value. In particular, we started from the mean

[6] https://figshare.com/articles/Widgets_mapping_and_matching_rate_over_consecutive_releases_Thesis_Appendix_2_/11777730

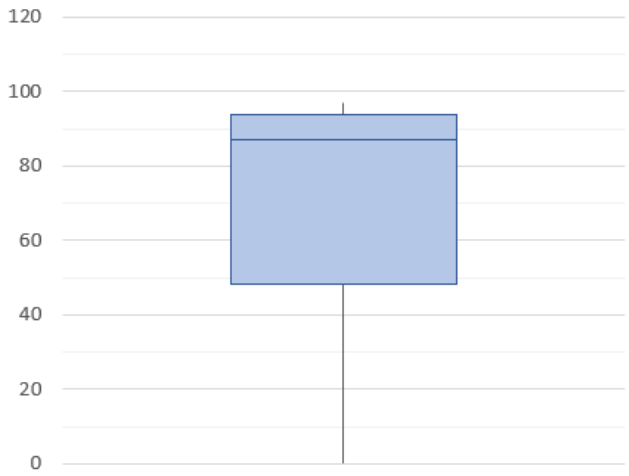


Figure 3.4. Box Plot for matching percentage

	All matches	**Without 100% matches**
Mean value	98,8%	71,1%
Median value	100%	87,1%

Table 3.2. Mean and median values of matches found in this phase for the analysed applications.

match value when not considering the 100% matches, and we lowered it a bit - to **65%** - to take into account possible valid matches that are under 70%: this means that matches under this threshold are not going to be considered valid matches by our tool.

To support this computation, we can observe that above the threshold, the 71,3% of the mappings (2214 out of 3104, not having a 100% match) have the same id, the same text or the same content description.

3.2 Experimental results of the tool

We conducted our experiment on three different applications: Lazy Cafe' [7], Glucosio[8], and TodoList[9]. The following criteria hold for each of them:

- We ran our tool working with two different folders: the first one containing the last working release (version A of the project, where all test cases passed successfully), while the second one contains the current release (the one under development, version B of the project, where some tests are broken);
- We analyzed the error log from running a single test class containing 10 test cases;
- The test cases have been custom-built using method calls taken among those in listed in Table 2.2, to cover all possible errors with a reference case number from 2 to 6 (see Table 2.3). The test cases tried to locate widgets that have been *added* to the main activity layout of each application, to be able to test our tool more easily than inspecting all releases to find test cases that suited our purpose - so covering each possible test breakage caused addressed in this Section.
- Case number 1 will be further discussed in Section 3.2.3. For this reason, the ids of the widgets involved in the tests have not been modified from version A to version B;
- Each time an equivalent widget was found in version B of the application, the refactoring was always performed, even in case the wrong widget was selected by our tool: we've done this to observe the tool's ability to locate the widgets correctly.

3.2.1 Widgets and results display format

To better analyze the data and the results of the experimental part, we created some tables. In particular, for each selected application, we'll have the following:

[7] https://github.com/BankkokBank/final_project_android/tree/master/LazyCafe

[8] https://github.com/Glucosio/glucosio-android

[9] https://github.com/seifane/TodoApp

1. **Widgets table:** it contains the information about the widgets added to versions A and B of the app, together with their values for the attributes *android:id*, *android:text*, *android:contentDescription* and *android:hint*.

   - widgets with number 1-3 are of class EditText, while those with number 4 and 5 are instances of the Button class;
   - widgets identified with "a" or "b" belong to version A or B of the application, respectively.
   - widgets numbers can be seen in the applications screenshots, also included in this section;
   - A dash (-) in an attribute column means that that attribute **does not** have a value for that given widget.

2. **Method calls table:** it's a table containing all the test cases method calls and their refactored version after running the tool. This table also shows the reference case number for each method call (as defined in Table 2.2);

   - A dash (-) in the *Refactored method call(s)* column means the call has not been affected by our tool, so it remained as it was before refactoring.
   - As each test case should locate one or two widgets inside the activity layout, the last column of a *Widgets table* shows the numbers of the test cases that should locate each widget. In this way, the test case number can be seen as a foreign key pointing to one of the rows of the respective *Method calls table*: with this information, we can double-check the results achieved by our tool.

3. **Results table:** this kind of table contains, for each method call on the previous table, the information on whether the method call makes the test pass or not in version B before and after the refactoring, together with its equivalence with the broken test case method call - i.e. if the two method calls can locate equivalent widgets in the two versions of the app (namely those having the same id, since we haven't changed ids in this experiment)

Any other information specific to one of the selected applications and/or its written test cases will be given in its dedicated part.

