POLITECNICO DI MILANO

School of Industrial and Information Engineering

Master of Science in
Computer Science and Engineering

SCRIPTING LAYER FOR ANDROID: UPDATED AND EXPANDED SUPPORT

Master graduation thesis:
Miguel Angel Palacio Delgado – Student ID: 816355

Supervisor:
Dr. Giovanni Agosta

Academic Year 2015-2016

Acknowledgements

I would like to thank my family for all their support throughout my master's studies, without it I would not have been able to complete them. I would also like to express my gratitude to those friends that somehow helped me during my stay in Milan.

I specially thank my supervisor Giovanni Agosta for all his understanding, cooperation and availability during the development of this work.

Milan, April 2016

Abstract

The Scripting Layer for Android (SL4A) is a software suite that brings scripting languages to Android by allowing the creation, edition and execution of scripts and the use of interactive interpreters directly on Android devices. These scripts have access to many of the APIs available to traditional Android applications, with the inherent ease of scripting languages. Despite being initially developed at Google, SL4A was never an official project and it is currently not maintained. The main purpose of this work is to update the SL4A libraries to support the most recent versions of the Android operative system as well as to extend the support for missing components in the user interface library.

Lo Scripting Layer for Android (SL4A) è una suite di software che porta linguaggi di scripting ad Android, permettendo la creazione, edizione ed esecuzione di scripts e anche l'uso di interprete interattivi direttamente su dispositivi Android. Questi script hanno accesso a tante delle API disponibili ad applicazioni tradizionali Android, con la facilità intrinseca dei linguaggi di scripting. Nonsostanto il fatto che SL4A fu inizialmente sviluppata in Google, non è mai stata un progetto ufficiale e non si è più sviluppata. Lo scopo principali di questo lavoro è aggiornare le biblioteche di SL4A per supportare le ultime versioni del sistema operativo Android, così come estendere il supporto per componenti mancanti nella biblioteca dell'interfaccia utente.

Keywords

android, sl4a, java, python, scripting languages

Contents

Chapter 1 Introduction

Mobile development is very popular. Nowadays, most of the smartphones sold have Android on their entrails which means that Android developers are in demand. Nevertheless, when it comes to develop a "true" Android application, it seems like there is only one programming language of choice: Java (although it is possible to work with C and C++ by using the Android NDK[1], it requires a lot more of effort).

The election of Java as the language for building applications was a wise decision from Google. Java is a well known language, many developers already know it and do not have to learn it. Additionally, Java runs in a virtual machine so there is no need to recompile applications for every phone.

This has been a clever strategy that has helped to skyrocket Android's popularity. There is no need to own specific hardware and software in order to get started with development of Android applications. All what is needed is a computer running Linux/Windows/OSX, along with a copy of Java and the Android SDK. Moreover, Google distributes the Android Studio IDE that eases the development and includes many useful tools.

All of this is definitely great for developers. Nevertheless, as an alternative to Java for Android development we have the Scripting Layer for Android, or SL4A as is commonly abridged. SL4A allows to write applications for Android using scripting languages. And despite the fact that there is nothing wrong with Java, it may be really excessive for some applications that only need to perform a bunch of simple operations.

However, you may still ask: "why would anyone prefer SL4A over Java?", and there are several reasons to prefer SL4A. The simplest reason is that not everyone likes Java. Another reason is that with Java, an "edit – compile – run" cycle is required and it might result too cumbersome for building simple applications. "I want to use Python/Ruby/etc to develop Android applications" is a valid reason too.

The Scripting Layer for Android is an open source application that permits programs written in a variety of interpreted languages to be executed on Android. Furthermore, SL4A provides a high level API that empowers these programs to interact with the Android device, which makes it easy to perform tasks like sending SMS, fetching sensor data, and more.

The Scripting Layer for Android is suitable for several kind of tasks, including:

1 The Native Development Kit (NDK) allows developers to implements parts of their applications using C and C++. However, its use is highly discouraged by Google.

- Writing scripts to test certain functionalities available in the Android APIs, as long as those functionalities are exposed through SL4A.

- Building utilities in order to automate repetitive jobs, as well as simple tasks that do not require highly sophisticated user interfaces for user interaction.

- Rapid application development (RAD), a software development methodology that facilitates the creation of application prototypes so that developers can test the viability of an idea.

All in all, the Scripting Layer for Android is targeting those developers who are looking for either a way to write simple Android applications or to write scripts to automate tasks on their Android devices. With this purpose, not only does SL4A provide an interactive console in which you can to enter a line of code and see the results immediately, but also provides an editor in which you can write more complex routines to be executed later on. The main idea is that SL4A makes it possible to write code for Android devices in languages other than Java and doing it in an interactive way.

Unfortunately, SL4A is no longer under active development and the last meaningful contributions to the project were nearly three years ago. As a result, a great part of the codebase is outdated, leading to a plethora of functions that are either deprecated or just not supported in newer versions of the Android platform.

The main objective of this thesis is to update the project so as to support the most recent versions of Android, as well as to extend the support for missing components primarily in the UI library. Beyond bringing up to date and extending the SL4A's API, the user interface of the application itself is to be modernized too.

In order to achieve these goals, a comprehensive study of the project codebase and its architecture has to be done in order to understand how SL4A works. The most relevant parts of the codebase have to be identified so that the modifications required are carried out. Subsequently, all the modified and introduced functionalities need to be tested thoroughly with the purpose of verifying its correct operation.

This document is organized in five chapters, as follows:

Chapter 1 provides an introduction to the Scripting Layer for Android, why it is important in the Android development scene and what is this thesis about. Chapter 2 reviews the state of the art regarding alternatives to the classical Android Java development. The Scripting Layer for Android is examined with more detail in order to have a good understanding of the way in which it operates. Finally, the current status of the project is analyzed in order to know how this work needed to be directed. Chapter 3 goes meticulously throughout the work that was carried out in order to bring up to date important components of SL4A such as the user interface of the application itself, as well as the UI APIs offered by it. Chapter 4 is dedicated to show tests and their respective results performed on SL4A in order to verify the correct functionality of the application according to the work done in this thesis. Chapter 5 draws some conclusions about the work carried out, highlighting the challenges and experiences learned that can be applied to other projects hereafter. Furthermore, reflections about how the development of the Scripting Layer for Android should be directed in the future are brought up.

Chapter 2
State of the Art

The Scripting Layer for Android was created by Damon Kohler, a software engineer at Google who worked on this project for about 20 months thanks to Google's “20 percent time” policy[2]. This project was first introduced on the Google Open Source Blog in June 2009 as Android Scripting Environment (ASE) and thereafter, other developers started contributing to help it mature.

Prior to SL4A's initial release, Java was the only choice for building Android applications. The announcement of SL4A generated excitement among those developers who were yearning to try other options for Android development different to Java. Some time later and as result of Android's popularity growth, different other alternatives to develop applications for the platform began to appear.

This chapter is dedicated to give an overview to some of the alternatives to Android Java development, as well as to study in more detail how the Scripting Layer for Android works and what is the current state of the project. In following chapters, the work carried out to bring SL4A up to date will be reviewed as well as the tests performed in order to assess it.

2.1 Alternatives to Android development with Java

As it was pointed out in chapter 1, Google adopted Java as the programming language to develop Android applications due to its popularity among developers and the amount of tools already available for the Java platform. Nonetheless, Java is not a language that appeals everyone. Therefore, as the Android ecosystem expanded, many alternatives to develop Android applications emerged.

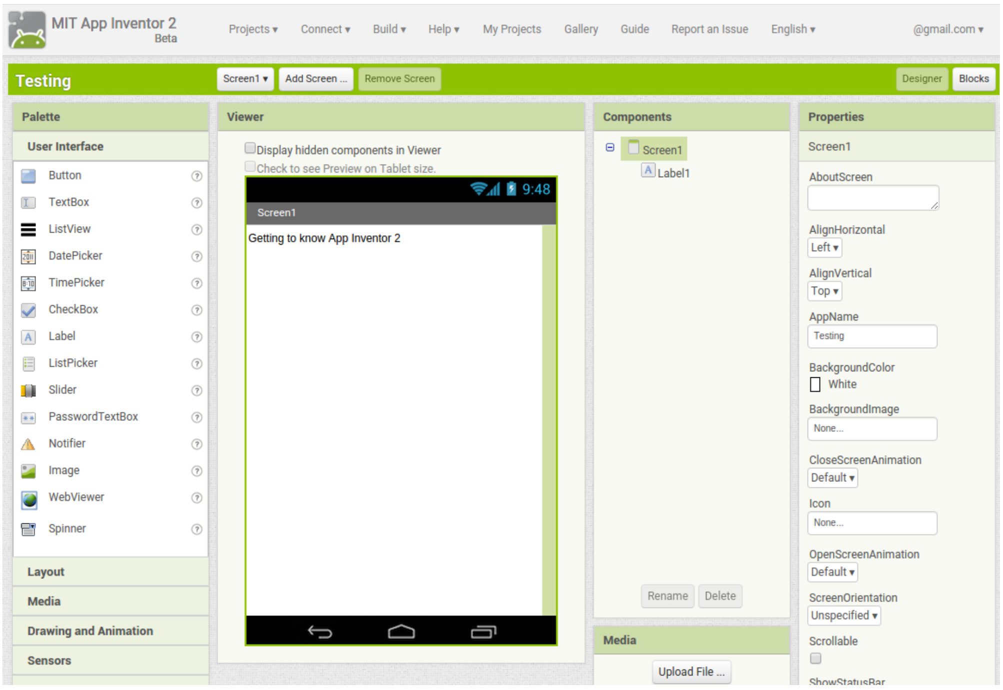
Some alternatives are remarkably good; however, they center on allowing you to create and package native applications using some otherwise unsupported language. That said, most of the alternatives are not meant to run scripts on Android, but to build whole applications with them in a more “practical” way. The succeeding subsections examine a few of these alternatives.

2.1.1 App Inventor for Android

App Inventor for Android is an open source web application aimed to people new in computer programming since it eases the way to create applications for Android. This application was originally offered by Google in December 2010, but during the second half of 2011 its maintenance was transferred to the Massachusetts Institute of Technology (MIT).

2 This policy encourages Google employees to spend about 20% of their time experimenting with ideas that might be beneficial for the company.

The application uses a graphical interface that let users *drag-and-drop* visual objects to create an application that can execute on Android devices. Figure 2.1 shows the main window of App Inventor. As we can see there is a rather good number of widgets, their functionality is limited but it is great for the objective of the project.



AboutScreen

AlignHorizontal Left ▾

AlignVertical Top ▾

AppName Testing

BackgroundColor White

BackgroundImage None...

CloseScreenAnimation Default ▾

Icon None...

OpenScreenAnimation Default ▾

ScreenOrientation Unspecified ▾

Scrollable

ShowStatusBar

Figure 2.1: App Inventor for Android

2.1.2 Kivy

Kivy is an open source Python library that facilitates the development of applications for several platforms. A developer can create an application written in Python that will run on Linux, Windows, OSX, iOS and Android. The idea of the library is to provide a tool for rapid application development.

However, since it is a tool designed to build multi-platform applications, it does not provide a direct way to access Android APIs. In this way, Kivy is nice to build entire applications using Python but it does not allow to write Python scripts and execute them on Android.

2.1.3 Qt for Android

As Kivy, Qt's main goal is to provide a tool to build multi-platform applications. Their idea is that software development must include a cross-platform user experience. Qt development is done in standard C++ and QML[3].

