Implementation and evaluation of a tool for translating Visual to Layout-based android tests

Simona Saitta

Master Degree in
Computer Engineering

Politecnico di Torino
Academic Year: A.Y. 2019-2020

Supervisors:
Morisio Maurizio
Ardito Luca
Torchiano Marco

Contents

List of Figures

List of Tables

Abstract

Context: Modern mobile GUI test tools are classified as first generation (coordinate based), second generation (layout based) or third generation (visual). All of them have different benefits and drawbacks, which could be leveraged through automated translation from one generation to another one. Changing device or some graphical aspects of the application could result in failure of visual scripts but not layout based ones. Visual scripts could be easily regenerated having available layout based ones.
Goal: The aim of this work is to implement a tool that can automatically translate third generation test scripts in the EyeAutomate syntax to second generation Espresso test cases.
Method: A software project in Java was developed to achieve the proposed goal and then an experiment was performed, to evaluate the success rate of the translated scripts. The experiment consisted in translating EyeAutomate visual test scripts, developed for two open source mobile applications (PassAndroid and OmniNotes), into Espresso layout based scripts.
Results: The translation proved feasible with the proposed tool architecture, and the evaluation yielded a quite high success rate on the considered software objects (58 out of 60 test cases successfully translated).
Conclusion: The study demonstrated that the translation from third generation (visual) to second generation (layout-based) test scripts is feasible in the mobile domain, and that translation is able to reduce the required effort for generating Layout-based test cases. More work is however required to extend the proposed tool and evaluate it in industrial practice, as well as to measure its capability of reducing fragility and maintenance effort for Layout-based test suites.

Chapter 1

Introduction

This chapter introduces the work done and the project in its contest and providing its motivations.

1.1 Software testing

Software testing is a process that allows to check that the actual result resembles the expected one, matching the requirements, and allows to detect possible errors. It is an important step of software development because software bugs could be quite dangerous.
When working on a large project it is easy to miss out key details in the requirements phase itself delivering a possible completely wrong product. It is also possible to make mistakes in later steps even before reaching the coding one. To monitor this and other processes the SDLC was established.

1.1.1 Software Development Life Cycle

The SDLC (Software Development Life Cycle) defines the steps involved in the development of software at each phase and covers the plan for building, deploying and maintaining it. There are usually six stages in this cycle: analysis, design, development and testing, implementation, documentation, and evaluation. One of the most used models is the V-model.

V-model

In software development the V-model, also known as Verification and Validation Model, is a Software Development Life Cycle (SDLC) model where instead of moving down in a linear way (as in the waterfall model) the process steps are bent upward having the development of each step directly associated with its testing phase.

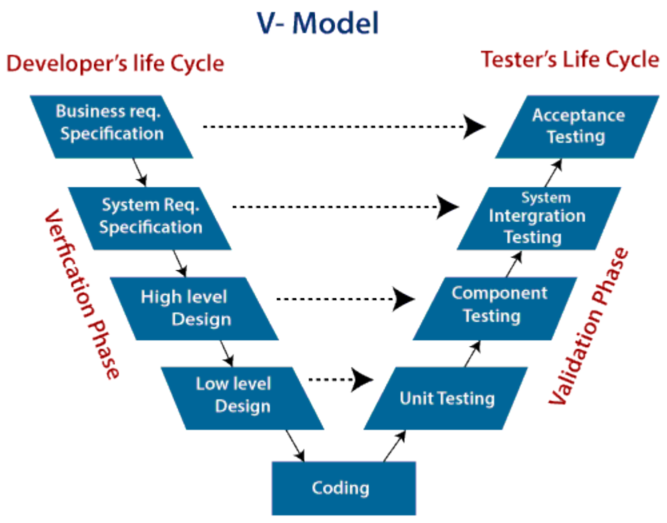


Figure 1.1: V-Model

The Verification phase starts with collecting the system requirements by analyzing the need of the users. In this steps a requirements document is generated. Later on the system is designed by studying the generated document.
Once the design is completed its the turn of the Coding phase that will take place using the coding guidelines and standards.
Last phase, going up the V, is the Verification one. In this step the tests designed with their verification step counter part are executed following the reverse order. In this phase we can identify four main level of testing:

1. Unit testing
2. Component testing
3. System integration testing
4. Acceptance testing

Among all of these main focus for this thesis must be put on the System integration testing one.

1.1.2 System testing and End-to-End testing

The **System testing** is a level of software testing which involves testing the fully and integrated software to make sure it matches the requirements. It is a type of Black Box Testing which involves the external workings of the software from the user's perspective. An alternative quite similar to it is the **End-to-End testing** which is a methodology used to test whether the flow of an application is performing as designed from start to finish.

Graphical User Interface testing

Graphical User Interface (GUI) testing is part of the system testing and is the process of checking every possible interaction the user can do with the Graphical User Interface (GUI). Even though it is a fundamental level of testing, especially in applications, it still often performed manually at high costs [18]. We can identify three generations of testing tools:

- **First generations testing tools** (also called **Coordinate-based testing tools**) identify elements in the layout through their coordinates. This technique is not much used due to its fragility. [17]

- **Second generations testing tools** (also called **Layout-based** or **Property-based testing tools**) are able to access the GUI components. These allows them to identify a component based on its label or ID or a property as the text contained. This type of test is more robust of the previous one but does not check the actual aspect of the GUI and has an high maintenance cost [16].

- **Third generation testing tools** (also called **Visual GUI testing tools**) identify elements through image recognition algorithms, which capture the actual appearance of the elements as displayed to the user.

1.2 Android

As can be seen in the graph of figure 1.2 the operative system Android [4] has nowadays a large control in the mobile market. It has therefore become quite relevant being able to develop a working application to release. Before diving on android testing we need to have a general understanding of the various types of applications.

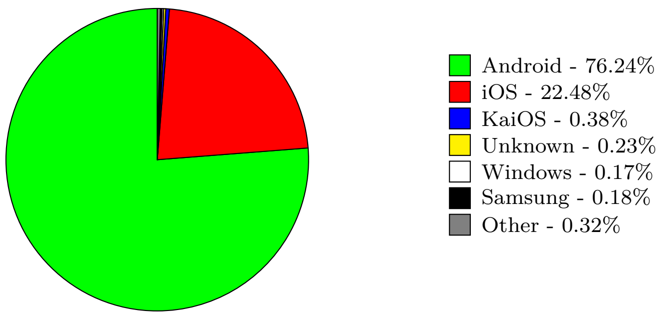


Figure 1.2: Mobile Operating System Market Share Worldwide - September 2019 [27]

1.2.1 Android Applications types

Three main types of android applications can be identified:

- Native applications
- Web applications
- Hybrid applications

The applications this thesis refers to are the native ones.

Native applications

A native application is the most common type of application. It is developed for a specific operating system allowing it to use all the OS features and the users to have a good experience. They also need to be downloaded from the application store, taking up device storage.

Web application

A web application use a browser to be run and the user experience depends on the connectivity. On the other hand they take up a really small quantity of android storage (e.g. cookies) and they do not need to be downloaded. They can be used in any device but they are unable to make use of a specific device API except for some small functionalities as localization.

Hybrid application

An hybrid application is built using multi-platform web technologies so they are generally web application disguised inside a native wrapper. For these reasons they have both advantages and disadvantage of the two types mentioned previously.

1.2.2 Android testing

As mentioned previously an important step in software development is testing. To test an android application there are different tools that can be used, based also on the generation technique.
For the purpose of this thesis we will examine two of these tools: Espresso [14] and EyeAutomate [7].

Espresso

Espresso is a framework by Google that allows to write layout based tests. It is able to identify when the application is in a waiting status increasing the reliability and execution speed. The API provides numerous methods but to better match the developer requirements it is also possible to create your own subclasses and methods.
The interactions usually use a view as an entry point (*onView()* and *onData()* methods). The view is located in the view hierarchy through *ViewMatchers* objects. It is expected that the matched view is one and only one in order to perform correctly the interaction. An exception would be raised otherwise. Examples of methods that return a matcher can be:

- **withId(int id)**: this method returns a matcher for all the views with that specific id.
- **withText(String text)**: this method returns a mathcer for all the views containing that specific text.
- **allOf(Matcher<? super T>... matchers)**: this method returns a matcher that satisfy all the other ones.

Commands and assertions can then be sent to the view through *ViewActions* or *ViewAssertions*. This is the reason the onView() method can refer to a single element.
ViewActions are objects that can be passed to the *ViewInteraction.perform()* method to execute simple interaction as for example clicking on a view. Some examples of methods can be:

- click(), longClick(), doubleClick()
- swipeDown(), swipeUp(), swipeLeft(), swipeRight()
- closeSoftKeyboard()
- pressBack()
- typeText(String stringToBeTyped)

GeneralClickAction and **GeneralSwipeActions** can be used to specify for example the specific coordinates in the view.
ViewAssertions are objects that can be passed to the ViewInteraction.check() to verify the status of the view. It does not have lot of methods:

- **doesNotExist()**: Returns an assert that ensures the view matcher does not find any matching view in the hierarchy.
- **matches(Matcher<? super View> viewMatcher)** Returns a generic ViewAssertion that asserts that a view exists in the view hierarchy and is matched by the given view matcher.
- **selectedDescendantsMatch(Matcher<View> selector, Matcher<View> matcher)** Returns a generic ViewAssertion that asserts that the descendant views selected by the selector match the specified matcher.

EyeAutomate

EyeAutomate is a tool by AUQTUS that allows to write Visual tests for any app on any platform using intelligent image recognition. Most of the commands are followed by an image of the element to operate into and coordinates may be added too. The scripts can be written with a simple text editor or with the provided one (EyeStudio [8]) that simplify the procedure. The used algorithm is called âĂIJThe EyeâĂİ. it allows to identify the images in any screen using all the available CPU improving the execution speed and efficiency of the test.

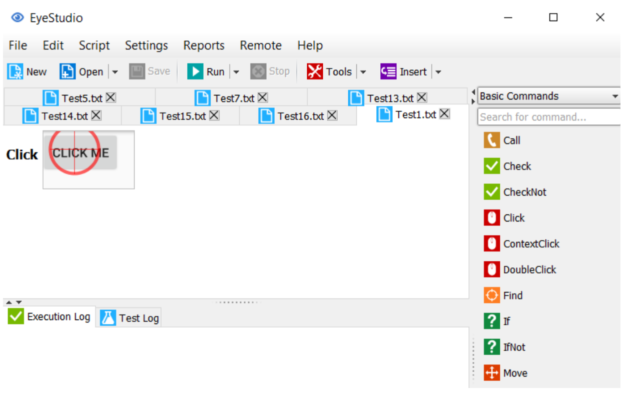

EyeStudio
File Edit Script Settings Reports Remote Help
New Open Save Run Stop Tools Insert
Test5.txt Test7.txt Test13.txt Test14.txt Test15.txt Test16.txt Test1.txt
Click CLICK ME
Basic Commands
Search for command...
Call
Check
CheckNot
Click
ContextClick
DoubleClick
Find
If
IfNot
Move
Execution Log
Test Log

Figure 1.3: EyeStudio

For the thesis purpose the tests have been executed through the jar file provided during the installation:

java -jar EyeAutomate.jar script.txt

The generated execution log file (execution_log.txt) was also relevant for the development of the tool since it reports all the script interactions results. Examples of some commands can be:

- **Check**: checks that the provided image is currently visible
- **Click**: clicks at the center of the provided image or at the specified position.
- **Move**: moves the mouse at the specified position provided through an image
- **MoveRelative**: moves the mouse of the specified amount
- **Type**: types the provided string
- **DragStart, DragDrop**: used for mouse drag instructions
- **MouseLeftPress, MouseLeftRelease**: generic mouse interaction that can be used either for click, long click, swipe, etc.

