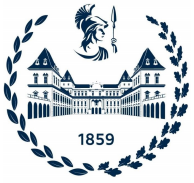


POLITECNICO DI TORINO

Master's degree course in Computer Engineering

Master's Degree Thesis

TOGGLE: a library for lightweight and automated generation of mobile visual GUI test sequences

Supervisors
prof. Luca Ardito
dr. Riccardo Coppola

Candidate
Vittorio Di Leo
Student id: 267584

Academic year 2020-2021

Summary

Mobile applications and mobile devices are becoming more and more pervasive in our everyday life supporting a wide range of activities in a variety of contexts. An "app" is a highly event-oriented software thought to react to a set of inputs typical of the mobile world. Furthermore, mobile devices themselves are characterized by a mixture of attributes that can affect in many ways how a graphical user interface (GUI) is displayed. This highlights the need to have applications that have been tested extensively and from different points of view, from their back-end to their presentation layer.

Despite these factors, Visual GUI Testing (VGT) is not a common activity in the mobile environment, especially in an industrial context. The existing tools introduce a collection of different approaches to this activity, each one with a specific testing goal and each one suffering from well-known weaknesses.

These frameworks can be categorized into three groups, according to how they identify graphical elements on the screen. First-generation VGT tools focus on interacting with the bitmap layer of the application's front-end, second-generation ones directly access the app's visual components to extract locators, and third-generation ones use image recognition algorithms to match areas on the screen of the host machine.

Issues that make VGT not feasible in an industrial development process are the diversity of both mobile operating systems and mobile hardware models, which makes tests not portable, the scarce performances, the test flakiness, and, finally, the tests fragility that makes their maintenance during lifecycle of a project unsustainable.

The focus of this thesis work is TOGGLE, a tool supporting the maintenance and the creation of VGT in the Android ecosystem, which covers the largest percentage of the market. Besides, TOGGLE must be used in combination with an Android Virtual Device (AVD) running on a personal computer.

A popular framework in the Android ecosystem is Espresso, a tool to

generate both first- and second-generation tests, that represent the starting point of TOGGLE. The main purpose of the framework is to automatically translate second-generation Espresso tests into third-generation ones. These tests can detect graphical features, such as the modification of the color of a button, that the source ones cannot identify, enlarging the overall testing capability of the suite. Third-generation tests use the Java APIs offered by SikuliX and EyeAutomate libraries.

At its state-of-the-art TOGGLE completed the translation while showing several weaknesses impacting both its usability and portability in a non-academic context.

The main purpose of this thesis work was to make TOGGLE a library that testers could use when performing VGT in an industrial development process. To achieve this result, some interventions on some of TOGGLE's most problematic aspects were required to increase the flexibility of the modules composing the system and their APIs, strengthen the translation flow against some of the weaknesses typical of this context, and add new features to increase the portability of the intermediate results. To summarize, the final goal was to make the tool as easy as possible for the end-user. Additionally, the Façade structural design pattern has been adopted to offer a clean and simple interface to the system, with methods suited for both casual and more experienced users.

Finally, the set of translatable inputs has been enlarged, adding the possibility to emulate the scroll interaction on the screen of the AVD. This is a typical mobile interaction that requires a dedicated translation mechanism.

The final version shows some considerable improvements, as will be presented in chapter 5. Particularly, the tool reached a 100% percentage of successfully translated tests, 99.09% of successful interaction emulation during the translated tests execution, and an overall 3^{rd} generation tests success percentage of 98.00%.

The experience gained during the empirical evaluation of TOGGLE proves that it is feasible to have robust and automatically generated third-generation test suites. Additionally, the tool tool has proven to be easy-to-use, requiring low effort for translating and executing the third-generation test suites. Moreover, the re-engineered version of the tool can be the foundation to further extensions enriching, for example, the collection of applications that could exploit TOGGLE or even adding the support for different second-generation test syntaxes. I hope that this work could help TOGGLE give its contribution to increasing the adoption of VGT in industrial projects.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past few years, mobile application development has been one of the main trending technologies, growing constantly year by year.

More in detail, devices running the Android Operating System occupy a high percentage of the market and this led to an increasing interest coming from all kinds of companies, from automotive to financial ones, from IT ones to public authorities.

The outcome is that devices running the Android Operating System pervade our everyday life, acquiring a fundamental role in it. This leads to the need of having well-tested and reliable applications. While it is an easy task to find tools that may help a developer testing his/her code, it is harder to find reliable ones that could help when performing end-to-end testing of mobile applications. Since this world is dynamic and fast-changing, it is not feasible to perform this kind of testing in a non-automated way. This often results in software that has not been tested deeply, either behaving unexpectedly due to some undiscovered flaws or, more in general, having a sequence of interactions causing a different result compared to the one the developers had in mind.

It is in this context that TOGGLE [12] (i.e. Translation Of Generations of GUI testing at Low Effort) places itself, trying to compensate for the lack of tools that can make E2E testing automated and easy to maintain during the life-cycle of the project. Its purpose is to generate third-generation test cases from second-generation ones automatically, lessening the effort needed to maintain them.

Generally, every generation of GUI tests uses some sort of locators, to isolate a graphical element, and of oracles, or expected outputs, to verify that the interaction has produced the expected result. According to the

testing approach under examination, both these elements will use a different factor to evaluate the success or the failure of the assertion.

The *second-generation test cases* group the layout-based tests of an Android application, namely the ones that, starting from a model of a Graphical User Interface decomposed in a hierarchy of layout components, can access the attributes of each one of them and use their values as locators (property-based locators), so that it is possible to isolate an element to be used as an interaction starting point, or as an oracle, verifying the result of a sequence of interactions based on the state of that component. These tests do not make any kind of assumptions on how the View is implemented or on what kind of algorithm is running in the background, their main goal is to test the output of a sequence of interactions. One of the most important and used frameworks to write second-generation tests is *Espresso*. Moreover, *Espresso* is the layout-based testing tool that TOGGLE can interact with.

The *third-generation test cases* are, instead, tests that, using screen-captures both as locators and as the expected output, can interact with a Graphical User Interface emulating the behavior of a real end-user, thus being the ones that are best suited for E2E testing. This makes them completely independent from the application code but unable to perform any kind of assertion on individual component objects since all the assertions that this kind of tests perform are based on the visual appearance of the application under test. This characteristic makes this approach unstable, needing constant maintenance and being deeply dependent on the machine the test is run on.

The consequence is that these tests usually require a maintenance effort that is often too high (surely higher than the second generation ones) and, therefore, they are not adopted in an industrial environment.

The two main tools for executing third-generation test cases that are presented in this paper are *SikuliX*[17] and *EyeAutomate*[3].

SikuliX is a tool using image recognition, powered by OpenCV, to identify any kind of GUI component or image displayed on the desktop of the computer. It is an evolution of another open-source project named *Sikuli* that was started by the User Interface Design Group at MIT in 2009. It supports many scripting languages and can be used with any Java-aware programming language. Once it has recognized an image, it can emulate any kind of interaction (click, type, double-click, and more).

EyeAutomate is a similar tool that, through image recognition, aims at automating any user scenario, seeing the application as a black box. It is a Java application and, for this reason, it can be used on any platform supporting Java. It uses both pixel-based and vector-based image recognition

in combination with AI.

In the following chapters, it will be introduced an analysis on both GUI testing and mobile application testing techniques, TOGGLE will be presented as it was before being re-engineered (by listing its features, its flaws, and all the challenges faced to improve it), then all the interventions on it will be presented, explaining what were the final goals and all the reasons behind every modification. Finally, I will present the results of an empirical application of the tool of a five open-source real applications.

Chapter 2

Background

Visual GUI Testing is a high-level testing approach for GUI-based software. It is known to have great potential since it allows to work with closed systems without needing access to their APIs.

For this reason, it is mostly code-independent since its main focus is the front-end of a system. With this approach tests usually ensure that: the application under test (from now on AUT) respects a given visual design (e.g. that each visual element is actually displayed at given coordinates and/or with given dimensions), messages for the end user are/are not displayed, a certain set of interactions result in the proper expected functionality, radio buttons or dropdown menus or other graphical elements are correctly aligned, the color of each visual element respects the theme of the AUT and more.

Succinctly, this approach makes it easier to test the system in its entirety to guarantee that the developed software follows its requirements.

Naturally, this approach presents some limitations that have discouraged its diffusion in an industrial environment. The outcome is that there is a lack of empirical data documenting both its applicability for testing purposes and its general cost for maintenance and development.

In general, it has been documented that Visual GUI Testing suffers from high sensitivity to GUI layout changes, it is deeply dependent on characteristics of the screen (pixel density, zoom factor, width, height...) and, if not automated, the development of a full set of tests is costly, time-consuming and highly error-prone.

Test automation, in this context, has been considered but there are still no frameworks making this kind of test actually feasible in an industrial ecosystem. It has been observed that it is more challenging to automate high-level tests, because they depend on a greater number of functional components of

the system.

2.1 GUI Testing

As previously stated a lot of effort (coming especially from the academic world) was put into the research for the best way to perform GUI testing and for an efficient way of automating this kind of tests. This resulted in a plethora of different approaches to GUI testing[22][14]:

Manual testing : this is the most basic approach. It involves a human tester that manually creates and executes test cases. Even if it requires the highest effort, it is convenient when the GUI is at the initial stage of development and it is likely to change in a short time. Manual testing allows developers to perform quick checks that can be done at any moment. This approach doesn't support any form of regression testing, meaning tests aiming at making sure that a change in the program has not adversely affected the existing features, and it is not scalable. Furthermore, the tester is required to have some expertise to be able to write tests that actually validate the design of an element;

Capture/record and Replay (from now on R&R): it is, as the name suggests, divided into two main phases. During the first phase (record phase), the user performs some interactions with the GUI and the framework captures and translates them automatically into a script. During the second one (replay phase), the script is replayed to repeat the recorded interactions. This approach supports automated regression testing and can interact with different windows as long as they are displayed on the screen. The drawback of R&R is that it is hard to maintain and it does not work well with tests requiring a lot of interactions with the GUI of the AUT. R&R can work at different levels of abstraction, from GUI bitmap level, where the framework interacts directly with the bitmap layer on the screen and works with coordinates, to the widget/-component level where, during the capture/record phase the information of the widget the test is interacting with are extracted and then, during the replay phase, these data are used to correctly locate them. Each level of abstraction has its weaknesses and strengths: the higher the level the more it suffers from GUI layout changes while being robust to code modifications, the lower the more it suffers from changes in the code structure or the APIs while being robust to GUI layout modifications. Several frameworks supporting this testing technique exist. Some

examples are *Espresso Test Recorder*[1], *Robotium Recorder*[2], *RERAN*[3], *Barista*[4] and *Xamarin Test Recorder*[5];

Model-based testing : this approach focuses on creating a model that will be used to understand and evaluate the behavior of the system. After having defined a model a tool should be used to automatically generate test cases from it. To make this possible it should be used either state charts representing the state of the system in each step or decision tables showing the results for each input. The generated test cases are usually filtered by a selection algorithm to discard all the inadequate ones.

Scripted testing [13]: this approach is, probably, the most common one. It requires that testers programmatically write down test cases emulating a sequence of interactions with the AUT GUI while having direct access to the code of the application. For this reason, they are often classified as white-box testing, meaning that they are written with deep knowledge of the actual implementation of the software. They enable regression testing but, since they have to be constantly and manually maintained during the whole lifecycle of the software under examination, they are costly and time-consuming.

Since the mobile world is the main focus of this paper, it is important to say that it presents some additional challenges and tools to perform visual GUI testing.

2.2 GUI Testing for Android applications

As previously stated, the mobile world is in rapid expansion and it needs tools that can help developers writing quality software. Testing is a key activity in the software development process and needs to be supported.

For Android application development, the general suggestion is to use well-tested libraries (like, for example, the Jetpack[6] ones) and to follow a

[1]https://developer.android.com/studio/test/espresso-test-recorder

[2]https://robotium.org/

[3]https://www.androidreran.com/

[4]https://github.com/AdevintaSpain/Barista

[5]https://marketplace.visualstudio.com/items?itemName=XamarinInc.XamarinTestRecorder2015

[6]https://developer.android.com/jetpack/androidx/explorer

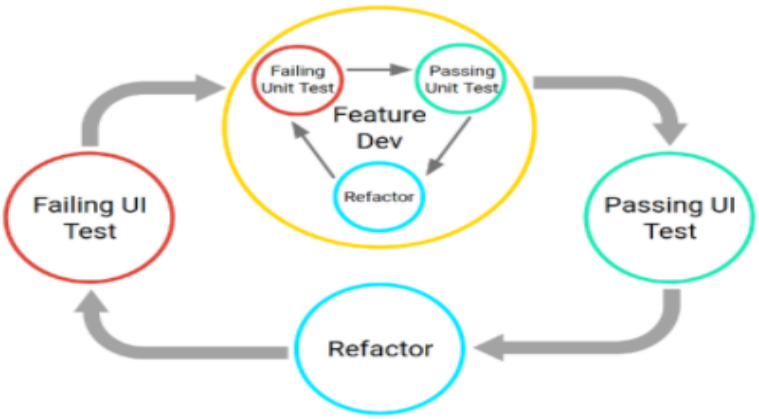


Figure 2.1. Iterative test driven development. From : https://developer.android.com/training/testing/fundamentals

test-driven development approach [5], organizing the code in a way that will ease the testing activity and performing unit tests during development and UI tests after passing the previous ones without failures.

Moreover, the test environment should be set following some best practices that are often enforced automatically by the integrated development environment (IDE). A common IDE when developing an Android application is Google's Android Studio that organizes tests in two different folders named:

androidTest : the directory where tests running on real devices or emulated ones are placed. Typically, here integration tests, acceptance tests, end-to-end tests, GUI tests could be found;

test : the directory where tests that run on the local machine are placed (typically unit tests).

Since mobile applications are strongly event-driven (both coming from users and environmental changes detected from a set of sensors and hardware components) then GUI testing has become more and more relevant.

Some examples of types of tools introduced to enforce this approach in the mobile world are[20]:

Automation APIs/Frameworks : they offer an efficient and powerful way of obtaining information on the visual hierarchy of GUI components and let testers interact with them programmatically, emulating the behavior of a user. Moreover, these tools offer some APIs to perform some assertions on the state of the components. These tools typically provide cross-device compatibility and support for basic user interactions and APIs, while still being susceptible to the difficulty of emulating complex user actions (such as scrolling, zooming, pinching, and more) and to the high maintenance cost. Some examples of frameworks that fall under this category are UIAutomator[7], Espresso[2], Appium[8], Robotium[9].

Record & Replay : this kind of framework has already been presented. As for the previous cases, in the mobile environment, they are an alternative to manual test scripts generation but they require a more limited knowledge to be used. They suffer from the fragmentation problem, since they often interact with the Linux kernel of the mobile device without having any knowledge of the actual GUI component and, for this reason, they are often tied to the screen size and density of the device. Another problem to be considered is that the higher the level of user action representations the more limited is the accuracy and the timing of events. Tools following this approach have already been presented in section 2.1.

Automated Test Input Generation Techniques : these tools automatically generate input for test scripts to lower the time spent by testers in creating test cases. They are usually used to reach a particular goal like achieving a high percentage of code coverage or discovering the highest possible number of bugs. Their main weaknesses are that it may be difficult to generate system events, it is costly in terms of performance to restart the application, certain complex inputs usually need to be specified manually, it is unlikely to have reproducible tests, and cross-device testing is not supported. This kind of tool may be further categorized into three different classes according to their input generation

[7]https://developer.android.com/training/testing/ui-automator

[8]https://appium.io/

[9]https://robotium.org/

technique: *random-based*, *systematic*, and *model-based*. Some examples of tools that fall under this category are Monkey[10], AndroidRipper[11], Google RoboTest[12], AppDoctor[13].

Bug and Error Reporting/Monitoring Tools : they are a common category of tools that have been included, in the past few years, in a lot of mobile testing workflows. As the name suggests, the main goals of these tools are bug reporting and error and resource consumption monitoring. The ones aiming at bug reporting, typically, describe issues with textual reports and, eventually, other additional files (like screenshots). The others may be included by developers in the application, otherwise, the only in-field option to retrieve this information are reviews (taking into account that real users usually don't have the required knowledge to provide the context to identify and solve the error) or automated crash reports (that often are not good enough). Some solutions to these limitations have been introduced but they all include the need for API calls to methods provided by third-party services, in order to collect data that may actually help developers locate the issue, and they are limited to crash reporting. Some examples are Airbrake[14], Appsee[15], BugClipper[16].

Mobile Testing Services : this is a crowd-based approach that, exploiting the effort of a delocalized group of workers (that may or may not be actual testers), lowers the cost of test case generation by splitting it among a large number of people. The main goal of these tools may vary among the following options: collecting bug reports to discover the maximum number of software flaws, measuring the usability of an application, stressing the security mechanisms of the AUT, ensuring that the AUT will work in different geographical regions. None of these tools is open source and typically they are not scalable since it is difficult to find a place for them in an agile development approach, which is

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

[11]https://github.com/reverse-unina/AndroidRipper

[12]https://firebase.google.com/docs/test-lab/android/robo-ux-test

[13]https://www.theappdoctor.com/

[14]https://airbrake.io/

[15]https://www.appsee.com/

[16]https://bugclipper.com/

becoming more and more common in software teams. Some tools of this type are Xamarin Test Cloud[17], Google Firebase[18], TestArmy[19], Apperian[20], Apptimize[21].

Device Streaming Tools : the purpose of this category of tools is to allow developers to mirror a device connected to their PC over the internet and to access it remotely. These tools could be used in combination with the previous category to stream secured devices to remote testers. Some examples of tools are OpenSTF[22], Appetize.io[23].

As for the actual test scripts, a general categorization that will be used in the remainder of this paper classifies them into:

First-generation tests : these are tests generated with frameworks that work with the bitmap layer of the front-end of the application;

Second-generation tests : they work directly on the visual components and their state. They are faster and more reliable but need the tester to have deep knowledge of the GUI of the application;

Third-generation tests : these tests are based on image recognition algorithms and have the highest abstraction level concerning the AUT. They are more flexible but very fragile and costly to be maintained.

The focus of this thesis work is on *Espresso*, a framework producing tests of second-generation that will be the input of TOGGLE. The tool will then generate tests of third-generation automatically, easing their maintenance cost.

In conclusion, other than the ones presented in this section, a lot of different frameworks and approaches exist and there is no absolute best choice. For this reason, while designing a mobile application, it is advisable to perform an analysis on what approach should be taken during Visual GUI Testing

[17]https://testcloud.xamarin.com/

[18]https://firebase.google.com/

[19]https://testarmy.com/en/

[20]https://digital.ai/app-management

[21]https://apptimize.com/glossary/ab-testing/

[22]https://github.com/openstf/stf

[23]https://appetize.io/

and on what should be actually tested according to both the specific testing needs and the team's knowledge of the adopted tool(s). Furthermore, a key point in this analysis should be the study about the limitations that each different approach brings with it to evaluate the effort required, both in developing the test scripts and in maintaining them, through the whole life-cycle of the application.

2.3 Issues for Android GUI testing

In the previous section, a lot of characteristics of Android GUI testing tools and techniques have been presented. Among those, it is possible to find some weaknesses that are common to all frameworks.

These issues are the reason why GUI testing is still struggling to find its applicability in industrial environments. In this section, the main problems will be presented and further detailed.

2.3.1 Fragmentation

One of the first weaknesses to be considered, and probably the most intuitive one, is *fragmentation*[19]. The fragmentation problem is not exclusive of the mobile world but, in this context, it found a greater emphasis.

By fragmentation, we mean the problem connected with the wide variety of devices supporting the Android OS and of the operating system itself. Furthermore, with the recent introduction of devices such as smart TVs, this issue has gained even more relevance than before.

With this premise, it is clear that nowadays an Android application must be developed keeping in mind that, in order to reach the majority of the market, it should run on a high percentage of the available devices without incurring any malfunctioning. Moreover, since OS versions change over the years the same device might run different OSs.

The fragmentation issue comes with important consequences for the testing practice too. A crowd-based approach might be used but, as previously stated, it is not compliant with agile DevOps practices.

It is easy to understand that this issue is a great challenge while performing Visual GUI Testing. With different configurations, there could be different screen dimensions, different screen densities, different Android APIs. This results in different visual appearances for the same layout. Moreover, if, while designing an application, some best practices are not followed, the same

visual component may appear larger on screens with a lower pixel density and smaller on screens with a higher one. GUI tests run the risk of being tied to a single configuration or, alternatively, they require the same test cases to be written for all configurations we expect our application to be executed into. Both options are, in the majority of cases, not good enough for the first option and not feasible in terms of maintainability, cost, and effort for the second one.

2.3.2 Test flakiness and performance

It is safe to assume that visual GUI tests have performance issues and this happens for many reasons. Firstly, knowing that they emulate user's interactions, they need more time to be executed than, for example, unit tests.

Some tools introduce by default a time-out between each instruction to correctly synchronize them that may or may not be customized.

Some other tools rely on an image recognition algorithm that needs a high computational effort (and, consequently, it takes more time to be executed) to locate the visual components on the screen.

Finally, the performance of this kind of test may be influenced by dependencies on external services. The latter introduces another related weakness of visual GUI tests.

This issue was highlighted by the diffusion of cloud-based web services. Even if the trend of both computational and storage capability of mobile devices is growing, sometimes applications rely on remote services. This happens for many reasons: the computational cost of an operation is too high to be performed on the device, the user wants to store some data in a shared repository or the developer decided to use a third-party service.

This comes with the introduction of non-predictable delays in the behaviors and outcomes of the application because the execution time is impacted by race conditions, loss of connectivity, data integrity issues, response timeouts. Moreover, for frameworks using image recognition algorithms, the transition from one state of the application to another may take more time than the one required to locate the image on the screen. This misalignment is hard to quantify and compensate for and may cause unpredictable testing results as well. The outcome is that the testing activity becomes, of course, more challenging, since non-determinism in the behavior of the application often results in test cases failing or succeeding due to external conditions. This issue is known as test flakiness.

Sources of flakiness may be, for example, codified response delays, lack of

identification mechanisms for unexpected behaviors, lack of mechanisms supporting different device states according to the number of remote resources available, lack of support for internal unpredictable events like the Garbage Collection.

Even if non-determinism does not impact exclusively Visual GUI Tests it is still an important problem in this context since, when writing test scripts, it is necessary to assume that a fixed set of interactions produces a predictable graphical result.