3 Qt Modeling Language. It is a user interface markup language. It has a JavaScript Object Notation (JSON) writing style that allows the design of user interfaces.

As from the release of version 5.1 in July 2013, Qt supports Android. Qt for Android lets the developer run QML applications on Android devices, it provides support for Android native style and allows the creation and deploy Application Packages (APK), among many other features.

2.1.4 Xamarin

Xamarin is probably one of the most advanced alternatives to Android development with Java nowadays. As Qt for Android and Kivy, it permits the development of multi-platform applications including Android. The language for development used in Xamarin is C#.

Xamarin also provides a easy way to build and package Android applications. It is thought to ease the life of C# developers so that they can build mobile applications that are compatible with Android, iOS, and Windows Phone.

2.1.5 RubyMotion

As the previous three alternatives, RubyMotion's main goal is to allow the creation of multi-platform applications for Android, iOS and OSX using Ruby. Support for the Android platform was added in December 2014.

RubyMotion supports the latest versions of Android (from version 3.2 to 6.0 at the moment of writing this thesis). It allows to call the entire set of Java APIs for these versions of the Android platform, and it is also possible to integrate 3rd-party Java libraries.

However, and in the same way as Kivy, despite the fact that Ruby is the programming language for developing, it is not possible to write Ruby scripts and execute them on Android.

2.2 The Scripting Layer for Android

In section 2.1, some of the alternatives to Android development with Java were reviewed in order to understand what makes SL4A different.

In a nutshell, the Scripting Layer for Android is a framework that enables scripting language engines which have been ported to Android, to use the Android application programming interface (API) by means of remote procedure calls (RPCs) to a server implemented as a standard Android Java application.

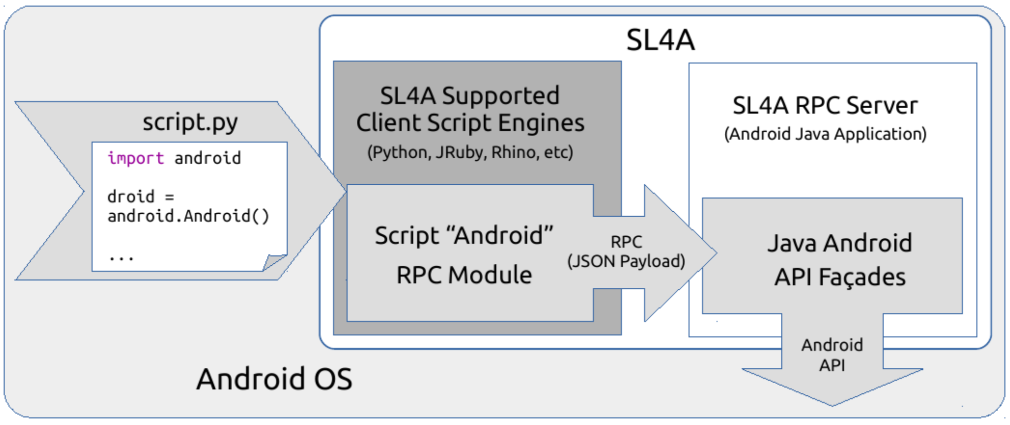
SL4A allows packaging applications too, but it is no more than an option. Having this as an option rather than an end, is something advantageous since in fact most scripts are not published on Google Play anyhow. With the Scripting Layer, a developer can start writing and executing scripts for the supported interpreted languages immediately, like she would do on any other system.

The Scripting Layer for Android is an Android application itself. It was designed and built around the Android OS. Thus, it requires Android to be useful but enables a much closer integration with the operative system.

2.2.1 SL4A architecture

The Scripting Layer for Android exhibits a set of Android APIs to be used by its clients. With the purpose of achieving this, a special module is implemented for each of the supported scripting languages. This module is in charge of arranging RPCs and their responses to and from a RPC server that is implemented as an Android Java application. Due to its nature (i.e., being an standard Android application), the RPC server has direct access to the Android API. The server behaves as a remote proxy using a set of façades that encapsulate and expose selected Android functionalities.

In order to support a scripting language on Android, its respective script engine needs to be ported to the platform in their purest form avoiding any source code changes. This suggests that the script engine is completely unaware that it is running on the Android OS. The way in which a scripting engine gets access to the Android API consists in using the special module previously mentioned. This module is generally implemented in the scripting language itself, and it can access the Android API through the remote SLA4 RPC server.


SL4A
script.py
SL4A Supported Client Script Engines
(Python, JRuby, Rhino, etc)
SL4A RPC Server
(Android Java Application)
Script "Android" RPC Module
RPC (JSON Payload)
Java Android API Façades
Android API
Android OS

```
import android

droid =
android.Android()

...
```

Figure 2.2: SL4A architecture simplified

In other words, any ported script engine runs in a self-contained process and it can access the Android API through bidirectional JSON-RPC[4] via a set of API façades. These façades have full access to the Android platform API and hence they serve as a sort of remote API proxy for the script engines. Figure 2.2 shows a simplified version of the SL4A architecture that summarizes the general idea.

4 It is a lightweight remote procedure call protocol encoded in JSON. For more info see: http://json-rpc.org/wiki/specification.

2.2.2 Understanding the special module

As it was stated, each supported script engine must implement a special module (from now on, the android module). This module works as a “wrapper” that provides an “Android object”. A user can invoke any of the methods provided by the SL4A API through this object and it is seen as a normal method in the respective scripting language.

For instance, if a user wants to show a toast notification on Android through SL4A using Python, she could write the following script:

```python
import android
droid = android.Android()
droid.makeToast(“Hello SL4A!”)
```

This is a very simple script but serves to illustrate the idea of the android module. In the first line the android module is made available in the script's namespace, whilst in the second line it is possible to see how it provides the Android object mentioned. The third line invokes a function of the Android object called `makeToast` that expects a string that will be displayed on the device as a toast notification.

Naively, one might assume that each of the methods in the SL4A API has to be implemented individually in the android module of each script engine, but that would be extremely inefficient and bitterly unmaintainable. Let us take a look at the android module implementation for the Python language:

```python
__author__ = 'Damon Kohler <damonkohler@gmail.com>'

import collections
import json
import os
import socket
import sys

PORT = os.environ.get('AP_PORT')
HOST = os.environ.get('AP_HOST')
HANDSHAKE = os.environ.get('AP_HANDSHAKE')
Result = collections.namedtuple('Result', 'id,result,error')


class Android(object):

def __init__(self, addr=None):
   if addr is None:
      addr = HOST, PORT
   self.conn = socket.create_connection(addr)
   self.client = self.conn.makefile()
```

```
22     self.id = 0
23     if HANDSHAKE is not None:
24        self._authenticate(HANDSHAKE)
25
26  def _rpc(self, method, *args):
27     data = {'id': self.id,
28             'method': method,
29            'params': args}
30     request = json.dumps(data)
31     self.client.write(request + '\n')
32     self.client.flush()
33     response = self.client.readline()
34     self.id += 1
35     result = json.loads(response)
36     if result['error'] is not None:
37        print result['error']
38     # namedtuple doesn't work with unicode keys.
39     return Result(id=result['id'], result=result['result'],
40                   error=result['error'], )
41
42  def __getattr__(self, name):
43     def rpc_call(*args):
44        return self._rpc(name, *args)
45     return rpc_call
```

As it can be seen, there is no trace of a `makeToast` method nor attribute in the Android class (which the one that ultimately provides the Android object). So why does it work? Well, Python offers a special built-in method called `__getattr__` that is called when an attribute lookup has not found the invoked attribute in the usual places (i.e., it is not an instance attribute nor it is found in the class tree).

Therefore, when `makeToast` (or any other method offered by the SL4A API) is called and it is not found, `__getattr__` is called. The `__getattr__` method receives the name of the method invoked (line 42), and along with any extra attribute (i.e.,, the parameters required by the method in particular) it marshals a RPC that is then sent to the SL4A server (lines 26-35). Afterwards, the server sends back the result of the procedure and that result is handled (lines 36-40).

This implementation allows any method to be invoked. If the called method has a corresponding procedure in the server and the parameters passed along with it are correct, the corresponding procedure is executed on the server and its result is sent back to the client script engine. If there is something wrong a response explaining the error is sent back. This android module is implemented similarly for each script engine, making the server agnostic to its clients.

2.2.3 Supported languages

One of the most interesting features of SL4A is the amount of scripting languages that it supports. Up to release 6, the languages that can be downloaded from SL4A are: BeanShell, JavaScript, Lua, PHP, Perl, Python and Ruby (the project's README file states that SL4A also supports Tcl, but it cannot be downloaded from SL4A directly). The following subsections will talk about these languages and their support on SL4A along with a simple snippet of code to give a general idea of how to use each of them.

2.2.3.1 BeanShell

BeanShell is a Java source interpreter with object scripting language features. It is written in Java, and it can execute standard Java syntax dynamically. It is very useful for interactive Java experimentation and debugging. Scripting Java allows developers to do rapid prototyping, rules engines, testing, dynamic deployment, etc. The lastest version of BeanShell supported by SL4A is the 2.0b4.

```
source(“/sdcard/com.googlecode.bshforandroid/extras/bsh/android.bsh”);
droid = Android();
droid.call(“makeToast”, “Hello SL4A”);
```

2.3.3.2 JavaScript

JavaScript is supported through Rhino, a JavaScript engine fully written in Java. One interest advantage of having a JavaScript interpreter is that if a developer wants to build a custom interface using HTML and JavaScript, she could prototype the JavaScript part and test it with the Rhino interpreter. The latest supported version of the Rhino JavaScript interpreter is the 1.7R2.

```
load(“/sdcard/com.googlecode.rhinoforandroid/extras/rhino/android.js”);
var droid = new Android();
droid.makeToast(“Hello SL4A!”);
```

2.3.3.3 Lua

By its description, Lua is a powerful, fast, lightweight, embeddable scripting language. Thanks to its embeddable features, it is easy to extend programs written in other languages using Lua. And since Lua is small, it is great for running it on mobile devices. All of this fits perfectly into SL4A. The latest supported version of Lua on SL4A is the 5.1.4.

```
require “android”
android.makeToast(“Hello SL4A!”)
```

2.3.3.4 PHP

PHP is one of the most used general-purpose scripting languages to create dynamic web pages. Given its popularity, supporting it on SL4A enables a countless number of PHP developers to start building applications on Android. Version 5.3.3 is the latest installable version of PHP on SL4A.

```php
<?php
require_once(“Android.php”);
$droid = new Android();
$droid->makeToast(“Hello SL4A!”); ?>
```

2.3.3.5 Perl

Perl is probably the oldest among the supported languages. Perl is a general-purpose language that was originally developed for text manipulation, but it is now used for a wide variety of tasks including web development, GUI development and more. This language is aimed to be practical rather than beautiful. Up to release 6.1.1 of SL4A, Perl 5.10.1 is the version supported.

```perl
use Android;
my $a = Android->new();
$a->makeToast(“Hello SL4A!”);
```

2.3.3.6 Python

Python (along with BeanShell) is undoubtedly the most popular language to build applications using SL4A, and for this reason it is better supported. Python is very popular since it is thought to be powerful and fast, but also friendly and easy to learn. The last supported version of Python is 2.6.2. Section 2.2.2 shows the use of Python and how the android module is implemented for its scripting engine.