Lazy Cafe'

The aspect of the main activity of this app in versions A and B is shown in Figure 3.5.

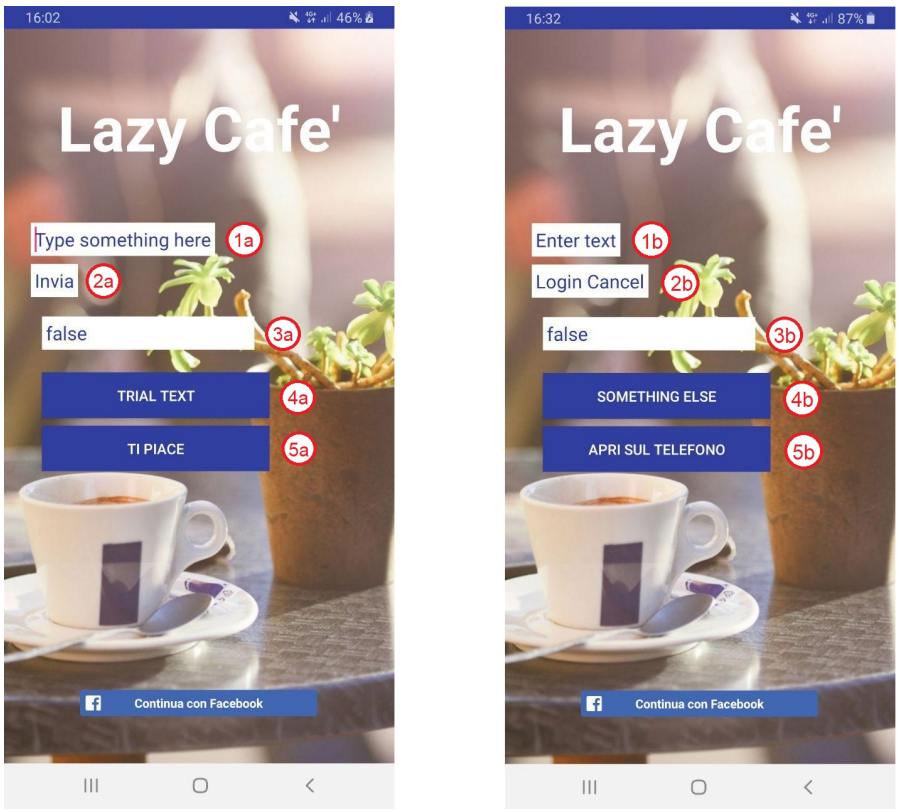


Figure 3.5. Main activity of Lazy Cafe' in versions A and B

Table 3.3 describes the added widgets in terms of the considered 4 attributes values.

Note that, in this case, we mainly changed the values of the attributes involved in the test cases method calls when creating version B of the app.

As shown in Table 3.4 all the method calls causing test breakage have been refactored, since the tool succeeded each time in finding a widget having a matching percentage higher than our threshold.

Eventually, we observe that if the test case contained two method calls that would cause test breakage, the test case execution stops after the first one: for how

Widget #	android:id	android:text	android:contentDescription	android:hint	Should be located in test case(s)
1a	@+id/input1	-	"Input field 1"	"Type something here"	[8, 9]
1b	@+id/input1	-	"Input field one"	"Enter text"	
2a	@+id/input2	-	"Input field 2"	@string/ com_facebook_send _button_text	[2, 4, 6]
2b	@+id/input2	-	"Input field two"	@string/cancel_login	
3a	@+id/input3	@id/input1	"Input field 3"	@id/share	[1, 3, 10]
3b	@+id/input3	@id/like	"Input field 3"	@id/detail	
4a	@+id/share	"Trial text"	@string/ com_facebook_share _button_text	-	[5, 7]
4b	@+id/share	"Something else"	@string/ com_facebook_loading	-	
5a	@+id/like	@string/ com_facebook_like _button_liked	@id/input1	-	[2, 6]
5b	@+id/like	@string/ common_open_on_phone	@id/input2	-	

Table 3.3. Widgets added to versions A and B of the app Lazy Cafe'

Test #	Reference case number	Method call(s) (version A)	Refactored method call(s) (version B)
1	2a	withText(R.id.input1)	-
2	2b	withText(R.string. com_facebook_like_button_liked)	withText(R.string. common_open_on_phone)
	5	withContentDescription("Input field 2")	withContentDescription("Input field two")
3	2c	withHint(R.id.share)	-
4	2d	withHint(R.string. com_facebook_send_button_text)	withHint(R.string.cancel_login)
5	3	withText("Trial text")	withText("Something else")
6	4a	withContentDescription(R.id.input1)	-
	5	withContentDescription("Input field 2")	withContentDescription("Input field two")
7	4b	withContentDescription(R.string. com_facebook_share_button_text)	withContentDescription(R.string. com_facebook_loading)
8	5	withContentDescription("Input field 1")	withContentDescription("Input field one")
9	6	withHint("Type something here")	withHint("Enter text")
10	3	withText("false")	-