1.3 Motivation

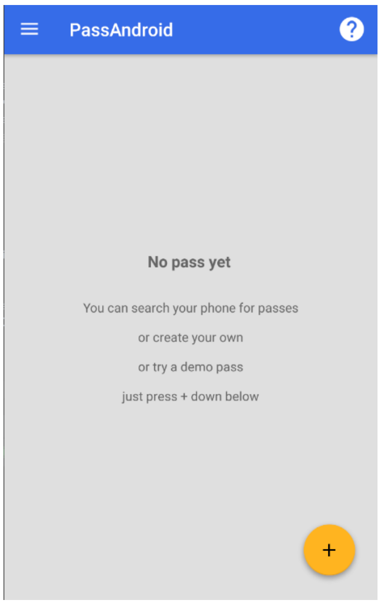
Even if there are evidence of applicability, feasibility and usefulness [3] of Visual GUI testing tools they are less commonly used than the Layout-based ones [5] for their lack of robustness and performance compared to the later [1]. However, Layout-based testing tools cannot fully emulate human user as interactions through GUI properties do not verify the system's appearance as shown to the human user. Most researches aimed to compare the two techniques have concluded that a hybrid approach is required where both generations are developed and used in parallel by the practitioner [2].
A first advantage of an hybrid approach is the possibility of reusing already existing Visual tests suites. This would reduce the development effort and increase the productivity for layout-based ones [28] [21]. On the other hand Device Fragmentation is a big issue of Visual tests [26]. By reusing the layout-based tests it would be possible to automatically generate the Visual ones for different devices without having to manually recapture all the required screens [24]. Reusing already provided test suites is also useful for new developer. Studies have demonstrate that many Android apps are poorly tested [9] or totally not tested [2] and developer prefer manually testing their application [22] [29]. This approach is not affordable on an industrial level. Studies have also identified repetition patterns in Android platform framework test case libraries that can be represented in generic form for further reuse using and design of generic test case libraries. [6]
Empirical measurements have proved that an hybrid approach would allow the tester/developer to focus on the development of a single test suite in one methodology and automatically generate the counterpart.
Other perk would be the mitigation of the fragility. Layout-based tests are sensible to code changes while Visual ones to visual changes. We can in fact identify five main causes of fragility [10]:

- Identifier change
- Text change
- Deletion or relocation of an element
- Change of Physical buttons into Action Bars
- Graphics changes

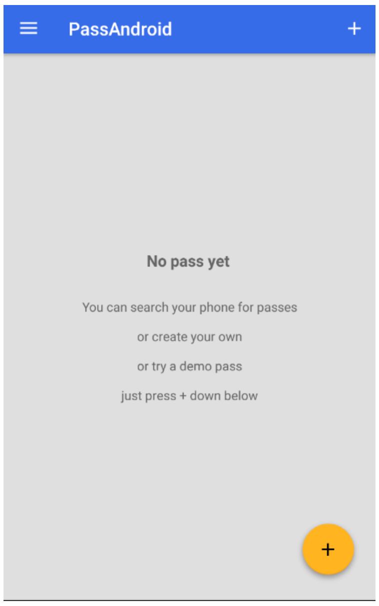
If tests of a generation do not work anymore it is possible to attempt translating the other one in order to greatly reduce the maintenance cost on which the fragility has a relevant impact [11].

1.3.1 Example

Let us assume we are writing a Visual test that uses the button in the top right corner of the screen (for example a simple check).



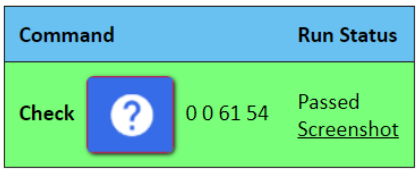
(a) Before editing



(b) After editing

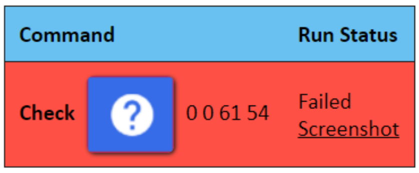
Figure 1.4: Application screen

As we can see in the picture below a Visual test will fail if changes are done to the button appearance. This will make it not reusable for future testing and will have to be recorded again.



Command			Run Status
Check		0 0 61 54	Passed Screenshot

(a) Before editing

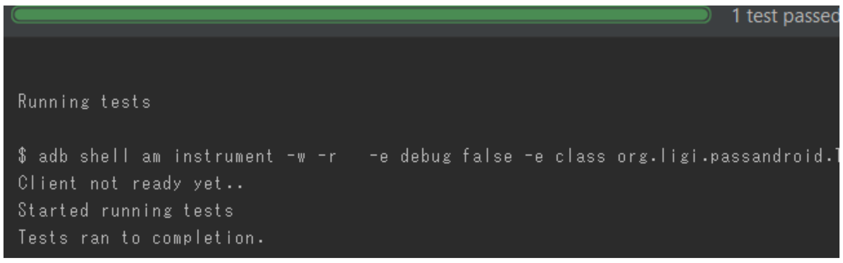


Command			Run Status
Check		0 0 61 54	Failed Screenshot

(b) After editing

Figure 1.5: Visual test on button

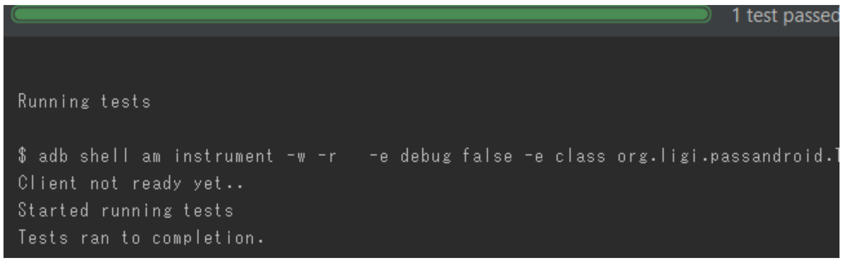
Let us now assume that the written test is a layout based one. This time the test will use the id of the button to find the element so any change to its appearance will not influence the result of the test.



```
Running tests

$ adb shell am instrument -w -r   -e debug false -e class org.ligi.passandroid.
Client not ready yet..
Started running tests
Tests ran to completion.
```

(a) Before editing



```
Running tests

$ adb shell am instrument -w -r   -e debug false -e class org.ligi.passandroid.
Client not ready yet..
Started running tests
Tests ran to completion.
```

(b) After editing

Figure 1.6: Layout test on button

1.3.2 The project

The tool I worked on is just part of a bigger project which objective is to allow an hybrid approach without requiring the tester/developer to have knowledge about both generations of test automation. [25]
“The project can hence provide benefits such as:

1. Automated generation of Layout-based tests from Visual GUI tests and the other way around, reducing development costs.

2. Reduced script maintenance costs through automated analysis of failing locators, and repair based on the other generation’s scripts.

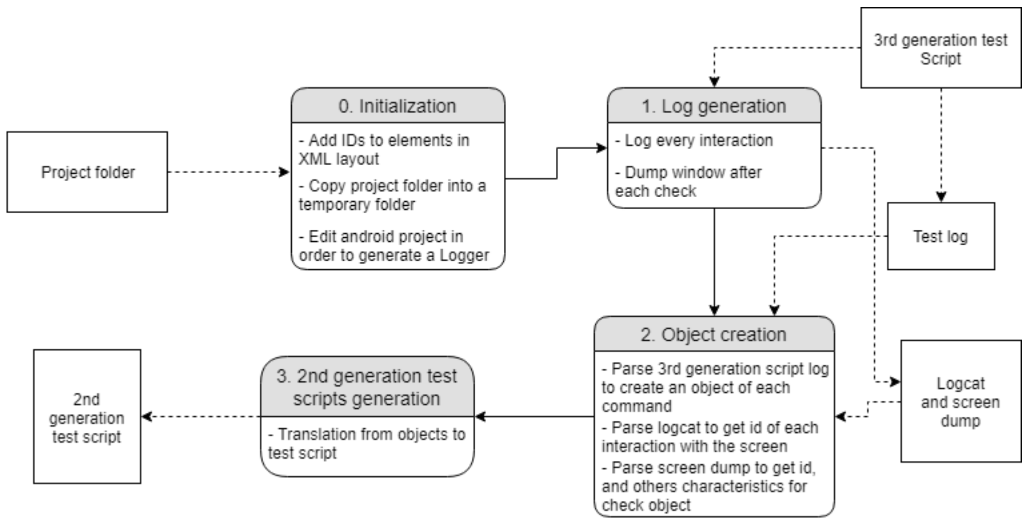
3. Reuse of existing Layout-based tests for testing the SUT’s visual appearance

4. Porting of visual scripts for different devices/configurations through strategic reuse of Layout-based tests, i.e. a single test can be run on a set of emulated devices to automatically obtain Visual GUI test scripts specific to each device.

5. Limited need for costly manual, repetitive, and error-prone regression testing.”

Point 4 of the previous list is the main focus of the tool I implemented.

1.4 From Visual GUI to Layout-based tests



3rd generation test Script
0. Initialization
- Add IDs to elements in XML layout
- Copy project folder into a temporary folder
- Edit android project in order to generate a Logger
1. Log generation
- Log every interaction
- Dump window after each check
Project folder
Test log
2. Object creation
- Parse 3rd generation script log to create an object of each command
- Parse logcat to get id of each interaction with the screen
- Parse screen dump to get id, and others characteristics for check object
3. 2nd generation test scripts generation
- Translation from objects to test script
2nd generation test script
Logcat and screen dump

Figure 1.7: Process steps

To allow the conversion from Visual GUI to Layout-based tests four main steps were identified:

0. Initialization
1. Log generation
2. Object creation
3. Second generation test script generation

1.4.1 Initialization

The initialization step purpose is to add the missing identifiers to the XML and prepare the android application for the next steps. The android project copied in a temporary directory will be edited in order to allow the generation of a logger with the second generation test script (in our case EyeAutomate) execution.

1.4.2 Log generation

Logs will be generated executing the EyeAutomate test. The logs will contain information about each interaction and a dump of the screen after each check command.

1.4.3 Object creation

In this step both the logcat and EyeAutomate script log are parsed in order to generate some object representing each command done by the script execution. Information to the check object will be added by the dump.

1.4.4 Second generation test script generation

In this last step the collection of objects is translated in a second generation (in our case Espresso) test script.

1.4.5 Input and Output

As a tool that automatically translates from Visual GUI to Layout-based tests the input and output are respectively scripts of these kinds.
As Visual GUI script type EyeAutomate was chosen while Espresso was chosen for the Layout-based one. This choice was made based on the fact that familiarity with these kinds of tests was already present due to the Software Engineering course.

Chapter 2

Tool preparation

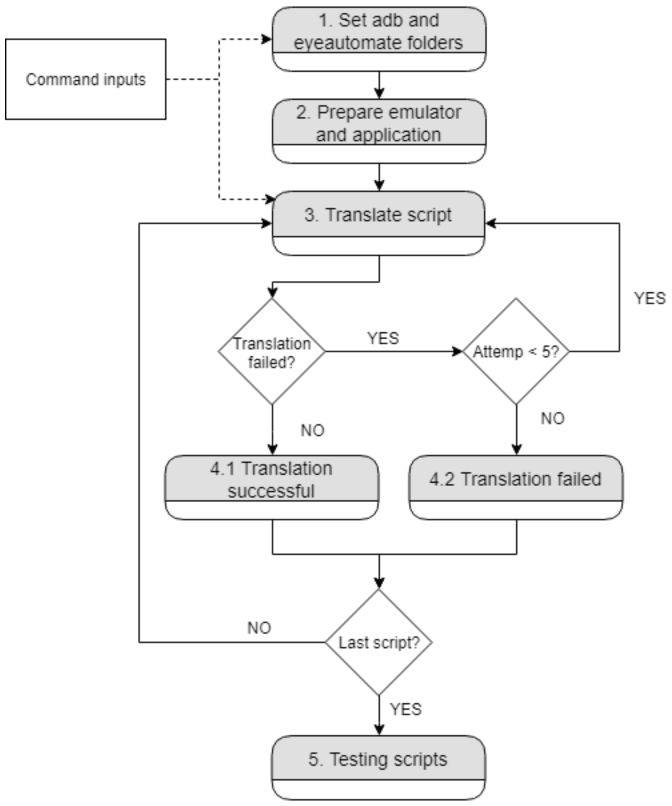


Figure 2.1: Main and executor steps

This chapter introduces the tool preparation step. As the name suggests this step consist in preparing the tool before starting with the translation steps. In this step the emulator is stared and prepared for the application execution.