2.3.3.7 Ruby

The Ruby programming language is intended to be simple and productive. Its syntax is elegant, which makes it clear to read and easy to write. It is supported on SL4A by means of JRuby, an implementation of the Ruby language atop the Java Virtual Machine. JRuby is written mostly in Java. The latest supported version of Ruby through JRuby is the 1.8.7p249.

```ruby
droid = Android.new
droid.makeToast “Hello SL4A!”
```

2.2.4 Current state of the project

As it was previously stated, SL4A development was carried out by Damon Kohler and other Google employees for a couple of years. Nowadays, none of them is currently working on the project and the last meaningful commit was more than three years ago.

SL4A was originally hosted on Google Code, but due to its shutdown[5] the project was migrated to Github in June 2015. SL4A new repository on Github remains untouched and Damon Kohler's last commit implies that he will not be accepting any pull requests. This commit suggests filling new issues and making pull requests to an active fork of development.

The version 6 of SL4A was the last release that could be downloaded from the old Google Code repository. This version can run on Android Lollipop without problems. However, there are issues with the majority of the script engines that are downloadable from the application. The Lua, Perl, Python and Rhino script engine do not work. Moreover, the PHP script engine is not even listed as a recognized interpreter by the application. The only script engines that work are BeanShell and JRuby.

There are several problems with the graphical user interface as well. One of these problems is that the interface itself is at least a old as Android Gingerbread, causing some issues in newer Android devices (e.g, this interface was thought for phones that have a physical "menu" button. New devices lack of this button and that impacts the usability of the application).

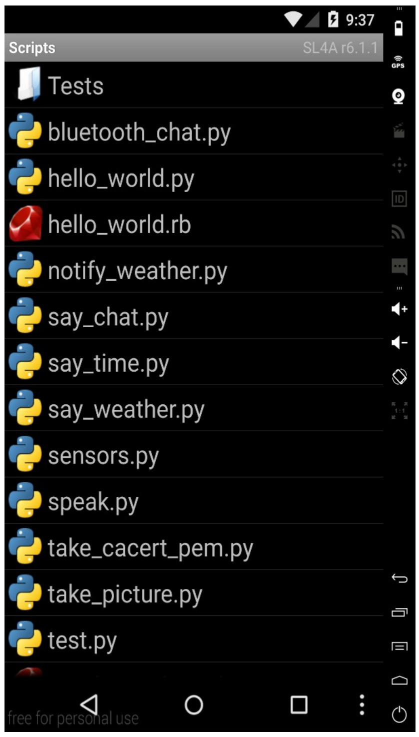


Figure 2.3: SL4A r6 GUI

As the former developers claimed, the project is still in alpha state. Hence, in addition to the aforementioned issues there are many other unsolved bugs. However, these do not interfere with

5 Bidding farewell to Google Code: http://google-opensource.blogspot.it/2015/03/farewell-to-google-code.html

the creation and execution of scripts (as long as the corresponding script engine works). As it has been stated, the idea of this work is to update SL4A so that it supports new features of the Android platform that are missing due to the lack of development. The next chapter is entirely dedicated to review the work carried out in this thesis.

Chapter 3
Modernizing SL4A

In the previous chapter some alternatives to the classic Android development with Java were reviewed, and it was also mentioned how these alternatives differed to SL4A. Additionally, the Scripting Layer for Android architecture was analyzed in order to understand its operation, and the current state of the application was outlined. In this chapter, the work carried out to bring SL4A up to date is examined at great detail.

Due to the shutdown of Google Code where the Scripting Layer for Android was hosted and the subsequent migration toward Github, this work started by reviewing the original repository of the project that is held by Damon Kohler.

Since the main goal of this thesis was to bring up to date the Scripting Layer for Android, the project targeted the latest Android API levels. At the moment of beginning this work (July 2015), the last stable release of the Android platform was Lollipop (version 5.1, API level 22). In spite of the fact that Android Marshmallow (version 6.0, API 23) was on the way, it was still a preview version and things could change in the API so the decision made was to stick to Android Lollipop.

The first phase of this work consisted in understanding the structure of the project, how the application worked and what needed to be modified in order to achieve the objectives. This phase also included an upgrade of the project to target the API level 22. The second phase involved a renewal of the graphical user interface since it was really outdated, hindering the use of the application. The third and last phase was intended to update the SL4A API focusing mainly on the UI functionalities. During the execution of these phases all the deprecated code seen along the way was either removed or replaced by newer and supported code.

3.1 Start up

This work began by doing a short research about the Scripting Layer for Android. Every post that talked about SL4A was rather dated and they all linked to the previous Google Code repository. This repository is closed but it provides a link toward the new repository on Github[6]. As it was stated in the previous chapter, the ownership of the new SL4A repository is held by Damon Kohler; however, he stated on the project's README file that the project is not under active development and that new issues should be filed against an active fork. This also implies that he will not be accepting pull requests and that they should be made against an active fork too.

6 https://github.com/damonkohler/sl4a

Fortunately, repositories on Github offer several graphs to measure their activity and the activity of their forks. Taking a look on the network graph, one could see that there was a fork under active development and that fork itself had been forked several times. As it can be seen in figure 3.1, this active fork belongs to the Github user kuri65536[7].

This user had already done a very good job migrating the project to Android Studio and they even included instructions about how to import it. Since the project had been inactive for three years, it was still configured to run on Eclipse. And despite the fact that Google provides an excellent tutorial to help developers with the migration of projects from Eclipse to Android Studio, the Scripting Layer for Android is a very complex project that have many modules and this makes the migration more traumatic.

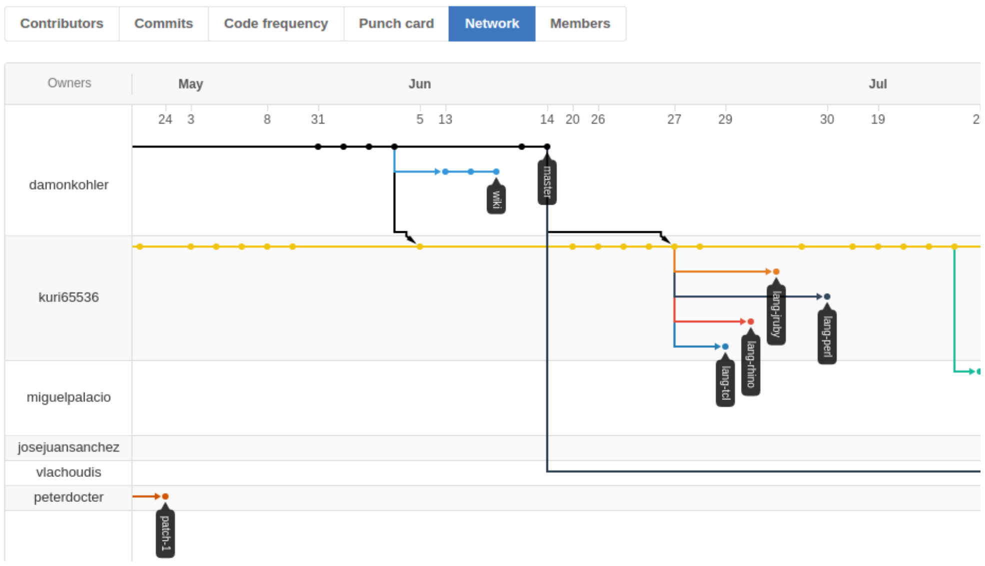


Figure 3.1: Damon Kohler's SL4A network graph

For this reason and because it was the most active fork, the decision was to fork it and start this work from there[8]. As it was stated before, kuri65536 did a great job making ready the project for Android Studio and the import to it was smooth. kuri65536 had already tagged the project as release 6.1.1; however, apart from the migration to Android Studio nothing had really changed compared to release 6.

The issues with the interpreters mentioned in chapter 2 were still present. Of the interpreters downloadable from SL4A, the only ones that worked were the BeanShell and JRuby interpreters. Nevertheless, there was a change made by kuri65536 in the structure of the project. Formerly, the Scripting Layer for Android held the script engines modules inside the project's folder, and now they are gone.

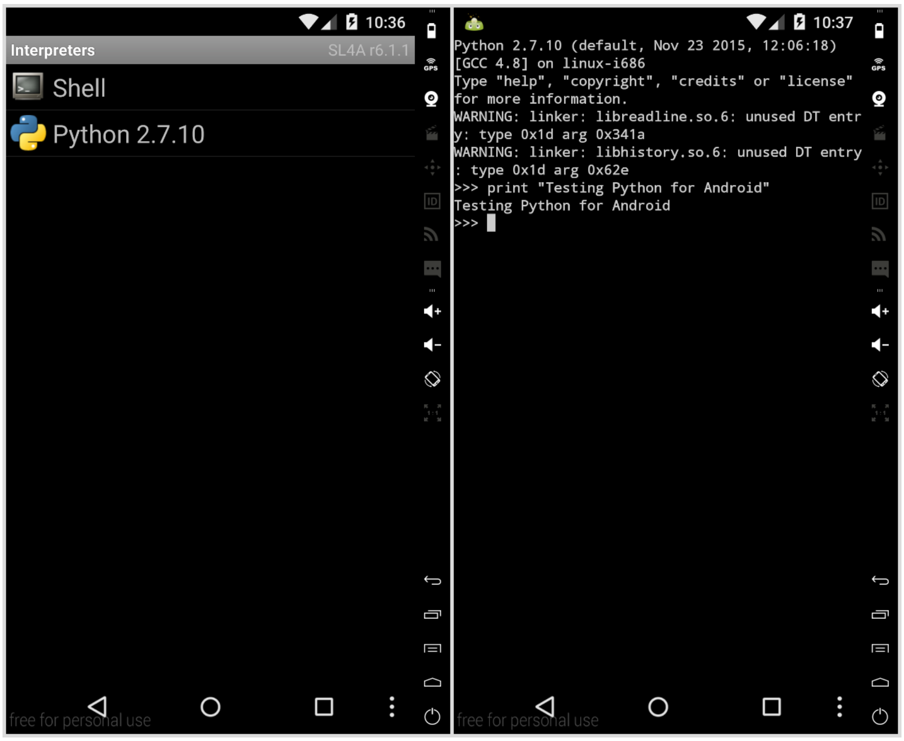
Each script engine was a separated Android application that did not intervene with the operation

7 https://github.com/kuri65536/sl4a

8 Incidentally, the repository of this work can be found at https://github.com/miguelpalacio/sl4a.

of SL4A itself. So they were removed from project's root. They are meant to be completely separated projects, and in fact kuri65536 created another repository called python-for-android[9] that takes care of the Python script engine. kuri65536's version of Python for Android cannot be downloaded from SL4A yet, but it works correctly on it. The Python version that this project provides is 2.7.10; however, the support to Python 3.x seems to be on the way.

Besides this change and several bug fixes, kuri65536's fork was nearly the same as the original SL4A provided by Damon Kohler. In order to bring SL4A up to date, it was necessary to do a renewal of the graphical user interface and make use of the new functionalities provided by the Android OS and that had not been yet adopted on SL4A. Section 3.2 is devoted to review this in great detail.



```
Python 2.7.10 (default, Nov 23 2015, 12:06:18)
[GCC 4.8] on linux-i686
Type "help", "copyright", "credits" or "license"
for more information.
WARNING: linker: libreadline.so.6: unused DT entr
y: type 0x1d arg 0x341a
WARNING: linker: libhistory.so.6: unused DT entry
: type 0x1d arg 0x62e
>>> print "Testing Python for Android"
Testing Python for Android
>>>
```

Figure 3.2: Python for Android on SL4A

As it was mentioned before, SL4A is a rather complex project with a huge codebase. One of the reasons is that the application itself makes use of many modules that are part of the project. Each of these modules counts as a sort of “independent” Android application, so the project is quite heavy. Figure 3.3 shows the project structure with the modules it has, being the main module the one highlighted.