Table 3.4. Method calls causing test breakage in version B of Lazy Cafe' and their refactored versions

Test #	Reference case number	Does the method call pass on version B?	Does the refactored method call pass on version B?	Do the two calls locate equivalent widgets?
1	2a	yes	yes	yes
2	2b	no	yes	yes
	5	-	yes	yes
3	2c	yes	yes	yes
4	2d	no	yes	yes
5	3	no	yes	yes
6	4a	yes	yes	yes
	5	no	yes	yes
7	4b	no	yes	yes
8	5	no	yes	yes
9	6	no	yes	yes
10	3	yes	yes	yes

Table 3.5. Refactored test cases success and equivalence to the broken ones for the app Lazy Cafe'

we built the test cases we have been lucky and - as can be seen in test number 2 - the call to *withContentDescription* was refactored anyway since the same call appeared in test case number 6, where it leads to test breakage.

As shown in Table 3.4, we considered just one method at a time, and all the test cases have been successfully repaired maintaining the same method calls: this was possible because each widget kept a value for the attribute we were trying to locate it with.

With the next two applications, we tried to change the type of value assigned to each widget - changing from version A to version B - and also removing the attribute from a widget, so that it had to be located using another method. This was mainly done to test out all the features of our tool, described in Section 2.5.1.

Glucosio

Figure 3.6 shows the aspect of the main activity of Glucosio in versions A and B.

In addition to the changes reported in Table 3.6, we need to list that - when transitioning from version A to version B of the layout:

- **Button 8a** had *android:textColorHint="@color/colorPrimaryDark"* (blue), while in **8b** the same attribute has value "@color/dark_red" (dark red);
- **Button 9a** had *android:background="@color/colorPrimaryDark"* (blue), while