The path of the scripts to be translated and the application project are also gathered and each test will counts a maximum of five retry attempts.

2.1 Main class and translation automation

To ensure the correct and easy translation of the script a main class was developed that, with the help of the Executor class, automatize the procedure by executing the tool from the command line. The first execution of the tool will require setting the Adb and EyeAutomate paths that will be saved in a config file. Application project and scripts path must also be provided with the following format.

.\tool [-setEye <eye path>] [-setAdb <adb path>] <application path> <script path> ...

2.2 Attempts

Scripts cannot always be translated on their first attempt: sometimes the script to translate could fail its execution (for example when executing swipe actions) and sometimes the time on the emulator may not coincide with our machine. To increase the possibility of translating a script it was decided to retry its translation five times.

2.3 Emulator

In order to execute the scripts an android emulator must be started. This emulator time may differ from our machine time and have a different clock. To ensure that the two will always have the same one it will be resetted before each script execution.

2.4 Adb, Gradle and Eyeautomate

In order to ensure the communication between the emulator and our machine and editing some setting of the emulator the adb (Android Debug Bridge) tool will be used. Installation and build of the application will instead be done with the gradle system.

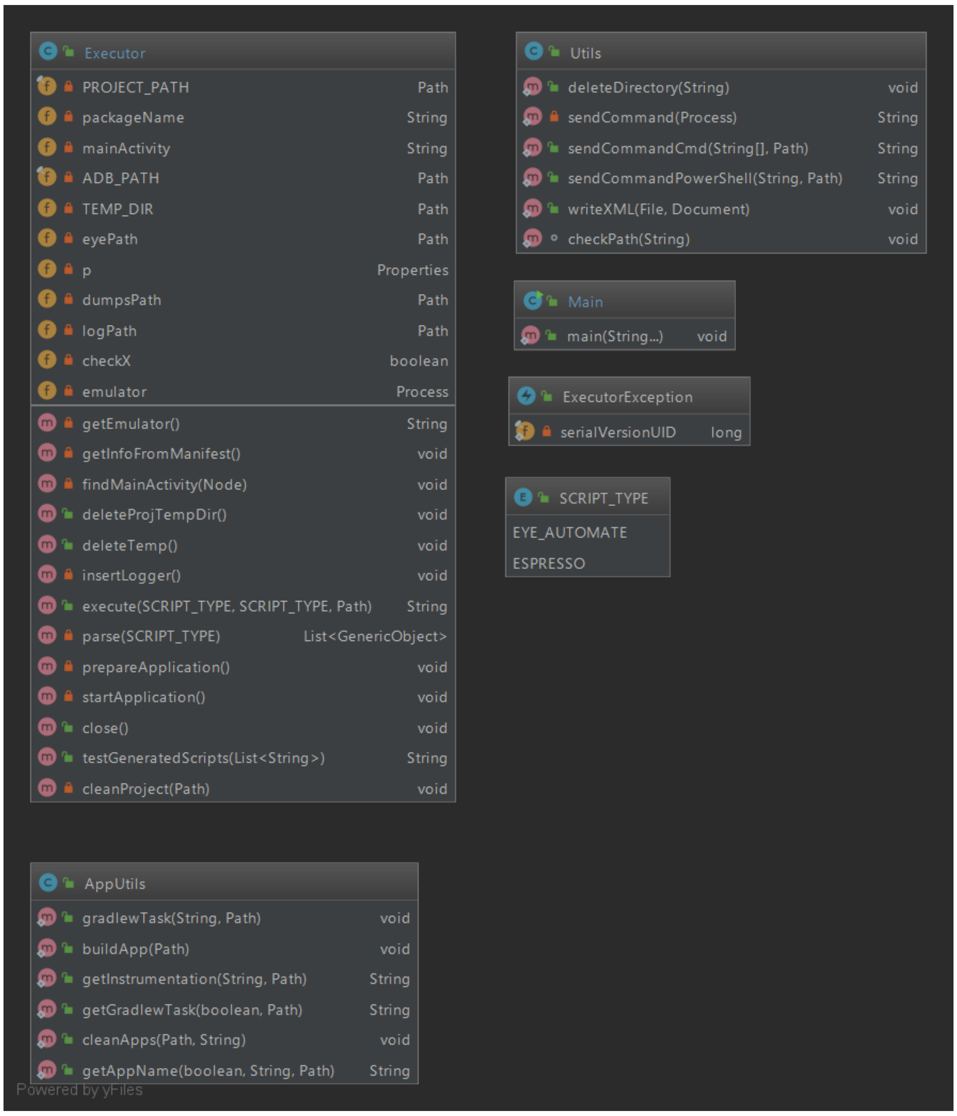

Executor
PROJECT_PATH Path
packageName String
mainActivity String
ADB_PATH Path
TEMP_DIR Path
eyePath Path
p Properties
dumpsPath Path
logPath Path
checkX boolean
emulator Process
getEmulator() String
getInfoFromManifest() void
findMainActivity(Node) void
deleteProjTempDir() void
deleteTemp() void
insertLogger() void
execute(SCRIPT_TYPE, SCRIPT_TYPE, Path) String
parse(SCRIPT_TYPE) List<GenericObject>
prepareApplication() void
startApplication() void
close() void
testGeneratedScripts(List<String>) String
cleanProject(Path) void
Utils
deleteDirectory(String) void
sendCommand(Process) String
sendCommandCmd(String[], Path) String
sendCommandPowerShell(String, Path) String
writeXML(File, Document) void
checkPath(String) void
Main
main(String...) void
ExecutorException
serialVersionUID long
SCRIPT_TYPE
EYE_AUTOMATE
ESPRESSO
AppUtils
gradlewTask(String, Path) void
buildApp(Path) void
getInstrumentation(String, Path) String
getGradlewTask(boolean, Path) String
cleanApps(Path, String) void
getAppName(boolean, String, Path) String
Powered by yFiles

Figure 2.2: Main and executor classes

Chapter 3

Initialization

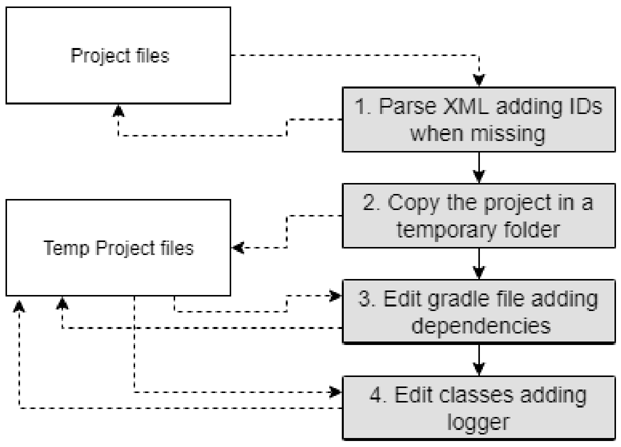
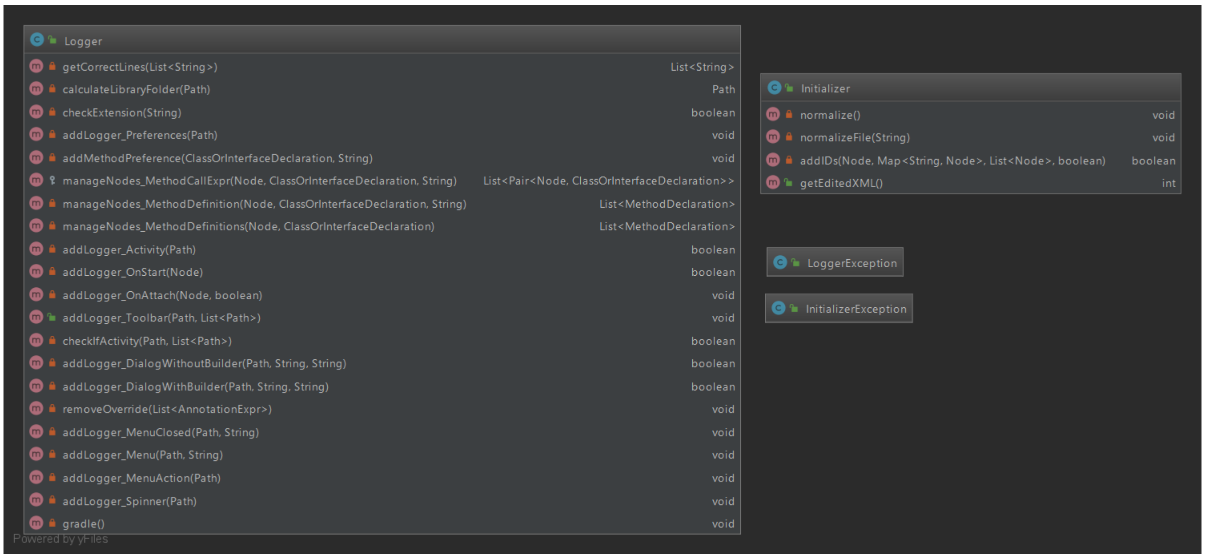

Project files
1. Parse XML adding IDs when missing
2. Copy the project in a temporary folder
Temp Project files
3. Edit gradle file adding dependencies
4. Edit classes adding logger

Figure 3.1: Initializer

This chapter introduces the initialization step. This step is the phase zero of the translation. It takes care of:

- normalizing the application files for the next steps, which means making sure that each element in each layout file of the application is provided with an ID;
- copying the android project in a temporary directory;
- edit the temporary project in order to generate a log line after each interaction.



Logger	
getCorrectLines(List<String>)	List<String>
calculateLibraryFolder(Path)	Path
checkExtension(String)	boolean
addLogger_Preferences(Path)	void
addMethodPreference(ClassOrInterfaceDeclaration, String)	void
manageNodes_MethodCallExpr(Node, ClassOrInterfaceDeclaration, String)	List<Pair<Node, ClassOrInterfaceDeclaration>>
manageNodes_MethodDefinition(Node, ClassOrInterfaceDeclaration, String)	List<MethodDeclaration>
manageNodes_MethodDefinitions(Node, ClassOrInterfaceDeclaration)	List<MethodDeclaration>
addLogger_Activity(Path)	boolean
addLogger_OnStart(Node)	boolean
addLogger_OnAttach(Node, boolean)	void
addLogger_Toolbar(Path, List<Path>)	void
checkIfActivity(Path, List<Path>)	boolean
addLogger_DialogWithoutBuilder(Path, String, String)	boolean
addLogger_DialogWithBuilder(Path, String, String)	boolean
removeOverride(List<AnnotationExpr>)	void
addLogger_MenuClosed(Path, String)	void
addLogger_Menu(Path, String)	void
addLogger_MenuAction(Path)	void
addLogger_Spinner(Path)	void
gradle()	void

Initializer	
normalize()	void
normalizeFile(String)	void
addIDs(Node, Map<String, Node>, List<Node>, boolean)	boolean
getEditedXML()	int

Figure 3.2: Initializer and logger classes

3.1 Instruments

3.1.1 Support Libraries and gradle file

Not always the support libraries are imported in the android project. During the testing with a batch of application two classes not always imported were found: the RecyclerView and support Preference libraries. To avoid any issue their dependencies are added in the gradle files.