Note: For future reference, when talking about a specific Java file or XML resource, a path of the following format will be used:

`Module_Name/Path_To_File_Directory/File`

9 https://github.com/kuri65536/python-for-android

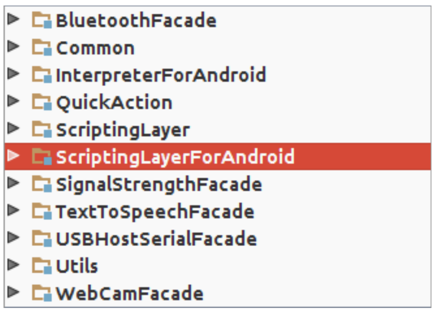

BluetoothFacade
Common
InterpreterForAndroid
QuickAction
ScriptingLayer
ScriptingLayerForAndroid
SignalStrengthFacade
TextToSpeechFacade
USBHostSerialFacade
Utils
WebCamFacade

Figure 3.3: SL4A project structure

3.2 Catching up on Android Lollipop

With the purpose of bringing the Scripting Layer for Android up to date, it was necessary to target the newest versions of Android. As it was mentioned before, the version chosen as a basis for this part was Android Lollipop. Although by July 2015 Android Marshmallow was already available for developers, it was only offered as a preview so that developers could get used to it. These preview versions have been known by having changeable API so taking Marshmallow as the target version was not advisable.

The following subsections provide a thorough explanation of the work done in order to get SL4A fully compatible with Android Lollipop.

3.2.1 Targeting new API levels

The first thing that needs to be addressed is the definition of API level. On the Android Developers site we find the following definition: API level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform. From this definition it can be inferred that for each version of the Android platform corresponds only one API level. The newer the version of the Android platform, the larger the integer that represents its matching API level.

All applications developed for Android can use this framework API to interact with the Android system. Applications must specify which API levels they cover and this determines the range of devices they can support. The larger the number of API levels covered the larger the number of devices supported, but this also makes the maintenance of an application more difficult.

The framework API is updated thinking on keeping backward compatibility with older versions of this framework. In this manner, if a device runs a version of the Android platform that delivers an API level that is newer than any of the API levels covered by an application, this device can execute the application. This is clearly the case of SL4A running on Android Lollipop without much inconvenience.

The number of API levels that an Android application covers is defined by setting two values:

minSdkVersion and *targetSdkVersion*. As it can be deduced, *minSdkVersion* specifies the minimum API level on which the application can run; whilst *targetSdkVersion* specifies the API level on which the application is designed to run. *targetSdkVersion* makes it possible to use specific functionalities that are available on the target API level rather than being restricted to what it is offered by the minimum API level.

Both the *minSdkVersion* and *targetSdkVersion* on SL4A were set to 4. This means that the application is meant to fully support Android Donut. However, Android Donut was released in 2009 and there has been a lot of development for the Android platform since then. Naturally, after nearly 6 years there is a plethora of missing functionalities on SL4A that more recent Android applications have.

The decision made in order to update the project was to set *targetSdkVersion* to 22 and *minSdkVersion* to 15. Thus, the oldest devices that can run SL4A are the ones with Android Ice Cream Sandwich. The choice of *targetSdkVersion* has already been explained. The choice of *minSdkVersion* was made after analyzing the Android Dashboards statistics[10].

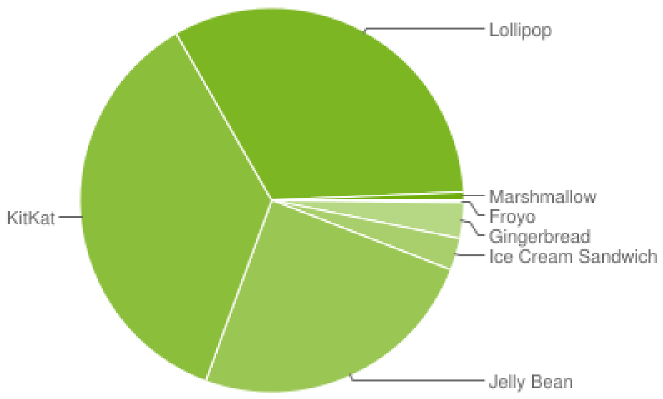


Figure 3.4: Android platform versions distribution

The Dashboards is a tool that provides information about the market share among Android devices. Not only does it supply information about the versions of the Android platform that they run, but also other useful information like their screen sizes and the Open GL version they support. These statistics are obtained by Google on a weekly basis by studying the devices that visit Google Play.

At the moment of deciding the oldest version of the Android platform to be supported, the idea was to support as many devices as possible. Initially, the idea was to set *minSdkVersion* to 10. The problem is that the lower the *minSdkVersion* supported, the more the deprecated code that needs to be handled. For this reason, the criterion used was to support at least the 90% of devices.

By August 2015, setting the minimum API level to 15 gave support to more than 90% of

10 https://developer.android.com/about/dashboards/index.html

devices. By January 2016 the same configuration gives support to more than 95% of the devices. As a side note, from figure 3.3 it is possible to see that devices with Android Marshmallow are starting to appear in the dashboards; nevertheless, their participation is still less than 1% of the total distribution.

Changing *targetSdkVersion* to 22 did not throw any errors upon compiling the project. However, there were a plethora of new warnings due to code that was now recognized as deprecated. The real issue emerged once SL4A was executed: it always crashed. The problem was located at `ScriptingLayerForAndroid/src/.../activity/CustomizeWindow.java`.

`CustomizeWindow.java` is in charge of setting a custom window title. It worked fine with API level 4, but on Android Lollipop the way in which the window title is handled has changed. The problem was that the default theme that Android applications use now does not allow to use custom titles; hence, when a custom title was requested it raised an exception making the application crash. The following lines on `CustomizeWindow.java` were ruled out:

```
activity.requestWindowFeature(Window.FEATURE_CUSTOM_TITLE);

activity.getWindow().
          setFeatureInt(Window.FEATURE_CUSTOM_TITLE, R.layout.title);
((TextView) activity.findViewById(R.id.left_text)).setText(title);
    ((TextView) activity.findViewById(R.id.right_text)).setText("SL4A r" +
        Version.getVersion(activity));
```

Additionally, in order to set the window title the following line was added:

```
activity.setTitle("SL4A - " + title);
```

The result can be observed in figure 3.5. It is possible to see now how the application begins to look more modern.

3.2.2 Interface renovation: Material Design

With the introduction of Android Lollipop a great deal of new features were made available for developers. Among the most important was Material Design[11], an expanded UI toolkit for integrating the new design patterns recommended by Google easily in applications. At the moment of carrying out this work, Material Design was the de facto standard for implementing Android applications UI.

3.2.2.1 From activities to fragments

Fragments were introduced with Android Honeycomb (API level 11) and they were designed thinking in reusable user interfaces, focusing on the support of large screens such as tablets.

11 http://www.google.com/design/spec/material-design/introduction.html

Since tablets screens are much larger than those of smartphones, there is more space to integrate different UI components.

This opens the possibility to divide an activity's layout into several fragments, and as each fragment is independent from each other (each has its own lifecycle) the result are activities with much greater flexibility.

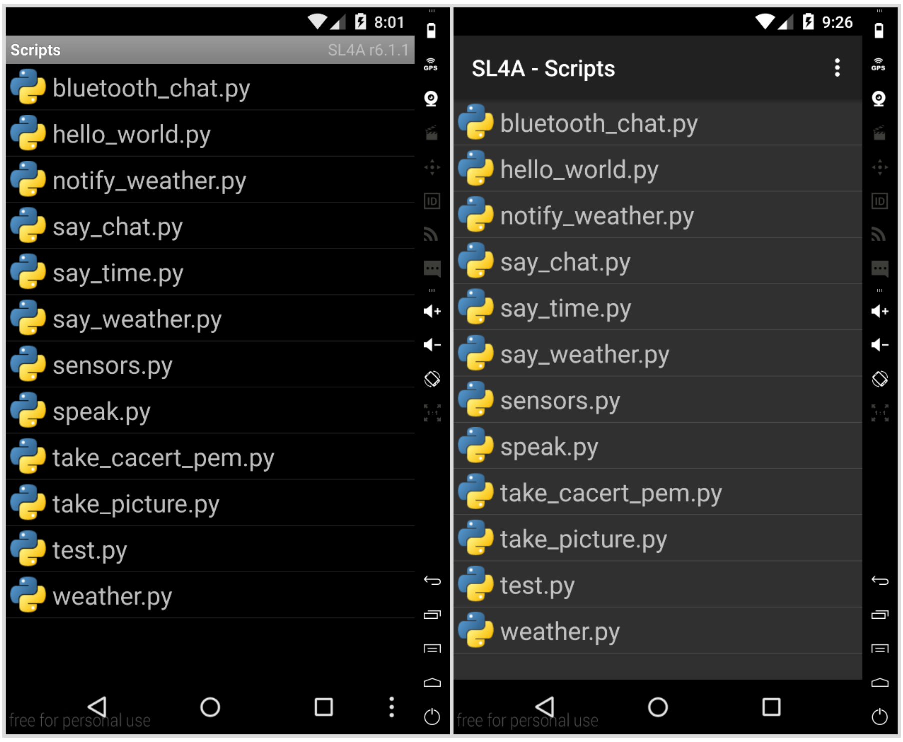


Figure 3.5: Target API 4 vs. Target API 22

In spite of the fact that it is possible to achieve some of the same things that fragments do using activities and layouts, fragments are now bound to the Android API. Firstly, the navigation through the application is improved by using them since transitions between fragments are smoother than transition between activities. Secondly, new widgets such as tabs and navigation drawer require them. Thirdly, fragments ease portability across different devices (tablets or phones). Even if an application is aimed at smartphones only, fragments should be used having portability in mind. Last but not least, development is easier if the API is used the way it is meant, and not using fragments is going against the Android API ideals.

However, fragments are not a silver bullet and not all activities should be replaced by them. In many cases, migrating an activity to a fragment would only increase the amount of code and Java classes without giving any real benefit to the application. For this reason, only a few activities on SL4A were migrated to fragments.

`ScriptingLayerForAndroid/src/.../activity/Preferences.java` had to be modified since the recommendation from the Android developer site is to use fragments to show an application's settings whenever the case. Their reason is that using fragments improves navigability and take advantage of bigger screens, as it can be seen in figure 3.6. Moreover, a large number of the methods of `PreferenceActivity` (`Preference`'s former superclass) is now deprecated so it was of utter importance to migrate it.