2.3.3 Fragility and maintainability

These are, probably, the most impacting weaknesses in this context. It is well known that GUI tests are inherently unstable and fragile. No approach can be considered immune to this problem and, as previously stated, this is one of the main reasons why GUI testing has not been widely adopted in every mobile application development process.

The reasons behind this fragility are multiple and not uniform among the various frameworks. A tool that has a higher-level view on the AUT is susceptible to GUI layout changes, theme changes, textual changes and, generally, any kind of graphical modification. A tool that, instead, has a lower-level view on the AUT is susceptible to APIs changes or, for example, modifications of the attributes of a resource.

Moreover, the mobile world is a dynamic ecosystem that is constantly evolving. This implies that applications too are often fast-changing both in terms of functionalities offered and in terms of graphical features, especially in the first phases of their life cycle.

To these problems, the fragmentation issue in Android devices and the ones connected to test flakiness (sections 2.3.1 and 2.3.2) should be added to have a clear picture of how many factors contributes to increasing the instability of GUI tests.

The most obvious consequence of test fragility is the high effort and cost they require to be maintained during the AUT life-cycle. It is not feasible to maintain and update tests that need to be re-written from scratch for each different device or each new feature introduced. Tools like *Espresso* partially try to decouple test scripts from device characteristics, but they are deeply connected to attributes of the View like text, ids, hints, and, consequently, they are fragile to internal changes of the application.

Another consequence of this problem is that regression testing might become an unfeasible activity. Tests passing on previous versions of the applications and failing with the most recent one do not necessarily highlight a real malfunctioning. Instead, their failure may be due to an error in the test scripts themselves (that, for example, may not be able to recognize an updated View).

Chapter 3

Adopted tool: state of the art

In the context presented in chapter 2 TOGGLE finds its place. TOGGLE is a framework designed to lower the maintainability effort and the cost required to perform the transition from second-generation test cases to third-generation ones.

The adopted tool to create second-generation tests is *Espresso*, while the tools used to run third-generation ones are *SikuliX* and *EyeAutomate*.

The reason why TOGGLE focuses on the transition from second- to third-generation tests is that the two methodologies have complementary advantages. The first category allows testing the values taken by the attributes of a View in the displayed layout like its id, its type, its position in an AdapterView, and similar characteristics not directly related to the visual appearance of the component. On the other hand, the second category, since it does not depend on any internal attribute, makes it possible to test how each component is displayed on the screen, whether it respects or not the theme of the application or if a given interaction on a specific View will result or not in the expected output on the screen.

Furthermore, running a test suite containing tests from both generations will increase the probability of identifying faults that would be impossible to discover using exclusively only one of the two approaches. More in detail if, in presence of a fault, both second- and third-generation test scripts fail it means that the visual component under test has radically changed or it is not displayed. When, in presence of a fault, only one of the two generations fails while the other one succeeds there could be two cases according to which one is failing:

- **Second generation test failure**: the fault is about a specific attribute of a View that might not be graphically displayed, like a non-existing id or a wrong class type of entries inside an AdapterView.
- **Third generation test failure**: the fault is related to the visual appearance of the component like, for example, a change in the color of a View or a wrong image is displayed inside an ImageView.

The last combination of events is when a fault is present but none of the generations detects it. This is an unlikely scenario.

The problem in coupling second- and third-generation tests is that the translation process is cumbersome since the testing objectives behind each generation are typically different, so the tester should be aware of what it is the actual purpose of the test before performing the translation. Additionally, the translation process itself is hard to perform because a second-generation assertion often results in several third-generation instructions. This problem is even more highlighted in the context of this case study since the translation, in this context, happens for tests that should run on an emulated device. This means that third-generation tools should reproduce interactions that are not standard ones on the host machine (typically a personal computer) but are the ones typical of the mobile world (scroll, tap, double-tap, swipe. and similar).

Finally, another issue is that it is even harder to maintain two different sets of Visual GUI Tests manually, especially third-generation ones that are the ones requiring the highest effort.

The main purpose of TOGGLE is to remove the translation cost from the picture by automating this activity, easing the combined adoption of second- and third-generation tests. The result is a deeper and more accurate GUI testing suite for the AUT.

3.1 Espresso

Espresso [2] is an open-source framework, developed by Google to test Android user interfaces, that supports JUnit4. It is strongly encouraged to use it as a white box testing tool, to be able to exploit all its potential even though it is possible to use it to create black-box tests too.

Espresso has become one of the most relevant testing tools for many reasons. For starting, by being developed by Google it is well integrated and supported in Android Studio (probably the most common IDE for Android

applications development) and the Android ecosystem as well. This means that it is aware of the internal mechanism of the Android Operating System, hiding the complexity needed to synchronize the test interactions: an interaction won't be performed until the message queue of the main thread of the application is empty, no instances of AsyncTask are currently executing a task, and all the resources that should be idle are actually idle. This automatic synchronization mechanism makes Espresso robust against some sources of test flakiness presented in section 2.3.2.

Furthermore, it has a clean and simple set of APIs with a smooth learning curve offering to the tester a flexible set of operations other than the possibility of validating and/or mocking Android Intents.

Tests written using Espresso are instrumented ones, meaning that they need the application to be installed and launched. The Android testing framework offers the AndroidJUnitRunner to run Espresso tests. This runner automatically handles the loading of both the Espresso test cases and of the AUT on the device (emulated or not), then it runs the tests collects the results.

This framework has four main components with different functionalities:

- **Espresso**: the starting point of every interaction offering methods like *onView()* and *onData()* to perform assertions or actions on the Views on the screen. Moreover, it offers some other APIs to perform actions that do not operate on any view like *pressBack()*;

- **ViewMatchers**: classes whose main purpose is matching UI components in the visual hierarchy by analyzing different attributes like their id, their text, their class, whether they have or not another child View inside themselves, and more;

- **ViewActions**: classes whose main purpose is to perform actions on a View. Objects of this class can be passed to the *perform()* method of the ViewInteraction class.

- **ViewAssertions**: classes whose main purpose is to assert that the actual Views on the screen (found through the usage of ViewMatchers) and the expected ones are the same. Objects of these classes can be passed to the *check()* method of the ViewInteraction class.

3.2 SikuliX and EyeAutomate

For what concerns third-generation tests the supported tools are SikuliX and EyeAutomate.

SikuliX [18] is a framework that aims at locating and performing actions on GUI components that are displayed on the screen of the machine executing the test script. Each script has a statement file and zero or more images. The statement file is a combination of instructions to wait for an image to be displayed on the screen with a customizable time out, actions to be performed on the located visual component, and other instructions implementing, for example, decisions and repetitions statements. Images are used as locators.

The image recognition algorithm is highly dependant on the number of pixels of the images. This means that a test will fail if the portion of the screen that it is trying to locate does not correspond to the relative image dimensions (with, of course, a small tolerance margin).

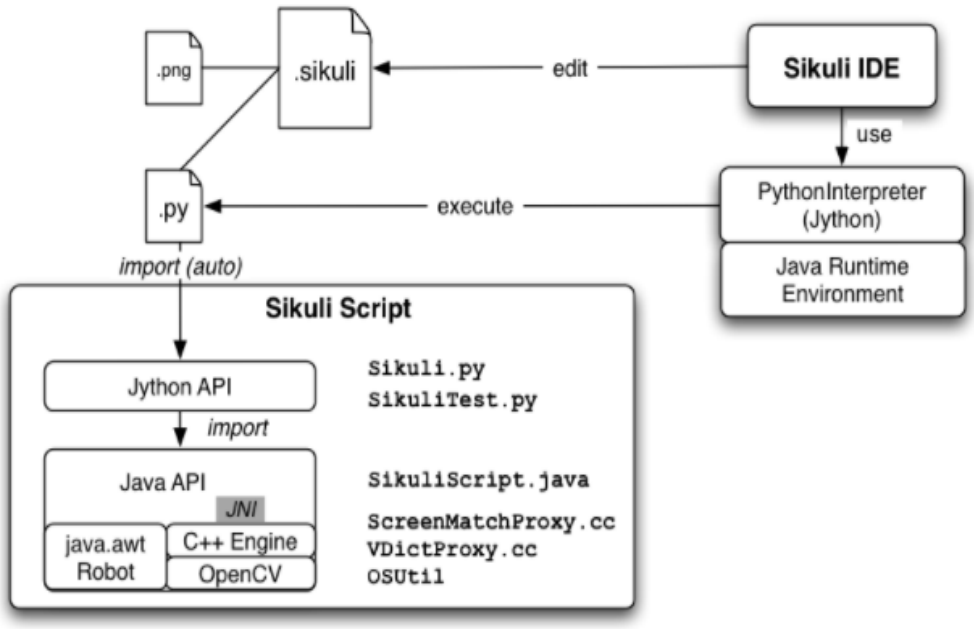


Figure 3.1. Sikuli architecture[1].

The core of this framework is the internal Java library that is composed of two different sections:

[1]https://sikulix-2014.readthedocs.io/en/latest/devs/system-design.html

- *java.awt.Robot*: a component in charge of handling mouse and keyboard events;
- *C++ Engine*: a component using the OpenCV library to locate on the screen the visual elements captured in the images of the script.

The two components interact through the Java Native Interface, a framework used to call native code (code specific for a given operating system) or, more in general, code written in other programming languages like, in this case, C++.

Additionally, above the Java API, a Jython layer is in charge of showing to the end-user simple and easy-to-use commands.

SikuliX provides APIs for several programming languages like Javascript, Python, Java, and Ruby. This case study will focus on the Java APIs that could be used by adding *sikulixapi.jar* to the dependencies of the project.

EyeAutomate [4] is a visual GUI testing framework completely developed in Java. For this reason, it can be executed on every platform supporting this programming language.

The core of this tool is called the "Eye" and it envelops the image recognition algorithm used during the execution of test scripts. This algorithm uses all available CPUs and, for this reason, its execution time depends on the resolution of the screen and the number of available CPUs. A visual test script is composed of four kinds of files:

- Textual files (*.txt): the ones containing the script itself;
- Image files (*.png): the ones containing the image to be recognized on the screen;
- Data files (*.csv): the ones containing data to be passed to the script, making data-driven testing possible. Each line will contain a parameter to be sent to the script and the framework will execute it one time for each row;
- Widget files (*.wid): the ones containing a script with some prioritized properties (that could be an image, a location, or an identifier) used as locators in a given interaction. Each line in the widget will be executed until a locator successfully locates the required visual element.

Inside an image, the selected area is a round section and the recognition will succeed or not depending on a correspondence that should be at least

equal to 95%. In case of multiple matches, all of them will be returned ordered by their distance from the current position (the upper-left corner of the screen at the beginning of the test script).

The image recognition AI supports the recognition of small screenshots by dividing an image into small areas of interest and then by searching for the best match. Moreover, this framework supports both vector- and pixel-based image recognition. The first one is less sensitive to zoom factors and to rendering details. The outcomes of each test script can be verified thanks to the following log files:

- execution_log.txt: a file where all the executed instructions are registered and where, in presence of a failure, an error message will be added;
- text_log.txt: a file where dates and hours of every test execution are registered;
- test_history.csv: a file listing all runs for a given script;
- test_steps.csv: a file listing all commands in a script.

It is possible to use EyeAutomate APIs from Java including the *EyeAutomate.jar* file to the dependencies of the project.

3.3 TOGGLE

The purpose of TOGGLE is to translate Espresso tests into both SikuliX and EyeAutomate scripts and into Java classes implementing the equivalent version of third-generation scripts, using instructions coming from EyeAutomate Java APIs, from SikuliX Java APIs, or from both libraries (in this case the test firstly tries one approach and then, if this fails, it tries with the other one).

Picture 3.2 shows the high-level architecture of this tool. The main components are:

- **Enhancer**: this component oversees parsing 2^{nd} generation tests, understanding the operations that are being executed, and injecting the lines of code that are necessary to later perform the translation. The purpose of the injected instructions is to collect some data on what is being displayed on the screen, to collect and log information on the operation to be translated, to create an XML dump containing data on the

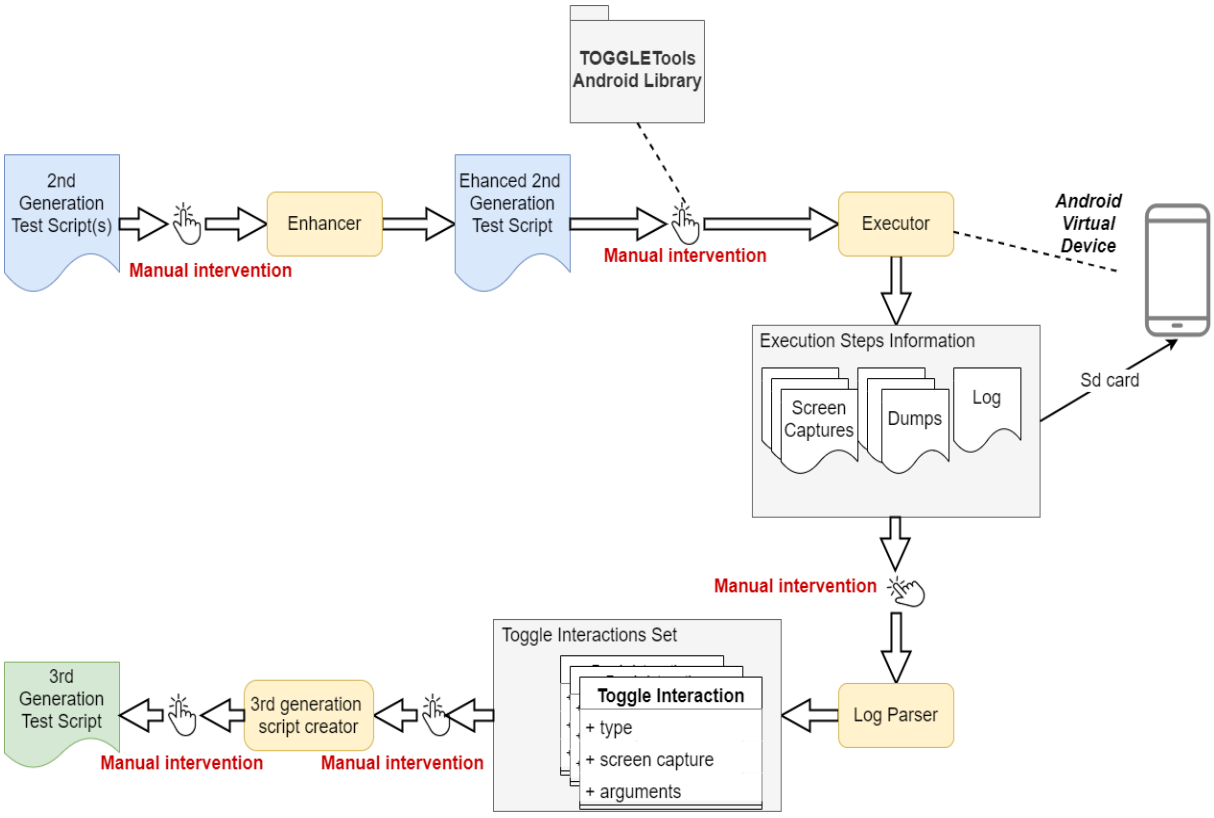


Figure 3.2. The translation process of TOGGLE.

displayed visual hierarchy, to take screenshots at each step, to handle a coherent naming convention that grants the uniqueness of names for both XML dump files and screenshot ones and, in general, to set up all the required information to create the 3^{rd} generation equivalent of the test under analysis. From now on I will refer to the class with the injected lines of code as *enhanced class*;

- **Executor**: this component executes the 2^{nd} generation enhanced test classes verifying their outcome and generating, consequently, the screen-captures, the XML dumps, and the log lines containing information on each visual interaction;

- **Log Parser**: this component oversees filtering the Android LogCat buffer to retrieve the logged lines. From these lines the properties of each interaction can be rebuilt and then translated into 3^{rd} generation test instructions connected to their correspondent visual locators;

- 3^{rd} **generation script creator**: this component performs the actual translation starting from the list of rebuilt interactions.

Another key element displayed in picture 3.2 is the *TOGGLETools* Android Library. This library should be added to the test package of the AUT. It contains the implementation of all the utility functions that the enhanced classes invoke.

In the remaining of this section, each component and all the other key elements of the translation process of TOGGLE will be detailed and the weaknesses of each one of them will be introduced.

3.3.1 Enhancer

This component is tool-specific, meaning that it can enhance exclusively test classes written using the Espresso library. It is possible, theoretically, to create an equivalent component supporting different 2^{nd} generation testing frameworks to make TOGGLE compliant with them too.

The input of this component is, as previously stated, an Espresso test class but, before starting the translation process, it is necessary to modify the size of the Android LogCat buffer [9].

To go into further details, the Android logging system is composed of multiple circular buffers maintained by an internal system process (named *logd*). Each buffer has a precise goal: the *main* buffer stores most of the log of the application, the *system* buffer stores messages coming from the Android Operating System, and the *crash* buffer stores all log lines generated after a crash. A library to interact with these buffers is available by including the header file *android/log.h*). Alternatively, it is possible to use a command-line instruction with the following syntax:

```
[adb] logcat [<option>] ... [<filter-spec>] ...
```

Given their nature, when these buffers reach their limit size the initial content will be overwritten and, consequently, will be impossible to be retrieved. This mechanism, if not correctly addressed, could heavily affect the translation process if, for example, the overwritten lines contain some of the logged instructions that are necessary to rebuild the visual interactions.

This is the first weakness of TOGGLE that is going to be presented. In its initial state, this operation had to be either manually performed via command line instruction or coded into the same Java project invoking the APIs of TOGGLE. In any case, the resizing of the Android Logcat buffers was

performed in a non-transparent way for the end-user. Moreover, this operation implies an invasive intervention of the original context of the application that could impact its functionalities.

An empirical evaluation demonstrated that a reasonable size for these buffers to have a fault-free translation process is 32MB. It is worth noticing that, for an application that performs logging intensively, this could be not enough.

The specific command-line instruction to resize the LogCat buffers is:

```
adb logcat -G 32M
```

Once the buffers have been successfully resized the enhancing activity can start. The Espresso test classes are parsed exploiting the *JavaParser* library[2] and then, once an Espresso assertion is identified during the analysis of a test method, instructions coming from the TOGGLETools library are injected to generate:

- *Screen-captures*: captions of the currently displayed screen. They will be saved as Bitmap files on the emulated external storage of the Android Virtual Device (AVD). These captures are generated thanks to the UI Automator framework. The naming convention for these files involves using the test method name with a suffix composed by a progressive identifier number;
- *XML Dumps*: an XML file containing the current screen hierarchy and reporting every layout component with some of its properties (such as its coordinates on the screen, its id, its class, and similar attributes). Similar to what happens to screen-captures, these files are saved on the external storage of the AVD and they follow the same name convention so that the two kinds of files will be paired in a one-to-one correspondence. The screen hierarchy is extracted through the usage of the UI Automator framework;
- *Log interactions*: using the Android LogCat library, lines containing attributes about the performed interactions are pushed into the LogCat buffers.

[2]JavaParser is the most popular parser for the Java programming language. Specifically, this library can analyze Java files with invocation level granularity, generate and inject Java code, and do some processing and refactoring. This project is open source and has many years of history with hundreds of contributors. More can be found at https://github.com/javaparser/javaparser

A typical log line has the following format:

```
<testName> <interactionIdentifier> <
   searchType> <searchKeyword> <
   interactionType> <optionalArguments>
```

Where *testName* is the name of the enhanced test method, *interaction-Identifier* is the unique name, identifying a couple composed by screenshot and an XML dump file, generated following the explained naming convention, *searchType* determines the type of attribute used to locate a specific view in the hierarchy (e.g. its id, its textual content...), *searchKeyword* is the value that *searchType* assumes, *interactionType* is the type of interaction to be performed on the located widget (e.g. click, doubleclick, typetext, fullcheck), and, finally, *optionalArguments* are optional parameters that should be coupled to the *interactionType* to correctly rebuild the visual interaction.

Other injected instructions are sleep instructions, used to correctly synchronize each step to give the system the time to generate the screen-captures and the XML dumps. The sleep time is of two seconds, an amount of time that has been proven to be sufficient to create the two files without incurring unexpected behaviors.

Another key factor to consider is that the initial version of the enhancement process supported most of the Espresso instructions using *Espresso.onView()* as their entry point, except for the ones using *scrollTo()* as ViewActions. This uncovers another issue that the state-of-the-art of this library presented. It completely lacked the support for all types of scrolling interactions (i.e. the interactions that are tested using *Espresso.onData()* as entry-point or *scrollTo()* as ViewActions).

In addition to this, the Enhancer module offered interface methods that let the end-user enhance one class at a time and were not able to distinguish whether the test class being parsed had already been enhanced or not. The outcome was that, if the AUT has a test suite with N test classes, the method to enhance a class had to be called N times. Furthermore, if an enhanced version of one of these classes had already been generated the enhancing method will simply overwrite it, wasting execution time and computational resources.

The output of the Enhancer module is an enhanced test class performing the same set of 2^{nd} generation tests of the source one and that can be executed in the same way but, additionally, it executes some other instructions whose purpose is to collect the required data to later complete the translation process.

3.3.2 Executor

Once the first phase of the translation has been completed, a set of preliminary operations needs to be performed before starting the execution of the newly generated tests.

Firstly, the application needs to have the permissions for reading and writing on the external storage to be granted so that it can correctly create and store the output files of the execution of the enhanced class.

In particular, in the manifest file of the application, the following permissions must be declared:

android.permission.WRITE_EXTERNAL_STORAGE : needed to be able to create and write on a file stored on the external storage of the device;

android.permission.READ_EXTERNAL_STORAGE : needed to be able to read from a file stored on the external storage of the device.

It is worth noticing that, starting from Android version 6.0 (API level 23), these permissions are considered dangerous, meaning that they could potentially expose user's private information or share operations with other apps, therefore they should be granted by the user at run time. Of course, forcing the end-user to click on the screen to grant these permissions during the execution of a test is not compatible with the concept of test automation and, consequently, with the purpose of TOGGLE.

In addition to this, another key element to be considered before launching the enhanced tests is that that it is mandatory to inject some library classes into the test folder. These classes are named:

- *TOGGLETools*: a Java class offering the implementation of all the additional operations needed to complete the translation. Particularly, its methods are used to log the interactions, to take the screenshots, to get the display size, and more;

- *BitmapSaver*: a Java class whose main purpose is to store on the external storage the bitmap version of the taken screenshot.

As introduced in section 3.3.1, the lines of code injected into the Espresso test classes use methods coming from these two library classes.