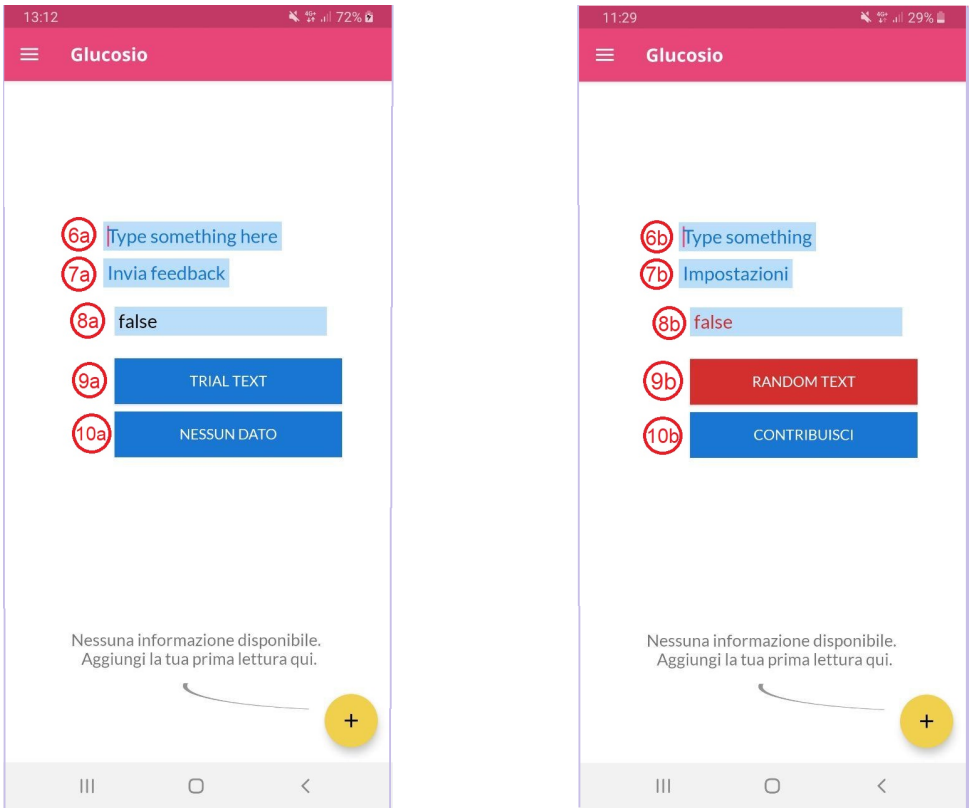


Figure 3.6. Main activity of Glucosio in versions A and B

Widget #	android:id	android:text	android:contentDescription	android:hint	Should be located in test case(s)
6a	@+id/input6	-	"Input field 6"	"Type something here"	[18, 19]
6b	@+id/input6	-	@string/assistant	"Type something"	
7a	@+id/input7	-	"Input field 7"	@string/action_feedback	[12, 14, 16]
7b	@+id/input7	-	-	@string/action_settings	
8a	@+id/input8	@id/input6	"Input field 8"	@id/credits	[11, 13, 20]
8b	@+id/input8	-	"Input field 8"	@id/credits	
9a	@+id/credits	"Trial text"	@string/about_credits	-	[15, 17]
9b	@+id/credits	"Random text"	@string/about_donate	-	
10a	@+id/no_data	@string/a11y_no_data	@id/input7	-	[12, 16]
10b	@+id/no_data	@string/about_contribute	@id/credits	-	

Table 3.6. Widgets added to versions A and B of the app Glucosio

in **9b** the same attribute has value "@color/dark_red" (dark red).

We did this to change up the widgets a little bit more and see if the tool was still able to associate them with one another across the two versions.

Test #	Reference case number	Method call(s) (version A)	Refactored method call(s) (version B)
11	2a	withText(R.id.input6)	withId(R.id.input8)
12	2b	withText(R.string.a11y_no_data)	withText(R.string.about_contribute)
	5	withContentDescription("Input field 7")	withId(R.id.input7)
13	2c	withHint(R.id.credits)	-
14	2d	withHint(R.string.action_feedback)	withHint(R.string.action_settings)
15	3	withText("Trial text")	withText(R.string.about_contribute)
16	4a	withContentDescription(R.id.input7)	-
	5	withContentDescription("Input field 7")	withId(R.id.input7)
17	4b	withContentDescription(R.string.about_credits)	withContentDescription(R.id.credits)
18	5	withContentDescription("Input field 6")	withContentDescription(R.string.assistant)
19	6	withHint("Type something here")	withHint("Type something")
20	3	withText("false")	-

Table 3.7. Method calls causing test breakage in version B of Glucosio and their refactored version

Test #	Reference case number	Does the method call pass on version B?	Does the refactored method call pass on version B?	Do the two calls locate equivalent widgets?
11	2a	no	yes	yes
12	2b	no	yes	yes
	5	-	yes	yes
13	2c	yes	yes	yes
14	2d	no	yes	yes
15	3	no	yes	no
16	4a	yes	yes	yes
	5	no	yes	yes
17	4b	no	yes	no
18	5	no	yes	yes
19	6	no	yes	yes
20	3	no	no	-

Table 3.8. Refactored test cases success and equivalence to the broken ones for the app Glucosio

Note that the modifications of widget number 6 (content description went from a quoted string in 6a to a string resource in 6b) and 7 (content description no longer present in 7b) were made to check if the tool was still able to identify the equivalent widgets and change the type of method call needed to identify them in the test cases - which happened successfully, as shown in Table 3.8.

Lastly, we notice that modifying widget 8 text attribute value, that was an id resource in 8a and is no longer set in 8b, made test number 20 impossible to be repaired: in fact, none of the widgets present in the activity layout in version B

has text equal to false or to an id resource - which we observed would behave as a "false" value while running the same test case on Lazy Cafe' (see test case number 10 in Table 3.4).

For the same reason, the first method call of test case number 16 passes successfully, even though there is no widget inside the layout having *@id/input7* as content description.

As shown in Table 3.8, this time our tool succeeded in refactoring 8 method calls causing breakage, but two of those ended up not being equivalent to the original calls.

We can imagine that this happened because we also changed the background color of widget 9 (from 9a to 9b, as previously written): this modification, together with the ones done on the attribute values we used to try to locate the button inside the activity layout, made button 10b the most similar to 9a, due to its consistent colour[10].

It is also very likely that if the background color was the only attribute value we changed, the tool would have picked widget 9b as the most similar to 9a and therefore repaired the test cases with other method calls able to locate the equivalent widgets.

TodoList

The aspect of the main activity of this app in versions A and B is shown in Figure 3.7.

Table 3.9 describes the added widgets in terms of the considered 4 attributes values. When transitioning from version A to version B, this time changed values to **all three** reference attributes.

As shown in Table 3.10, some of these test cases involve two widgets and, in particular, both of the method calls would cause test breakage but that isn't shown in the log output since, as stated before, the first breakage makes the test case execution stop.