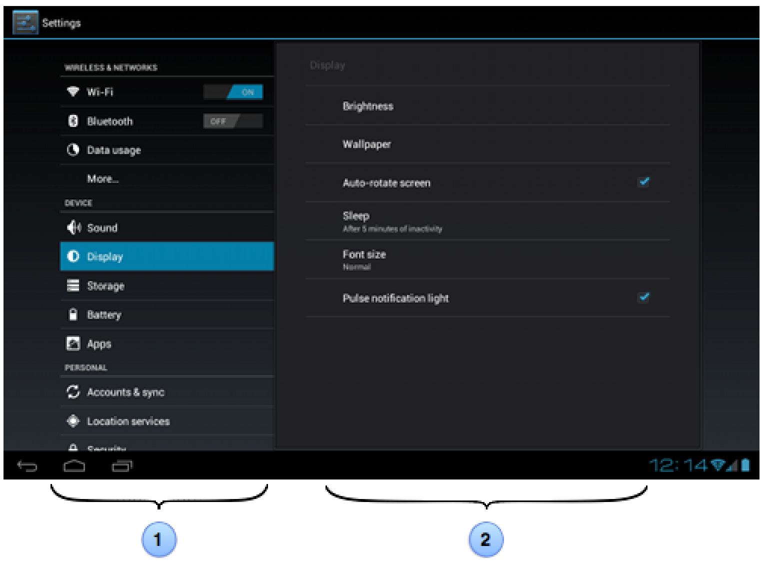


Figure 3.6: Settings screen using fragments. Subgroups of settings can be placed on different fragments in order to improve navigability (*image taken from the Android developer's site*)

The idea was to replace `PreferenceActivity` for a standard activity class that holds `PreferenceFragments`. Modifying `Preferences` to achieve that was a rather simple task. `Preferences` superclass `PreferenceActivity` was replaced by `AppCompatActivity` (a base class for activities that use recent features of the Android OS while giving support to older devices). `PreferenceList` (a `PreferenceFragment` subclass) was added to hold the content of the view that `Preferences` held. Additionally, a `Toolbar` widget was added to the modified `Preferences` activity in order to follow the Material Design patterns and have more consistency with the rest of the application.

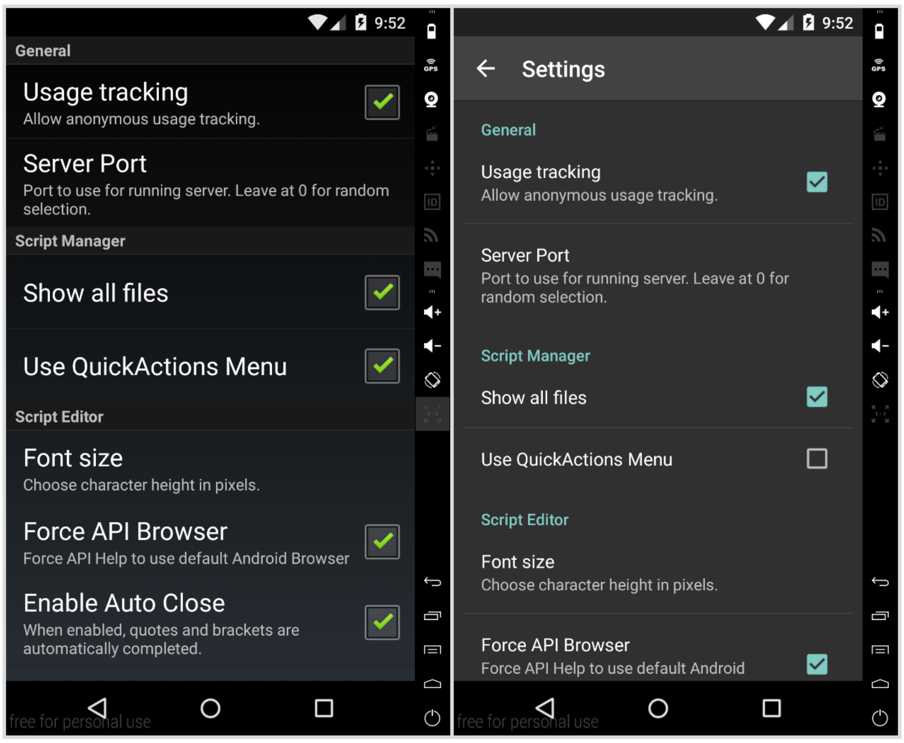
Following the project standards, a package was created so that it holds all the fragments. Its path is `ScriptingLayerForAndroid/src/.../android_scripting/fragment/`. Figure 3.7 shows the modified `Preferences` activity. Note that the application's theme was changed to the one provided by the AppCompat library for Android Lollipop[12] with some customizations.

The other activities that were replaced by fragments were: `ScriptManager`, `InterpreterManager`, `TriggerManager` and `LogcatViewer`. All of these Java classes are now found at `ScriptingLayerForAndroid/src/.../fragment/`. These classes provide the main screens through which the users interact with the application. In order to switch from one activity to other the user had two options: performing a swipe gesture (from left to right and

12 http://android-developers.blogspot.com.co/2014/10/appcompat-v21-material-design-for-pre.html

vice versa) or popping up the menu and from there going to the wanted screen.

Google's guideline to increase the navigability of applications is to use a navigation drawer widget or tabs. For this project, a navigation drawer was deemed as the better choice. For this widget to work as expected, the transition between screens must be smooth and neat. In order to achieve this smoothness, it is necessary that the different screens are implemented as fragments.


9:52

General

Usage tracking ✓
Allow anonymous usage tracking.

Server Port
Port to use for running server. Leave at 0 for random selection.

Script Manager

Show all files ✓

Use QuickActions Menu ✓

Script Editor

Font size
Choose character height in pixels.

Force API Browser ✓
Force API Help to use default Android Browser

Enable Auto Close ✓
When enabled, quotes and brackets are automatically completed.

9:52

Settings

General

Usage tracking ✓
Allow anonymous usage tracking.

Server Port
Port to use for running server. Leave at 0 for random selection.

Script Manager

Show all files ✓

Use QuickActions Menu ☐

Script Editor

Font size
Choose character height in pixels.

Force API Browser ✓
Force API Help to use default Android

Figure 3.7: Old `Preferences` activity vs. New `Preferences` activity using a `PreferenceFragment` and the toolbar widget.

To implement the navigation drawer, the four previously mentioned Java classes were changed to fragments. An activity named MainActivity was created with the purpose of holding the navigation drawer, a `Toolbar` widget and a container which in turn holds the fragments. Figure 3.8 shows one of the previous ways to access the different main screens as well as the implementation of the navigation drawer. The navigation drawer is accessible from any of the four main screens, and also provides a way to access the Settings screen (i.e., the `Preferences` activity).

Apart from enabling the use of the navigation drawer, the migration of these activities to fragments did not change their aesthetics at all. All of these four activities were in fact subclassing `ListActivity`, and after the migration they subclassed `ListFragment`. These classes require the use of a `ListView` object bound to a data source. Figure 3.5 shows a list of scripts in the `ScriptManager` screen, the list of scripts is displayed by using the mentioned `ListView` object.

Nevertheless, due to several limitations of the `ListView` class, a new type of view called `RecyclerView` was released on Android Lollipop with the purpose of replacing the old `ListView` and provide many more functionalities. The following section will give a better insight about `RecyclerView` and how it was used on this project.

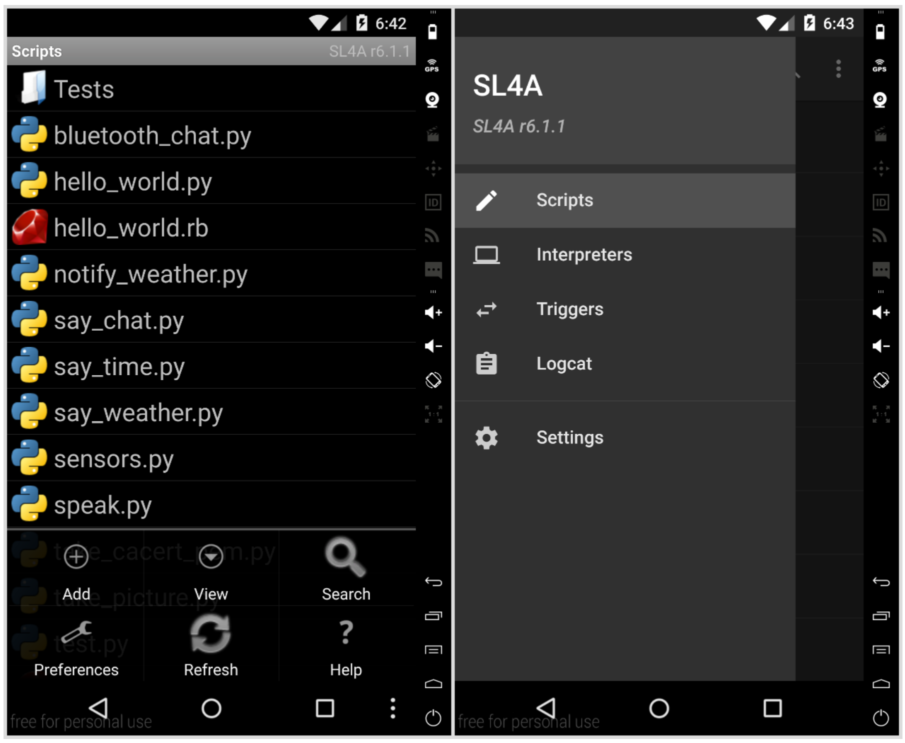


Figure 3.8: Old menu to access different screens (after clicking on "View" a submenu listing the screens is displayed) vs. New navigation drawer

3.2.2.2 Replacing ListView with RecyclerView

`RecyclerView` was introduced with Android Lollipop in the support-v7 library[13] (enabling its use on all versions of Android from API level 7), and it allows to display a collection of items in an arbitrary disposition. `RecyclerView` goes in the same direction of `ListView` but it is much more flexible. For instance, with `RecyclerView` is possible to have elements of different size whereas with `ListView` all the elements are constrained to have the same size. Another example can be seen in figure 3.8, the whole content of the navigation drawer is in fact a single `RecyclerView` element that holds a header, a bunch of entries and a divider.

However, with `RecyclerView's` flexibility also comes a new layer of complexity. With `ListView` and `GridView` an object implementing an `Adapter` interface is used and it is in charge of pulling the content from a source and transforming it adequately so that it is placed into the list. The implementation of this `Adapter` interface was rather short and simple. With `RecyclerView` a whole adapter class must be implemented, and depending on how you wish to position the elements the implementation changes substantially.

13 https://developer.android.com/tools/support-library/features.html#v7

As it was mentioned in the previous section, in SL4A many of the activities were implemented by subclassing `ListActivity`, and most of the main screens were migrated to `ListFragment`. The purpose of `ListActivity` is to host a `ListView` so that it can show a set of items, and the purpose of `ListFragment` is not different. After the implementation of the navigation drawer, the four main screens were subclassing `ListFragment`, but by using `RecyclerView` this class is of no use so they now subclass `Fragment`.

Most activities (i.e., screens) that were subclassing `ListActivity` were modified to use `RecyclerView` and now subclass `AppCompatActivity`. In total, six screens (either activities or fragments) were modified to use a `RecyclerView` widget. For them, it was necessary to implement five different adapters due to the nature of the elements of each list. Table 3.1 shows which screens were modified to replace `ListView` with `RecyclerView`, their previous and current superclass and the adapter they use to bind the data with the views within the lists. All the classes that implement the adapters for the `RecyclerView` widgets can be found in the package `ScriptingLayerForAndroid/src/.../custom_component/item_lists/`.

Screen (activity/fragment)	Previous superclass	Current superclass	Adapter
`ScriptManager`	`ListFragment`	`Fragment`	`ScriptListAdapter`
`InterpreterManager`	`ListFragment`	`Fragment`	`InterpreterListAdapter`
`TriggerManager`	`ListFragment`	`Fragment`	`TriggerListAdapter`
`LogcatViewer`	`ListFragment`	`Fragment`	`LogcatListAdapter`
`ScriptPicker`	`ListActivity`	`AppCompatActivity`	`ScriptListAdapter`
`ScriptProcessMonitor`	`ListActivity`	`AppCompatActivity`	`ScriptMonitorListAdapter`

Table 3.1: Screens were `RecyclerView` was used to replace `ListView`.