Finally, the application needs to be re-built, installed, and correctly instrumented on the Android Virtual Device so that its APK will include both

the enhanced classes and the library ones and it is possible to launch the enhanced tests through command-line instructions.

All these steps either were not completely automated or their complexity was not hidden to the end-user. More in detail, the permissions had to be manually granted acting directly on the AVD, the library classes had to be copied into the test folder of the AUT and there was no support for the automatic generation of the new APK of the application.

Moreover, the reference to the instrumentation of the application had to be retrieved manually by the programmer and then sent, as a string, to the method in charge of executing the enhanced test case.

A partial solution to these problems was offered by an extension of TOGGLE implementing its GUI interface. In this case, all these steps were interactive with the end-user and the framework became more a stand-alone tool than an open-source library, which is, instead, the actual goal of this case study.

Once the test environment has been correctly prepared, the enhanced test class can be executed thanks to the following Android Debug Bridge (ADB) command-line instruction, launched via Java code:

```
adb shell am instrument -w -e class <
   testInjectionPath>.<testClassName> <
   instrumentation>
```

Where *testInjectionPath* is the path to the folder of the enhanced test cases, *testClassName* is the name of the test class we want to execute, and *instrumentation* is the test instrumentation of the AUT.

The method launching this command needs to be executed as many times as the number of enhanced test classes that we wanted to execute.

This highlights two different vulnerabilities. The first one involves the fact that, as already presented in section 3.3.1, the complexity of the execution is not truly hidden to the end-user meaning that, even if the test classes are all in the same directory, there is no support for executing them all with one single instruction. The second vulnerability is the high memory consumption that these tests cause on the external storage of the AVD. In particular, each enhanced test method execution results in, at least, as many bitmap files and XML dumps as Espresso assertions plus an additional couple of these files (one per each type) that is needed for the final check of the screen (this is a further operation injected at the end of each test method). It is easy to understand that the external storage can be quickly filled and, for this reason, it has to be manually cleared before each test execution.

Another problem that is directly connected to the previous one is that, in presence of test classes with many test methods, the translation process could fail even addressing all the previous issues with the correct countermeasures. This happens because the execution granularity by test class has proven to be too coarse leading to consuming the available memory on the external storage of the AVD before having completed the execution of the whole class. Consequently, in these cases, it is not possible to create the needed files for all test methods.

The initial version of this framework did not address these issues, leaving the responsibility of preventing them to the end-user.

Finally, the executor checks the outcome of each 2^{nd} generation test assertions, and, in presence of a failure, it notifies the tester and aborts the translation process. This is a reasonable consequence since it is useless to translate a faulty test method.

The output of this module is a set of bitmap files and XML dumps with a 1-to-1 correspondence and a set of logged lines in the Android LogCat buffer.

3.3.3 LogParser

This module is in charge of initiating the actual translation process, starting from the output of the Executor, by collecting the information generated after the execution of the enhanced test classes.

The first operation is filtering the log file so that the logged lines, describing the visual interactions, can be isolated. This has a computational complexity proportional to the total number of lines in the LogCat buffer.

Once all lines of interest have been collected, the second operation that has to be performed is pulling out from the external storage of the AVD, for each interaction line, the screen-captures and the XML dumps.

Finally, starting from the information collected from the log lines and the dumps, an object of the class corresponding to the actual interaction is created. During the creation process of these objects, the screen-captures are resized according to the screen size of the host machine and, using the coordinates taken from the corresponding dump file, cropped, isolating the component on which the interaction has to be performed. The final image will be used as a locator in the equivalent 3^{rd} generation test script.

The resizing of the images is necessary since the screen-captures, considering that they come from the AVD, have a screen resolution that does not reflect the one used to graphically display the emulated screen on the computer hosting the emulator.

The crop operation, instead, is useful because, by using as locator only the portion of the screenshot containing the visual component, every dependency on its rendered position will be eliminated. This is important because this factor could slightly differ according to which device the AUT is being executed on, to its orientation, and similar factors that usually affect the stability of GUI tests.

The dump file and the optional arguments could be further used to extract some other information, if needed, to correctly emulate the visual interaction.

This module presents, probably, one of the most challenging vulnerabilities of TOGGLE. Programmatically retrieving the screen density of the host machine, and resizing the cropped screen-captures accordingly, is not an easy task because it is directly related to two of the biggest Visual GUI Test weaknesses: the dependency on the pixel density of the screen and the small tolerance to width and height differences between the locators and the front-end bitmap layer of the AUT.

In particular, the library offers no method to automatically retrieve the width and height of both the screen of the host machine and the screen of the AVD neither it offered an automatic way to dynamically handle the differences in these measures when changing the host machine. These dimensions were hard-coded for a specific host machine and a collection of different types of mobile emulators, thus they are not portable among different computers. Consequently, the whole process becomes not maintainable in the long term nor in an industrial development process.

The output of this module is a set of Java objects containing all the instructions and information, other than the references to their locator image, that are necessary to complete the translation. Each of these objects inherits from a common parent class, named ToggleInteraction, that offers some basic common functionalities. Picture 4.3 shows a schematic representation of the key elements inside this class.

3.3.4 Third generation script creator

Starting from a collection of ToggleInteraction objects, the third generation script creator is in charge of generating GUI tests using both EyeAutomate and SikuliX instructions.

It is useful to highlight two important factors to correctly analyze this translation process. The first one is that, since the scripts are meant to be executed in an ecosystem, the computer one, that is different from the mobile one, the basic interactions will be different too. For example, in the mobile

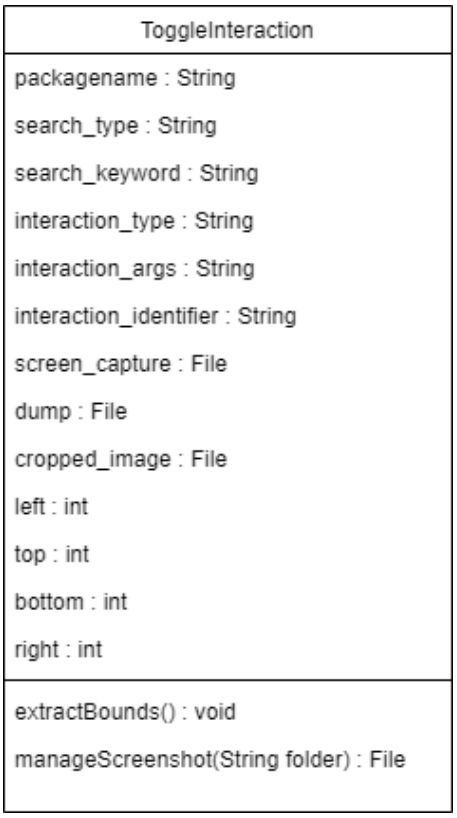


Figure 3.3. ToggleInteraction.

world, a user could perform actions like swiping or pinching that are typical of a device with a touchscreen while not so common on a personal computer.

The second factor is the difference between the entry points of the two generations of tests. A 2^{nd} generation test, especially an Espresso one, has a lower-level approach and, for this reason, it can act directly on the View objects without necessarily having to perform any graphical interaction on them, that is, instead, the typical action of a real user. This happens, for example, when the test method is executing the Espresso instructions to enter some text on a View. In this case, the text will appear inside the visual component without having to move the focus on it.

On the contrary, a 3^{rd} generation test access the AUT through its GUI and has no lower-level attachment to the software. This means that, for example, to type some text on a View a test needs to execute a preliminary instruction to tap on it, moving in this way the focus to the component under exam, and then one or more other instructions to actually insert the text.

The same problem presents itself when a test needs to emulate the following interactions: swipe down/up/left/right, replace some text, clear some text, double-click, and long-click.

Both factors explain why it is impossible, especially for more complex interactions, to have 1-to-1 correspondence between 2^{nd} and 3^{rd} generation instructions.

Another key element to be considered while transitioning from one generation to the other is that, since a 3^{rd} generation test has a higher-level entry point on the AUT, the instructions have to be correctly synchronized so that the GUI can have the time to reach a stable state after each interaction.

This synchronization mechanism is implemented by a customizable time-out set on each instruction, needed to wait for the widget to be displayed on the screen, and some sleep instructions placed to additionally temporize the interactions.

An empirical evaluation proved that a reasonable time-out value to grant the fact that the GUI can reach a stable state is $30s$. The sleep instruction placed between the interactions, instead, can use a different value according to the situation. For example, a double-click needs a sleep time that is, of course, shorter than the one needed to perform a long-click.

In addition to this, after each group of instructions generated in the translation of a 2^{nd} generation assertion, an additional sleep instruction of $1s$ is inserted to further decrease the probability of performing graphical interactions too quickly for the GUI engine to intercept them.

The outputs of this module are two textual scripts, one per each 3^{rd} generation testing tool, and a Java project with the classes containing the instructions that, using the APIs offered by both EyeAutomate and SikuliX, can execute the same test.

The textual script can be executed by the two correspondent IDEs (*EyeStudio* and *SikuliX IDE*).

The Java project offers, instead, a wider range of options to the tester. While being semantically equivalent to the textual scripts, the Java classes let the tester use them as the foundation for a more complex test case (for example it is possible to generate a test that, before the execution of the actual set of interactions, wants to set up a given scenario in the database of the AUT).

Furthermore, TOGGLE exploits the possibility of using instructions coming from both EyeAutomate and SikuliX APIs in the same test method to increase the stability of the generated visual GUI tests. The Java project is composed of four different classes:

1. an EyeAutomate-only class, containing exclusively instructions coming from this library;
2. a SikuliX-only class, containing exclusively instructions coming from this library;
3. a Java class combining the two libraries, executing first the EyeAutomate version of the test and, if it fails, trying to execute the SikuliX version;
4. a Java class combining the two libraries, executing first the SikuliX version and, if it fails, trying to execute the EyeAutomate one.

Table 3.1 shows the collection of instructions needed to emulate some of the most common interactions [12].

3.4 Translation example

In this section, an example of the translation of basic interaction is going to be presented, showing the generated data and files. The goal is to give the reader an actual example of how a translation takes place and the appearance of every file involved in it.

```
@SmallTest
public void testClickExample(){
    onView(withId(R.id.fab_expand_menu_button)).perform(click());
}
```

Figure 3.4. (1) Espresso source test method.

The starting point is the source Espresso test method shown in figure 3.4. This method will be parsed to generate the enhanced version and then this class will be executed to create the log lines, the XML dump files and the screen-captures. Figures 3.5, 3.6, 3.7, and 3.8 report, in order, an example for each step.

Once these files have been created and pulled out from the external storage of the AVD, then the Click interaction is created and the screen-capture is resized and cropped to extract the locator shown in figure 3.9.

This locator will be used by the generated test files to make assertions on it and to perform interactions. The same sequence of files and, generally, of steps will be executed for the "fullcheck" log line, whose purpose is to verify

Table 3.1. 3^{rd} generation test syntax

Logged interaction	EyeAutomate commands	SikuliX commands
clearText	i. Click *img* ii. Type [BACKSPACE] (*arg1* times)	i. click(*img*) ii. type (Key.BACKSPACE) (*arg1* times)
click	i. Click *img*	i. click(*img*)
closesoftkeyboard	i. Type [CTRL_PRESS] ii. Sleep 10 iii. Type [BACKSPACE] iv. Sleep 10 v. Type [CTRL_RELEASE]	i. keyDown(Key.CTRL) ii. sleep(0.01) iii. type(Key.BACKSPACE) iv. sleep(0.01) v. keyUp(Key.CTRL)
doubleclick	i. MouseDoubleClick *img* ii. Click *img* iii. Type *arg1*	i. hover(*img*) ii. mouseDown(Button.LEFT) iii. sleep(0.001) iv. mouseUp(Button.LEFT) v. sleep(0.001) vi. mouseDown(Button.LEFT) vii. sleep(0.001) viii. mouseUp(Button.LEFT)
longclick	i. Move *img* ii. MouseLeftPress iii. Sleep 500 iv. MouseLeftRelease	i. hover(*img*) ii. mouseDown(Button.LEFT) iii. sleep(0.5) iv. mouseUp(Button.LEFT)
typetext	i. Click *img* ii. Type *arg1*	i. click(*img*) ii. type(*arg1*)
openactionbarmenu	i. Type [CTRL_PRESS] ii. Sleep 10 iii. Type m iv. Sleep 10 v. Type [CTRL_RELEASE]	i. keyDown(Key.CTRL) ii. sleep(0.01) iii. type(m) iv. sleep(0.01) v. keyUp(Key.CTRL)
pressback	i. Type [CTRL_PRESS] ii. Sleep 10 iii. Type [BACKSPACE] iv. Sleep 10 v. Type [CTRL_RELEASE]	i. keyDown(Key.CTRL) ii. sleep(0.01) iii. type(Key.BACKSPACE) iv. sleep(0.01) v. keyUp(Key.CTRL)
presskey	i. Type *arg1*	i. type(*arg1*)
pressmenukey	i. Type [CTRL_PRESS] ii. Sleep 10 iii. Type h iv. Sleep 10 v. Type [CTRL_RELEASE]	i. keyDown(Key.CTRL) ii. sleep(0.01) iii. type(h) iv. sleep(0.01) v. keyUp(Key.CTRL)
replacetext	i. Click *img* ii. Type [BACKSPACE] (*arg1* times) iii. Type *arg2*	i. click(*img*) ii. type(Key.BACKSPACE) (*arg1* times) iii. type(*arg2*)
swipedown	i. Move *img* ii. Sleep 10 iii. MouseLeftPress iv. MoveRelative "0" "250" v. MouseLeftRelease	i. r = find(*img*) ii. start = r.getCenter() iii. stepY = 250 iv. run = start v. mouseMove(start); wait(0.2) vi. mouseDown(Button.LEFT); wait(0.2) vii. run = run.below(stepY) viii. mouseMove(run) ix. mouseUp() x. wait(0.2)

```
@SmallTest
public void testClickExample() {
    FutureTask<Boolean> capture_task = null;
    Instrumentation instr = InstrumentationRegistry.getInstrumentation();
    UiDevice device = UiDevice.getInstance(instr);
    int num = 0;
    Activity activityTOGGLETools = getActivityInstance();
    capture_task = new FutureTask<Boolean>(new TOGGLETools.TakeScreenCaptureTaskProgressive(num,  test_name: "testClickExample", activityTOGGLETools));
    try {
        runOnUiThread(capture_task);
    } catch (Throwable t) {
        t.printStackTrace();
    }
    TOGGLETools.LogInteractionProgressive( className: "TestHomeActivityTOGGLE", num,  test_name: "testClickExample",  search_type: "id",  search_kw: "fab_expand_menu_button",  interaction_type: "click");
    TOGGLETools.DumpScreenProgressive(num,  test_name: "testClickExample", device);
    onView(withId(R.id.fab_expand_menu_button)).perform(click());
    try {
        Thread.sleep( millis: 1000);
    } catch (Exception e) {
    }
    num++;
    activityTOGGLETools = getActivityInstance();
    capture_task = new FutureTask<Boolean>(new TOGGLETools.TakeScreenCaptureTaskProgressive(num,  test_name: "testClickExample", activityTOGGLETools));
    Rect currdisp = TOGGLETools.GetCurrentDisplaySize(activityTOGGLETools);
    try {
        runOnUiThread(capture_task);
    } catch (Throwable t) {
        t.printStackTrace();
    }
    TOGGLETools.LogInteractionProgressive( className: "TestHomeActivityTOGGLE", num,  test_name: "testClickExample",  search_type: "-",  search_kw: "-",
            interaction_type: "fullcheck",  interaction_params: currdisp.bottom + ";" + currdisp.top + ";" + currdisp.right + ";" + currdisp.left);
    TOGGLETools.DumpScreenProgressive(num,  test_name: "testClickExample", device);
}
```

Figure 3.5. (2) Espresso enhanced test method.

```
testClickExample; testClickExample0; id; fab_expand_menu_button; click
testClickExample; testClickExample1; -; -; fullcheck; 1776;72;1080;0
```

Figure 3.6. (3) Log lines.

that the final screen of the AVD matches the expected one. Figure 3.10 shows an example of a SikuliX test script.

```
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
<hierarchy rotation="0">
  <node bounds="[0,0][1080,1776]" visible-to-user="true" selected="false" password="false" long-clickable="false" scrollable="false" focused="false" focusable="false" enabled="true" clickable="false" checked="false" checkable="false" content-desc="" package="org.ligi.passandroid" class="android.widget.FrameLayout" resource-id="" text="" index="0">
    <node bounds="[0,0][1080,1776]" visible-to-user="true" selected="false" password="false" long-clickable="false" scrollable="false" focused="false" focusable="false" enabled="true" clickable="false" checked="false" checkable="false" content-desc="" package="org.ligi.passandroid" class="android.widget.LinearLayout" resource-id="" text="" index="0">
      <node bounds="[0,72][1080,1776]" visible-to-user="true" selected="false" password="false" long-clickable="false" scrollable="false" focused="false" focusable="false" enabled="true" clickable="false" checked="false" checkable="false" content-desc="" package="org.ligi.passandroid" class="android.widget.FrameLayout" resource-id="" text="" index="0">
        <node bounds="[0,72][1080,1776]" visible-to-user="true" selected="false" password="false" long-clickable="false" scrollable="false" focused="false" focusable="false" enabled="true" clickable="false" checked="false" checkable="false" content-desc="" package="org.ligi.passandroid" class="android.view.ViewGroup" resource-id="org.ligi.passandroid:id/decor_content_parent" text="" index="0">
          <node bounds="[0,72][1080,240]" visible-to-user="true" selected="false" password="false" long-clickable="false" scrollable="false" focused="false" focusable="false" enabled="true" clickable="false" checked="false" checkable="false" content-desc="" package="org.ligi.passandroid" class="android.widget.FrameLayout" resource-id="org.ligi.passandroid:id/action_bar_container" text="" index="0">
          <node bounds="[0,240][1080,1776]" visible-to-user="true" selected="false" password="false" long-clickable="false" scrollable="false" focused="false" focusable="false" enabled="true" clickable="false" checked="false" checkable="false" content-desc="" package="org.ligi.passandroid" class="android.widget.FrameLayout" resource-id="android:id/content" text="" index="1">
            <node bounds="[0,240][1080,1776]" visible-to-user="true" selected="false" password="false" long-clickable="false" scrollable="false" focused="false" focusable="false" enabled="true" clickable="false" checked="false" checkable="false" content-desc="" package="org.ligi.passandroid" class="androidx.drawerlayout.widget.DrawerLayout" resource-id="org.ligi.passandroid:id/drawer_layout" text="" index="0">
              <node bounds="[0,240][1080,1776]" visible-to-user="true" selected="false" password="false" long-clickable="false" scrollable="false" focused="false" focusable="false" enabled="true" clickable="false" checked="false" checkable="false" content-desc="" package="org.ligi.passandroid" class="android.widget.RelativeLayout" resource-id="" text="" index="0">
                <node bounds="[0,240][1080,1776]" visible-to-user="true" selected="false" password="false" long-clickable="false" scrollable="false" focused="true" focusable="true" enabled="true" clickable="false" checked="false" checkable="false" content-desc="" package="org.ligi.passandroid" class="androidx.recyclerview.widget.RecyclerView" resource-id="org.ligi.passandroid:id/content_list" text="" index="0"/>
                <node bounds="[0,240][1080,1776]" visible-to-user="true" selected="false" password="false" long-clickable="false" scrollable="false" focused="false" focusable="false" enabled="true" clickable="false" checked="false" checkable="false" content-desc="" package="org.ligi.passandroid" class="android.widget.TextView" resource-id="org.ligi.passandroid:id/emptyView" text="No pass yet You can search your phone for passes or create your own or try a demo pass just press + down below " index="1"/>
                <node bounds="[255,512][1032,1728]" visible-to-user="true" selected="false" password="false" long-clickable="false" scrollable="false" focused="false" focusable="false" enabled="true" clickable="false" checked="false" checkable="false" content-desc="" package="org.ligi.passandroid" class="android.view.ViewGroup" resource-id="org.ligi.passandroid:id/fam" text="" index="2">
                  <node bounds="[810,1506][1032,1728]" visible-to-user="true" selected="false" password="false" long-clickable="false" scrollable="false" focused="false" focusable="true" enabled="true" clickable="true" checked="false" checkable="false" content-desc="" package="org.ligi.passandroid" class="android.widget.ImageButton" resource-id="org.ligi.passandroid:id/fab_expand_menu_button" text="" index="8" NAF="true"/>
                </node>
              </node>
            </node>
          </node>
        </node>
      </node>
    </node>
    <node bounds="[0,0][1080,72]" visible-to-user="true" selected="false" password="false" long-clickable="false" scrollable="false" focused="false" focusable="false" enabled="true" clickable="false" checked="false" checkable="false" content-desc="" package="org.ligi.passandroid" class="android.view.View" resource-id="android:id/statusBarBackground" text="" index="1"/>
    <node bounds="[0,1776][1080,1920]" visible-to-user="true" selected="false" password="false" long-clickable="false" scrollable="false" focused="false" focusable="false" enabled="true" clickable="false" checked="false" checkable="false" content-desc="" package="org.ligi.passandroid" class="android.view.View" resource-id="android:id/navigationBarBackground" text="" index="2"/>
  </node>
</hierarchy>
```


Figure 3.7. (4) XML dump file.

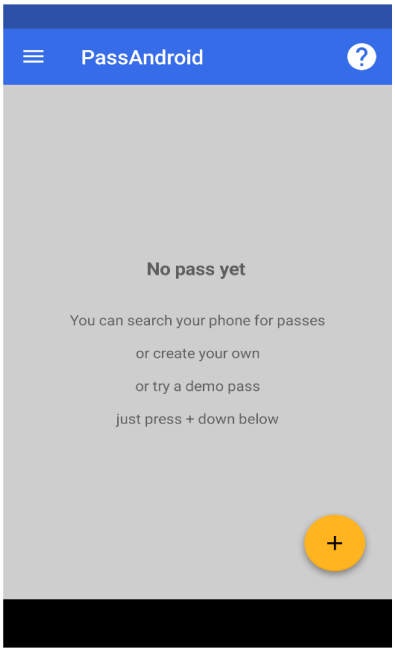


Figure 3.8. (5) Screen-capture.

Figure 3.9. (6) Locator.