AndroidX

An analysis of the gradle files must be done before starting in order to find out if AndroidX is been used. AndroidX is an improvement of the Support Library which provides backward-compatibility across Android releases.The support library packages have been mapped into corresponding **androidx.*** packages [13]. This will influence all the packages referencing in the inserted code.

3.1.2 Javaparser

Until now we have seen the code placing from the android project point of view. When we need to automatize this process we need to be able to easily manage the java file. To do so the JavaParser tool [20] has been used. This tool is open-source and allows to easily generate, analyze and process java.

3.2 Layout

A layout resource is an XML file which defines the User Interface. Each View or View Group that is defined by the file can be associated to an ID which uniquely identify each element in the current inflated layout. Same identifiers can be used in different layouts but it is good practise to avoid it.

3.2.1 Layout folders

The layout of an Android Application must be placed in one of the folders starting with "layout" or "menu". The later contains all the XML layout files that define the user interface, the further the XML files that define app menus [12]. Both of these folders must be placed inside the resources folder "res". These information are important in order to be able to localize the right XML file to normalize among all the application files.

3.2.2 Identifiers and Layout-based tests

In Layout-based tests it is easier to identify a view using its identifier but this can also include some risks: the test will check each layout file for an identifier and it will call an Exception if two matching views are found. When generating the new identifiers during the normalization, it is important to try avoiding redundancy in order to reduce risks. In the tool this was done by adding to a fixed string the name of the layout file and a random number (in case of two layout files with the same name but different folders).

3.2.3 Dynamic generated layout

During the execution of the application some layout may be generated dynamically. This may cause the presence of duplicate identifiers. To avoid this issue the unique parent identifier is instead used in the logger and the element can be referred using the child feature.

3.2.4 Programmatically added views

Not all the views are created through an XML file: some views can be programmatically added to the layout. This will not assure that all the views have an ID as we need to. If none of its parents has an ID the root view will be used in the future steps.

3.3 Normalization

The normalization is executed by selecting all the layout files and for each one:

1. a random number, useful for the ID generation, is selected;
2. all the elements with and without an ID are collected;
3. an ID to all the elements without one is added by paying attention that it was not already used by another one.

The random number is used in order to try avoiding common IDs among different layout files with the same name. The name of the layout file will also be used in the ID creation in order to reduce risks.
An ID will be added to all the elements in the layout XML file apart from those defined by the include type. This is done because all the views of the included files will have had an ID added in this step.

3.4 Application logger

The purpose of the application logger is to generate a log for each interaction with the application from the EyeAutomate test script which will help in the objects creation.

3.4.1 MyWindowCallback

A MyWindowCallback class has been defined in order to edit each window callback interface so that a log is generated after every click or swipe. This class extends the android.view.Window.Callback class and edits the behaviours after specific events:

- Touch events
- Focus change events
- Detached from window events
- Dispatched key events

onTouchEvent

When a touch event is registered a different behaviour is taken based on the kind of event.

- android.view.MotionEvent.ACTION_DOWN: it is called when the screen is being pressed. When the event matches this case:

    1. it is checked whether this is or not the first instance of the click or swipe (the swipe will cause other calls of the onTouchEvent method with an ACTION_DOWN event type but different coordinates);
    2. the event time is saved;
    3. the event coordinates are saved;
    4. the clicked view is calculated:
        (a) every element is added in a map as a key and the pair parent-child index as value.
        (b) every element whose rectangle comprehend the coordinates of the click event is added to an array
        (c) each element in the array is checked and if it is clickable it takes the place of the current selected view to be reported in the log. If the view is not clickable it does not replace the view that has to be reported if this one is clickable or child of a clickable parent. The string containing the information of the interaction is also created in this step. If the view does not have a unique ID a parent-child index relationship will be used to identify it.

- android.view.MotionEvent.ACTION_UP: it is called when the screen is being released. When the event matches this case it is checked whether the event is a click or a swipe by checking the distance between the starting and ending position:

    – in case of a click event it is checked if it is a short click or a long click
    – in case of a swipe event the distance is added to the reported string.

onWindowFocusChanged

When a focus change event is registered the Fragment Manager and the fragment list is used to check all the current Fragments. We will talk about this fragment list on section 3.4.3. If a DialogFragment is found the custom window callback interface is attached to the Dialog window inside the DialogFragment.

dispatchedKeyEvent and onDetachedFromWindow

The *dispatchedKeyEvent* method is edited in order to intercept the pressing of the back button.
The *onDetachedFromWindow* method is edited in order to intercept dialogs dismiss.

3.4.2 AdapterView and RecyclerView

During the initialization step each element in the layout obtained its own ID. This does not guarantee that there will not be two elements with the same ID: it is the case of AdapterView and RecyclerView. These views allow to represent a dynamic list whose children have all the same IDs but can be told apart from the child number. This information must be added when the clicked element id is given by the logger, especially in case of a RecyclerViews. An "R" will be in fact placed before the child indexes of a RecyclerView in order to identify it: they will be handled differently in later steps.

3.4.3 Dialog and DialogFragment

A Dialog is a window that either gives out some information or prompts the user to input or accept something and floats on top of the rest of the content.
A DialogFragment is a Fragment that displays a Dialog and allows to control it through itself.
Dialogs not managed by a DialogFragment are not detected by the Fragment manager and have to be managed by separately. The first step is detect which objects are Dialogs: to do so a set of application was chosen to detect some of the possible Dialogs types. These classes where collected and a class extending each of them was created, adding the custom window callback to them.
The Dialogs can be grouped based on the presence or absence of a Builder:

- Dialogs with a Builder
  - MaterialDialog
    (com.afollestad.materialDialogs.MaterialDialog)
  - AlertDialog
    (android.support.v7.app.AlertDialog or android.app.AlertDialog)

- Dialogs without a Builder
  - DatePickerDialog
    (android.app.DatePickerDialog)
  - TimePickerDialog
    (android.app.TimePickerDialog)
  - BottomSheetDialog
    (android.support.design.widget.BottomSheetDialog)

The dismiss method of a dialog is called before the onTouch one. In case of dialogs which classes have been extended the cancel listener can be edited in order to report this event. In case of DialogFragments the onDetachedFromWindow method comes to our rescue as was mentioned before.

3.4.4 Other cases

Studying the set of applications I was able to identify some other cases that were not covered due to change of focus like in the Dialogs case.

Preferences and Spinners

Preferences and Spinners are special kind of Dialogs that can be managed by editing the behaviour of a single method.

- Preferences allow the user to change functionality and behaviour of an application. The method to be edited is the onPreferenceChange one.
- Spinner provide an easy way to select a value from a set. The method to be edited is the onItemSelected one. Special attention should be placed on this method for two reasons:
  - the method could be called without clicking on a value
  - the method is not called when the clicked value is the same as the already selected one

In order to simplify later steps the text of the selected preference/spinner is logged.

Toolbars

During the creation of the Activity or Fragment the window callback interface may be overwritten by setting a Toolbar to act as an ActionBar for the activity.

android.app.Fragments

Management of fragments it is not easy when the fragments are not from the support library. These fragment are managed by the FragmentManager which does not have a method to keep track of the fragments. To simulate this behaviour a list of Fragments can be stored in the window callback through the onAttachFragment method called by the Activity every time a Fragment is attached to it (section 3.6.1)

Overflow menu

Another special case that has to be managed manually are the items in the menu. Some of the menu items may be available through a window which is different from the activity one (overflow menu). A log for each item from the menu selected can be provided in order to have a report for these items as well.
As in the previous case the text content of the menu option will be logged. The closure of the overflow menu must be detected through a different method: the *onPanelClosed* method. This method must be extended from the activity class.

Back button key

As mentioned in section 3.4.1 the back button is logged through the *dispatchedKeyEvent* method. Unfortunately this method will not intercept all the events since the closure of the keyboard will be managed by another callback.

3.5 Issues with logger implementation

3.5.1 Logger placing

One of the main issues of this step was deciding where to place the logger.
A first solution was editing the click behaviour of each view by reporting their id every time they were clicked. This solution obviously has some disadvantages:

- each element of the view would become clickable causing change of behaviour of the activity or fragment;
- changes of the click listener must be monitored because they would bring to loosing the logger;

- there would be issues in managing external libraries.

A second approach was editing the onTouch event handler by directly putting the code inside the activity. This behaviour as well has some disadvantages:

- it does not allow a big flexibility in setting the logger;
- big chunk of code should be placed in various point of the class;
- there would be many repetitions of pieces of code.

For these reasons a third approach was taken which comports in changes to the window callback interface which is not usually edited.

3.5.2 Logger in custom activity

As it was just mentioned a third approach was taken which comports in editing the window callback interface. To do so the window callback interface must be edited at the beginning of the activity life-cycle. The onStart method was chosen for this purpose because it takes place after the window has been created.
If an activity extends another custom activity it is not always needed to change the onStart method behaviour. If the activity calls the extended activity onStart method it would be useless to change its behaviour.
If an activity extends an android.app.Activity class the support fragment manager is not available. For this reason a check on the kind of activity will be done before setting the callback.

3.5.3 Logger in Dialogs

When trying to add the logger to the Dialogs a first approach has been editing the dialog listeners. This has some disadvantages:

- it must be kept track of changes on these listeners;
- the listeners must be manually set in order to be manageable;
- listeners could be easily overwritten.

The next approach has been editing the dialog class in order to add the callback when the dialog is created. This approach encountered problems when the dialog is created using a builder that will return a dialog of the class our custom dialog extends from.
In case of Dialogs with builder the build method is edited, changing the

window callback interface of the returned dialog. Editing the build method is not enough because the builder standard methods usually return a builder of the extended type.

Dialog cancel

As mentioned before a dialog dismiss skips the onTouch method call. To record the dialog cancel call a first approach was to set the dialog as not cancelable and call the cancel method when clicking outside the dialog. This would not cover back pressing or programmatically cancel/dismiss of the dialog and will not cancel the dialog. The approach mentioned in previous sections has then been chosen.

3.6 Extended methods

In the logger development three main methods had to be extended:

- onAttachFragment
- onStart
- onPanelClosed

Of these three methods we will see the first two in the next sections while the third one has already been addressed in section 3.4.4

3.6.1 onAttachFragment

The onAttachFragment is not always called after the onStart method: in some cases it could in fact be called during the creation phase. If the window callback has not be set when the fragment is attached it can be set before the onStart method, which will check the callback interface in order to avoid overwriting it.
As the name suggests this method is called every time a fragment is attached to the activity. With this method we can add to the list of fragments of the window callback the new fragment to be handled.

3.6.2 onStart

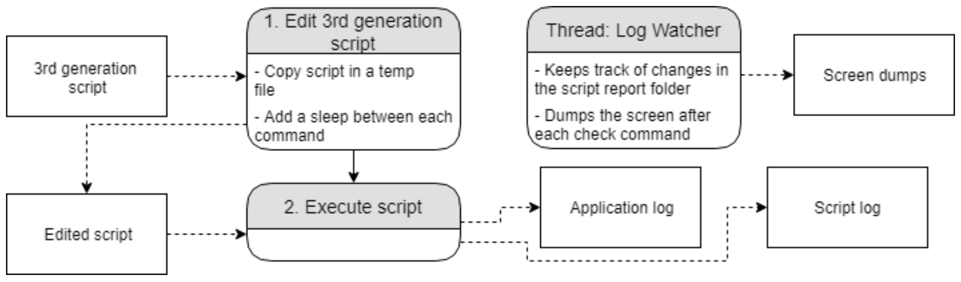
The onStart method is used to set the window callback. In order to reduce rendundance the method is extended only if the class does not its super method or the extended class is not a custom one.