As it can be seen in table 3.1, the reuse of the adapters is minimal. The only adapter that is reused is `ScriptListAdapter`, and the reason for this is that both screens (`ScriptManager` and `ScriptPicker`) use exactly the same layout and data collection. The problem with the implementation of these adapters is that even the slightest difference between two lists, can make it be very difficult to use the same adapter for both. There are ways to use the the same adapter for all the lists (or collections of objects) that need to be displayed but the result would be a huge and very hard to maintain piece of code.

All the adapters listed in table 3.1 inherit from the `RecyclerView.Adapter` abstract class. This class provides all the methods that need to be overwritten so that Android OS handles the binding between a specific data set to views that are displayed within a `RecyclerView` widget. The following chunks of code are used to explain how the adapters are implemented. The `RecyclerView` used to implement the content of the navigation drawer shown in figure 3.8 will be used as a example.

```
public static class ViewHolder extends RecyclerView.ViewHolder { … }
```

This class is implemented inside of the adapter and serves to bind the data with the views inside a `RecyclerView`. The same `ViewHolder` can be used for different view types. Its job is to bind data to `TextViews`, `ImageViews`, etc. For instance, the navigation drawer header's data is bound to two `TextViews`, whilst the list items' data is bound to an `ImageView` for the icon and a `TextView` to display the entry's name.

```
public void onCreateViewHolder(ViewGroup parent, int viewType) { … }
```

This method inflates a layout depending on `viewType`. For the case of the navigation drawer, it inflates three different type of layouts: one for the header, one for each item of the list, and a very simple one to be used as a divider (the line separating Logcat from Settings). Once a layout is inflated, a new `ViewHolder` is created.

```
public void onBindViewHolder(ViewHolder holder, int position) { … }
```

This methods fills the `ViewHolder` with data from item at position `position` of the collection of data that wants to be represented using the `RecyclerView`.

```
public int getItemCount() { … }
```

This method returns the number of views that will be present in the whole `RecyclerView`. The reason to implement this method is that the number of views in the `RecyclerView` not necessarily match the number of items in a collection of data. For the navigation drawer, the number of entries of the list cannot be used as the number of views in the corresponding `RecyclerView` since no data is assigned to the divider, and the data used for the header comes from a different source as well.

```
public int getItemViewType(int position) { … }
```

This method is implemented to return a view type depending on the position of the item within the `RecyclerView`. Here, the the order in which the items are arranged within the `RecyclerView` is defined. The result is then used by `onCreateViewHolder`.

In spite of the fact that the navigation drawer was used to illustrate how a `RecyclerView`'s adapter works, that specific adapter used for the navigation drawer has some particularities. It does not extend `RecyclerView.Adapter` but `SelectableAdapter`. The latter allows to keep a register of which items were selected. Figure 3.8 shows that the “Scripts” entry was just selected since it appears highlighted.

Figures 3.9 to 3.13 show the results obtained after implementing the adapters and carrying out the necessary modifications in each Java class corresponding to the screens in which

`RecyclerView` replaced `ListView`. The lists were implemented so that they matched the previous design. Also note that some of the icons for the lists were renewed to match more current trends. Also note that as stated in section 3.2.2.1, `MainActivity` holds a `Toolbar` object and the fragment that contains the `RecyclerView`.

There was obviously a lot of work in modifying these screens (Java classes) in order to make them work using `RecyclerView`. To begin with, the modification of the superclass from which they were inheriting. This single change introduced a lot of errors due to specific methods of the previous superclasses (see table 3.1). However, this errors clearly underlined what it had to be implemented in order to match the previous functionalities and not leaving out any.

Each screen represented a different challenge so speaking thoroughly about the changes made in each of them gets out of the scope of this text. It is recommended to head to the Github repository of this project in order to examine the changes with a level of detail that this text cannot possibly hope to have.

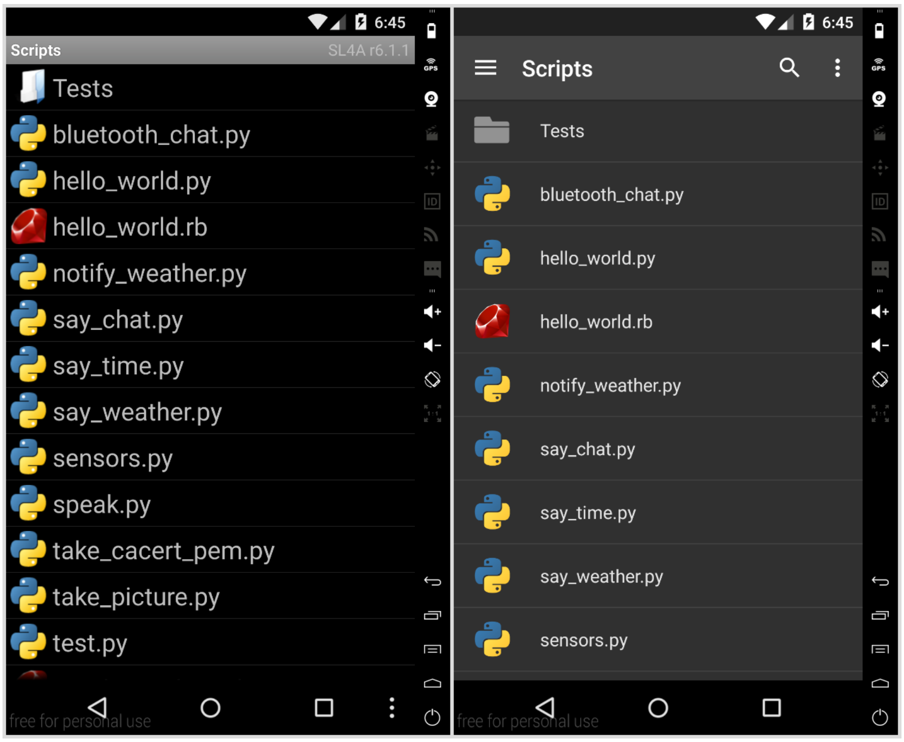


Figure 3.9: Scripts screen: using ListView vs. RecyclerView.

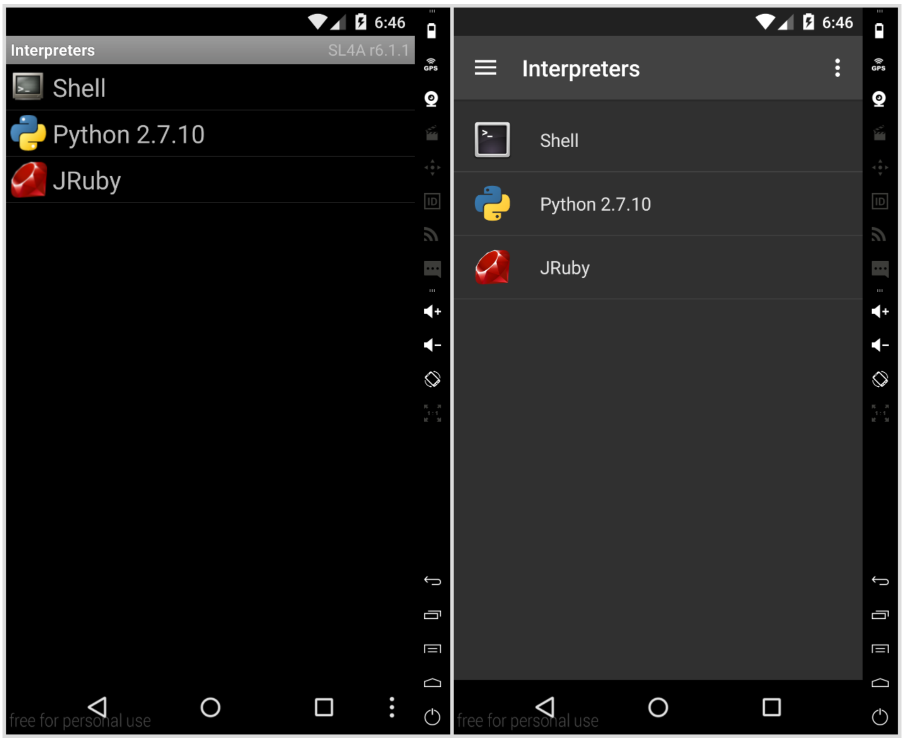


Figure 3.10: Interpreters screen: using ListView vs. RecyclerView

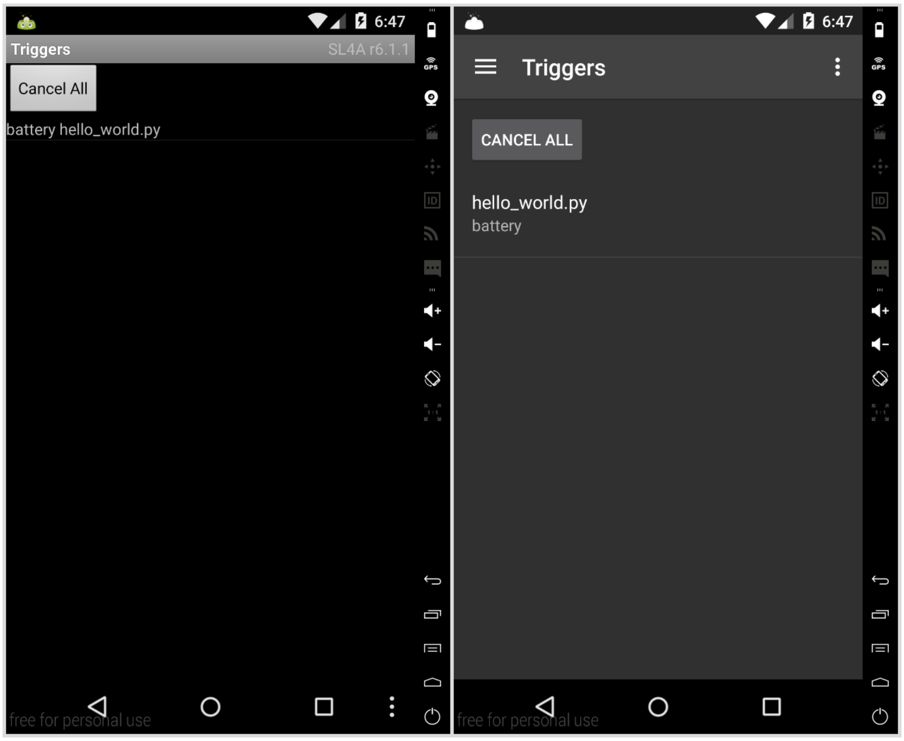
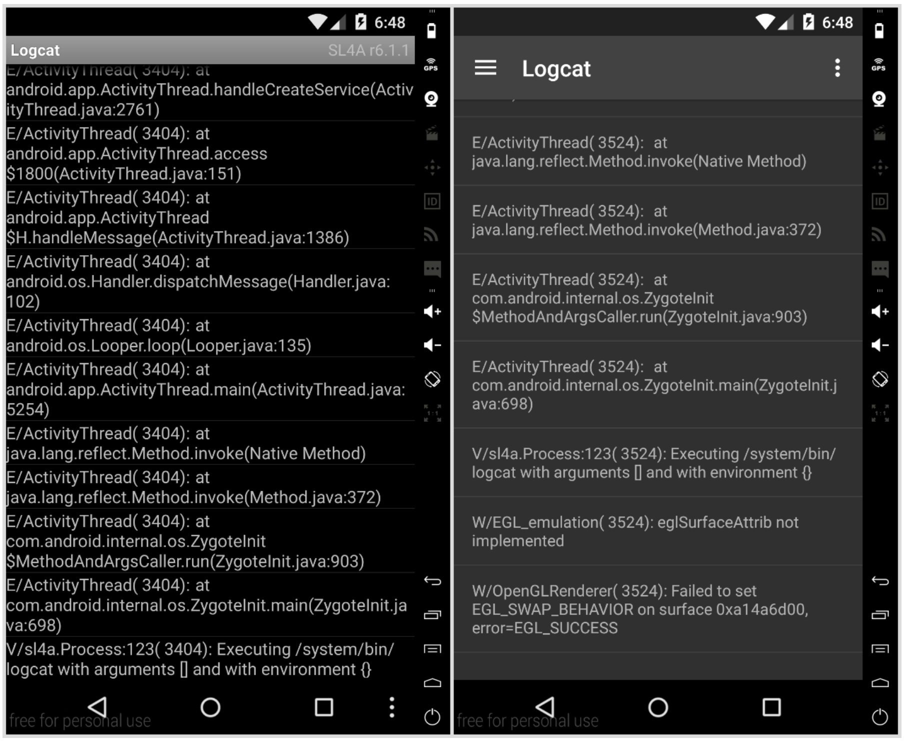