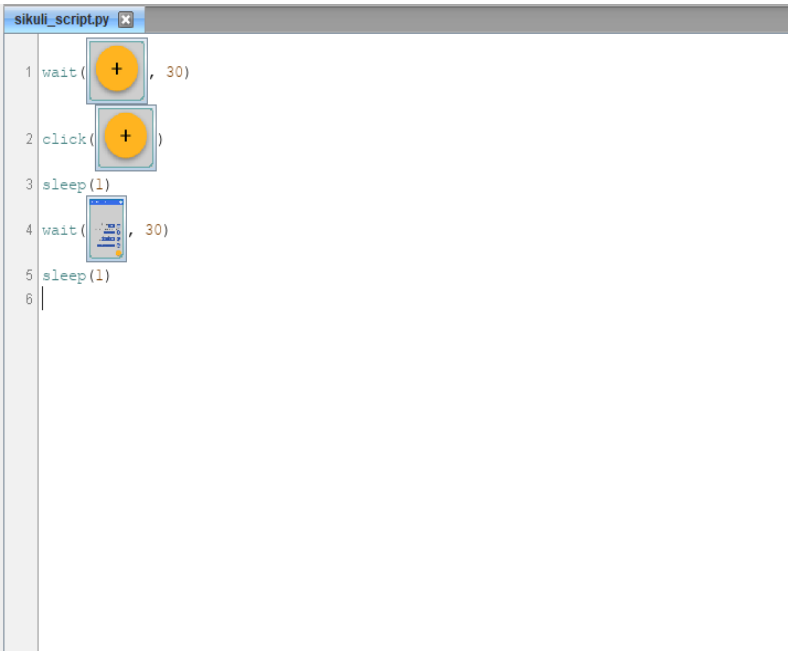


Figure 3.10. (7) 3^{rd} generation test script.

Chapter 4

TOGGLE Reengineering

Chapter 3 introduced TOGGLE, a support tool whose purpose is to ease the transition from 2^{nd} generation tests to 3^{rd} generation ones. TOGGLE, being conceived as an academic tool, proves the feasibility of an automated translation but, due to all the flaws and weaknesses presented in chapter 3, it is not good enough to be adopted in an industrial development process.

For this reason, we felt that the framework needed to be re-engineered to make its learning curve smoother.

This process may be categorized into three main phases:

tool improvement : in this phase, the vulnerabilities of the tool, presented during its state-of-the-art description, were addressed and solved. The main goal was to make the whole process as automated as possible by reducing the manual interventions requested to an end-user to the lowest level possible;

re-design : in this phase I re-designed the architecture of the library to hide its complexity, offering a clear, flexible, and easy-to-use interface that can act as the main entry-point for the end-user. The adopted design pattern is the *Façade* pattern;

new features : in this phase, I implemented some missing features of TOGGLE. Specifically, the tool lacked a translation mechanism supporting the scroll interaction so a mechanism to translate it has been introduced.

Another additional phase may be placed before the previous ones since the starting point was to change the build automation mechanism from Maven to Gradle.

The re-engineered version of TOGGLE can be found at https://github.com/VittorioDiLeo21/TOGGLE, together with a zipped folder reporting the source Espresso files and the corresponding 3^{rd} generation translation used for the experiment reported in chapter 5.

4.1 Gradle

The first step taken during the re-engineering process of TOGGLE was to change the build automation tool. TOGGLE initial version used Maven.

Even though Maven, since it is older than Gradle [7] [8], has higher support in the most common IDEs, the decision of migrating to Gradle was carried out because this tool offers higher flexibility and better performances.

Additionally, Google adopts Gradle as the default build tool for Android applications. This choice was made not only because its build scripts are shorter and clearer than an equivalent Maven one but also for the flexibility and customization possibilities that this build tool offers. Gradle, in fact, is an extensible framework and it can be used in projects written in a wide range of programming languages, including C/C++, while Maven is mainly Java-oriented.

Furthermore, since a Maven script is written using the XML syntax and the build flow follows some predefined targets to which goals are attached, it is more rigid and it does not support incremental compilation for classes (i.e. compiling only the classes that have actually changed).

On the other hand, Gradle uses a Domain Specific Language (DSL) derived from Groovy. It is based on tasks composed in an acyclic graph and supports incremental compilation resulting in overall higher flexibility.

In addition to the mechanism to avoid unnecessary work, Gradle exploits both a build cache, to reuse the outputs of any previous build having the same inputs, and a long-lived process (called Gradle Daemon), that keeps data related to the build process ready to be re-used.

Another factor affecting the build performances is the difference in dependency management between the two tools. In particular, Gradle lets the user define customized dependency scopes and substitution rules, avoiding unwanted dependencies in the project so that it is possible to obtain better-modeled and faster builds.

As it can be seen in picture 4.1, that is showing a comparison between Gradle and Maven execution times in three different contexts, Gradle is proven to have better performances than Maven.

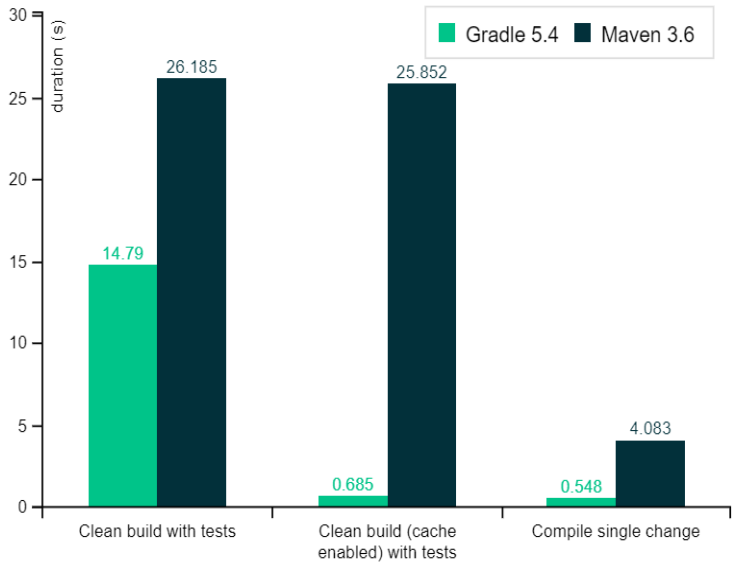


Figure 4.1. Comparison between Gradle and Maven build performances. From : https://gradle.org/maven-vs-gradle/

Moreover, since TOGGLE is a mobile-oriented framework and, more in detail, an Android-oriented one, I felt that Gradle was the best option to maintain coherency with the official Android build tool.

4.2 Tool improvement

As presented in sections 3.3.1, 3.3.2, 3.3.3, and 3.3.4 the modules composing TOGGLE presented some weaknesses that worsen its maintainability. In this section these issues will be further analyzed, presenting the proposed solutions and all the various improvements introduced.

4.2.1 Log files

One of the first issues highlighted in chapter 3 was the need to resize the Android LogCat buffers to avoid the overwriting of their content once they have been filled up to their maximum size.

Even though a first temporary solution, using the ADB command to manually resize these circular buffers, had already been introduced, this option was not sustainable in a long-term generic library. First of all, it introduces an invasive intervention on the environment of the AUT while the tool should

be as transparent as possible but, more importantly, this solution was not universally valid because it was heavily dependent on the log frequency of the AUT. An application that, during its execution, logs a lot of lines in these buffers might easily fill these buffers even after they have been resized.

For this reason, the lines containing the information about the translation process have been redirected into separate log files stored on the external storage of the device, avoiding any kind of manipulation on the Android LogCat buffers.

Having separate files introduces several benefits. For starting, there is an actual separation of concerns between the logic of the AUT and the translation process, avoiding mixing information coming from the two processes. This highlights another improvement related to the previous one: since all log lines are concentrated in the same location, there is no need to add a subsequent filtering operation to retrieve them. A simple sequential read is all it is needed, thus resulting in a shorter translation execution time.

Going into further details on how the log phase has been changed, it is necessary to specify that the logged lines have been further divided into one file for each test class. By using different files, the flexibility of the translation process has increased since the execution and the translation of every single class have become independent from one another.

The outcome is that if only a few Espresso test classes in the whole test suite have changed then only these classes need to be included in the re-translation process to update accordingly the correspondent 3^{rd} generation tests, leaving untouched all the other log files that can maintain their validity. The results are better performances and easier maintainability.

A consideration to be done is that creating, updating, and saving a file on an Android device is not a trivial task. The application should declare in its manifest file the permissions to write and read on the external storage. Given that these permissions are needed to execute the enhanced tests too (as already explained in section 3.3.2), the conclusion was that this improvement is not adding any new constraints to the adoption of the library.

To automate the phase of permission granting another ADB command has been used:

```
adb shell pm grant <package-name> <
  permission-name>
```

Where *package-name* is the package name of the application and *permission-name* is the name of the permission to be granted. In this case, the latter will assume the following values:

android.permission.WRITE_EXTERNAL_STORAGE ;

android.permission.READ_EXTERNAL_STORAGE .

4.2.2 TOGGLETools library injection

The library classes named *TOGGLETools* and *BitmapSaver* had to be copied in the folder containing the enhanced tests to be able to execute them. These classes explicitly depend on libraries belonging to the Android ecosystem, thus they cannot be included in a generic Java project that does not use a Software Development Kit (SDK) compliant with the Android APIs.

TOGGLE, instead, is a tool thought to be used on a personal computer supporting Java, starting from a mobile emulator but generating images, test scripts, and test classes that will be stored and eventually executed on the first kind of machine. For this reason, it should be considered external to that environment, thus it cannot use an Android SDK.

Knowing this, it is easy to understand why both *TOGGLETools* and *BitmapSaver* library classes need to be directly injected into the environment of the AUT, so that they can be built as part of the application itself, and could not be used as internal library classes of TOGGLE, adding a simple dependency to the test classes of the AUT.

Of course, manually pasting these classes into the test folder of the application is not feasible nor maintainable in the long term and for big projects. The implemented solution includes some Java methods that, receiving as input the test directory of the AUT, can detect whether the two classes have already been injected or not and, if not, they are able to re-create them in the specified folder.

For this purpose a class, named *ToggleToolFinder*, has been introduced to TOGGLE, offering the implementation of the methods to perform these operations other than a copy of both library classes. The latter is actually excluded from the build process of the library, they are used as source files when it is needed to inject them into the project.

4.2.3 Enhance method update

The enhancing process, at the state-of-the-art of the tool, had some weaknesses. For starting, when launching a new enhancing sequence of instructions, there was no support for detecting whether an Espresso test class had already been enhanced or not.

To understand if there is the actual need to enhance a class or not there are many options. Generally, it should be enough to search for a file with the same name plus the word "Enhanced" appended to it but, to add a further check, it is possible to parse this test file searching for a reference to any of the APIs offered by the *TOGGLETools* class or, simply, for the import declaration for *TOGGLETools* itself. The second option has been selected because it is a reasonable compromise between good performances and trust in the output result. In any case, to force the re-enhancement of a test class, it is enough to simply delete the previous version from the folder.

Another problem was that to enhance a test suite with multiple test classes, the same method had to be invoked repeatedly for as many times as the number of classes to be enhanced, passing as method parameter the class name. This condition heavily affects the usability of the library.

For this reason, starting from the test folder, a feature has been implemented to offer the possibility of analyzing every Java file inside it to find out whether it contains an Espresso assertion, if, of course, it is not an enhanced file, and if it does not already have an enhanced equivalent. If all three conditions are verified then the test file needs to be enhanced and the whole enhancing process will be executed. This sequence is repeated until all the files in a given folder have been processed. In this way, the library lifts from the end-user the burden of having to invoke the same instruction over and over. Moreover, it automates the update process in case of a new Espresso test file added to the given directory: it will be enhanced (while all the other unchanged classes will be ignored) as long as the new file is in the same folder.

4.2.4 Build and install

Once *TOGGLETests* and *BitmapSaver* have been successfully injected into the test folder and all the Espresso test classes under exam have been enhanced, the application needs to be built and deployed on the Android emulator.

An Android application uses Gradle as its build-automation tool and the recommended way to execute any Gradle build is by exploiting the Gradle Wrapper[6]. The Wrapper is a script whose main goal is to quickly set up the build automation tool, eventually downloading automatically the declared version of Gradle and reducing the overall build time.

To include a Gradle Wrapper in a project it is necessary to have a Gradle runtime version installed on the machine and to execute the *wrapper* task.

This script is usually automatically generated by the IDE when a new project for an application is created but it is possible to use the *gradlew* instruction with the relative options to execute the build tasks with the Wrapper through command-line instructions. This is the adopted solution to run tasks to build and install the application from a third-party project.

Knowing that this script is placed in the base folder of the AUT, which is known to the library since the enhancing phase, it is possible to exploit this path to reach the script and execute it to build the application, generate the APK, and install it on the target device.

In particular, there are two solutions for performing these operations. The first one is by executing the following command-line instruction:

```
<applicationPath>\gradlew assemble
```

Where *applicationPath* is the path to the gradlew script that has been previously mentioned. This command will assemble all AUT versions and an APK will be created for each of them.

Another option is to execute the following combination of commands. The first one is:

```
<applicationPath>\gradlew tasks
```

That returns the list of all the available executable Gradle tasks. Then, this list needs to be filtered in order to collect all tasks that should be executed to build the project of the application.

Thereupon, for each identified task, the following instruction needs to be executed:

```
<applicationPath>\gradlew <taskName>
```

Where, of course, *taskName* is the name of the task to be executed.

If one of the previous tasks fails an exception will be thrown so that the end-user can be notified and it can avoid performing all the subsequent instructions (that would likely be pointless if this step was not successful).

If, instead, the build and deploy phase succeeds the next operation to be performed, to be able to execute the enhanced tests, is to automatically retrieve the instrumentation of the application. The following ADB command returns the list of all the active instrumentation:

```
adb shell pm list instrumentation
```

To isolate the single instrumentation needed to execute the enhanced Espresso test methods this list should be filtered, searching for a line containing both the package name of the AUT and the string *AndroidJUnitRunner* in it. From this line, it is possible to obtain the reference to the instrumentation of interest.

4.2.5 Enhanced test execution

As presented in section 3.3.2, the execution phase had several vulnerabilities to be fixed.

The first issue that is going to be presented involves the external memory usage: a generic enhanced test execution, especially after the improvements to the log phase introduced in section 4.2.1, outputs, other than the actual test results, a set of files that can quickly fill the emulated external storage. For this reason, it is necessary to clear up the memory, deleting any residual file of previous executions, so that it will be possible to generate and store the new ones.

Moreover, at its state-of-the-art, TOGGLE executed the enhanced tests with a class level granularity. As already presented, this introduces another memory-consumption-related problem. Even if the external storage of the device is cleared up before and after any enhanced test execution if a test class contains a lot of test methods the external storage of the AVD may be filled up before completing the execution of the whole class. This results in a fragmented set of files, due to the lack of memory, that does not include all the ones necessary for a complete test translation.

This means that the initial version of TOGGLE introduced an indirect and undesirable constraint on the number of test methods that could be included in a test class. This is an element that might further discourage an end-user from using this translation automation tool.

Additionally, even a single method may cause excessive memory consumption but, in this case, an empirical evaluation proved that the average number of assertions in an Espresso test method results in a reasonable memory usage so that it is unlikely that a single test completely consumes all the available space on the external storage. From this starting point, a finer granularity has been adopted, moving from the class-level to the method-level one.

Executing enhanced tests with this finer level of granularity is not sufficient to solve the memory usage problem by itself. Another complementary operation to be carried out is to pull out all the generated output files after each method execution so that they could be safely deleted from the device

without causing any loss of information, which might affect the translation process.

All these actions can be executed through some ADB command-line instructions. In detail, the following commands are used to remove any bitmap or dump file, generated from any previous execution of TOGGLE, and to erase any residual log file respectively:

```
adb shell rm sdcard/TOGGLE/*.bmp sdcard/TOGGLE
   /*.xml
adb shell rm -f /sdcard/TOGGLE/*TOGGLE.txt
```

Then, for each enhanced class, every test method can be executed and, if all assertions succeed then it means that the translation process can continue safely and that all the generated bitmap and XML files have been pulled out from the AVD external storage. The instruction to pull all the output files of the method is:

```
adb pull /sdcard/TOGGLE <destinationFolder>
```

Where *destination folder* is the directory where all test scripts, test classes, locator images, and log files will be placed.

It is worth noticing that the sub-directory named "TOGGLE" has been introduced during this phase of the re-engineering process of the tool. There are two different reasons behind its introduction. Firstly, as it happened for the log phase, to establish an actual separation among entities that logically belong to separate processes so that the generated files of TOGGLE do not mix with files coming from different sources. Secondly, in a more pragmatic way, the use of this sub-directory drastically lowers the effort required during the pull phase since it makes it possible to pull the whole set of files that is wrapped in this folder with only one command-line instruction.

Once all files have been pulled out from the device, the screen-captures and the XML dumps must be deleted from the external storage of the device. The ADB command used to complete this task is the same one that has been introduced while explaining the operations needed to reset the log files, with the only difference in the string used to match the file to be removed. In fact, in this case, the command is:

```
adb shell rm -f /sdcard/TOGGLE/<methodName>*
```

Where *methodName* is the name of the method that has just been executed.

To execute a single method through an ADB command, a piece of additional information (i.e. the name of the method) should be included in the instruction used in the starting version of TOGGLE to launch a test. Specifically, the ADB command becomes:

```
adb shell am instrument -w -e class <
   testInjectionPath>.<testClassName>#<
   testMethod> <instrumentation>
```

The granularity change in the test execution introduces an additional difficulty. As a matter of fact, this modification leads to a deeper information level of detail required when using TOGGLE.

Initially, to execute the enhanced test classes, a list containing all the names of the test classes had to be passed as a parameter of the method. Furthermore, there was no automatic way to retrieve this information and, for this reason, the end-user had to manually collect and hard-code this information before invoking the execution method.

Knowing this, it was necessary to automate this process, and, facing the new challenge that the method-level execution granularity introduces, a deeper re-work of this process was the natural consequence to try and solve it.

Starting from the enhance phase, since the parsing mechanism analyzed the test class method by method, I introduced, during the analysis of a test class, a mechanism to collect all the methods names of the class in a list that will be returned to the invoker of the enhance method.

Then, since the final goal was to use a Façade class to hide the complexity of the whole translation execution flow, as it will be later explained in section 4.3, a JavaBean-like class has been introduced, meaning a serializable class that encapsulates many properties that can be later accessed by its getter and setter methods, so that all data about an enhanced class can be wrapped in one single object and easily stored. This class is named *ClassData* and it contains the list of test methods of an enhanced class and the name of its log file (i.e. *<className>TOGGLE.txt*).

Additionally, to automatically collect the name of all the test classes, a new class, named *EspressoTestFinder*, has been introduced. This class offers a method to filter the names of the Java files in a given directory according to the following conditions:

1. It should contain an Espresso import declaration;
2. It should contain an Espresso assertion in one of its methods;

3. It shouldn't contain a TOGGLETools invocation (i.e. it should not be an enhanced test file).

This utility class can be used in all the preparatory translation phases since its output (i.e. the actual list of test files to be enhanced in a given directory) is what needs to be given in input to the Enhancer module.

Having both the test class name and its matching *ClassData* object it is possible to group them in a *java.util.Map<K, V>* having as key the class name and as value the *ClassData* object. In this way, it is possible to finally use a single library's method invocation to execute a whole test suite obtaining, as a result, an improvement in the overall usability and maintainability of the library.

Finally, the log file remains at class granularity level and, for this reason, it will be pulled out only after the class execution has been completed, and then it will be deleted from the external storage of the AVD. This implies that the log file will remain in memory for the whole class execution time. In this case, differently from what happened with both screen-captures and XML dumps, the memory consumption of log files is limited (since they are .txt files) thus it is not problematic to keep them in memory for a longer time.

4.2.6 ToggleInteraction generation process

After that all the generated files have been pulled out from the AVD to the host machine, the next tasks of the translation process were the filtering of the LogCat content and the interventions on the screenshots to extract the locators. Particularly, the first one aimed at isolating the lines containing the information on each interaction while the second one, exploiting the information coming from both the log lines and the dump file corresponding to the screen-capture under examination, consisted of a combination of resizing and cropping interventions on the screenshots.

The updates in the whole translation process that have been introduced up to this point made the filtering operation useless. As already explained in section 4.2.1, each log file already groups all the lines that are necessary to generate the various instances of the *ToggleInteraction* class matching the interactions to be translated, grouping them in one file for each enhanced test class. This leads to an improvement in the performances of the library performances. Furthermore, the introduction of the log files makes it possible, for the end-user, to execute independently the instructions to generate

both the required *ToggleInteraction* objects for a specific enhanced class and the relative 3^{rd} generation script files. More specifically, these instructions have become independent both from the first two steps of the translation process and from any other enhanced class.

The outcome is that it is possible, as long as an enhanced test file has not changed and as long as its output couples of screen-capture/dump file are available on the host machine, to re-execute the last two steps of the translation process in different moments. Moreover, since the size of the screen-captures is tied only to the width and height of the mobile device, thus being independent of the host machine, it is possible, simply by sharing the output files of the enhance phase, to generate a different version of the locators for every different model of host machine we want to run the final 3^{rd} generation test scripts on. This is a notable improvement in terms of flexibility and maintainability of the generated GUI tests because it makes it possible to execute once the enhanced tests (that is the most critical and time-consuming step of the whole process) and then share the results, for example, among all team members so that everyone might be able to have its own version of the locators, compliant with the screen resolution of his/her device.

Another relevant element to be improved during the creation process for each *ToggleInteraction* object concerns the number of pixels occupied from the screen of the AVD on the host machine. These dimensions are needed, for example, to be able to generate locators with the right width and height. The initial version of TOGGLE had these dimensions hard-coded. It is easy to understand that, to make the library portable, it was fundamental to address this problem.

The first step of the implemented mechanism to solve this issue was to introduce the *Java Native Access* (JNA) library among the dependencies of TOGGLE. This library makes it possible to use native library methods without further requiring native code of any kind.

More in detail, this library has been introduced because it offers some classes that can search, among all the opened windows, for the one containing a given string in its name, evaluate programmatically the actual width (or height) in pixels of the selected window and obtain the screen width and height of the host machine.

Thanks to these features, a solution has been implemented to evaluate, with sufficient precision, the actual width and height of the screen of the AVD.

Going step by step, it is fundamental to observe that, to resize a given

screenshot so that it can assume the resolution of the emulated screen (and not the resolution of the real mobile device), a ratio between the measures of the physical device and the actual ones must be computed. TOGGLE compute this measure with the following formula:

$$r = \frac{AVDWidth}{PhysicalDeviceWidth} \tag{4.1}$$

It is worth noticing that this measure can be considered as a reduction factor, thus it could be computed on one dimension only being valid for both of them. In other words, the computation of this ratio on the height dimension will produce the same ratio.