This means that on the first run of our tool, some method calls will not be

[10]This can be seen looking at the weights we gave to each attribute (Table 3.1), where the background had a weight that's bigger than those of id and text.

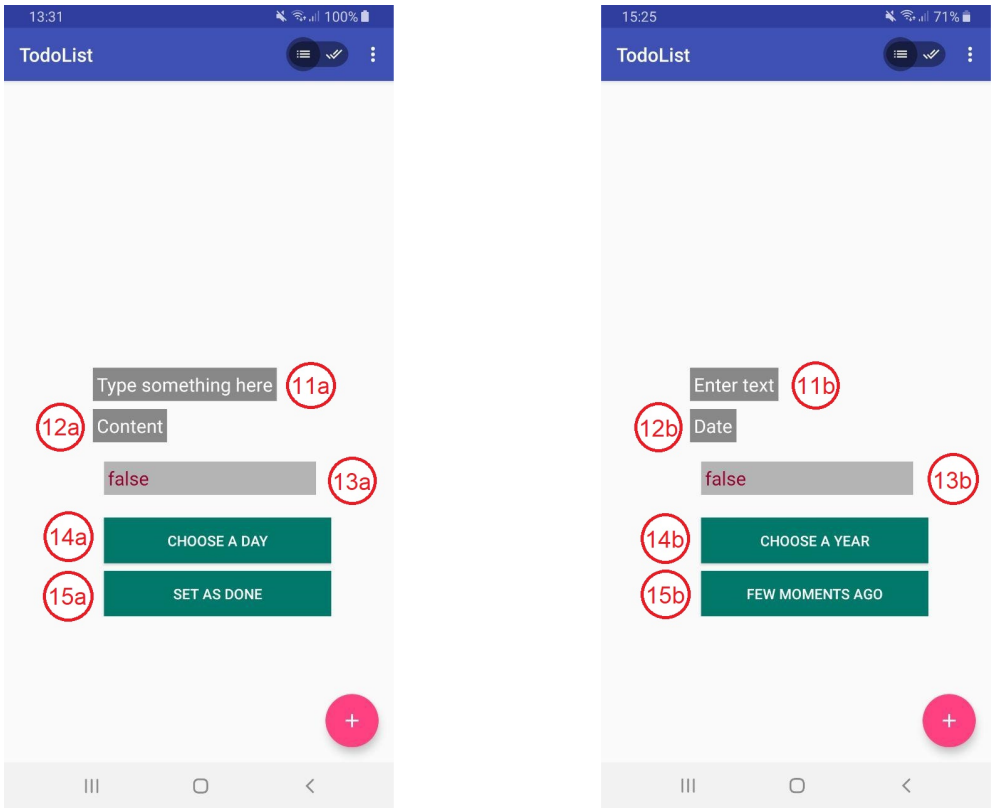


Figure 3.7. Main activity of TodoList in versions A and B

Widget #	android:id	android:text	android:contentDescription	android:hint	Should be located in test case(s)
11a 11b	@+id/input11 @+id/input11	- -	"Input field 11" "Input field eleven"	"Type something here" "Enter text"	[21, 23]
12a	@+id/input12	-	"Input field 12"	@string/hint_content	[21, 25, 29, 30]
12b	@+id/input12	-	"Input field twelve"	@string/hint_date	
13a 13b	@+id/input13 @+id/input13	@id/set_done @id/input12	"Input field 13" "Input field thirteen"	@id/input11 @id/set_done	[22, 24, 26]
14a	@+id/select_date	"Choose a day"	@string/ mdtp_select_day	-	[22, 24, 28]
14b	@+id/select_date	"Choose a year"	@string/ mdtp_select_year	-	
15a	@+id/set_done	@string/ notification_set_done	@id/select_date	-	[23, 25, 26, 27]
15b	@+id/set_done	@string/ time_ago_now	@id/input13	-	

Table 3.9. Widgets added to versions A and B of the app TodoList

Test #	Reference case number	Method call(s)	Refactored method call(s) (those refactored after the second run are signaled with [*] and in bold)
21	5	withContentDescription("Input field 11")	withContentDescription("Input field eleven")
	2d	withHint(R.string.hint_content)	**[*] withHint(R.string.hint_date)**
22	2a	withText(R.id.set_done)	-
	4b	withContentDescription(R.string.mdtp_select_day)	withContentDescription(R.string.mdtp_select_year)
23	4a	withContentDescription(R.id.select_date)	-
	6	withHint("Type something here")	withHint("Enter text")
24	2c	withHint(R.id.input11)	-
	3	withText("Choose a day")	withText("Choose a year")
25	2b	withText(R.string.notification_set_done)	withText(R.string.time_ago_now)
	5	withContentDescription("Input field 12")	**[*] withContentDescription("Input field twelve")**
26	5	withContentDescription("Input field 13")	withContentDescription("Input field thirteen")
	4a	withContentDescription(R.id.select_date)	-
27	3	withText("Set As Done")	-
28	5	withContentDescription("Seleziona mese e giorno")	-
29	6	withHint("Content")	-
30	6	withHint("false")	-