3.7 Tool custom classes and modules

The package containing the tool custom classes (MyWindowCallback and Dialogs) should be placed inside each module source folder. The packages in each module cannot contain the same classes since we cannot be certain that each module has the same dependencies.

Chapter 4

Log Generation



3rd generation script
1. Edit 3rd generation script
- Copy script in a temp file
- Add a sleep between each command
Thread: Log Watcher
- Keeps track of changes in the script report folder
- Dumps the screen after each check command
Screen dumps
Edited script
2. Execute script
Application log
Script log

Figure 4.1: Log Generation steps

This chapter introduces the log generation step. With the initialization completed it is time to execute the Visual GUI scripts that are going to be translated. This execution allows the generations of logs that are going to be used for the translation. Two types of logs are going to be generated:

1. Application log
2. Script log

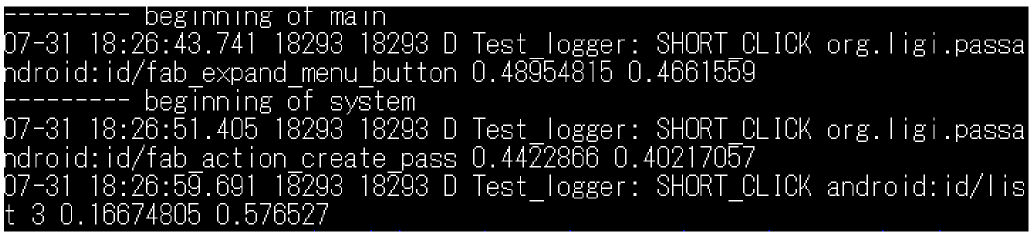
Screen dumps are also going to be generated after every check.

4.1 Initialization

Before executing the script it has to be edited. The executed one will be an edited temporary version on which a sleep is added between each command in order to easily differentiate each click operation and complete the screen dumping.
A thread (LogWatcher) is also executed with the script which will use a

WatchService to keep track of changes in the folder with the script report and it will dump the screen every time a check command is added to it.

4.2 Application logs



```
--------- beginning of main
07-31 18:26:43.741 18293 18293 D Test_logger: SHORT_CLICK org.ligi.passa
ndroid:id/fab_expand_menu_button 0.48954815 0.4661559
--------- beginning of system
07-31 18:26:51.405 18293 18293 D Test_logger: SHORT_CLICK org.ligi.passa
ndroid:id/fab_action_create_pass 0.4422866 0.40217057
07-31 18:26:59.691 18293 18293 D Test_logger: SHORT_CLICK android:id/lis
t 3 0.16674805 0.576527
```

Figure 4.2: Application log

This log is going to be generated by the application execution thanks to the code inserted to its project during the initialization step. A log message is mainly composed by a time-stamp, a tag and a string. Other values are in between the time-stamp and the tag but they can be ignored.

- The time-stamp is used to associate each operation of the script with its log message.
- The tag is used to find the log messages we need among all the others and the string "Test_logger" was chosen for it
- The string contains the log message we are going to use

4.2.1 Structure of the log string

Let us now examine the log string. This string as well has a specific structure.

1. ACTION:
   The first part of the string and only mandatory part of the string. It indicates the type of action the log represent. It can be one of the following types:
   - SHORT_CLICK
     It is used for a simple short click
   - LONG_CLICK
     It is used for a simple long click

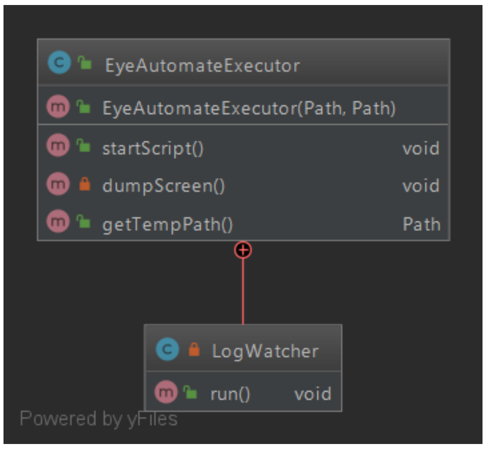


Figure 4.3: Log generation and script executing classes

- SWIPE
  It is used for a swipe or drag and drop
- BACK
  It is used when the back button is pressed during the navigation
- SPINNER
  It is used to indicate the new Spinner value when it is a string
- PREFERENCE
  It is used to indicate the Preference value when it is a string
- MENU
  It is used to indicate the value of the menu option pressed when t is a string. It used in case of overflow menu in which click is not intercepted by the activity
- MENU_DISMISSED
  It is called when the overflow menu is being closed
- DIALOG_CANCEL
  It is called when a dialog has been canceled

2. IDs or TEXT:
   The second part of the string is optional and can show either the ID of the clicked view or the text value of the clicked menu/preference/spinner option.
   In case it contains the IDs it will show the clicked view id, if it has one. If the the view does not have an ID or it is not unique in the current visible layout a list of indexes will be added. These indexes represents the parent-child connection between the view and the first parent with

a unique id. If there is a parent belonging to the RecyclerView class in this hierarchy it will be pointed out by adding an "R" to the index and preceding it with its id. If the string is empty a view was not found which means that the click was outside the window (e.g. a click outside a dialog). It is to be noted that if the click outside a dialog cancels the window a DIALOG_CANCEL type will be returned instead.

3. COORDINATES:
   The third part of the string indicates the position of the click in the view. It is returned in the form of percentage x and y.

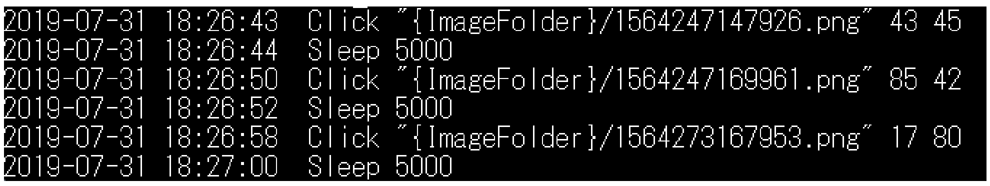
4. DISTANCES:
   In cases of swipes or drags a last information is returned which is the distance x and y from the start position.

4.2.2 Not logged cases

Unfortunately there are some cases that are not recorded.

- Not all the Dialogs can be intercepted for the reasons discussed in the previous chapter.
- The back button to close the keyboard. This can be supposed to have happened when a type has just been performed and the next click was not recorded.
- Spinners and menus value that are not texts.

4.3 Script log



```
2019-07-31 18:26:43  Click "{ImageFolder}/1564247147926.png" 43 45
2019-07-31 18:26:44  Sleep 5000
2019-07-31 18:26:50  Click "{ImageFolder}/1564247169961.png" 85 42
2019-07-31 18:26:52  Sleep 5000
2019-07-31 18:26:58  Click "{ImageFolder}/1564273167953.png" 17 80
2019-07-31 18:27:00  Sleep 5000
```

Figure 4.4: Script log

The script execution log is the second one that will be used for the translation. It reports each executed operation with its details. Each log message is composed of a time-stamp and information about the executed operation. A LogWatcher is going to be used to determinate when a check operation is happening: this allows the dumping of the current window for future commands. The script log will be used to get the list of executed commands and get information about the check coordinates.

4.4 Screen dump

UI Automator is a testing framework that provides a set of APIs to build UI tests that perform interactions on user apps and system apps. It also provides a UiDevice class to access and perform operations on the device on which the target app is running. Its methods can be called to access device properties such as current orientation or display size [15].
The dump of the window is done through this UiDevice after each check command, recognized thanks to the LogWatcher, and it helps translating this command in later steps. The screen is dumped as an XML file and contains information about visible views. These views are not ordered as in the layout files but based on their visual position. This must be considered in the object creation if the index is referenced.

4.4.1 When to dump

The main issue of this step was deciding when to dump the screen. A first approach was to dump the screen after each click operation from the application. This was more difficult than what it may look like since the click operations are not always intercepted in the same way as was previously seen. Another problem was that the operation is intercepted before the click has happened and this would show the screen before the click operation making it unusable for the check. For these reason another approach was taken which is to dump the screen with the adb shell every time a check operation is done by the script.

Chapter 5

Object creation

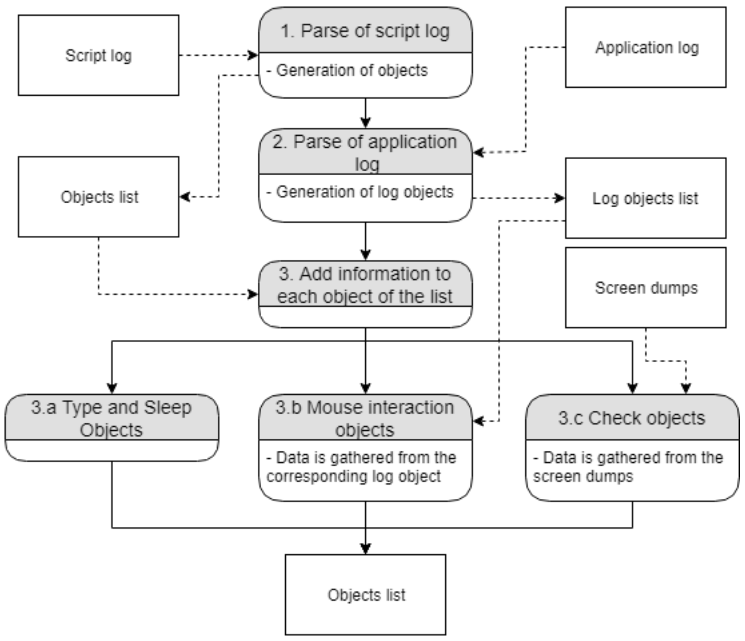

Script log
1. Parse of script log
- Generation of objects
Application log
2. Parse of application log
- Generation of log objects
Objects list
Log objects list
3. Add information to each object of the list
Screen dumps
3.a Type and Sleep Objects
3.b Mouse interaction objects
- Data is gathered from the corresponding log object
3.c Check objects
- Data is gathered from the screen dumps
Objects list

Figure 5.1: Object creations steps

This chapter introduce the object creation step. A list of object, that will be later on translated, is generated in this step through three distinct parsing operations: parsing of the application log, script execution log and screens dump. The script log will be used as base for the objects list while the other two will enrich these objects information.

5.1 Initialization

Before starting with the parsing some information of the emulator must be gathered: the screen coordinates and the screen size.

5.2 Parsing of the script log

The first parsed log is the script one. This log is used to create the list of objects whose information will be added with the next parsing operations. Not all the commands are being parsed but only the ones that could be used in the testing of an Android Application (for example Log or OpenBrowser commands have been ignored).
The parsed commands are:

- Click
- DragStart
- DragDrop
- Check
- MouseLeftPress
- MouseLeftRelease
- Type
- Sleep

Some other commands may be needed but they can be easily managed by reusing the already implemented ones (e.g. DoubleClick).
An object for almost each parsed command is generated in this step (some objects may be grouped together e.g. DragStart and DragDrop).
Different data is stored based on the type of command:

- In case of mouse interaction commands the only stored information will be the timestamp (two in case of composed commands).
- In case of type commands the typed string is stored in the object.
- In case of check commands the checked rectangle coordinates and size are stored, calculated relatively to the emulator.