The *PhysicalDeviceWidth* can be obtained through an ADB instruction:

```
adb shell wm size
```

The *AVDWidth*, instead, is more challenging to be computed automatically. In the initial version of TOGGLE, as previously presented, this value was predetermined and fixed. This made it impossible to have a portable version of the test cases. To obtain this measure automatically a new utility class has been implemented, named *WindowUtils*, that groups all the methods implementing the operations that are necessary for extracting the searched values. The first variables to be obtained are the dimensions of the window containing the emulated device. To achieve this result, a function has been introduced. Particularly, this method, starting from a sub-string that should belong to the name of the window under exam, can programmatically retrieve both its height and width thanks to the APIs of the JNA library. The default value for the sub-string is "Android Emulator", which is a fixed value that is always present in the default name assigned to a new AVD from the Android Studio IDE.

This process highlights a constraint: there should be at most one running AVD during the translation process. This is a reasonable limitation for several reasons. For starting because the Espresso test cases will run on one device per time. Additionally, each device has its resolution thus it will produce different versions of the same locators and, for this reason, it makes sense to perform separate translation processes for separate mobile hardware models. Finally, none of the presented command-line instructions would work with more than one running emulator.

After having collected the dimensions of the window, the next step is the computation of the percentage of these dimensions that are actually occupied

by the screen of the AVD. An AVD window typically contains, besides the emulated screen itself, some other visual components like, for example, the device frame of the emulator (different according to the selected AVD skin) and an options menu. It is important to say that all the computations presented in this section refer to emulators with the device frame enabled.

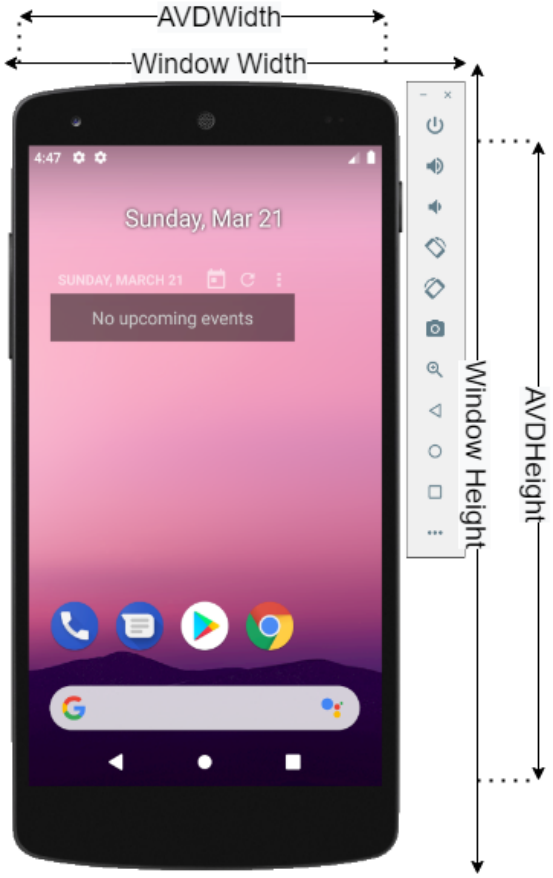


Figure 4.2. Measures on the emulator window.

This percentage has been evaluated empirically for some of the default AVD skins that it is possible to select while creating a new emulated device from Android Studio IDE. The followed mathematical procedure starts from some basic measures like the pixels per inch (ppi) of the host machine, its pixels per centimeter (ppcm) obtainable by computing the ratio between ppi and the number 2.54, the dimensions of the window, and the measured width and height in centimeters of the emulated screen. Having computed these measure, to compute the number of pixels of the width of the emulated screen

the formula to be applied is:

$$px = measuredCm * ppcm \tag{4.2}$$

Where px is the number of pixels of the emulated screen, $measuredCm$ is the measured size in centimeters and $ppcm$ is the number of pixels per Centimeter. This formula can be applied interchangeably to compute both width and height. This value is computed automatically and not predefined but it is still dependant on the host device, meaning that, to reach the goal of having a library as portable as possible, a further step had to be taken. In particular, the ratio between the dimensions of the emulated screen and the window ones has been computed to have a value that is not tied to the actual number of pixels in a particular screen but, instead, a value that can be used regardless of the characteristics of the host machine.

$$ratio = \frac{px}{windowDim} \tag{4.3}$$

In this case, $windowDim$ can be either the window width or height. Thanks to this formula it is clear that, if both $windowDim$ (as previously presented, it is possible to retrieve this value programmatically) and the ratio values are known then it is possible to obtain the value of px without needing any further assumption on the characteristics of the host machine, thus increasing the level of portability of the library.

Table 4.1 shows a list of the presented measures collected on an Acer Aspire A715-71G-743, with an Intel i7 of 7^{th} generation processor, a screen resolution of 1920x1080, and running Windows 10 OS.

Finally, using the computed ratios, it is possible to retrieve both the width and height of the emulated screen starting from the dimensions of its wrapping window. It is worth noticing that, in this case, the ratio should be considered as the percentage of the total width or height occupied by the correspondent measure of the emulated screen, and not as a simple reduction factor. In fact table 4.1 shows that there are different ratio values for height and width, highlighting that each one applies exclusively to a single dimension. This is a consequence of both the difference between the screen resolution of the AVD skins and the way these skins affect the window dimensions.

Table 4.1. Ratios and resolution per skin of the emulator

Emulator skin	Window dimensions (px)	Screen dimensions(px)	Emulator resolution	Width ratio	Height ratio
GALAXY NEXUS	557x865	366x651	720x1280	0.6570915619389587	0.7526011560693642
NEXUS 4	493x864	396x661	768x1280	0.8032454361054767	0.7650462962962963
NEXUS 5	492x864	391x695	1080x1920	0.7947154471544715	0.8043981481481481
NEXUS 5X	484x865	385x684	1080x1920	0.7954545454545454	0.7902665121668598
NEXUS 6	536x864	408x726	1440x2560	0.7611940298507462	0.8402777777777778
NEXUS 6P	489x864	385x684	1440x2560	0.787321063394683	0.7916666666666666
NEXUS ONE	536x863	355x589	480x800	0.6623134328358209	0.6825028968713789
NEXUS S	546x865	369x612	480x800	0.6758241758241759	0.707514450867052
PIXEL	467x864	369x656	1080x1920	0.7901498929336188	0.7592592592592593
PIXEL 3	407x864	371x742	1080x2160	0.9115479115479116	0.8587962962962963
PIXEL XL	482x864	377x670	1440x2560	0.7821576763485477	0.7754629629629629

4.2.7 3^{rd} generation script creator

The improvements affecting this module involve mainly three aspects: the mechanism to launch the generated test suites, the mechanism to reset the state of the AUT after a test case execution, and, finally, how the jar files containing both EyeAutomate and SikuliX APIs are inserted into the new project.

The state-of-the-art version of the library required a manual intervention to inject the jar files into the project and had no method to run at once all 3^{rd} generation test classes. Specifically, every Java class had a main method that could be launched to run all the test methods of the class so, to run completely the test suite, it was necessary to launch manually one by one every Java main or to modify the generated project. This was, of course, not maintainable nor usable in an industrial context. For this reason, the translated Java classes have been updated by substituting their *main* method with an equivalent *run* one and by including in the generated project a Java class including a unified *main* containing the code to launch every *run* method.

Furthermore, a utility module offering two Java methods has been implemented. The first one is used to start the AUT on the emulator through the following ADB command-line instruction:

```
adb shell am start -n <appPackage>/<
  pathToMainActivity>
```

Where *<appPackage>* is the package name of the AUT and *<pathToMainActivity>* is the relative path to the main activity file. The second one stops the AUT with the following ADB command:

```
adb shell am force-stop <appPackage>
```

The purpose of this new class is to automatically reset the state of the AUT so that every 3^{rd} generation test case can start from the same initial state, emulating the behavior of the Espresso source tests.

Finally, another improvement that has been introduced is the automatic injection of the jar files containing the Java API of the 3^{rd} generation adopted frameworks. In this case, since the injection operation is costly in terms of execution time, a mechanism has been implemented to detect whether these files have already been injected or not and to copy them in the project folder only the first time the translated project is generated so that all following translations won't suffer from this time overhead.

4.3 Re-design

The improvement phase, even though it addressed most of the issues of TOGGLE, especially the ones related to its maintainability and portability, pointed out that the library still had a usability problem. The main reason is that to have a working translation process, it is mandatory to use in the correct order all the various modules composing the library.

This implied that an end-user willing to adopt TOGGLE to translate his/her Espresso test classes into 3^{rd} generation ones had to experience a steep learning phase that could likely discourage him/her from adopting the library.

Furthermore, the whole sequence of method calls, since it involves different modules and different execution phases, could likely be a source of errors even for experienced users.

This analysis highlighted the necessity to make the learning curve for a user more shallow so that both the normal translation flow and a more customized set of tasks could be achieved with a lower effort.

For this reason, a complete re-design of the library structure was introduced. Particularly, a structural design pattern, known as the "Façade pattern", has been adopted to offer a cleaner interface to the features of the library, adding the possibility to use its modules both from a high-level point of view and a lower-level one. The first one is more suitable for normal users that need to use the library for a basic translation process while the second one offers some methods that let the user access every single module with their functionalities without exposing the deeper level implementation details that are automatically handled by the library itself.

4.3.1 Façade pattern

A design pattern, by its definition, is a way of solving well-known problems through the application of some best practices that a programmer should follow when facing similar situations. One of the first and, probably, most influential categorization of design patterns was made by the GoF (Gang of Four, a nickname for the authors of *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) [15] that classified these patterns into three categories:

creational patterns : they focus on the instantiation process so that the system implementing this pattern can be independent of the object creation. This can be done, for example, by exploiting inheritance so that, according to some external conditions, with one single instantiation instruction it is possible to create objects that are instances of several different classes. Another example could involve the adoption of some Factory classes, which are classes in charge of creating other objects in place of the user. These patterns increase the flexibility of a system, making it independent from object creation, encapsulating the instantiation complexity, and hiding who or what generates an instance of a class and when this instance has been created;

structural patterns : they focus on how different modules of a system can be linked to group them into bigger structures. The purpose of these patterns can be merging two different systems into one, grouping different objects so that the programmer can be able to introduce new functionalities, using "proxy" objects acting in place of other objects (for example in place of remote ones), sharing objects without suffering of memory overhead, using different parts of a system from a single interface object, separating an object abstraction from its actual implementation so that this can be changed safely or enriching an object dynamically by adding to it new characteristics and responsibilities;

behavioral patterns : they focus on algorithms and the responsibilities of objects. They model the control flow of communication among objects so that it can be easily maintained and updated. Patterns belonging to this category use inheritance to share common behaviors, common algorithms, and common communication motif among different classes. This can be done, for example, by offering some abstract classes implementing an algorithm that uses some base operations whose implementation is delegated to the concrete classes, so that changing them will actually

make the same algorithm to be portable in several contexts. Another example could be the usage of class composition, which focuses on how to make different objects cooperate so that it is possible to implement a new feature that was not obtainable otherwise. Other improvements offered by these patterns are the possibility of gaining a looser coupling in the system, the possibility of dynamically maintaining dependencies among objects, and some solutions encapsulating in a single object a whole set of behaviors so that they can be reused in the system.

Another criterion, used to further distinguish patterns belonging to the same category, is their scope. A design pattern may have:

class scope : patterns having this scope focus on static relationships (i.e. established at compile-time). This kind of relationships is typical of classes and sub-classes and it is established through inheritance;

object scope : patterns having this scope focus on dynamic relationships among objects, meaning a relationship that might change at run-time. The majority of the design patterns fall under this scope.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight (195) Observer State Strategy Visitor

Figure 4.3. Design patterns.

The Façade pattern is one of the twenty-three well-known design patterns presented in *Design Patterns: Elements of Reusable Object-Oriented Software*[15] and belongs to the structural category with object scope. Picture 4.4 shows the sequence flow of a generic system adopting this pattern.

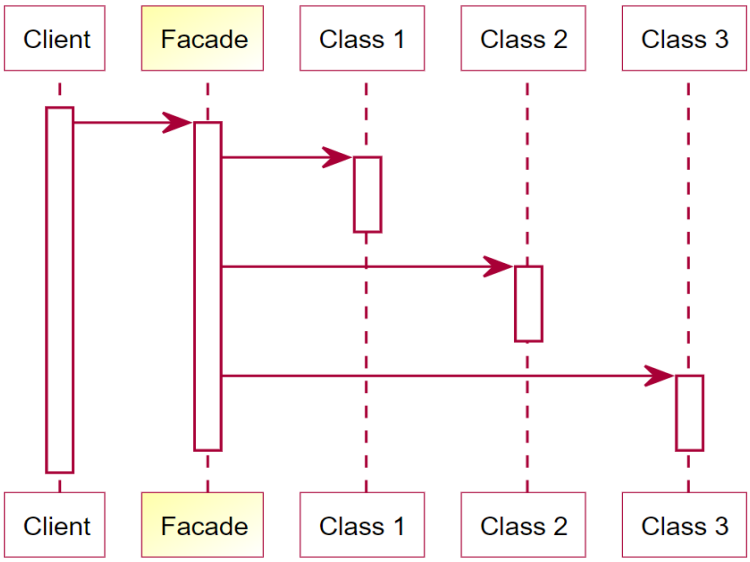


Figure 4.4. Façade pattern sequence flow[1].

There are many reasons to adopt this design pattern. The most popular ones are the need to isolate parts of a system by interposing among them one façade class for each subsystem that will act as the entry-point for its reference module, the need to decouple clients with the implementation of abstract interfaces and, finally, the need to provide a simple interface to a system (or to part of it) that will be good enough for most clients, leaving the direct access to the subsystems to the few clients needing for customized behaviors.

As a matter of fact, by embracing this pattern the adoption entry barrier to TOGGLE for an end-user has been drastically reduced, offering a set of APIs both hiding completely the complexity of the translation flow, thus making it possible to execute the whole translation with one single method invocation, and ensuring the correct usage of every single module separately. In particular, as already presented in sections 3.3 and 4.2, some modules of TOGGLE need some preliminary actions that, at the state-of-the-art, are

[1]From: https://en.wikipedia.org/wiki/Facade_pattern

under the direct responsibility of the programmer even though they involve some internal mechanism. An example might be the Executor component that requires the classes *TOGGLETools* and *BitmapSaver* to be injected into the test folder. For this reason, even though, for example, after all the interventions the library offers some methods to inject these classes where needed, some additional interface methods have been included in the façade class to lift the burden of ensuring that all the preliminary steps have been correctly completed from the end-user.

Another key factor in favor of the adoption of this pattern is the possibility to improve the decoupling between the library implementations and the code using them. A façade class, in fact, simply by standing in between the clients and the system, facilitates the concealing of the evolution of internal library modules, making them independent from how the functionalities offered by the system are exploited by the end-users.

To summarize, by re-designing the architecture of the library with the adoption of the façade pattern, the goal was lowering the learning effort required to use the library without giving away the possibility to a more customizable use of it, which could be suited for more experienced users.

4.3.2 TOGGLE class

The façade class of TOGGLE is thought to be placed between the modules of the library and the end-user, offering different abstraction levels and, consequently, a wide range of operations.

The main incentive to its introduction was the complex set of interactions among the components of the tool and with the AUT, which often led to unwieldy Java methods implementing the whole translation. More in detail, these methods required a high number of instructions to be invoked in the correct order, other than some hard-coded values. The outcome was a method having an excessive number of lines of code, that was not portable and highly error-prone.

For this reason, the introduction of the Façade structural design pattern was the right solution to re-design the interface of the library. The starting point was to isolate the fundamental data that should come as input from the end-user so that it can be possible to carry out the translation process without any further user intervention. These data are:

- **rootTestDirectoryName**: commonly, it will assume either the value "androidTest" or, eventually, "test";

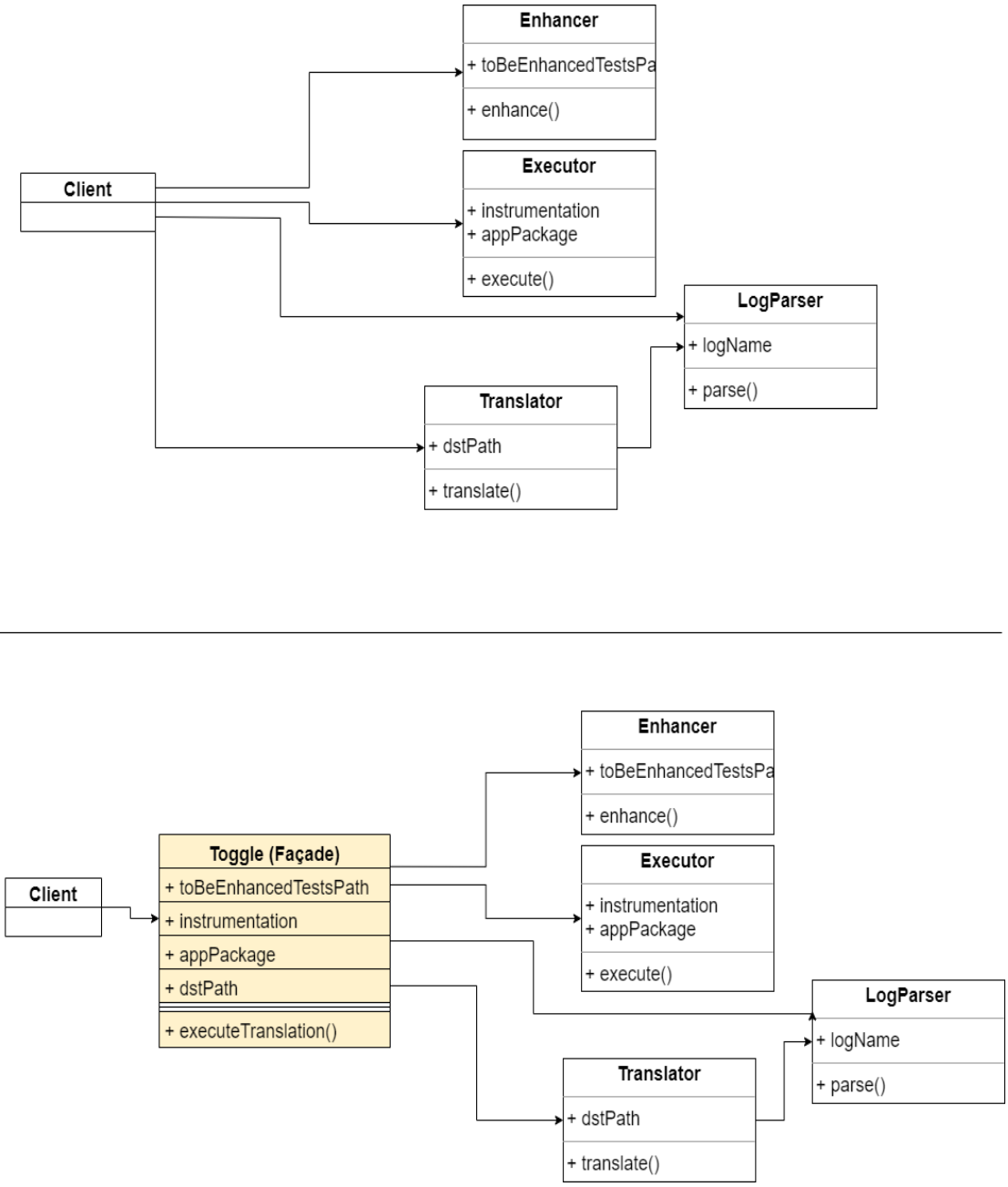


Figure 4.5. Toggle façade class.

- **guiTestPath**: the path to the directory where translated GUI test should be generated into;
- **appPackageName**: the package name of the AUT;

- **testDirectoryPath**: the path to the directory containing the Espresso test classes to be enhanced and translated;

- **device**: the hardware model of the AVD.

From this set of information it is possible to extract automatically, for example, the root folder of the application, that will be used to run the Gradle instructions, or the graphical reduction factor, to programmatically evaluate the dimensions of the emulated screen.

Once the input data to create a *Toggle* instance has been defined, the next step was selecting a set of interface methods to be offered to the end-user.

The most important ones wrap the sequence of instructions required to complete a whole translation process, offering the possibility to execute it with one single method invocation.

Two versions of this method are offered: the first one with method-level execution granularity, which was introduced with the re-engineering of the library in section 4.2.5 and that has no constraints on the number of test methods in a single class, and the second one with class-level execution granularity, that requires a test class to have a limited number of Espresso methods and assertions.

Then, to increase the interface flexibility so that an end-user requiring a customized set of interactions with the components of the library is not forced to directly access the internal modules, some interface methods performing lower-level operations have been included in the façade class. More in detail this class offers a method to inject (if They are not already present) TOGGLETools, BitmapSaver, and ScrollHandler (that will be introduced in section 4.4) into the Espresso test folder, a method to enhance one single test class starting from its absolute path, a method to enhance a whole test folder starting from its absolute path, some methods to execute, after performing all the preliminary operations, the enhanced tests with both granularity levels, a method to re-install the application on the target device, some methods to pull, clear and delete all the different kinds of files generated after the execution of an enhanced test, a method to grant the needed permissions to the AUT, a method to retrieve the string identifying the test instrumentation of the AUT, a method to generate the 3^{rd} generation test methods and more.

As will be later presented in chapter 5, an empirical evaluation proved that the introduction of this class has drastically improved the usability of the library, reducing the development time and its time-to-learn.

4.4 New features

After the completion of both the phase of tool improvement and the redesign one, TOGGLE had a solid base for the development of some further extensions. Specifically, the lack of support of the tool for the "scrolling" interaction with the screen of the emulator was still an issue to be overcome, especially knowing that this kind of input is widespread in the mobile world. This interaction can be executed through two different *Espresso* instructions:

onData : an instruction that interacts on an *AdapterView* (or on an object that is an instance of one of its descendant classes), finding among its child elements the one that matches the specified conditions and automatically scrolling to it;

scrollTo : an instruction that interacts on a "scrollable" View (i.e. an instance object of the following three classes: *ScrollView*, *HorizontalScrollView*, or *ListView*), searching among its children the one that matches the specified conditions and scrolling until it is displayed on the screen.

Additionally, this specific interaction presents some challenges that required considerable effort and the definition of some constraints to be surmounted. For example, even though the final interaction is actually the same, the semantics behind a scroll executed after an *onData()* instruction differs from the one behind the same operation but following a *scrollTo()*.