Table 3.10. Method calls causing test breakage in version B of TodoList and their refactored version

repaired: after the first ones discovered have been repaired by our tool, we need to run the test class again on the app and use the output of this second execution as input for the second iteration of our tool.

That's why the second method call of test cases number 21 and 25 have been refactored only on the second iteration.

Method calls of test cases 27-29 couldn't be repaired: we were trying to locate the widgets using the quoted string values found in the values/strings.xml file - while the widget attribute values were defined as *R.string.resource_name* (as shown in Table 3.9). This makes it impossible for our tool to correctly locate the widgets in version A and, consequently, to not repairing the test cases in version B.

3.2.2 Handling test breakage caused by widget id

A test case involving the value of the *android:id* attribute can break in two ways:

(a) The test case may not even compile because the id we are looking for is no longer present in the whole project.

(b) or it may run to completion but breaking because the widget having that id has been moved to another activity.

Test #	Reference case number	Does the method call pass on version B?	Does the method call pass on version B after the [second] refactoring?	Do the two calls locate equivalent widgets?
21	5	no	yes	yes
	2d	-	[yes]	yes
22	2a	yes	yes	yes
	4b	no	yes	yes
23	4a	yes	yes	yes
	6	no	yes	yes
24	2c	yes	yes	yes
	3	no	yes	yes
25	2b	no	yes	yes
	5	-	[yes]	yes
26	5	no	yes	yes
	4a	-	yes	yes
27	3	no	no	yes
28	5	no	no	yes
29	6	no	no	yes
30	6	yes	yes	yes

Table 3.11. Refactored test cases success and equivalence to the broken ones for the app TodoList

In case (a) the compiler output would be something like *error: cannot find symbol variable resource_name*: we can parse this output to find the reference widget in version A, but we don't have any additional information on the type of resource - id, string or others - we should look for.

In case (b), the output of the execution would look like the one that can be found in Table 2.2, in particular with reference case number 1a: we'd then have to extend our search for a matching widget in all layout files, then either communicate to the user that the widget has been moved to a different screen or repairing the test case making it perform some actions to navigate to the activity containing the moved widget and run the remaining of the test case afterward. This would be of course a complicated task, and it has not been implemented in this first version of the tool.

Chapter 4

Threats to validity

This tool has been developed to repair test cases broken by the change of some attribute values inside its layouts. However, during the implementation phase, we made some assumptions on the Android projects we were trying to repair and this may have led to a restriction of its field of applicability.

Threats to Internal validity

Internal validity threats concern the variables used in the experimental section of the study and the conclusions that are based on them.

We defined a widget as *equivalent* to another - in the same screen but different versions of the app - based on how similar their attribute values were. More specifically, the matching percentage between two widgets was calculated as shown in Equation 2.7 (Section 2.4).

The *Attribute Change metrics*, proposed in Section 2.1.1, have been computed for all attribute categories listed in Table 2.1, and for a sample set of 705 Android projects found on GitHub. Their computation could have been more precise by extending the list of considered attributes and using a larger sample of projects, but it is not proved in any way to be the best way to compute the equivalence rate between widgets.

Threats to External validity

External validity threats concern the generalizability of the conducted study to other domains or experimental subjects.

Our tool works only for applications developed in Java, but it could be in theory extended to those written in Kotlin. If we wanted to use our tool on a Kotlin based app, the modules of the tool regarding layout parsing could be kept as they are, since the layout files would still be written in XML format. Similarly, the *Test Refactoring* module (Figure 2.1) would just need some adjustment to comply with Kotlin syntax.

As *JavaParser* doesn't work on Kotlin, the only thing that would need some real effort to make our tool work on Kotlin projects, would be having an appropriate parser to analyze the code of the Activity Under Test when looking for its respective layout resource name.

The applications we used to test our tool belong to the same *Utilities* categorization (McMillan et al., 2011), but in general there should be no structural contraindications in using apps belonging to any other category.

Threats to Construct validity

Construct validity threats regard possible errors in the computational steps and in the construction of the experimental core of the study. The ones related to our tool are listed in the following.