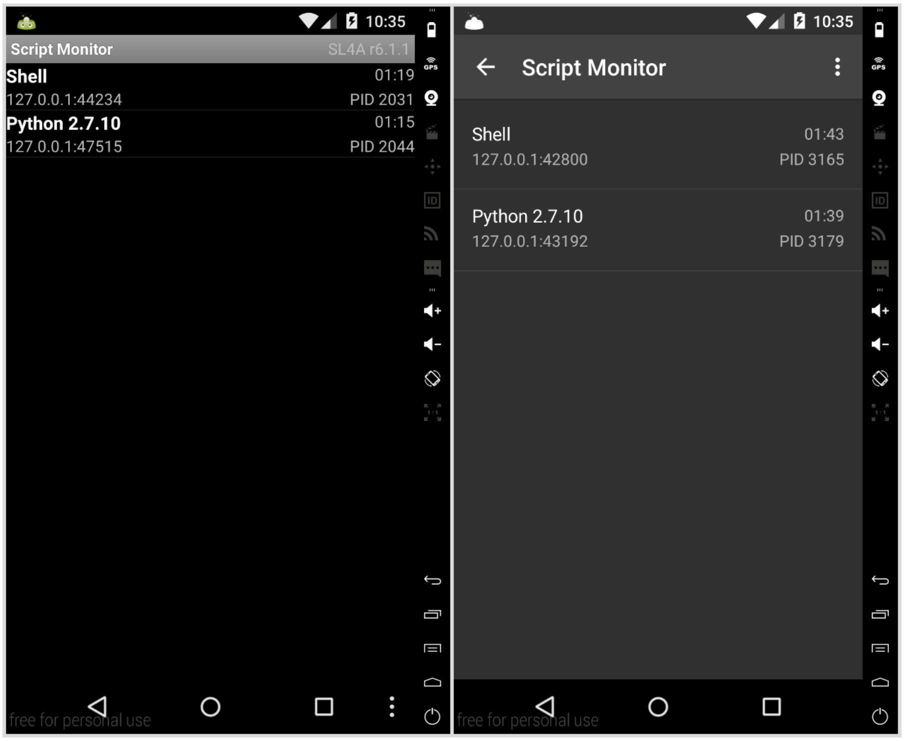
Figure 3.11: Triggers screen: using ListView vs. RecyclerView



```
E/ActivityThread( 3404): at
android.app.ActivityThread.handleCreateService(ActivityThread.java:2761)
E/ActivityThread( 3404): at
android.app.ActivityThread.access
$1800(ActivityThread.java:151)
E/ActivityThread( 3404): at
android.app.ActivityThread
$H.handleMessage(ActivityThread.java:1386)
E/ActivityThread( 3404): at
android.os.Handler.dispatchMessage(Handler.java:
102)
E/ActivityThread( 3404): at
android.os.Looper.loop(Looper.java:135)
E/ActivityThread( 3404): at
android.app.ActivityThread.main(ActivityThread.java:
5254)
E/ActivityThread( 3404): at
java.lang.reflect.Method.invoke(Native Method)
E/ActivityThread( 3404): at
java.lang.reflect.Method.invoke(Method.java:372)
E/ActivityThread( 3404): at
com.android.internal.os.ZygoteInit
$MethodAndArgsCaller.run(ZygoteInit.java:903)
E/ActivityThread( 3404): at
com.android.internal.os.ZygoteInit.main(ZygoteInit.ja
va:698)
V/sl4a.Process:123( 3404): Executing /system/bin/
logcat with arguments [] and with environment {}
```

```
E/ActivityThread( 3524):  at
java.lang.reflect.Method.invoke(Native Method)

E/ActivityThread( 3524):  at
java.lang.reflect.Method.invoke(Method.java:372)

E/ActivityThread( 3524):  at
com.android.internal.os.ZygoteInit
$MethodAndArgsCaller.run(ZygoteInit.java:903)

E/ActivityThread( 3524):  at
com.android.internal.os.ZygoteInit.main(ZygoteInit.j
ava:698)

V/sl4a.Process:123( 3524): Executing /system/bin/
logcat with arguments [] and with environment {}

W/EGL_emulation( 3524): eglSurfaceAttrib not
implemented

W/OpenGLRenderer( 3524): Failed to set
EGL_SWAP_BEHAVIOR on surface 0xa14a6d00,
error=EGL_SUCCESS
```

Figure 3.12: Logcat screen: using ListView vs. RecyclerView



```
Shell                      01:19
127.0.0.1:44234         PID 2031
Python 2.7.10              01:15
127.0.0.1:47515         PID 2044
```

```
Shell                      01:43
127.0.0.1:42800         PID 3165

Python 2.7.10              01:39
127.0.0.1:43192         PID 3179
```

Figure 3.13: Script monitor screen: using ListView vs. RecyclerView

Roughly, these are the most important changes introduced in the graphical user interface of the application. Details such as making the interface coherent throughout the different API levels supported were skipped in this text, but again, if the reader wish to go through all the technicalities, it is advised to visit the repository of this work and its commit history.

In the following section, the longest part of this project will be reviewed: the modifications and enhancements done in the API offered by SL4A.

3.3 SL4A API updates

The main objective of this work was to update the API offered by SL4A. Bringing up to date the whole SL4A API is something way beyond the scope of this work since there are many façades and each façade offers numerous functions. As it has been stated several times in this text, the focus is to extend the support for UI library. Hence, all the work of this phase was about updating and extending the façade in charge of the UI: the UiFacade.

3.3.1 The UiFacade

The UI façade gives access to a series of dialog boxes for general user interaction. It also offers the `webViewShow` and `fullShow` functions. The former permits the use of interactive HTML pages whilst the latter allows to inflate any Android XML layout that is passed to it.

Some of the dialog boxes supported are input dialog, alert dialog, date picker, horizontal progress dialog, seek bar dialog and spinner progress dialog. Each dialog can be customized with features such as setting a list of items (to be used like a menu), a multi-choice list of items, or a single-choice list of items; setting positive, negative and neutral buttons, etc.

With `webViewShow` it is possible to display a `WebView` with the given URL. It is possible to display external URLs as well as “internal” URLs (HTML pages written by the user and that reside on the phone local storage), where there is some JavaScript code that can be later executed by the JavaScript interpreter of the browser.

full`Show` and `fullSetProperty` are the functions that took the most attention in this façade, since they are some of the most complex functions offered by the SL4A API. These functions are closely related. The idea of `fullShow` is to provide a way to display XML layouts, so it must be able to inflate any layout that is passed to it. `fullSetProperty` allows to modify view attributes of inflated layouts; however, the `fullShow` uses the same implementation of `fullSetProperty` to set the attributes of the views that it is inflating.

These functions are also some of the most powerful since they allow to implement complex user interfaces for applications built using SL4A only. This goes beyond the creation of “simple” scripts that use user data as input and then show some results.

3.3.2 Revision of the UiFacade status

The first thing that had to be done was checking the UiFacade to determine what was working fine and what was not. In this way, it was necessary to test all the functions provided by this façade. Table 3.2 lists all the functions offered by the UiFacade and the state in which they were.

Function	State
`addContextMenuItem`	OK
`addOptionsMenuItem`	OK
`clearContextMenu`	OK
`clearOptionsMenu`	OK
`dialogCreateAlert`	OK
`dialogCreateDatePicker`	The dialog was not dismissed automatically after selecting a choice, nor after pressing the hardware back button. The only way to dismiss it was by explicitly calling `dialogDismiss()` in the script.
`dialogCreateHorizontalProgress`	Same problem of `dialogCreateDatePicker`.
`dialogCreateInput`	OK
`dialogCreatePassword`	OK
`dialogCreateSeekbar`	OK
`dialogCreateSpinnerProgress`	Same problem of `dialogCreateDatePicker`.
`dialogCreateTimePicker`	Same problem of `dialogCreateDatePicker`.
`dialogDismiss`	OK
`dialogGetInput`	OK
`dialogGetPassword`	OK
`dialogGetResponse`	OK
`dialogGetSelectedItems`	It did not work. The selected items were not returned, the result was always the items selected by default.
`dialogSetCurrentProgress`	OK
`dialogSetItems`	OK
`dialogSetMaxProgress`	OK
`dialogSetMultiChoiceItems`	OK
`dialogSetNegativeButtonText`	OK
`dialogSetNeutralButtonText`	OK
`dialogSetPositiveButtonText`	OK

`dialogSetSingleChoiceItems`	OK
`dialogShow`	OK
`fullDismiss`	OK
`fullKeyOverride`	OK
`fullQuery`	OK
`fullQueryDetail`	OK
`fullSetList`	OK
`fullSetProperty`	It did work. However, the support was rather limited. This function along with `fullShow` are truly complex, for that reason section 3.3.4 is entirely dedicated to them.
`fullSetTitle`	OK
`fullShow`	It worked. Although if the function was given an empty string (i.e., no layout) the application crashed. The main problems were related with the setting of attributes. See section 3.3.4 for more details.
`webViewShow`	It worked but it had a couple of issues: only local URLs were supported, and once this function was called it was no possible to go back to SL4A without a forced closure. It also had some security issues due to deprecated code.

Table 3.2: UiFacade status of the functions.

3.3.3 UiFacade general adjustments

In the previous section all the functions offered by the UiFacade were listed and their state was specified. This section is dedicated to show how the problems of the functions that were not working fine were addressed. As it was stated, fullSetProperty has section 3.3.4 dedicated to it so it will not be seen in this section.

3.3.3.1 Dialog dismissal problem

As it was seen, after calling `dialogCreateDatePicker`, `dialogCreateHorizontalProgress`, `dialogCreateSpinnerProgress` and `dialogCreateTimePicker`, these dialogs were not dismissed automatically after the user tapped on any dialog button, and it was not possible to dismiss them by pressing the hardware back button. The only way to dismiss these dialogs was to call `dialogDismiss` explicitly in any script that used the previously mentioned functions.