As displayed in picture 4.6, considering the case of a scroll-down interaction, the first one will scroll to the matched item until either it is displayed as the first element of its container or it is not possible to scroll down anymore, meaning that the AdapterView does not contain enough elements after the matched one. On the other hand, *ViewActions.scrollTo()* scrolls to the matched view until it appears on the screen. This implies that it does not try to bring it as the first displayed element of its container. Picture 4.7 provides an example of how a *scrollTo* instruction behaves in case of a scroll-down interaction.

Another challenge that this interaction presents is related to the possibility of performing a diagonal scrolling with one single Espresso instruction. This may happen if, for example, the container ScrollView is bigger than the screen of the device and, consequently, it is possible to scroll the screen in all directions. To translate these kinds of interactions there are two options, according to how a specific scroll interaction is translated. If the translated scroll interaction will happen, in any case, along with one single direction

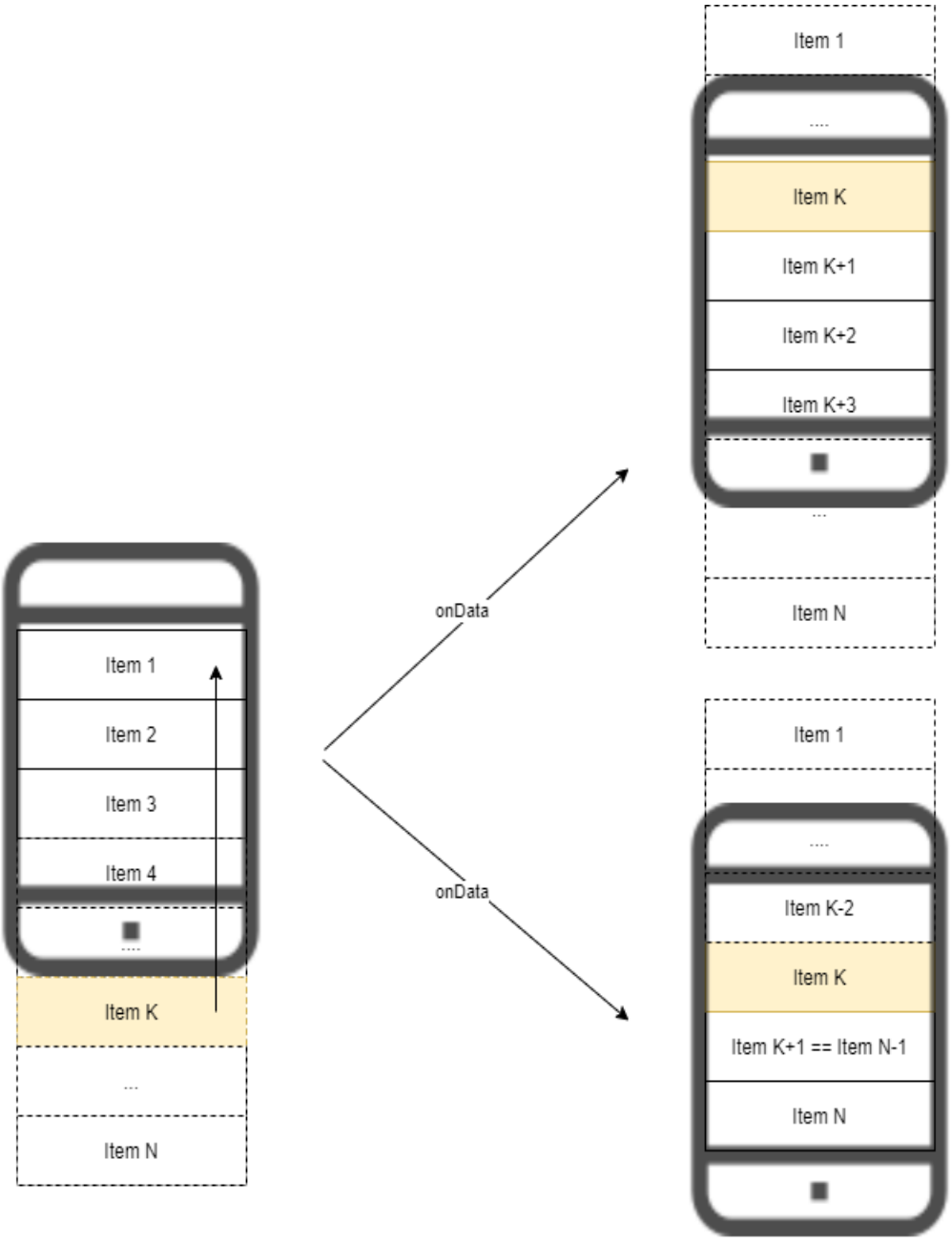


Figure 4.6. Espresso.onData() scrolling semantics.

then, to translate a diagonal movement, it is necessary to combine multiple translated interactions to emulate the diagonal original one. Alternatively, if the translated interaction can follow a diagonal path too, it is possible to translate every scrolling interaction in every direction with one single

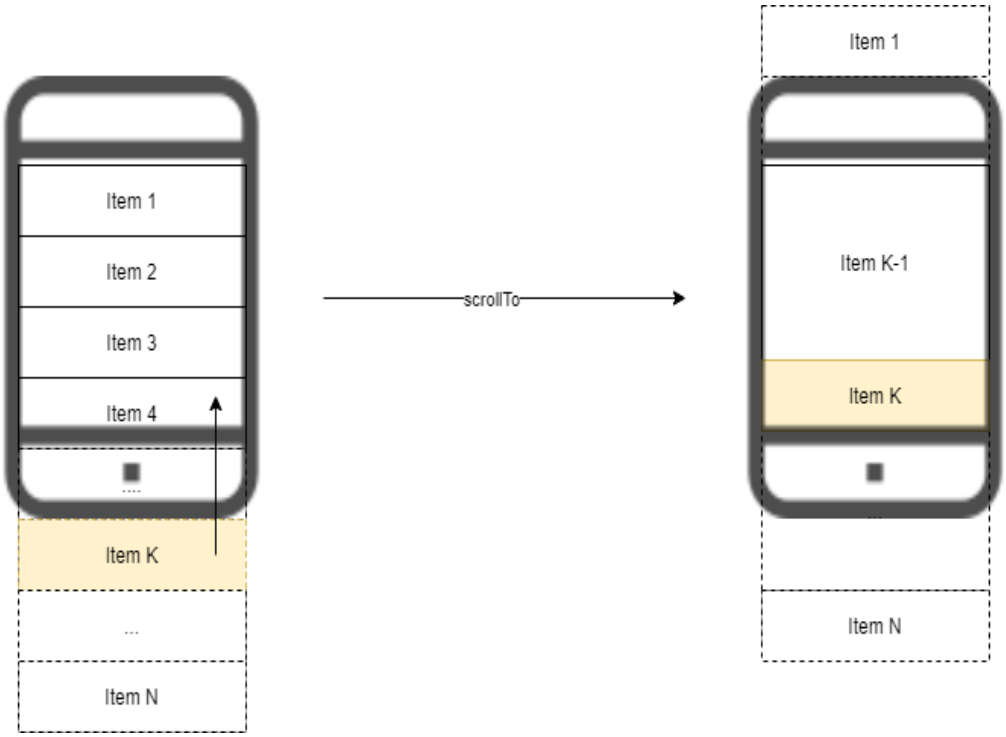


Figure 4.7. ViewActions.scrollTo() scrolling semantics.

ToggleInteraction object.

The second solution has been adopted for various reasons. For starting, even though the creation of the single object containing the information needed to emulate one movement was simpler with the first solution, the translation complexity increased drastically. The main reason is that a lot of additional measures had to be collected. Furthermore, it was necessary, while creating the *ToggleInteraction* objects, to detect whether the scrolling happened in one single direction or multiple ones so that the right number of *ToggleInteraction* objects could be created. Another issue was related to the need for "intermediate" locators. While the Espresso instructions perform this movement at once, if the translated interaction had to be split into multiple ones, one for each dimension, it was necessary to have a locator for each one of them. This implies that the execution order of the translated instruction was a key factor to be considered. Moreover, a mechanism to collect the "intermediate" locators had to be developed, increasing both the overall complexity and the fragility of the 3^{rd} generation tests. These issues, plus the different semantics of the Espresso scroll instructions, made the

adoption of the first solution prohibitive.

The development of the second solution introduces the following four *ToggleInteraction* inheriting classes: *ScrollDownRight*, *ScrollDownLeft*, *ScrollUpRight*, and *ScrollUpLeft*. The measures collected during the execution of the enhanced test method containing these kinds of interactions are the starting X and Y coordinates, the final X and Y coordinates, the width and height of the container, and its top-left coordinates inside the screen. To go into further details, the conceptual order of the actions that should be carried out during the execution of an enhanced test class to correctly collect all information and to translate a 2^{nd} generation scroll interaction into a 3^{rd} one is:

1. Get a reference to the container of the searched item;
2. Take a screen-capture, that will be used to match the starting point of the scrolling interaction;
3. Create the XML dump file, that will be coupled with the screen-capture of point 2;
4. Collect the starting coordinates inside the container;
5. Execute the 2^{nd} generation test assertion;
6. Collect the final coordinates inside the container;
7. According to the initial and final coordinates, identify the scrolling direction;
8. Log a new line containing all the collected information;
9. Take a screen-capture of the displayed screen after the scroll interaction;
10. Create the XML dump file, that will be coupled with the screen-capture of point 9;
11. Log a new line performing the original assertion on the located View.

Some additional intermediate operations could be needed, according to what kind of scrolling interaction is being executed, and they will be presented in sections 4.4.1 and 4.4.2.

Point 7 of the list presents a core decision that should be further analyzed. Since we are considering diagonal movements, the case of a mono-directional

scrolling interaction (that is the most common one) should be detailed to explain how it can be translated with the adopted mechanism.

To explain how this could work it is necessary to detail the specific translation process of a scroll interaction, starting from the enhance phase. As it can be seen from the sequence of actions that have been previously listed, a couple of files screen-capture / XML dump is taken before the execution of the actual scroll movement, performed by the Espresso test assertion, so that the data on the initial status of the application, meaning the one before the movement such as, for example, its starting X and Y coordinates inside the container View, can be collected. Then, after the Espresso assertion, the scroll has been completed and it is possible to collect the data on the state of the AUT after this operation (particularly its final X and Y coordinates inside the container View) and, finally, to log all these information.

This should be considered as the first phase of the translation, that is the one aiming at the definition of the scroll direction and of how many pixels have been scrolled in that direction. The key pieces of information in this phase are the X and Y starting and final coordinates, which identify the initial and the final offsets plus its direction. In fact, if the final X coordinate differs from the initial ones it means that there has been a movement along the horizontal axis (that might be either right or left according to whether the final X coordinate is, respectively, greater or smaller than the initial one), if instead, the two measures are equal it means that the scroll movement did not affect the horizontal axis and, consequently, it was only a vertical scroll. The same concepts can be applied to the vertical axis.

By convention, if a scroll interaction does not impact the horizontal coordinates it will be assigned to the right direction and if it does not impact the vertical ones it will be assigned to the down direction. This information will be logged too, specifically in the interaction-type field.

The next step involves the generation of the *ToggleInteraction* object. The constructor of the specific scroll interaction instance analyzes the log line to extract a locator from the screenshot and a couple of start and end points inside the area of the container. Particularly, the first operation is to extract a locator. This operation depends on the source Espresso instruction and it will be further detailed in sections 4.4.1 and 4.4.2 but, generally, its main objective is to extract the bounds of an element inside the area of the container so that it could be used as the starting point for the subsequent computations determining all the parameters required to emulate the movement. It is important to specify that, since the screen-capture frames the state of the AUT preceding the scroll interaction, it is not important

to locate a precise element, this step is needed simply to determine a fixed element to be located during the 3^{rd} generation test execution from which the emulation of the interaction could start.

Once the bounds of this element have been extracted from the XML dump and the screen-capture containing it has been correctly cropped and re-sized, two points inside the area of the container should be determined. They will act as starting and ending points of the movement giving, consequently, a measure of the number of pixels that could be scrolled along both axis within the margins of the container. The computation of the coordinates of these points will be performed separately for each dimension. Specifically, if the logged line, relative to the interaction under examination, reports that on the X-axis there has been a movement then the two points will have a horizontal component that is proportional to the number of pixels to be covered in this direction. The same thing will happen along the Y-axis. In this way, it is possible to have movements following a single dimension simply by putting the component on the other axis to 0. The source of this reference system is the center of the locator whose bounds in each direction have been extracted before the start of these computations.

Each point will keep a 20-pixel margin from the limits of the container to avoid the risk of starting the scroll interaction by clicking on an external point or on the border of the View, which would result in a faulty translation. The value 20 has been determined empirically and it has proven to be the optimal solution to ensure that the start and the end of the movement are constrained within the borders of the container. Two more measures to be established are the total distance to be scrolled in each direction. These measures could be easily determined from the initial and final offsets reported into the log line. It is important to specify that each measure, that should be used in the translated test, should be resized by multiplying it by the resize factor (specific for each dimension) defined in section 4.2.6.

The latter computation concludes the set of operations to be carried out during the construction of a single interaction instance.

Finally, it is important to detail how, starting from these measures, a scroll movement could be emulated. Particularly, a scroll movement is composed of a pressure on a point on the screen, a movement along a direction maintaining the pressure, and, finally, the release of the pressure. This set of operations could be repeated in a loop until the View of interest is displayed on the screen. The steps that will compose the emulated interaction follows exactly this template, using as the point to start the pressure and as the point to release it respectively the two points whose coordinates have been established

in the construction phase of the interaction, and as the number of pixels to be scrolled the specific measure computed in the same occasion.

The last trait to be determined is the number of repetitions of this movement. This measure will be computed by dividing the total distance to be covered by a scroll-step, a value measuring the distance between the start- and the end-point of the movement. Of course, this ratio could have a remainder that must be taken into account to emulate with the maximum precision possible the scroll movement.

Furthermore, an empirical evaluation, performed during the experiment detailed in chapter 5, proved that it is important to deal with the sensitivity of the emulator and, particularly, it is necessary to avoid moving the cursor too fast after the beginning of the pressure so that no pixels would be lost due to a non-deterministic behavior of the emulator when dealing with quick movements. Another factor affecting the precision of the translation involves the duration of the time slot that should elapse between the end of the movement and the release of the pressure. This event should happen after keeping the cursor stationary for a while so that the scroll won't continue after the pressure has been released. Both these issues have been solved by interposing some sleep instructions between the actual commands driving the cursor. Particularly, the movement is always divided into three segments of equivalent length, and, between the two movements, a sleep instruction of $0.25s$ is executed. This approach has proven to be the most effective one to obtain precise and effective scroll movements.

Additionally, between the start of the pressure and the actual movement of the cursor, there is a sleep time of $0.1s$ and, finally, after reaching the ending point, an additional sleep instruction of $1s$ is inserted to stabilize the state of the AUT.

If the number of pixels to be scrolled is either smaller than the scroll-step or not a multiple of it, the "remainder" number of pixels needs to be computed. This value may be easily retrieved by, yet, another ratio computed with the following mathematical formulas:

$$totToBeScrolled = \sqrt{(\text{toBeScrolledY}^2 + toBeScrolledX^2)} \tag{4.4}$$

$$totScrollStep = \sqrt{(\text{scrollYStep}^2 + scrollXStep^2)} \tag{4.5}$$

$$repetitions = \frac{totToBeScrolled}{totScrollStep} \tag{4.6}$$

$$remainder = totToBeScrolled \bmod totScrollStep \tag{4.7}$$

$$ratio = \frac{remainder}{totScrollStep} \tag{4.8}$$

Where *toBeScrolledY* and *toBeScrolledX* are the overall number of pixels to be scrolled along the Y- and the X-axis while *scrollYStep* and *scrollXStep* are the numbers of pixels of a single scroll-step divided into the two orthogonal dimensions. These four values have been determined during the construction phase. Coherently, *totToBeScrolled* is the overall distance to be covered, *totScrollStep* is the total length of a scroll-step, *repetitions* is the integer ratio among the previous two values and represents the number of times a movement of *totScrollStep* pixels has to be executed, *remainder* is the result of the modulus operator and represents the remainder number of pixels exceeding the ones covered during the *repetitions* iterations and, finally, *ratio* is the percentage of *scrollYStep* and *scrollXStep* that, coupled together, generates a movement of *remainder* pixels.

Finally, after the scroll interaction has been completed, an additional group of screen-capture, XML dump file, and line in the log file should be considered (points 9, 10, 11 of the previous list). These components have the purpose of either performing a ViewActions assertion on the destination element (e.g. the one that should be displayed on the screen after the scroll interaction) or checking if it is shown as expected.

All the utility methods implementing the code required to retrieve programmatically the measures, involving this kind of interaction, to be inserted in the log lines are offered by a class named *ScrollHandler*, injected together with the *BitmapSaver* and the *TOGGLETools* ones during the enhance phase.

It is important to specify that, while these kinds of interactions are performed automatically in 2^{nd} generation Espresso tests, the same cannot be said for what happens in the translated 3^{rd} generation ones. Specifically, the latter willingly emulates the movement that a human user would do, and, for this reason, it is unlikely that the final screen displayed after the interaction would match exactly the one that follows the execution of an Espresso equivalent assertion. Other causes behind this behavior are the non-deterministic number of pixels lost during a single movement due to the screen sensitivity of the AVD, which could be mitigated by the use of slow movements but not completely deleted, the margin of error caused by the adoption of the

reduction factors and, finally, the margin of error caused by the resizing of the screenshots.

This is not, by itself, a source of faulty translations since the purpose of these tests is to scroll in a certain direction until the requested element is displayed, and not until the same exact screen status is displayed. Knowing this, the decision was, while trying to mitigate as much as possible the source of errors in the number of pixels to be scrolled, to remove any *FullCheck* that would follow a scroll interaction unless the final operation, to be carried out on the scroll destination element, causes a switch in the appearance of the AUT, meaning a change in the displayed Activity/Fragment or, for example, the opening of a new Dialog Window. The FullCheck operation is an additional check placed at the end of every translated test method to verify the final state of the application after the set of interactions composing a test case. This assertion is likely to fail if, after a scroll movement, the final state does not match exactly the one displayed in the screen-capture taken during the execution of the source enhanced test method, even though the searched element has been correctly found by any precedent assertion. Of course, as previously stated, if the operation on this element causes a radical change in the appearance of the application the FullCheck operation would be enabled again.

Finally, sections 4.4.3 and 4.4.4 will explain two additional minor features that have been added to expand the set of translatable assertions and their combinations.

4.4.1 OnData

After the general description of how the translation of a scroll interaction is performed, in this section, the case of the *onData()* Espresso instruction will be detailed. This method can be considered the entry point for all tests that want to perform some assertions on elements contained in an *AdapterView*[1]. An *AdapterView* is an Android View whose purpose is to display a group of child elements contained in an adapter. An adapter is an object connecting the AdapterView with the data relative to its children, providing access to it. Furthermore, it is responsible for instantiating the View objects that will be used to graphically display the items in the data set. Android offers some predefined specific adapters according to how the data of the items are arranged. Some examples are *ArrayAdapter*, *CursorAdapter*, and *SimpleCursorAdapter*. Furthermore, the *AdapterView* class has some derived classes offering different behaviors. The ones directly introduced by Android

are *ListView*, *GridView*, *Spinner*, and *Gallery*. Picture 4.8 shows a general schema to explain how objects of these classes interact with one another.

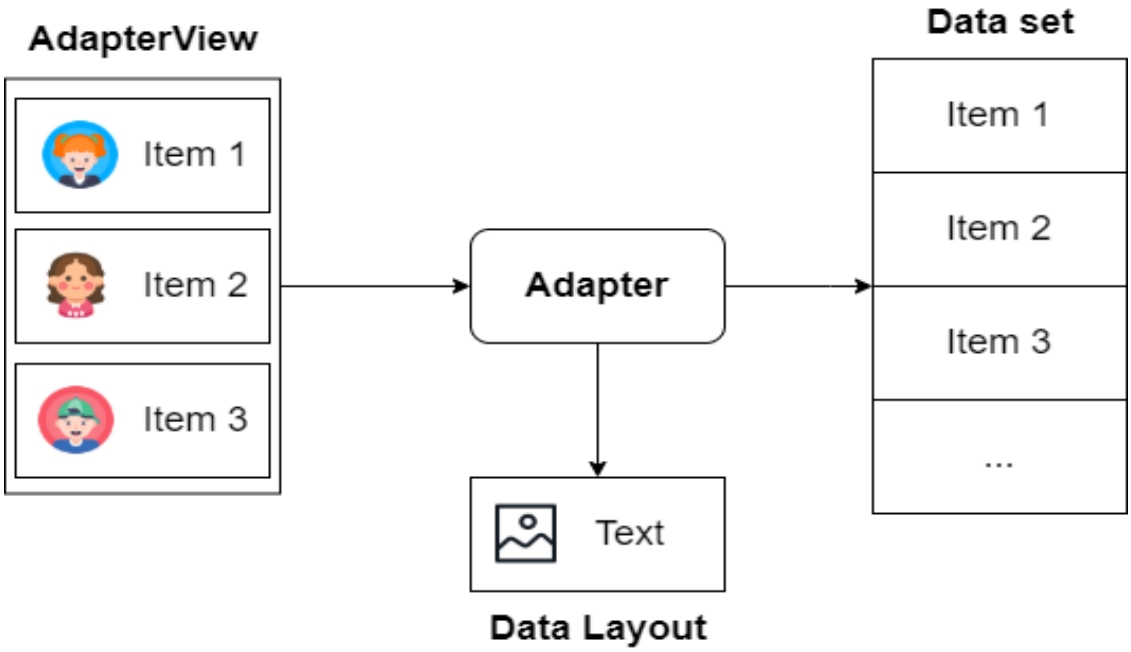


Figure 4.8. AdapterView schema.

It is worth mentioning that the child elements might not be rendered on the screen of the emulator at the moment of execution of the *Espresso.onData()* instruction. Specifically, this instruction will return a *DataInteraction* object related to the matched element and accepting all *ViewMatchers*.

By default, it will search for the element of interest among all the elements contained inside every displayed AdapterView and, then, it will automatically scroll to it. It is possible to restrict the search space to all the elements of a specific AdapterView through the method *inAdapterView()*, which accepts a ViewMatcher to uniquely identify the AdapterView that should contain the element corresponding to the DataInteraction object. This method has proven to be extremely useful since the scroll movement should start inside the area of the AdapterView to be correctly executed. Furthermore, to find the first displayed element, as explained in section 4.4, it is necessary to isolate the container. For these reasons, a new constraint has been introduced making mandatory the adoption of this method while executing an onData Espresso assertion. Another reason behind its adoption, as explained in section 4.2.3, is that thanks to this method, it is possible, while parsing

the Espresso source class, to inject the lines of code necessary to obtain a reference to the instance object corresponding to the specific AdapterView.