- When comparing the layout files in the two consecutive app releases, we made the assumption that the file has the same name in both releases, but that may not be the case, so further inspection of the tested activity is needed in case this assumption does not hold;

- The tool considers only two different ways of retrieving the Activity Under Test (see Section 2.2.2), so it may fail in case said activity is declared or used in a different way (for example, in Espresso tests, with the method *getActivity()*, we could retrieve the current activity after an action has started another activity);

- If the class is not found in any of these ways, we should try to see if the test class extends another one.
- Our tool doesn't take into account the chance that a layout or parts of it are imported from another layout file (e.g. toolbars, navigation drawers, menus, card layouts for recycler views, etc.): if the widget we are looking for in version A is from an imported (included) layout, it will not be found directly in the containing layout;
- For how this is coded inside the tool, in case two widgets in version B have the same matching rate with the reference widget in version A, we select the last one that goes through the *computeMatching()* method - that computes the matching percentage as defined in Equation 2.7.

Chapter 5

Conclusions

Test classes maintenance can be a very high invoice in the application development process, especially in an industrial setting. Due to the constant evolution of the graphical user interface from a release of the app to the following one, many layout-based test cases are broken in the transition.

This thesis proposes a tool to improve automated GUI testing procedures and in particular to repair broken layout-based Espresso test cases - but the same approach could be extended to other testing tools and techniques.

In case the test is broken because a widget can't be located in the current activity layout using the value of one of its attributes - because this value has changed since the last release where the test passed successfully - the tool tries to find in the current release the widget that is most similar to the one the test located in the previous release.

This search is done based on a matching criterion between the previously located widget and those currently present in the activity layout file. The matching percentage between two widgets is computed considering the main *android:* attributes of the *View* class, where each of them has a weight computed based on how much that attribute changes during the evolution of the application.

5.1 Summary of the results

We built 30 test cases on three different applications to test our tool's features and ability to repair several breakages that could happen when transitioning to a new release of an app. The test cases were custom-built to cover all possible combinations between the 4 selected *ViewMatchers* method calls and the 3 possible arguments these methods would accept - all listed in Table 2.2.

Out of the total 40 method calls contained in these tests, only 30 needed refactoring, 24 (80%) of these were correctly repaired - i.e. the repaired method call makes the test case pass successfully and is **equivalent** to the previous one since it locates the widget with the same *android:id* in the current layout. Detailed results are shown in Table 5.1.

Application	**Method calls that needed refactoring**	**Method calls correctly refactored**	**Tool success rate on method calls**	**Test cases that needed refactoring**	**Test cases correctly refactored**	**Tool succes rate on test cases**
Lazy Cafe'	8	8	100%	7	7	100%
Glucosio	10	7	70%	9	7	77,78%
TodoList	12	9	75%	9	6	66,67%
Total	30	24	**80%**	25	20	**80%**

Table 5.1. Tool results on the selected applications and built test cases.

We also noticed that, based on how many broken widget locators are present in a test case, we may need more that one iteration of our tool to be able to correctly refactor all the broken method calls.

5.2 Future work

In the following, we list a few improvement points for this tool. These can be aimed towards increasing its precision in locating equivalent widgets or simply ideas for possible additional features.

- To compute the matching percentage between two widgets, we analyzed only the most common widget attributes based on a visual analysis of the layout files (listed in Table 2.1): by the way, it would be possible to make a more

extensive analysis on actually all the possible layout widget attributes (so those of the View class and of all of its subclasses);

- Extending the tool to take into account the chance of imported/included layouts within other layout files, to avoid a Construct threat to the tool's validity;
- Having a deeper knowledge of *JavaParser* and its implementation, we could make the user choose if they want to repair a single test case instead of the whole test class;
- In case the wrong widget was selected for refactoring and the user discards it, we could also consider the widget with the second-highest matching. This feature would have correctly repaired a few method calls that the tool wasn't able to, such as test cases number 15 and 17 performed on the Glucosio app (as we can see in Table 3.8);
- If a test class does not compile because an id or string resource is no longer present in the project, as written in Section 3.2.2, we can parse the compiler output to retrieve the resource name that can't be found in version B, look for the reference widget in version A and then run our tool looking for a matching widget - possibly on all layout files inside the project;
- Handling the case of a widget that has been moved from an activity layout to another. It could be done asking the user if they want to extend the search to other activities layouts (in case no widget with a match above the threshold is found - or in general, because the same widget may just have been moved, with no modifications to the values of its attributes).

Bibliography

Alégroth, E., Feldt, R., & Kolström, P. (2016). Maintenance of automated test suites in industry: An empirical study on visual gui testing. *Information and Software Technology*, *73*, 66–80.