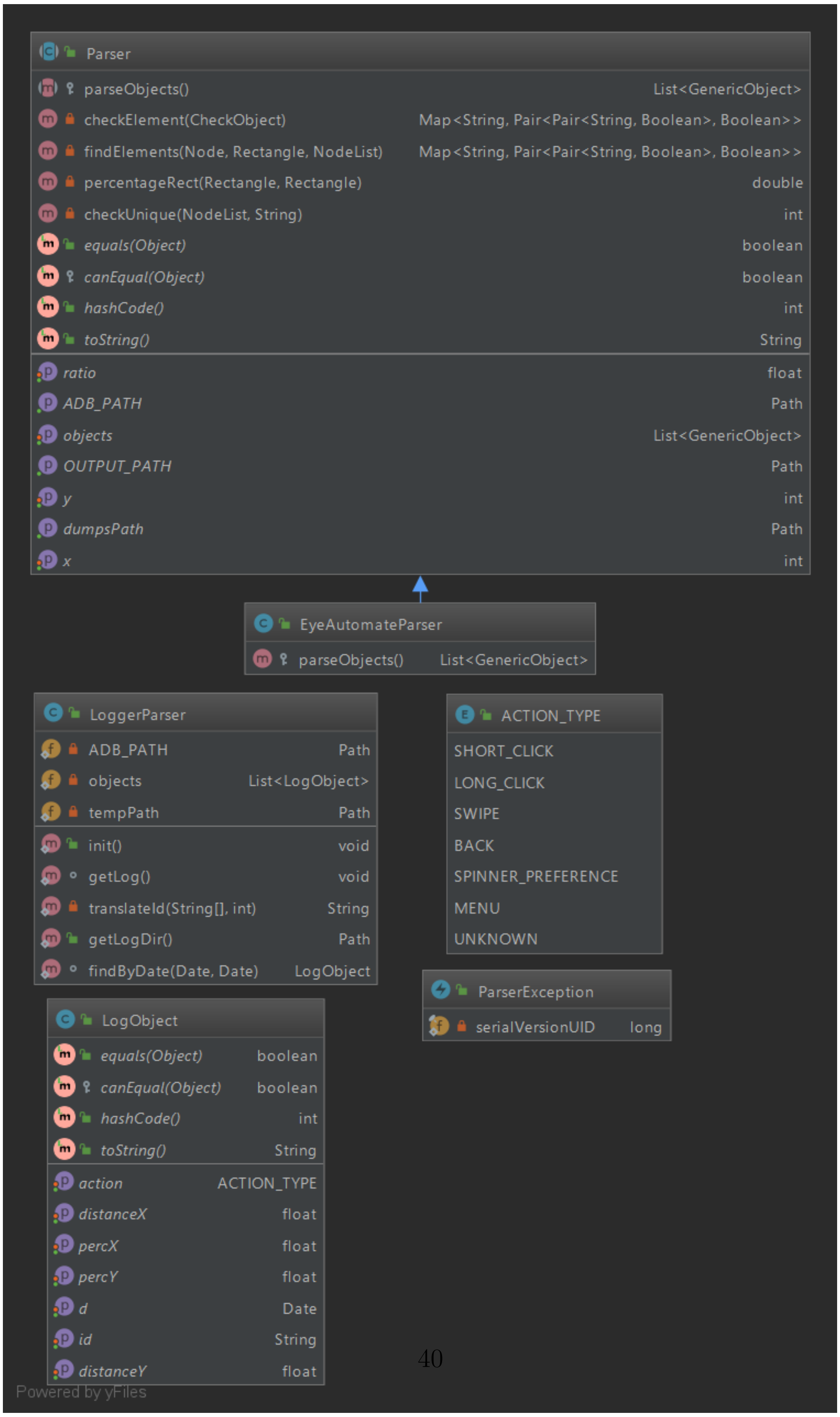


Figure 5.2: Object creation classes

5.3 Parsing of the application log

A list containing each intercepted click is generated in this step by parsing the application log. Each object will have:

- a timestamp (mandatory) used to identify each click
- an action type (mandatory) used to identify each click type. The types can be seen in figure 5.2
- a string (optional) containing either the id, the parent id-child index list or the text contained in the clicked menu/preference option.
- the pair of float numbers (optional) indicating the percentage position (mentioned in the Initialization chapter)
- the pair of distances x and y of the swipe operations (absent in the other cases)

5.4 Object information filling

At this point we have two list of objects: the one that will be used for the translation step is the one generated by the script log even thought it is missing many information. In order to fill the missing data each object will be analyzed one by one. Type and sleep objects are the only one which do not need extra information.

5.4.1 Mouse interaction objects

If the object is an instance of either a ClickObject a DragObject or the generic MouseInteractionObject the needed data will be gathered from the log objects. These object are in fact all mouse interactions which should have been intercepted by the application. In order to find the right log object the timestamp is used. The gathered informations are then added to the object.

Not logged click

A log object cannot always be found.
If a typing operation has just been completed and the script closes the keyboard by pressing the back button this operation is not logged. I have decided to assume that if a click is not logged after a type operation the keyboard has been closed: obviously this is not always true.

The opening of the overflow menu is not intercepted. If a menu action type is registered after a not logged click I have decided to assume that this is the case.
All the remaining cases are not logged clicks.

5.4.2 Check objects

In case of check commands the correct dump file must be parsed. The dump file with the closest timestamp is picked. Not all the views intercepting, containing or contained by the rectangle are chosen but only whose rectangle are in a certain percentage included in the checked rectangle. This is done in order to avoid picking every view such as their containers. After some experimenting the percentage was fixed to 10%. If no view is found the check is considered not logged and the operation fails.
As with the click objects either the IDs or the parent-child index relationships are stored in case of views without a unique id.

Chapter 6

Translation

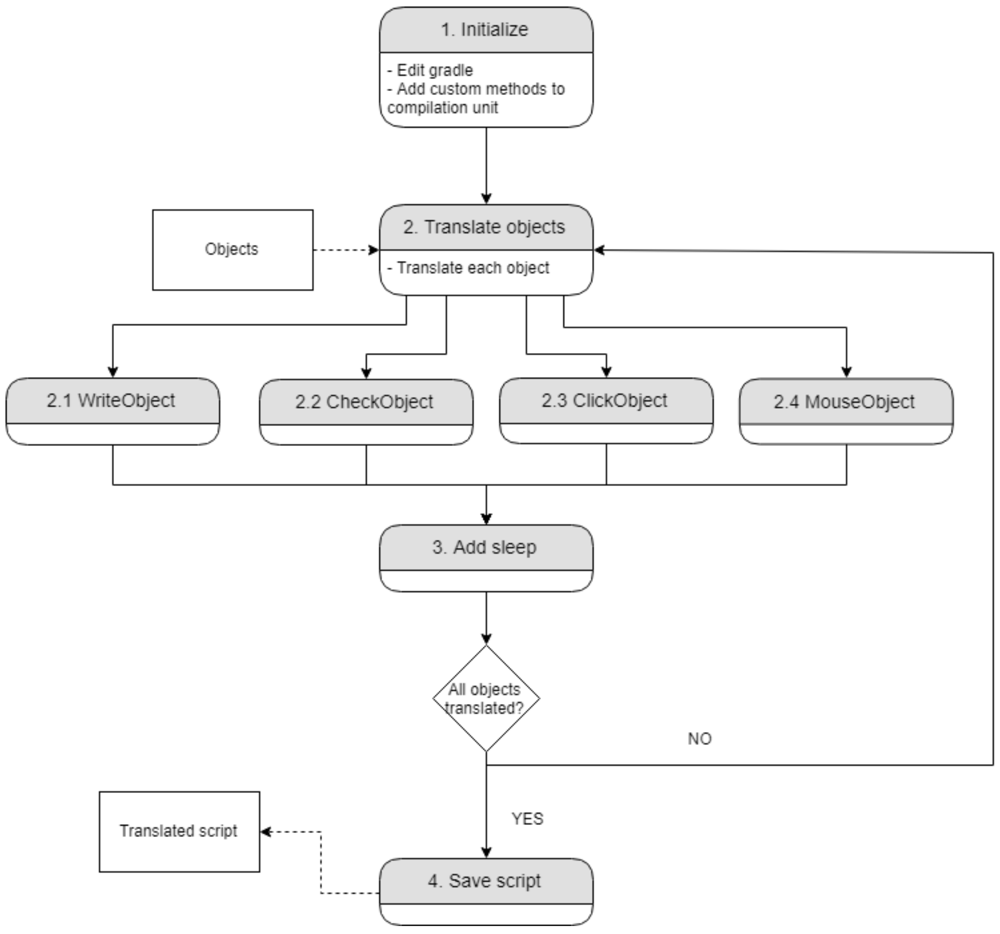


Figure 6.1: Translation steps

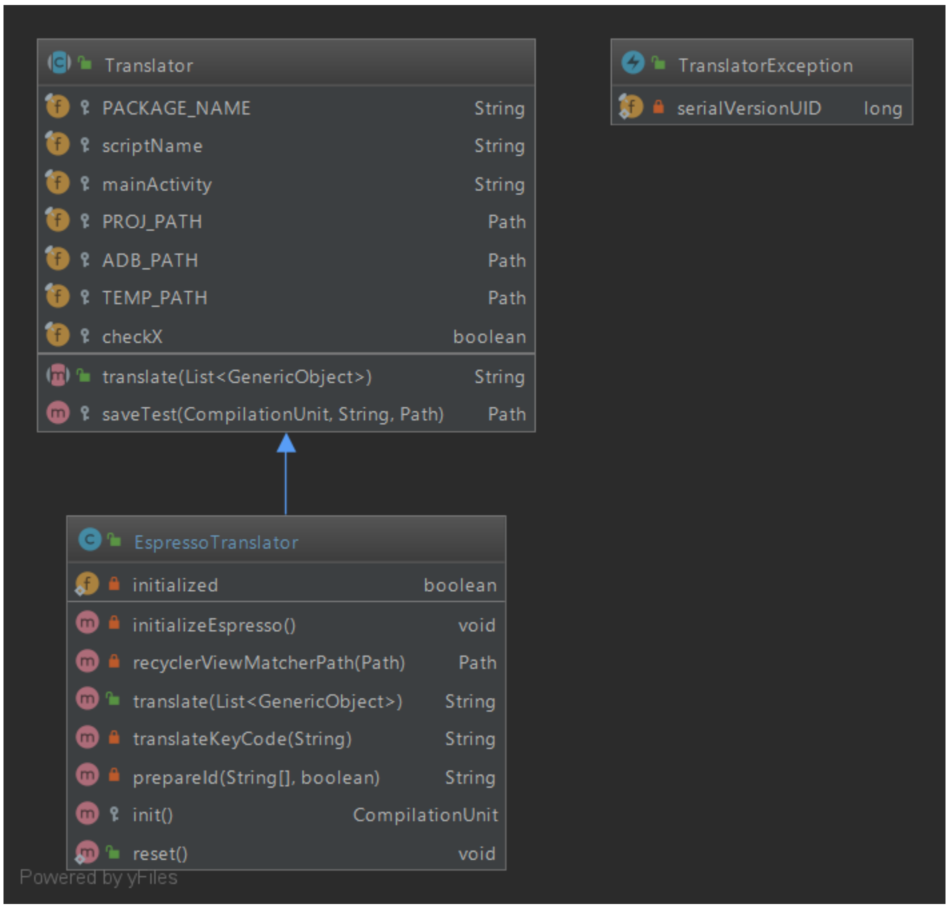


Figure 6.2: Translation classes

This chapter introduces the translation step as the last one of our tool. We now have a list of objects that must be translated into a second generation script. Before starting with the translation we need to make sure that the RecyclerView library is included in the gradle file. This library will be used in by the script when translating clicks on RecyclerView children. Each object from the list is then translated with the appropriate instructions.