Another unique method, typical of this context, is the *atPosition()* one, matching an element placed at a given position inside an AdapterView. For this method, the translation support has been added thanks to a dedicated algorithm that will be invoked when the log line field *searchType* matches the string "atposition". In this case, the *searchKeyword* will contain the id of the AdapterView of interest, the index searched position, the index of the first element displayed, and the index of the last one, each of them separated with a specific separator. Respectively, these pieces of information will be used to identify the container inside the XML dump, to find out the exact position of the element of interest (this field will be equal to the value passed as a parameter to the Espresso *atPosition()* method), and to find out how many elements at once can be displayed on the screen, a piece of fundamental information to translate the searched position into a valid index. In fact, the indexes reported in the log lines usually assume a value that is different from the one of the XML dump file. More in detail, the index attribute of the child elements of an AdapterView in an XML dump file will always assume a value between 0 and N, where N is the total number of elements displayed minus one. To make an example, starting from the following Espresso instruction:

```
onData(allOf())
    .inAdapterView(
        withId(
        R.id.settings_category_list
        ))
    .atPosition(12)
    .check(matches(isDisplayed));
```

and knowing that the AdapterView of interest has twelve elements displayed over 15 total ones, the log line relative to the atPosition assertion (that will follow the one relative to the specific scroll movement), will assume a similar format:

```
<testName>; <testName><number>; atposition;
   settings_category_list_;12_;3_;14; check
```

where "_;" is the separator. Following this example, the XML dump file would have the following content:

In picture 4.9 the red squares highlight the id of the AdapterView and the index values.

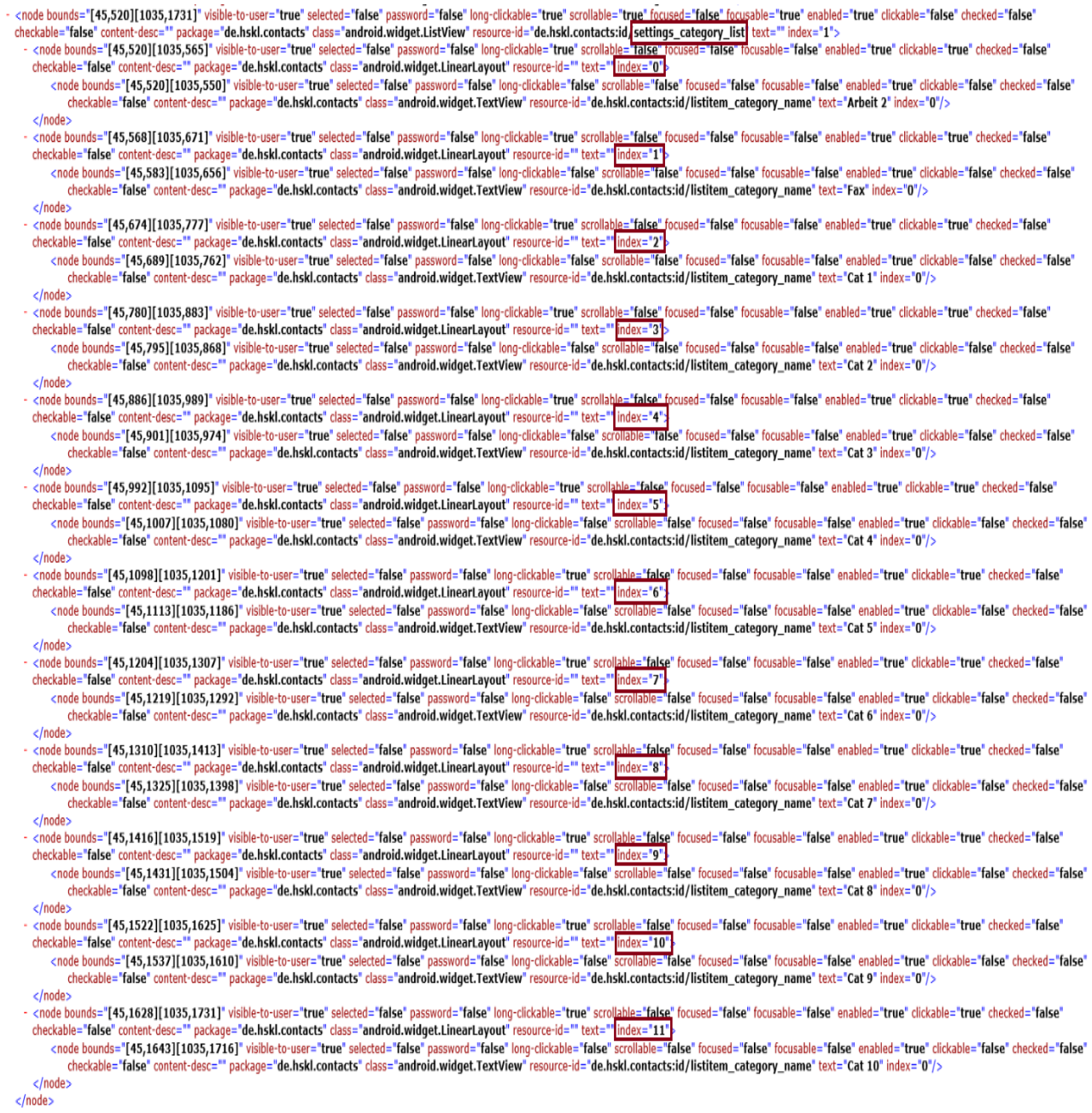


Figure 4.9. XML dump file for atPosition search type.

In this case, keeping in mind the information reported in the log line, the node with index 0 corresponds to the third element inside the AdapterView while the one with index 11 corresponds to the fourteenth one. Consequently, the searched item will be the one with index 9.

This example highlights behavior that is typical of the onData instruction and that has already been introduced in section 4.4: this instruction tries to scroll until the element of interest reaches the top of the screen or, in case we are scrolling down, until it is not possible to scroll down anymore since there are not enough elements left in the AdapterView to fill the screen. This implies that there will always be a scroll movement, even if the element is already displayed on the screen, with the only exception of the first element, which requires no scroll by definition.

Another important consideration to be made involves the item whose bounds will be extracted during the construction phase of a ToggleInteraction scroll instance. In fact, in this case, the algorithm to extract these values will search among all child elements of the container (i.e. the AdapterView) for the one with the largest area, and, in case of multiple nodes having the same area, it will select the first one of them. This mechanism has been implemented to select the best possible reference point (that will be the source of the orthogonal axis introduced in section 4.4), meaning the one that is more likely to be matched without a failure by the 3^{rd} generation test libraries. Furthermore, in this way we are avoiding the possibility of having a first element that is partially displayed and that could cause problems in the methods that crop and resize the area of the locator of screen-capture.

Finally, an onData assertion could end both with a ViewAssertions and with a ViewActions. This could be a problem since a ViewActions might result, for example, in a change on the displayed activity or fragment causing the impossibility of creating the files capturing the appearance and the data on the searched element after the scroll movement has taken place. To solve this issue, in case an onData assertion ends with a ViewActions, the Espresso instruction will be split in two, one ending with a ViewAssertions whose purpose is simply to verify that the element is displayed and to let the enhanced test class generate the required data to match it during the translation, and another one executing the real interaction on the element of interest.

4.4.2 ScrollTo

ScrollTo is the second command to reproduce a scroll movement offered by Espresso. Different from what happens with onData, scrollTo is a ViewActions thus it is the conclusion of an Espresso assertion and not its entry point. This characteristic prevents the possibility of having a context switch in the AUT since the assertion itself ends with the scrolling interaction. This

implies that it will be always possible to create the files that must precede and follow the scroll movement without any modification to the Espresso instruction.

This ViewActions object applies to scrollable Views, typically instances of the ScrollView and HorizontalScrollView classes, and it can scroll to an existing layout element (differently from what happens with the AdapterView elements) until it is displayed, if it is not already part of the displayed screen, otherwise it will have no effect. This behavior, as already introduced in section 4.4, differs from what happens with an onData command since it won't perform any movement if the element is already displayed when launching the Espresso assertion, even if it is not the first one among all displayed children of its container. Additionally, if an actual scroll movement is required, it will be limited to the number of pixels that are necessary to make the element appear on the screen without any further movement.

For these reasons, two different approaches were taken to extract the information needed to emulate the scroll interaction in the construction phase of the ToggleInteraction. Particularly, a scrollTo ViewActions object has a radically different approach to the extraction of the bounds of the element whose center will be the source of the reference system used for the subsequent computations, as introduced in section 4.4. The log line will contain, as its *searchKeyword*, the class name of the scrollable view and its top-left coordinates. Thanks to these pieces of information it is possible to extract the bounds of the container itself from the XML dump file so that the center of the whole scrollable View will be the reference point. From this point, it is possible to later extract the coordinates of the start- and end-points of the scroll movement. This difference has been introduced mainly because, in this case, the elements inside the container have no constraint on their appearance, unlike the case of an AdapterView whose children ViewHolders are created by the Adapter instance and should respect a predefined pattern. For this reason, it is not possible to make any assumption on the appearance of any child element, so it is safer to use the whole container area. This approach eases the computation of the two points since their coordinates will be equally spaced and mirrored with respect to the source point.

Finally, during the execution of the enhanced test class, the measures of the start and the end coordinates along both axes, used to evaluate the number of pixels to be scrolled in each direction, don't assume the value of the offset from the top and left margins of the container but, instead, they are equal to the number of pixels between the searched element and the nearest margin of the container.

4.4.3 InstanceOf

ViewMatchers.instanceOf() is a method returning an object matching an element in the visual hierarchy of the AUT according to its class. This method has proven to be particularly useful when dealing with AdapterViews or when multiple elements share the same id and it is needed a further discrimination factor to match a particular one among them.

The support to this method has been added by adding a new value, named "class", to the ones that the field *searchType* of the log line could assume. In this case, the field *searchKeyword* of the log line will assume as value the name of the class that is the one passed as a parameter of the instanceOf() method. Finally, when extracting the bounds to isolate the locator to be used during the 3^{rd} generation test execution, the XML dump file will be parsed to find a node with the attribute "class" assuming as value a string containing the one reported in the *searchKeyword*.

4.4.4 AllOf & anyOf

Another update that has been introduced in the re-engineered version of TOGGLE involves the *allOf()* and *anyOf()* methods. These Espresso commands are used to express a logical relationship among various ViewMatchers objects. Their purpose is to express more than one condition to locate a View putting them either in a logical "AND" or in a logical "OR" condition among themselves.

Particularly, allOf is the equivalent of the logical "AND" condition and, consequently, it will search for a View matched by all the ViewMatchers objects passed as parameters. The support for this instruction has been achieved by implementing a concatenation mechanism of all the *searchType* fields, corresponding to every ViewMatchers object, in a unique *searchType* reporting all conditions, separated by an "&". Consequently, all the corresponding *searchKeywords* will be concatenated in the same way. The next step is the translation of these conditions in an equivalent expression able to match the searched node in the XML dump file. Once this operation has been completed it is possible to extract the bounds of the element and to crop its area from the screen-capture coupled to the XML dump, following the normal flow of events of a translation.

On the other hand, anyOf is a similar command corresponding to the logical "OR" operator, matching a View that satisfies at least one of the expressed conditions. The support for its translation has been implemented

following the same pattern that has been used for the allOf case, with the only differences in the separator for the concatenation of both the *searchTypes* and the *searchKeywords*, that in this case is the symbol "|", and in the operator interposed among all conditions to isolate a node of the XML dump.

By adding the support for these operators it was possible to expand the horizon of the translatable tests, including, in this way, all the cases using more complex conditions. Additionally, the test suites used to make the empirical evaluation of the new version of TOGGLE have been fully translated, as later explained in chapter 5.

Chapter 5

Empirical evaluation

This chapter will present an empirical evaluation performed on the updated version of TOGGLE. This experiment was executed on a set of five Android applications, selected after a mining period on the F-Droid[10] portal.

F-droid is a catalog of FOSS (Free and Open Source Software) whose adoption is widespread in the research community[11][16][21] since it grants access to the source code of the applications listed inside it. It classifies applications according to a category and offers both a WordPress front-end and a client acting as an equivalent of the Android PlayStore, offering a way to download, install and update applications directly on a physical mobile device. Additionally, the F-droid server maintains details of multiple versions of every application. This project is continuously growing and counts, in March 2021, over 3400 available applications. Furthermore, to use F-Droid it is not mandatory to have an account and the client does not record any user's private data.

My research aimed at selecting five applications, which will be further detailed in section 5.3, suitable for this experimental experience. Particularly, the results have been filtered searching for those Android native applications offering the possibility of testing the new features introduced in this paper. Certainly, other criteria based on the testability of the application, meaning that it should not be a toy application and it should be applicable in a real use case, on the guidelines followed, that should not be too outdated, and on their code, that should be publicly available, have been applied.

The results have been narrowed down to the following five applications:

1. Budget Watch;
2. PDF CONVERTER: Files to PDF;

3. Contact Book;
4. Stoic Reading;
5. Simple Calendar.

It is important to specify that every test case starts from the same state of the application and does not depend on the result of any previous one so that it is not necessary to make any assumption on the execution order. The Activity used as the entry-point was always the default Main Activity of the application.

Section 5.1 will introduce the three research questions, explaining the aspects they want to verify and the mathematical formulas used to compute a result. Sections 5.2 and 5.4 explain how the experiment has been carried out and in what kind of environment, and, finally, the results to the research questions will be detailed in 5.5. Finally, the last part of this chapter will focus on a high-level analysis of the experimental conclusions and on some aspects that have relevance in the final evaluation of the tool.

5.1 Research Questions

In parallel with the execution of every test suite, some data has been collected to answer a set of research questions whose purpose was to evaluate both the quality of the translator and the effectiveness of the 3^{rd} generation Visual GUI Tests.

Particularly, the goal was to find an answer to the following research questions:

- **RQ1 - Translator Performance**: what is the ratio between the number of successfully translated test cases and the overall number of tests?

To answer RQ1, an analysis must be carried out on the enhanced test classes, other than on the log files and the third-generation test execution, to detect whether a 2^{nd} generation test assertion has been correctly translated or not. The ratio has been computed with the following mathematical expression:

$$ratio = \frac{\#correctlyTranslatedTests}{\#tests} \tag{5.1}$$

where *tests* is the number of tests and it is equal to 30 if we are computing this value for a single application or equal to 150 if we are considering the whole set of tests used for this experimental evaluation. Consequently, *#correctlyTranslatedTests* is the number of successful test translations and will assume a value between 0 and *tests*.

- **RQ2 -** 3^{rd} **generation test effectiveness**: What is the percentage of interactions correctly executed by the translated tests?

To answer RQ2 a key factor was to collect the total number of interactions composing every single test case, other than the number of both those that have been successfully executed and those who have failed. The mathematical expression to evaluate the percentage is:

$$perc = \frac{\#successfulInteractions}{\#interactions} * 100 \tag{5.2}$$

Where *#interactions* is the overall number of interactions composing the 3^{rd} generation test cases and *#successfulInteractions* is the number of interactions that have been successfully executed. This percentage has been computed at two granularity levels: first, this percentage has been evaluated for each interaction type, and then, it has been evaluated considering all interactions grouped together.

- **RQ3 -** 3^{rd} **generation test effectiveness**: What is the percentage of successful translated test cases?

The data required to compute the result for RQ3 is easier to be retrieved than the one needed for the previous research questions. Particularly the percentage has been computed through this formula:

$$perc = \frac{\#successfulTests}{\#tests} * 100 \tag{5.3}$$

Where *#tests* is the overall number of translated test executions while *#successfulTests* is the number of executions ended with success and its value will range between 0 and *#tests*.

5.2 Methodology

For each experimental subject, a test suite, composed of a total of 30 Espresso tests, has been developed, resulting in a total of 150 test cases. Furthermore, each test suite has been translated and then executed 10 times. The outcome is an overall number of 3^{rd} generation test executions equals 1500.

Table 5.1 shows the low-level distribution of Espresso assertions for each application while figure 5.1 shows how these instructions are globally rationed.

Table 5.1. Distribution of Espresso assertions per application.

Interaction	Budget Watch	Contact Book	PDF CONVERTER	Simple Calendar	Stoic Reading
allOf	1	1	34	8	0
anyOf	2	0	0	0	0
atPosition	21	9	0	0	0
click	37	30	7	30	20
inAdapterView	21	9	0	0	0
instanceOf	0	0	34	0	0
isDisplayed	17	24	11	12	9
matches	23	24	11	12	9
onData	21	9	0	0	0
onView	44	46	37	63	52
scrollTo	3	0	19	12	23
typeText	2	1	0	9	0
withContentDescription	2	0	4	5	0
withId	55	53	33	62	50
withText	9	3	0	4	2
All	258	209	190	217	165

Over these 150 tests, on average, 45% of them execute at least one scroll interaction. Specifically, the application with the minimum number of tests performing this kind of interaction is Contact Book with a percentage equal to 27% of the total ones contained in its suite, while the one with the highest percentage value is Budget Watch with 80% of its tests performing a scroll interaction.

Finally, the generated Java projects have been updated to automatically collect the data required for responding to *RQ1*, *RQ2*, and *RQ3*. Particularly, the customization introduced aimed at creating two separate CSV files for every experimental subject. The first one is composed of one row for each executed test case, meaning that the number of rows for this experiment, always equals 300. Each row reports the following data:

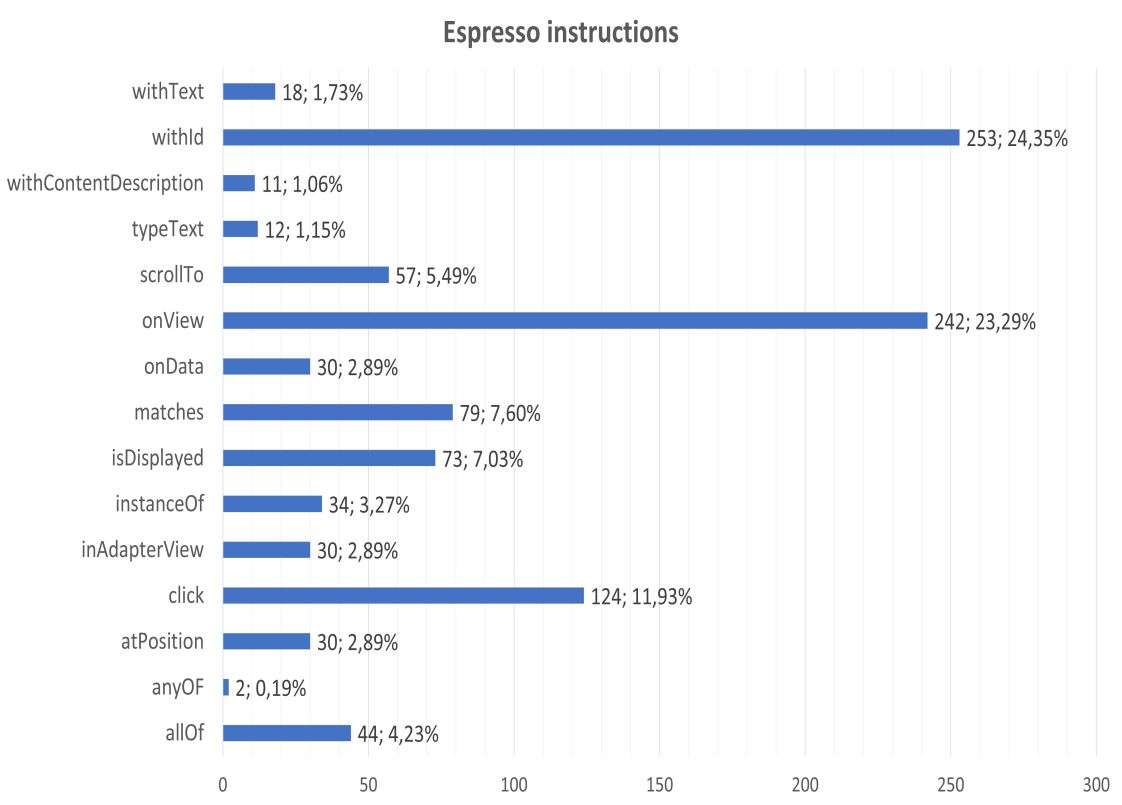


Figure 5.1. Espresso instructions distribution among all tests of the experimental test suites.

```
<TestClassName>; <TestCaseName>;
  <TotalInteractions>; <SuccessfulInteractions>
```

Where *TestClassName* is the name of the 2^{nd} generation test class, *TestCaseName* is the name of the test case, *TotalInteractions* is the total number of interactions in the specific test case, and *SuccessfulInteractions* is the number of successful interactions emulated during specific test execution.

The second CSV file has a finer granularity. Specifically, it is composed of one line for each interaction and every line reports the following data:

```
<TestCaseName>; <InteractionId>; <Result>;
  <InteractionType>
```

Where *TestCaseName* is the name of the test case of interest, *InteractionId* is a sequential number that is unique in a specific test case execution, *Result*

can either be "p" or "f" according to, respectively, if the interaction is successful (pass) or if it failed (fail), and *InteractionType* is a string identifying the interaction type and can assume values like "Scroll", "Click", "FullCheck", "Check", and more.

These two files have been further processed and adjusted to collect more statistical data on the five test suites and to compute the results presented in section 5.5.

5.3 Experimental Subjects

The experimental methodology presented in section 5.2 has been applied to all five test subjects. Particularly, the applications selected are:

1. **Budget Watch**: an application to create and manage personal budgets and to keep track of the user's expenses, by recording his/her transactions and assigning them to one of the defined budgets. It is available both on F-Droid and on Google Play;
2. **PDF CONVERTER: Files to PDF**: an application, with a particular focus on usability, whose purpose is to offer several options to manipulate, update and create new PDF files from different sources. It is available both on F-Droid and on Google Play;
3. **Contact Book**: an application aiming at being an alternative to Google contacts, with a focus on the user's privacy. It prevents data sharing with third parties by using a personal local database. Among its features, it enables the possibility of sharing and adding new contacts by using a QR code. It is available on F-Droid;
4. **Stoic Reading**: an application to consult, without any need for an internet connection, several stoic writings, ranging from Emperor Marcus Aurelius to Seneca. It is available on F-Droid;
5. **Simple Calendar**: a calendar application to handle customizable events and their reminders over time. It offers a monthly view and the possibility to create events with colored labels, different duration and repetitiveness, and many more customizable features. It is available both on F-Droid and on Google Play.