Berner, S., Weber, R., & Keller, R. K. (2005). Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on software engineering* (pp. 571–579).

Chang, N., Wang, L., Pei, Y., Mondal, S. K., & Li, X. (2018). Change-based test script maintenance for android apps. In *2018 ieee international conference on software quality, reliability and security (qrs)* (pp. 215–225).

Coppola, R., Ardito, L., & Torchiano, M. (2019). Fragility of layout-based and visual gui test scripts: an assessment study on a hybrid mobile application. In *Proceedings of the 10th acm sigsoft international workshop on automating test case design, selection, and evaluation* (pp. 28–34).

Coppola, R., Morisio, M., & Torchiano, M. (2017). Scripted gui testing of android apps: A study on diffusion, evolution and fragility. In *Proceedings of the 13th international conference on predictive models and data analytics in software engineering* (pp. 22–32).

Daniel, B., Gvero, T., & Marinov, D. (2010). On test repair using symbolic execution. In *Proceedings of the 19th international symposium on software testing and analysis* (pp. 207–218).

Daniel, B., Jagannath, V., Dig, D., & Marinov, D. (2009). Reassert: Suggesting repairs for broken unit tests. In *2009 ieee/acm international conference on automated software engineering* (pp. 433–444).

Daniel, B., Luo, Q., Mirzaaghaei, M., Dig, D., Marinov, D., & Pezzè, M. (2011). Automated gui refactoring and test script repair. In *Proceedings of the first international workshop on end-to-end test script engineering* (pp. 38–41).

Gao, Y., Liu, H., Fan, X., Niu, Z., & Nyirongo, B. (2015). Analyzing refactorings' impact on regression test cases. In *2015 ieee 39th annual computer software and applications conference* (Vol. 2, pp. 222–231).

Grechanik, M., Xie, Q., & Fu, C. (2009). Experimental assessment of manual versus tool-based maintenance of gui-directed test scripts. In *2009 ieee international conference on software maintenance* (pp. 9–18).

Hammoudi, M., Rothermel, G., & Tonella, P. (2016). Why do record/replay tests of web applications break? In *2016 ieee international conference on software testing, verification and validation (icst)* (pp. 180–190).

Imtiaz, J., Sherin, S., Khan, M. U., & Iqbal, M. Z. (2019). A systematic literature review of test breakage prevention and repair techniques. *Information and Software Technology*, *113*, 1–19.

Jobe, W. (2013). Native apps vs. mobile web apps. *International Journal of Interactive Mobile Technologies*, *7*(4).

Joorabchi, M. E., Mesbah, A., & Kruchten, P. (2013). Real challenges in mobile app development. In *2013 acm/ieee international symposium on empirical software engineering and measurement* (pp. 15–24).

Leotta, M., Clerissi, D., Ricca, F., & Spadaro, C. (2013). Improving test suites maintainability with the page object pattern: An industrial case study. In *2013 ieee sixth international conference on software testing, verification and validation workshops* (pp. 108–113).

Li, X., Chang, N., Wang, Y., Huang, H., Pei, Y., Wang, L., & Li, X. (2017). Atom: Automatic maintenance of gui test scripts for evolving mobile applications. In *2017 ieee international conference on software testing, verification and validation (icst)* (pp. 161–171).

Linares-Vásquez, M. (2015). Enabling testing of android apps. In *2015 ieee/acm 37th ieee international conference on software engineering* (Vol. 2, pp. 763–765).

Linares-Vásquez, M., Moran, K., & Poshyvanyk, D. (2017). Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 ieee international conference on software maintenance and evolution (icsme)* (pp. 399–410).

McMillan, C., Linares-Vasquez, M., Poshyvanyk, D., & Grechanik, M. (2011). Categorizing software applications for maintenance. In *2011 27th ieee international*

conference on software maintenance (icsm) (pp. 343–352).

Mongiovi, M. (2011). Safira: a tool for evaluating behavior preservation. In *Oopsla companion* (pp. 213–214).

Soares, G., Cavalcanti, D., Gheyi, R., Massoni, T., Serey, D., & Cornélio, M. (2009). Saferefactor-tool for checking refactoring safety. *Tools Session at SBES*, 49–54.

Stocco, A., Yandrapally, R., & Mesbah, A. (2018a). 12 vista: Web test repair using computer vision. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 876–879).

Stocco, A., Yandrapally, R., & Mesbah, A. (2018b). Visual web test repair. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 503–514).

Tomassetti, F., Smith, N., Maximilien, C., & Kirsch, S. (2017). *Javaparser.*