6.1 Script directory

All the translated scripts will be added into the androidTest directory under the source folder. The filename of these scripts will be the same of the one they will be translated from. A method called "TestMethod" will hold the script content.

6.2 Initialization

After the gradle file has been edited it is time to insert all methods and classes needed for the translated script execution.
A file containing the RecyclerViewMatcher class will be added in the scripts directory. This class will implement a Matcher interface that matches the child of a RecyclerView.
Each script file will also have all the custom methods that could be needed for the script execution. This will increase the length of the script and add redundant code among all the translated script. A possible solution would be adding these methods into a custom class in order to avoid these issues.
Imports and all the basic code needed in an Espresso test script is also added in this step. All the code is saved in a CompilationUnit object, a class provided by JavaParser representing the entire compilation unit which is denoted by a java file. This class allows to easily edit and add Java code and print it in a file at the end of the translation operation.

6.2.1 Custom methods

As mentioned in the previous section some custom methods are needed to allow the translated script execution.

Matcher<View> childAtPosition(final Matcher<View> parentMatcher, final int position)

This method matches the view whose parent is matched by the first parameter and has *position* as an index. It is the same method that is autogenerated by the Espresso Test Recorder, a tool that helps in the generation of Espresso tests by interacting with the emulator.

Matcher<View> childAtPositionCheck(final Matcher<View> parentMatcher, final int position)

This method is quite similar to the previous one but it is used for the check operations. The index provided in the previous step may in fact not be the correct one. It is calculated based on the current visible views, starting with index 0. If the ViewGroup has been scrolled or there is a not visible view in between it is not considered in the index counting. It has also to be considered that the index does not correspond to the container position but to the drawing order position.

Matcher<View> typeMatcher()

This method matches the view it has to be typed into. Matching the view with focus is not enough since its parent will have it as well. This would generate an Ambiguous match. To avoid the problem the view is matched if it has focus and none of its children (if it has any) have it.

float[] coordinatesFunction(View view, double px, double py, double x, double y)

This method returns the coordinates the view has to be clicked at. They are calculated using the x and y left top corner position of the view and the proportion calculated in the previous steps.

6.3 ViewMatcher and id

In Espresso a view has to be provided in order to perform an action on it or check it. This can be accomplished by using a ViewMatcher. The method prepareId was implemented to simplify the preparation of the ViewMatcher.

1. The onView entry point is selected
2. allOf matcher that matches the view matching all of the specified matchers.
3. isDisplayed matcher for check operations, isDisplayingAtLeast matcher with a value of 90 for the other ones.
4. the withId matchers is used to match a view with an id (either the view we are looking for or a parent one). If no id is provided the isRoot matcher is used instead.
5. the childAtPosition, childAtPositionCheck and RecyclerViewMatcher matchers mentioned previously are instead used for the numeric part of the id string (which represents the index).

onView(AllOf(isDisplayed, ...)
onView(AllOf(isDisplayingAtLeast(90), ...)
withId(...)
childAtPosition(withId(...), n)
childAtPosition(childAtPosition(withId(...), n), n)

6.4 Object translation

After completing the initialization procedures it is time to translate the various objects.
Different behaviours are taken based on the type of object.

6.4.1 WriteObject

WriteObjects indicate type operations. This is performed using the typeMatcher method to find the view to type into and the perform and TypeTextInfoFocusedView methods to perform the action. The text to be typed is taken from the object.

onView(typeMatcher()).perform(typeTextIntoFocusedView(...))

6.4.2 CheckObject

CheckObjects indicate check operations. For each id in the id array of the object a check operation has to be performed. After preparing the entry set the check view assertions are added through the check method. The matches() assertion with the isDisplayed matcher is always added. It assures that the selected view is displayed. isChecked and isNotChecked matcher will be used with the assertion in cases of checkable views. With hint or with text is then used to check the view text content with the equalToIgnoringCase matcher which ignores the text upper or lower case. This is need because text inserted with EyeAutomate and Espresso may differ in the first capital letter.

onView(...).check(matches(isDisplayed))
onView(...).check(matches(isDisplayed)).check(matches(isChecked()))
onView(...).check(...).check(matches(withText(
    equalToIgnoringCase(...))))

6.4.3 ClickObject

ClickObjects indicate click operations. ClickObjects have different types.

- Back type - indicates a back action and can be executed with a single line code
  onView(isRoot()).perform(ViewActions.pressBack())
- Spinner_Preference type - indicates that the click selected a spinner or preference value whose value is specified in the object. Instead of

performing a click operation on a view with an id it is done on a view with that specific text.

```
onView(withText(...)).perform(click())
```

- Menu type - indicates that the click either opened the overflow menu or clicked on an item of it. It can be determined by the id string value: it can either be a string (which will be the menu option value) or null (if it is an overflow menu opening). The first case is translated as in the Spinner_Preference type case, the second one using the openContextualActionModeOverflowMenu method.

```
openContextualActionModeOverflowMenu()
onView(withText(...)).perform(click())
```

- Normal - the remaining case is an easy click on a view that could be performed with a simple click action but a GeneralClickAction action was used instead. This action allows to specify if the click is long or short and the specific coordinates of the view in which to click. The calculateCoordinates method explained before is used inside the click method.

```
onView(...).perform(new GeneralClickAction(Tap.SHORT,
   new CoordinatesProvider(){
      @Override
      public float[] calculateCoordinates(View view) {
         return coordinatesFunction(view, px, py , 0, 0);
      }
   },
   Press.FINGER))
```

6.4.4 MouseInteractionObject

MouseInteractionObjects indicate swipe operations. These swipe can be not only up, left, right or top swipe but have also different angles. They are translated similarly to the normal click operations but a GeneralSwipeAction is used instead of a GeneralClickAction.

```
onView(...).perform(new GeneralSwipeAction(Swipe.FAST,
  new CoordinatesProvider(){
     @Override
     public float[] calculateCoordinates(View view) {
        return coordinatesFunction(view, px, py , 0, 0);
     }, new CoordinatesProvider(){
        @Override
        public float[] calculateCoordinates(View view) {
            return coordinatesFunction(view, px, py , 0, 0);
        }
    },
    Press.FINGER))
```

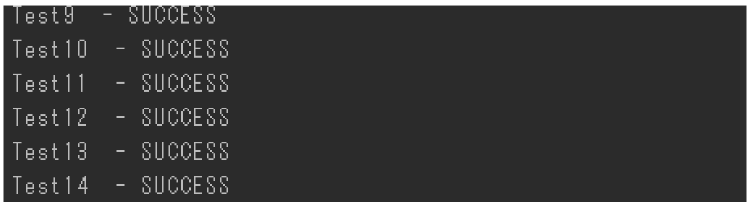
6.4.5 Sleep

Between each operation a sleep operation is inserted to avoid failure due to screen still loading. A time of three seconds was chosen to wait.

```
    Thread.sleep(3000)
```

6.5 Translation completed and testing the scripts

Once the translation has been completed it is time to save the test. The test is saved in the path previously established of the original project. After all the scripts have been translated they are then tested. The original idea was to copy the script in the real project path only if the test had been successful. The scripts had to be tested after each translation in case of build failure but this would have increased the translation time (the application would have to be built multiple time during the translation). Another found problem was that sometimes the gradle would not correctly build the test instrumentation failing test which would have instead passed. To avoid deleting possible working tests I decided to opt for copying all the translated scripts and only printing the testing results.



```
Test9  - SUCCESS
Test10  - SUCCESS
Test11  - SUCCESS
Test12  - SUCCESS
Test13  - SUCCESS
Test14  - SUCCESS
```

Figure 6.3: An example of testing output

6.6 Issues and improvements

A possible issue in the script translation could be the presence of ambiguous view in case of those matched with the text content. Generally is unusual for a spinner, menu or preference to have two options with the same value

but it is not assured the uniqueness.
A possible improvement could instead be grouping in a method the GeneralSwipeAction and GeneralClickAction code in order to avoid redundancy.

Chapter 7

Experimental Validation

This chapter introduces the experimental validation. Now that our tool is completed it is time to test its effectiveness in translating second generation scripts. To do so two test suites of thirty tests were generated for two distinct open source applications.

7.1 Experiment subjects

To evaluate the tool performance two open-source applications have been chosen: OmniNotes [19] (v6.0.0 Beta 7), a note taking application, and PassAndroid [23] (2.5.0), an electronic tickets manager. The code of both applications can be found on GitHub and both of them were released on PlayStore. These two applications were chosen for their different aspect and behaviour. Their charactersiscs can be found in table 7.1.
For the evaluation two test suites of thirty EyeAutomate tests cases each were written in order to evaluate as much behaviour and aspect as possible. Each test was designed to be executable independently in order to avoid failure in cascade. They should be launched with a clean application started normally with its launcher activity. The scripts contain from 2 to 43 commands mostly check and clicks.

7.2 Procedure and materials

The test cases have been run on a laptop PC with an Intel i7-6700HQ CPU at 2.60GHz clock, with 16GB RAM and Window 10 Operating System. The development of the test suites and execution of the EyeAutomate test cases were performed in EyeStudio 2.1. The apps have been first launched and on an emulated Nexus 6P API 26 (Android 8.0) with disabled device frame and

Table 7.1: Characteristics of selected apps (as of September 2019)

	OmniNotes	PassAndroid
Number Of Downloads	100,000+	1,000,000+
Number Of Releases	120	100
Tested release	6.0.0 Beta 7	2.5.0
Java LOCs	48,116	32,309
Number of Activities	13	17
Number of Layout Files	52	19

disabled animations.
The experimental evaluation aimed to answer the question:

RQ What is the success rate of the layout-based test scripts generated through translation?

To answer the question for each test case we computed the Success Rate (SR) which was defined as:

$$SR_t = \frac{N_s}{N_{ex}}, \tag{7.1}$$

where N_s is the number of generated test scripts whose execution ended with a success, and N_{ex} is the total number of attempted translated scripts. Failed tests can be of two types:

- Tests failed to be generated
- Tests generated but failed their execution

The test cases are summarized on table 7.2 and 7.3.