Table 5.2 presents some more detailed data on all the introduced applications.

Table 5.2. Information on the experimental subjects.

Application	PlayStore downloads	GitHub stars	GitHub commits	Lines of code	Last update
Budget Watch	29	70	477	9860	27/01/2021
Contact Book	-	7	26	2980	02/08/2019
PDF CONVERTER	952	607	639	19110	23/03/2021
Simple Calendar	6805	2397	4350	28762	22/03/2021
Stoic Reading	-	14	235	31836	05/02/2021

5.4 Experimental Environment

The hardware infrastructure used to execute all test cases is an Acer Aspire A715-71G with an Intel(R) Core(TM) i7-7700HQ CPU at the frequency of 2.80GHz, 16GB of RAM, and running Windows 10 Home as its operating system. Furthermore, all Espresso test cases have been written using Android Studio 4.1.1 as IDE and executed on a Nexus 5 API 28 (Android 9) with the animations disabled.

All 3^{rd} generation test executions have been performed on a black background, to avoid any disturbances to the image recognition algorithms, and using the SikuliX-then-EyeAutomate approach, which has proven to be one of the most robust 3^{rd} generation techniques.

5.5 Experimental Results

This section will introduce the results, obtained by studying the data collected after the experimental execution of the test cases, and all the interventions that have been introduced to obtain the presented achievements.

5.5.1 RQ1

The result of RQ1 can be summarized by saying that all Espresso test assertions have been successfully translated, obtaining a *ratio* value equal to 1.00. It is important to highlight the fact that, in this case, an intervention was necessary to implement the support for some key Espresso methods that are commonly used. Particularly, as previously mentioned in section 4.4, the range of supported Espresso assertions has been expanded by including the possibility of isolating the node of interest using its class name, through the

support for the ViewMatchers.instanceOf() matcher, and by including the support for the scroll interactions, both when originated by the scrollTo() and by the onData() methods. Furthermore, the onData() instruction was strictly connected to two other Espresso commands, namely inAdapterView() and atPosition(), whose support has been implemented as well. Finally, as mentioned in section 4.4.4, the support for multiple logically related conditions, used to match a single View, has been added by creating a translation mechanism for both the allOf() and the anyOf() instructions.

All these new features made it possible to successfully write and translate a collection of test suites that are complete and exhaustive.

By studying the data reported in table 5.3, it is possible to observe how the translation accuracy and the ratio used to evaluate the performances of the translator would be much lower without the support for the new features.

Table 5.3. Comparison between the state-of-the-art version of the library and re-engineered one.

Application	Translatable tests - state-of-the-art	Translatable tests - re-engineered	Translation ratio - state-of-the-art	Translation ratio - re-engineered
Budget Watch	5	30	0.17	1.00
PDF CONVERTER	0	30	0.00	1.00
Contact Book	22	30	0.73	1.00
Stoic Reading	10	30	0.33	1.00
Simple Calendar	11	30	0.37	1.00
All	48	150	0.32	1.00

5.5.2 RQ2

To evaluate the percentage of interactions correctly translated, and whose execution was successful, the data on the 3^{rd} generation interaction types and the result of the EyeAutomate and SikuliX relative assertions have been collected.

Particularly, two granularity levels have been adopted to compute an exhaustive response to this research question. The main goal was to make a first analysis on the effectiveness of the 3^{rd} generation tests, by focusing on the quality of every specific translated interaction and their success rate.

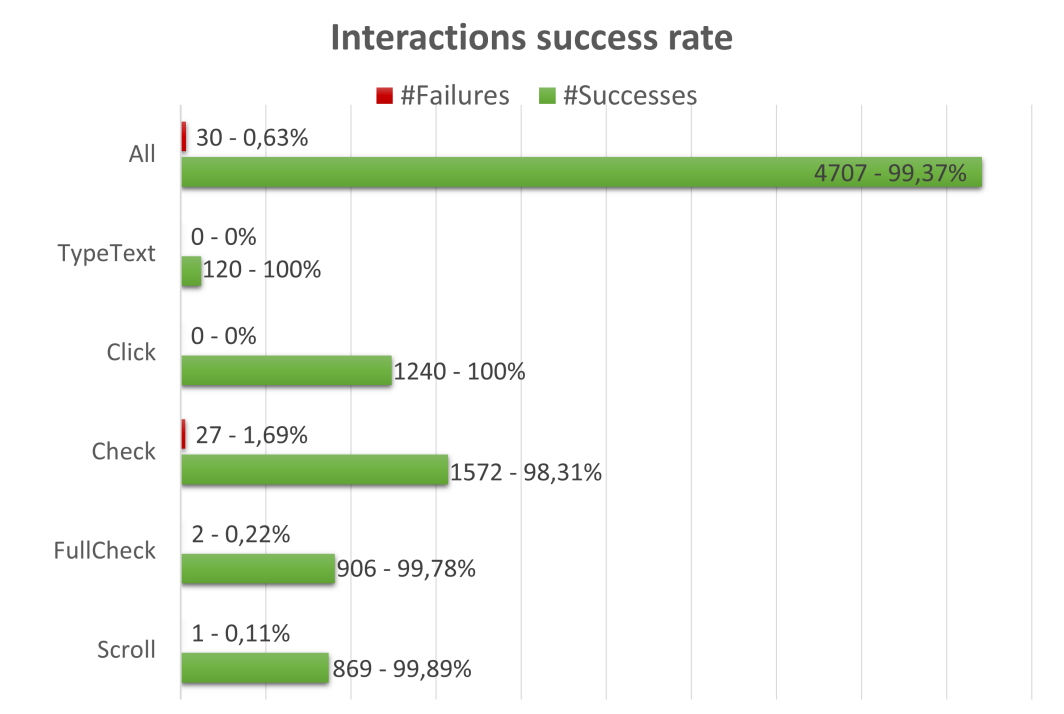
The finer granularity level takes into consideration the number of actually executed interactions, meaning that the *#interactions* variable won't

necessarily be equal to the absolute total number of translated interactions, because a failure in an assertion will stop the execution of the specific test case preventing the execution of all following assertions.

On the other hand, while computing the percentage of successful interactions at the coarser granularity level, the *#interactions* variable has been considered equal to the total number of translated interactions. It is important to highlight the fact that, in this case, the interactions that have been never reproduced will be considered as failing.

In both cases, the values refer to the complete experimental execution for all five applications.

Figure 5.2 reports the results obtained from the analysis by interaction type. It is worth noticing that the highest failure rate is relative to the "Check" interactions, which are generated not only after an actual Espresso check assertion but after every scroll interaction too.



	Scroll	FullCheck	Check	Click	TypeText	All
#Failures	1	2	27	0	0	30
#Successes	869	906	1572	1240	120	4707

Figure 5.2. Success rate of the specific 3^{rd} generation interaction types.

Figure 5.3 reports the result relative to the global evaluation performed on all interactions. In this case, the success rate is equal to:

$$perc = \frac{4707}{4750} * 100 = 99.09\% \tag{5.4}$$

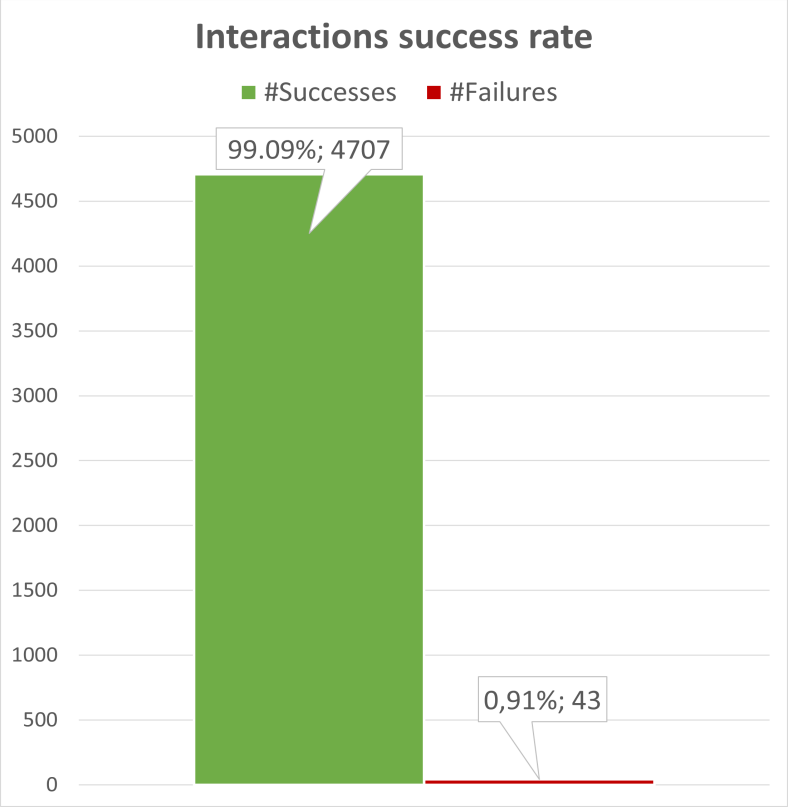


Figure 5.3. Success rate of all 3^{rd} generation interactions.

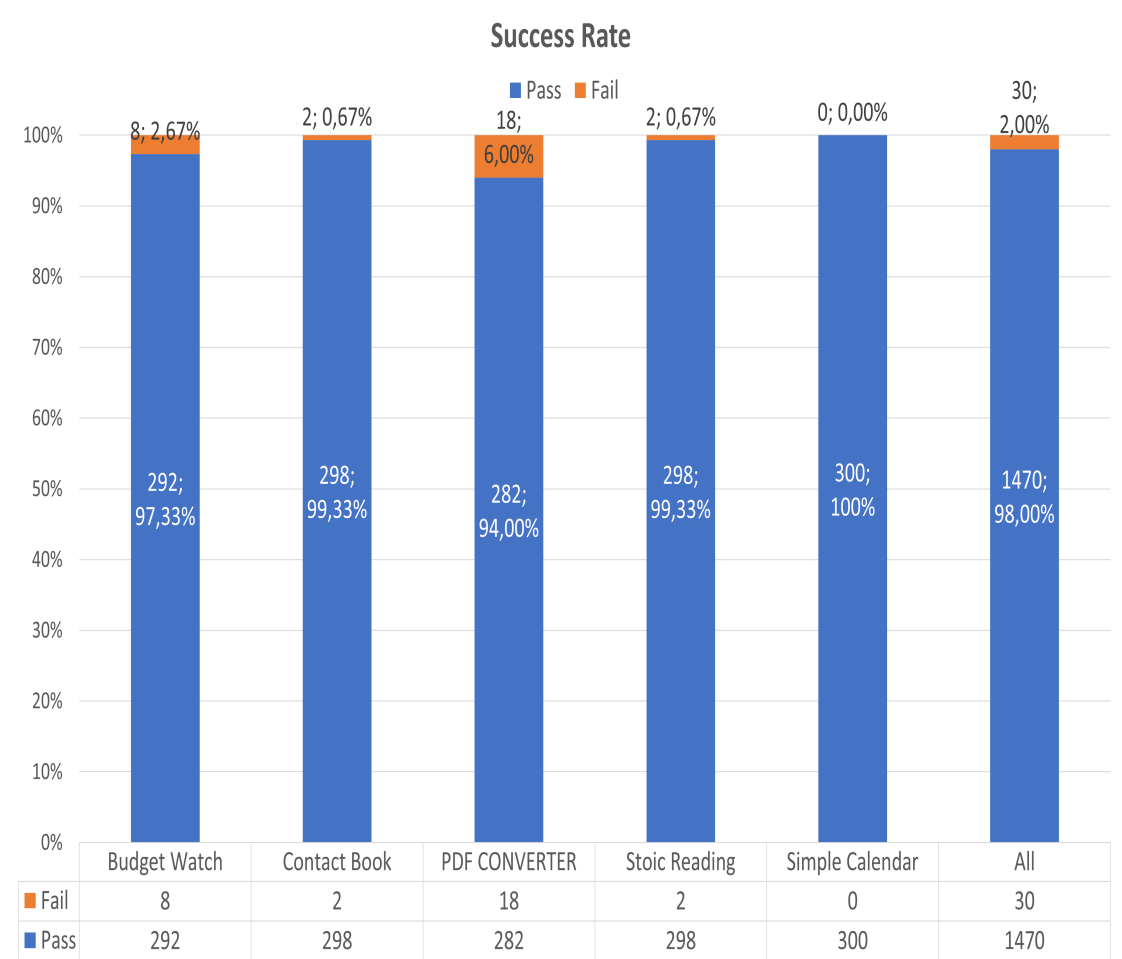
5.5.3 RQ3

The goal of this research question was, again, to evaluate the effectiveness of the translated test suites. In this case, the analysis has been performed from a higher-level point of view than to research question 2, by recording the results of each test execution and computing both the global success rate and the one specific for the single application.

As for what happened with RQ2, the data that is being presented refers to the complete execution of all the experimental test suites, meaning after that each 10^{th} execution of every translated test has been completed.

Consequently, the variable *#tests*, used during the computation of the success percentage, was equal to 300, during the analysis of the performance of a suite of tests for a single app, and equal to 1500 during the evaluation of the global success rate.

Table 5.4 and figure 5.4 present both a tabular and a graphic representation of the results collected to answer this question.



	Budget Watch	Contact Book	PDF CONVERTER	Stoic Reading	Simple Calendar	All
Fail	8	2	18	2	0	30
Pass	292	298	282	298	300	1470

Figure 5.4. Success rate for all 3^{rd} generation tests of all applications considered during the experimental evaluation.

Table 5.4. Translated tests success rate.

Application	Successful tests	Failing tests	Total	Success percentage
Budget Watch	292	8	300	97.33 %
PDF CONVERTER	282	18	300	94 %
Contact Book	298	8	300	99.33 %
Stoic Reading	298	8	300	99.33 %
Simple Calendar	300	0	300	100 %
All	1470	30	1500	98.00 %

5.6 Results overview

After this experimental evaluation of the performances of TOGGLE, some high-level results have been collected to obtain feedback on the quality of the improvements to the library.

For starting, an important preface to be made is that, after the improvements to the 3^{rd} generation script creator module, the complexity and the effort required to complete the whole experimental execution has drastically decreased in comparison with what happened in other previous empirical applications of TOGGLE. In fact, the automatic injection of the libraries offering the Java APIs of both EyeAutomate and SikuliX, coupled with both a unique Java main method launching all tests and an additional module automatically restarting the application before each test execution, made the experimental execution a lot easier. The only manual intervention required was executing the Java main.

Furthermore, RQ1 has proven that it is possible to write exhaustive test suites for a wide variety of applications with the Espresso methods supported in the re-engineered version of the library, even though, in this area, there is still some space for improvements. The two key features that made this result possible are: the support for the scrolling interactions, which are typical of the mobile world and, for this reason, highly exploited in various contexts, and the support for complex logical conditions thanks to the translation mechanism for the allOf() and anyOf() methods.

RQ2 and RQ3 proved that it is possible to have robust 3^{rd} generation test suites, especially when they are executed more than once to balance their flakiness, whose results are coherent with the expected ones, other than with the ones obtained after the execution of the source 2^{nd} generation suite. An important consideration regarding the fail condition of a scroll interaction

should be made. As introduced in section 5.5.2, interactions of this kind will result, actually, into two 3^{rd} generation ones, the first scrolling in a direction and the second checking that the searched element is displayed after the scroll movement.

A failure in the check phase implies that either too many pixels were scrolled or too few. Additionally, the empirical evaluation proved that it is more common, due to the sensitivity of the emulator, that often causes a limited and unpredictable loss in the actual scrolled pixels during a scroll movement, to have scrolled fewer pixels than expected. In this context, it is important to say that slowing down the scroll movement was an effective countermeasure that has drastically reduced this non-deterministic behavior.

On the other hand, a failure in the scroll phase is rarer, as figure 5.2 reports, and it is caused by the impossibility to locate on the screen the expected starting status of the application, meaning that no scroll movement has taken place before the failure because it was not possible to match the initial locator on the screen. This error presented itself only once and its source was, yet, the test flakiness. Particularly, this failure was caused by a longer time than usual taken by the application to start up, so that the test failed before the GUI of the application was completely displayed on the screen of the emulator.

Chapter 6

Conclusion and Future Work

This thesis presented, starting from the existing infrastructure of TOGGLE, a deep re-engineering of the library to improve both its usability and portability other than the introduction of some new features expanding the horizon of translatable interactions.

To go into further details, the purpose of the re-engineering phase was to transform TOGGLE into a library whose adoption in an industrial software development process could be feasible. To reach this objective a first analysis of the tool and all its weaknesses has been performed, with a particular focus on all those issues causing a high maintenance effort, low portability of the library, a high entry-barrier for an end-user, the non-scalability of the translation process in big projects, and any phase needing the manual intervention of an end-user.

It is worth noticing that, some of the issues presented are common to the ones preventing the diffusion of Visual GUI Tests and, consequently, they were particularly urgent to be solved, especially because TOGGLE is a library whose global goal is to make this kind of testing approach more common in mobile applications development processes. This phase could be further divided into two high-level steps: the re-engineering of each module, whose goal was to solve the issues of every specific component of the library, and the re-design of the library, aiming at improving the usability of TOGGLE, starting from the output of the previous step and providing, as its output, a version of the tool that could be easily enriched with new features after low effort interventions involving both their development and their integration with the library.

After the completion of this phase, the focus was moved to the implementation of some new features. In this case, the main goal was to provide an effective translation mechanism for the scroll interaction, that is as complex to emulate on a personal computer as it is common in a mobile environment. This phase took advantage of what has been done to improve the architecture of the library, making it easier to correctly place the new features in the re-engineered ecosystem of TOGGLE. Moreover, the scroll interaction required a preliminary analysis of its semantics and of the Espresso instructions that could generate it because, even though it could be started by multiple Espresso commands, it should have an unambiguous 3^{rd} generation translation. This factor was the one that required the greatest effort to be completed.

Lastly, an experimental appraisal of TOGGLE was performed to evaluate the results of the previous phases and to find out the aspects of the tool that could need some further improvements. Particularly, this phase highlighted that the scroll interaction suffered from a non-deterministic behavior, caused by the speed of the movement and by the sensitivity of the screen of the emulator. This required the development of some corrective interventions on the behavior of the translated interaction to lower the impact of this issue. As a consequence, the experiment has been repeated multiple times (after each modification to the scroll instruction) before obtaining the results presented in chapter 5. The last execution of the experiment proved that a robust and low effort translation mechanism is possible, presenting satisfactory results, both in terms of 3^{rd} generation tests success rate and of specific 3^{rd} generation interactions success rate, which are both similar to the ones collected with the initial version of the library [12]. The actual difference with the state-of-the-art variant of the tool is that these results have been obtained with a much less complex, low-effort, and automated use of the tool.

Furthermore, this empirical experience highlighted some aspects that offer a fertile ground for improvements. An example may be the extension of the Enhancer module to support the Kotlin programming language, which is becoming increasingly important in the Android mobile application context so that it could be possible to parse Kotlin Espresso test classes and to generate the equivalent enhanced version. Another improvement might involve the development of a new Enhancer module supporting different 2^{nd} generation testing frameworks, even though Espresso is probably the one with the highest diffusion among Android mobile projects. Moreover, the set of translatable interactions and Espresso commands supported by TOGGLE could be expanded. Finally, the adoption of this library in an existing industrial

project is encouraged so that it could be possible to obtain an evaluation of its performances in that context.

Bibliography

[1] Adapterview. https://developer.android.com/reference/android/widget/AdapterView. Accessed: 2020-12-10.

[2] Espresso. https://developer.android.com/training/testing/espresso/. Accessed: 2020-12-5.

[3] Eyeautomate. https://eyeautomate.com/eyeautomate/. Accessed: 2020-11-25.

[4] Eyeautomate. https://eyeautomate.com/wp-content/themes/EyeAutomateTheme/resources/EyeAutomateManual.html#. Accessed: 2020-11-25.

[5] Fundamentals of testing. https://developer.android.com/training/testing/fundamentals. Accessed: 2020-10-15.

[6] Gradle user manual. https://docs.gradle.org/current/userguide/gradle_wrapper.html. Accessed: 2020-12-20.

[7] Gradle vs maven comparison. https://gradle.org/maven-vs-gradle/. Accessed: 2020-12-20.

[8] The gradle wrapper. https://docs.gradle.org/current/userguide/userguide.html. Accessed: 2020-12-20.

[9] Logcat command-line tool. https://developer.android.com/studio/command-line/logcat. Accessed: 2020-10-19.

[10] What is f-droid? https://f-droid.org/. Accessed: 2021-03-01.

[11] Riccardo Coppola, Luca Ardito, and Marco Torchiano. Characterizing the transition to kotlin of android apps: A study on f-droid, play store, and github. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics*, WAMA 2019, page 8–14, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Riccardo Coppola, Luca Ardito, Marco Torchiano, and Emil Alégroth. Translation from layout-based to visual android test scripts: An empirical evaluation. *Journal of Systems and Software*, 171:110845, 2021.

[13] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. 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*, PROMISE, page 22–32, New York, NY, USA, 2017. Association for Computing Machinery.

[14] Robert Feldt Emil Alégroth and Lisa Ryrholm. Visual gui testing in practice: challenges, problemsand limitations. *Empir Software Eng*, 20:694–744, 2015.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[16] Giovanni Grano, Andrea Di Sorbo, Francesco Mercaldo, Corrado A. Visaggio, Gerardo Canfora, and Sebastiano Panichella. Android apps and user feedback: A dataset for software evolution and quality improvement. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics*, WAMA 2017, page 8–11, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Raimund Hocke. Automate what you see on a computer monitor. http://sikulix.com/. Accessed: 2020-11-28.

[18] Raimund Hocke. Sikulix - general aspects of scripting. https://sikulix-2014.readthedocs.io/en/latest/scenarios.html. Accessed: 2020-11-28.

[19] Muhammad Kamran, Junaid Rashid, and Muhammad Nisar. Android fragmentation classification, causes, problems and solutions. *International Journal of Computer Network and Information Security*, 14:992–999, 09 2016.

[20] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 399–410, 2017.

[21] Iván Arcuschin Moreno. Search-based test generation for android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, page 230–233, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Hiral Poshiya. Gui testing tutorial – understanding the basics. https://reqtest.com/testing-blog/gui-testing-tutorial/. Accessed: 2020-10-28.