7.3 Experimental results

As can be seen in figure 7.1 the experiment had a quite high success rate (96.7%, 29/30, in both test suites with a total of 96.7%, 58/60). Figure 7.3 shows a summary of all the developed test cases for the two applications and the individual test cases results. As we can see from the table the two scripts translation failed on different steps.
The OmniNotes test script failed in the generation step. The cause was the closure of the keyboard without any input. If there had been any input the tool would in fact have translated the missing click in a keyboard closure. This could be fixed by implementing the logger keyboard interception.
The PassAndroid test script failed in the testing step. It was in fact generated

TEST	**DESCRIPTION**	**NUMBER OF COMMANDS**
Test1	Test create note	7
Test2	Test trash	10
Test3	Check trash content	18
Test4	Test archive	11
Test5	Test swipe notes left and right	36
Test6	Test add reminder	13
Test7	Test create checklist	8
Test8	Test disable checklist	15
Test9	Test enable checklist	16
Test10	Test search	14
Test11	Test search wrongly	11
Test12	Test setting options swipe notes	5
Test13	Test setting options privacy and error reporting	4
Test14	Check floating action create Text note	2
Test15	Check floating action create Checklist	2
Test16	Check floating action add Photo	2
Test17	Test reduce and expand view	6
Test18	Check drawer menu	2
Test19	Check order menu	3
Test20	Check find operation	4
Test21	Test delete note after creation by swiping and undo operation	11
Test22	Test delete note from archive by swiping and undo operation	16
Test23	Test selected notes value is correct	20
Test24	Test order notes by last creation date	16
Test25	Test order notes by last modification date	16
Test26	Test add shortcut	13
Test27	Test discard changes to title before creating note	7
Test28	Test discard changes to content after note has been created	11
Test29	Test sketch	7
Test30	Test removing category	8

Table 7.2: Description of OmniNotes test case

TEST	**DESCRIPTION**	**NUMBER OF COM-MANDS**
Test1	Test set typed time	11
Test2	Test check to calendar	11
Test3	Check color wheel	18
Test4	Check images tab	13
Test5	Test scroll to last and first page	43
Test6	Test set description	6
Test7	Test set QR	7
Test8	Test set PDF41	7
Test9	Test set AZTEC QR	7
Test10	Test set message	8
Test11	Set alternative message	8
Test12	Check create pass	3
Test13	Check scan for pkpass files	3
Test14	Check add demo pass	3
Test15	Check open file	3
Test16	Check floating actions menu button	3
Test17	Check info	2
Test18	Check drawer menu	3
Test19	Check create pass	5
Test20	Test set description	13
Test21	Test delete pass from details	8
Test22	Test delete pass selected	11
Test23	Test edit pass	6
Test24	Test pass details toolbar	7
Test25	Check selected pass toolbar	10
Test26	Test set boarding pass	3
Test27	Test set generic pass	3
Test28	Test set coupon	3
Test29	Test set store card	3
Test30	Test set pass event and generic	4

Table 7.3: Description of PassAndroid test case

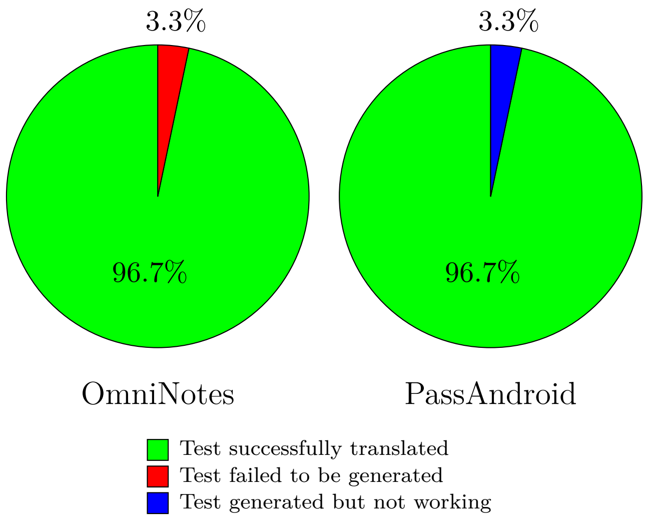


Figure 7.1: OmniNotes and PassAndroid success rate

a non working Espresso script. The script tried in fact to type inside the focused view but none had any focus during the script execution. To fix this typing may have to be handled differently in order to not have to guess on which view it has to be typed to.
In conclusion we can be satisfied with the tool results and high success rate that proves the tool capabilities even if some little improvements may be needed for industrial uses.

Test	SUCCESS	FAILED GENERATION	FAILED EXECUTION	Test	SUCCESS	FAILED GENERATION	FAILED EXECUTION
Test1	X			Test1			X
Test2	X			Test2	X		
Test3	X			Test3	X		
Test4	X			Test4	X		
Test5	X			Test5	X		
Test6	X			Test6	X		
Test7	X			Test7	X		
Test8	X			Test8	X		
Test9	X			Test9	X		
Test10	X			Test10	X		
Test11	X			Test11	X		
Test12	X			Test12	X		
Test13	X			Test13	X		
Test14	X			Test14	X		
Test15	X			Test15	X		
Test16	X			Test16	X		
Test17	X			Test17	X		
Test18	X			Test18	X		
Test19	X			Test19	X		
Test20		X		Test20	X		
Test21	X			Test21	X		
Test22	X			Test22	X		
Test23	X			Test23	X		
Test24	X			Test24	X		
Test25	X			Test25	X		
Test26	X			Test26	X		
Test27	X			Test27	X		
Test28	X			Test28	X		
Test29	X			Test29	X		
Test30	X			Test30	X		

Table 7.4: OmniNotes and PassAndroid test results

7.4 Threats to validity

7.4.1 Threats to Construction Validity

The tool effectiveness was measured in terms of success rate. A test is considered successful if it passes. Another aspect would be making sure that the translated Espresso script executes the correct instructions in order to avoid false positive. We decided to keep this limitation considering that avoiding false negatives is more relevant.

7.4.2 Threats to Conclusion Validity

Standard statistical test were applied to test the tool effectiveness. The results are consistent with the visual representation (96.7%).

7.4.3 Threats to External Validity

The experimental design includes some unfairness as only interactions and dialogs supported by the translator were used. Therefore the results of this evaluation are not generalizable to any EyeAutomate test suite. However, since the supported commands of the tool include the most common commands used in EyeAutomate and the dialogs were selected from a set of applications, we believe that the result is still meaningful to show the strengths of the tool, especially since the translated third generation test cases are representative of typical tests for Android applications in terms of sequences of interactions performed. Moreover, some EyeAutomate were created based on the Espresso ones already present in the GitHub project of PassAndroid and were used for the experiment, hence adding to its external validity. Furthermore, the limitations of the current translator are only temporary technical limitations in the tool, which will be addressed in future versions, and thereby do not affect the results. Apps with a very different graphical appearance or logic or used dialogs types may induce results that vary significantly from those reported.

Chapter 8

Conclusion

This chapter concludes the thesis recapping the experimental results and future works that can be done to the tool and for the future of the project.

8.1 Discussion and open issue

Even though the tool does not support the translation of every possible interaction with the application the results indicate that the users will still get the benefit of using it. However it should be considered that the translated tests will be faulty if faults were present in the original test suits. A possible fault could be for example an interaction including an element changing based on the current time: even if the translated test appeared working at the translation moment it could fail on a later execution. Another consideration is that the custom window callback will not be overridden during the translation which would cause loss of the application log.
As explained in the implementation section the tool still does not cover all the possible interactions. Among the not implemented ones the most urgent are the dialog, preference, menu and spinner.
Dialogs, as mentioned, are not always easily interceptable so additional methods are provided to assure a log of their interaction. For this reason the support Fragment Manager and the onAttach method are used on addition to a new implementation of the dialgos. Further testing should be done to ensure if all the dialogs interactions are intercepted or if further work should be done on it.
Preference, menu and spinner interactions have a lot in common. Right now they are cover only cases when their value is textual.
Last but not least the keyboard still needs further coverage. Right now keyboards interactions are covered only if expressed through the type EyeAuto-

mate command. Other types of interactions are not covered and comports to failure in translation.

8.2 Conclusion and future works

The results from the previous chapter confirm that it is possible to translate test cases from third to second generation. Future works would cover implementing the missing feature and perform a more through validation. A further step would be ensure that the tool works properly with the previous developed one, translating from third to second generation. The complete project would in fact simplify the creation of test cases for different devices starting from one of them. Visual tests written for a specific emulator would in fact be translated in Espresso test cases with this tool and then re-translated in visual ones using the previously developed one for all the desired devices.

Bibliography

[1] Maurizio Leotta , Diego Clerissi , Filippo Ricca , and Paolo Tonella. Visual vs. dom-based web locators: An empirical study. In Sven Casteleyn, Gustavo Rossi, and Marco Winckler, editors, *Web Engineering*, pages 322–340, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08245-5.

[2] E. Alegroth, Z. Gao, R. Oliveira, and A. Memon. Conceptualization and evaluation of component-based testing unified with visual gui testing: An empirical study. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, April 2015. doi: 10.1109/ICST.2015.7102584.

[3] Emil Alégroth and Robert Feldt. On the long-term use of visual gui testing in industrial practice: a case study. *Empirical Software Engineering*, 22(6):2937–2971, Dec 2017. ISSN 1573-7616. doi: 10.1007/s10664-016-9497-6. URL `https://doi.org/10.1007/s10664-016-9497-6`.

[4] Android. Android operative system. URL `https://www.android.com/`.

[5] Haneen Anjum, Muhammad Imran Babar, Muhammad Jehanzeb, Maham Khan, Saima Iqbal, Summiyah Sultana, Zainab Shahid, Furkh Zeshan, and Shahid Bhatti. A comparative analysis of quality assurance of mobile applications using automated testing tools. *International Journal of Advanced Computer Science and Applications*, 8, 07 2017. doi: 10.14569/IJACSA.2017.080733.

[6] Suriya Asaithambi and Stan Jarzabek. Towards test case reuse: A study of redundancies in android platform test libraries. pages 49–64, 06 2013. doi: 10.1007/978-3-642-38977-1_4.

[7] AUQTUS. Eyeautomate, . URL `https://eyeautomate.com`.

[8] AUQTUS. Eyeautomate, . URL `https://eyeautomate.com/eyestudio/`.

[9] Riccardo Copola, Luca Ardito, Marco Torchiano, and Maurizio Morisio. Mobile testing: New challenges and perceived difficulties from developers of the italian industry. 2019.

[10] Riccardo Coppola, Emanuele Raffero, and Marco Torchiano. Automated mobile ui test fragility: an exploratory assessment study on android. pages 11–20, 07 2016. doi: 10.1145/2945404.2945406.

[11] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. Mobile gui testing fragility: A study on open-source android applications. *IEEE Transactions on Reliability*, 68:67–90, 2019.

[12] Google. App resources overview, . URL `https://developer.android.com/guide/topics/resources/providing-resources`.

[13] Google. Androidx overview, . URL `https://developer.android.com/jetpack/androidx`.

[14] Google. Espresso, . URL `https://developer.android.com/training/testing/espresso`.

[15] Google. Ui automator, . URL `https://developer.android.com/training/testing/ui-automator`.

[16] Mark Grechanik, Qing Xie, and Chen Fu. Experimental assessment of manual versus tool-based maintenance of gui-directed test scripts. pages 9–18, 09 2009. doi: 10.1109/ICSM.2009.5306345.

[17] Ellis Horowitz and Zafar U. Singhera. Graphical user interface testing. 2012.

[18] Ellis Horowitz and Zafar U. Singhera. Graphical user interface testing. 2012.

[19] Federico Iosue. Omninotes. URL `https://github.com/federicoiosue/Omni-Notes`.

[20] JavaParser. Javaparser github. URL `https://github.com/javaparser/javaparser/`.

[21] Ajay Jha, Deok Kim, and Woo Lee. A framework for testing android apps by reusing test cases. 03 2019. doi: 10.1109/MOBILESoft.2019.00012.

[22] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. Understanding the test automation culture of app developers. 04 2015. doi: 10.1109/ICST.2015.7102609.

[23] ligi. Passandroid. URL `https://github.com/ligi/PassAndroid`.

[24] Mario Linares-VÃąsquez, Kevin Moran, and Denys Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. pages 399–410, 09 2017. doi: 10.1109/ICSME.2017.27.

[25] Marco Torchiano Luca Ardito, Riccardo Coppola and Emil AlÃĺgroth. Towards automated translation between generations of gui-based tests for mobile devices. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, Association for Computing Machinery, Inc*, 2018.

[26] Je-Ho Park, Young Park, and Hyung Ham. Fragmentation problem in android. pages 1–2, 06 2013. ISBN 978-1-4799-0602-4. doi: 10.1109/ICISA.2013.6579465.

[27] StatCounter. Stats of mobile operating system market share worldwide. URL `https://gs.statcounter.com/os-market-share/mobile/worldwide`.

[28] Rajeev Tiwari and Noopur Goel. Reuse: reducing test effort. *ACM SIGSOFT Software Engineering Notes*, 38:1–11, 03 2013. doi: 10.1145/2439976.2439982.

[29] Mario Linares Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. How do developers test android applications? *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 613–622, 2017.