The solution was fairly simple. All the Java classes in charge of handling these dialogs (`DatePickerDialogTask`, `ProgressDialogTask` and `TimePickerDialogTask` in `Common/src/.../androd_scripting/facade/ui/`) had to be modified in the same way: a call to `dismissDialog`, a function of their superclass. Dialogs without the dismissal issue (such as `AlertDialogTask`) had it. This function was in charge of dismissing the dialog upon user request. The following code applied for all the aforementioned classes:

```
mDialog.setOnDismissListener(new DialogInterface.OnDismissListener() {
   @Override
   public void onDismiss(DialogInterface dialog) {
      // Some code specific for each dialog
      ...
      dismissDialog();
   }
}
```

Where `mDialog` is an object of a class subclassing `DialogTask`. In this class, `dismissDialog` is implemented and it is the function that is in charge of dismissing (i.e., destroying) any existing dialog. The problem existed because `dismissDialog` was not being called when each dialog called its `onDismiss` method (that was invoked when the user pressed the hardware back button or tapped on any dialog button). Simply adding a call to `dismissDialog` in `onDismiss` fixed the issue.

3.3.3.2 dialogGetSelectedItems

This function is used along with `dialogSetMultiChoiceItems` and `dialogSetSingleChoiceItems`. As mentioned in table 3.2, the problem with this function is that it was not returning the item or items selected in the dialogs. The supposedly selected items were always the same as the items selected by default, regardless of which items were selected by the user.

The issue occurred if the result was requested just after the dialog was created, returning the set of items selected by default since it was impossible for the user to be faster than the execution of the script. In spite of the fact that this problem could be circumvented via scripting, it was deemed better to solve it internally in the function implementation. It was solved by postponing the retrieval of the selected values until the user pressed the positive button in the corresponding dialog.

3.3.3.3 fullShow

`fullShow` is in charge of inflating a layout based on the string it is passed. The string corresponds to an XML file used on Android to represent layouts. Typically, what is done in the scripts written for SL4A, is that an external XML file with the layout is loaded and then converted into a single string that is then passed to `fullShow` so that it can be inflated.

This worked relatively fine, however a corner case was missing: if this string was empty, the application went into an exception and crashed. This happened because when the method to inflate a layout receives an empty string, it simply returns a view with value `null`. This was easily solved by checking the outcome of the inflater and in case it is `null`, it is replaced by a default view. In `Common/src/...android_scripting/facade/ui/FullScreenTask.java`:

```
@Override
public void onCreate() {
   ...
   try {
      // Try to inflate view
   } catch (Exception e) {
      // Catch any error when trying to inflate view
      // (null strings were not caught)
   } finally {
      if (mView == null) {
         // Replace view with null value with default view
      }
   }
}
...
```

3.3.3.4 webViewShow

The `webViewShow` function permits to open a custom HTML page in a `WebView`. This custom HTML page can be specified by passing a URL (which also includes the possibility of using local HTML files by providing `file://...` paths). The attractiveness of using `WebViews` is the possibility of executing JavaScript scripts that access the Android APIs through SL4A.

As it can be seen in table 3.2, this function had several issues. One of the issues was that it was not possible to open external URLs. In order to solve this problem the decision was to delegate this responsibility to a dedicated browser. It is not optimal since scripts meant to use the SL4A API will not work; however, allowing external scripts to use this API poses an enormous security risk. Local files are still handled within `WebViews`.

A second issue was that once `webViewShow` was called, it was impossible to return to SL4A unless the application was closed by force. In order to overcome this problem, it was necessary to subclass `WebView` and then override the function regarding the hardware back button. Since this new class is not needed anywhere else, it could be safely implemented as an inner class within `Common/src/.../interpreter/html/HtmlActivityTask.java`.

```
public class MyWebView extends WebView {
   Activity mActivity;

   public MyWebView(Context context) {
      super(context);
      mActivity = (Activity) context;
   }

   @Override
   public boolean onKeyDown(int keyCode, KeyEvent event) {
      if (keyCode == KeyEvent.KEYCODE_BACK) {
         mActivity.onBackPressed();
         return true;
      } else {
         return super.onKeyDown(keyCode, event);
      }
   }
}
```

The last issue was due to deprecated code. From API level 17 all public methods that need to be accessed from JavaScript have to be annotated with `@JavaScriptInterface`, otherwise they will be ignored. Obviously, due to the long time of abandonment of the project these annotations were not present and without them it was not possible to execute any JavaScript within `WebViews`. Simply annotating the objects to be injected in the `WebView` solved the issue.

As a caveat, these annotations allow JavaScript to control the host application. As it was previously stated, there is a big security risk with this. However, since SL4A is aimed to developers they must know the security risks involved. Additionally, and as it was stated too, it is only possible to execute scripts in `WebViews` locally.

3.3.4 Improving the inflation of layouts

As it could be inferred from the previous section, `fullShow` and `fullSetProperty` are the most complex functions offered by the UiFacade. `fullShow` allows to inflate layouts which makes it possible to create complex applications using SL4A. Normally, applications created natively for Android use layouts created in XML and by compilation these layouts are transformed into something that Android understands.

However, due to the nature of SL4A these layouts cannot be pre-compiled (the idea is that a developer can create any layout on the fly so that it is inflated). This makes it really complex since the XML file has to be parsed in runtime within the application. This task is delegated to `Common/src/.../facade/ui/ViewInflater.java`. In order to be able to inflate any XML layout, SL4A makes use of a metaprogramming strategy called reflection that offered by Java.

Reflection is a computer process that embraces self-awareness. It is the capacity of a computer program to analyze and transform its own structure and behavior at runtime. Reflection allows to modify the execution of a program based on the same execution. This is normally achieved by dynamically designating program code at runtime. In Java, reflection permits to inspect classes, interfaces, fields and methods at runtime without knowing their names at compile time. It also permits to instantiate new objects and invocate methods of generic objects.

What it is done in `ViewInflater` when a layout is passed, is parsing it and building every view contained by the layout along the way. The first task is done by the `inflateView` method with the help of an `XmlPullParser` object that provides parsing functionality for XML. `inflateView` makes a recursive call if the view being parsed is an instance of `ViewGroup` (i.e., a view that groups other views). In this case the child views are constructed by calling `inflateView` recursively and then appending them to the current view group.

What `inflateView` does internally is calling `buildView`. `buildView` is is charge of the second task: creating the actual views. Here is when reflection comes into the game for the first time: by using the name of the XML element that is being parsed, an instance of the corresponding view is created. The way in which Android was thought allows this, since the names of the elements in an XML layout, coincide with the Java classes names that represent those layout elements (or views).

Thanks to the `XmlPullParser` object the name of the element being parsed can be easily known, and since this name corresponds to a Java class of Android it is possible to create an view object of that class using reflection. The following lines of code explains it better. Please note that this is just an example and that the actual code in `ViewInflater` that does this task is more complex.

```java
// Get name of the XML element being currently parsed
String name = xmlPullParser.getName();

// Create a class object and obtain a constructor object from it
Class<? extends View> viewClass = Class.forName(name).asSubclass(View.class);
Constructor<? extends View> ct = viewClass.getConstructor(Context.class);

// Create the view using the constructor object
View view = ct.newInstance(getActivity());
```

In this way it is possible to create any view by parsing an XML layout file/string. In fact, no problems regarding the inflation of views were found. In all the ahtests done there were no problems inflating the views themselves. The situation becomes really complex at the moment of setting attributes for the views created. The `fullSetProperty` of the UiFacade suffers the same problem since this function uses the same implementation that `fullShow` uses to set attributes for views.

3.3.4.1 Explicitly supported view attributes

The setting of attributes was performed in `setProperty` by checking a bunch of conditionals, and calling the corresponding methods to set the attributes for the views. The rest of the attributes were attempted to be set by `setDynamicProperty` with the use of reflection. Table 3.3 lists the attributes that were explicitly supported by this set of conditionals just mentioned and the state in which they were.

View attribute	Initial state of the setting method
background	It had some deprecated code.
digits	OK
gravity	OK
height	It had problems when using “mm” as the units of measurement. The resulting views were larger than expected.
id	It worked partially. The only way to set an id to a view was by prefixing the id with “@+id/” or “@id/”. This is accurate when inflating a layout but it does not make much sense while setting an id using `fullSetProperty`.
inputType	OK
layout_*	Some attributes related with `RelativeLayout` presented issues and were not set properly.
nextFocus[Down|Forward|Left|Right|Up]	Related to the id attribute problem. It worked fine as long as the given id had was prefixed by “@+id/” or “@id/”. This is accurate while inflating a layout but it is makes no sense while setting the property using `fullSetProperty`.
padding	OK
src	OK
stretchColumns	It worked, although it had some issues with some strings. “*” which is a valid value for this attribute was not supported either.
textColor	It worked fine as long as the given color was in the RGB color model format.
textColorHint	Same problem as with textColor.
textColorHighlight	It did not work.
textSize	It sort of worked, but the text was abnormally large for the values that it was given.
textStyle	OK
typeFace	OK
width	Same problem as height.

Table 3.3: Initial support of view attributes on SL4A.

The following paragraphs are dedicated to explain how the problems listed in table 3.3 were solved.

background

The deprecated code could not be entirely removed because it was necessary to support devices with older versions of the Android platform. A conditional was introduced so that newer devices use the new code, whilst older devices that do not support the code keep using the deprecated code. This solution is used in many other instances of SL4A where a complete removal of deprecated code was not possible.

```
if (Build.VERSION.SDK_INT < Build.VERSION_CODES.JELLY_BEAN) {
   // Old deprecated code
   view.setBackgroundDrawable(getDrawable(value));
} else {
   // Newer implementation
   view.setBackground(getDrawable(value));
}
```

`Build.VERSION.SDK_INT` is an integer that is equal to the API level in which the device is, whereas `Build.VERSION_CODES` contains a set of integer named after the released versions of the Android platform, and their values correspond to the API levels the releases. This version code correspond to the last API level before the code was deprecated.

height and width

The problem was that when the units used for these attributes were millimeters (mm), the resulting view sizes did not correspond to what was expected. These views were clearly larger. The problem was simply a wrong conversion of units.

```
if (units.equals(“mm”))
   return (float) (inches / 2.54);  // This converts to cm. Changed to 25.4
```

id

As it was stated in table 3.3, setting an id always required it to be prefixed by “@+id/” or “@id/”. This is accurate while loading a layout (e.g., when `fullShow` is invoked); however, it was no sensible when modifying the id of an existing view.

In order to make SL4A coherent with Android Java development, “@+id/” is required to set a view id while loading a layout, whereas “@id/” must only be used if the id is an existing id on the same layout or in a built-in Android class (e.g., in the android.R.id class). To change an id after the view is created, does no require the new id to be prefixed anymore.

layout_*

Most attributes of this set worked without issues. The problems belonged to a subset of attributes used only with `RelativeLayout`. These attributes were not correctly supported, and when they were attempted to be set, errors raised and attributes in other views were not set either. The issue was that for all `RelativeLayout` attributes, the value given was always treated as an id of an anchor view. Nevertheless, attributes such as `layout_alignParentTop` expect a boolean and the anchor view is implicitly defined as the parent view.

nextFocus*

The issue with these attributes was closely related to the one with the id attribute. In this case, the id passed as the value to set these attributes had to be prefixed by “@+id/” or “@id/”. It is now not necessary (which makes the calls to these function more natural), but it is still possible to use them so that older scripts do not stop working. This solution also apply to a plethora of attributes that require an id as an input parameter.

stretchColumns

This attribute accepts strings of the type “1,2,4”, in which each number corresponds to a column of a `TableLayout` view that needs to be stretched. It worked as long as the string did not have any space character in between. It is not a serious issue but might be tricky in some cases. It now accepts strings with spaces (as long as they are well formed). Moreover, it now supports the string “*” as input when it is wanted to stretch all the columns of a `TableLayout` view.

textColor and textColorHint

As it is expressed in table 3.3 these attributes worked. Although they only accepted strings in the RGB model format (i.e., `“#RGB”`, `“#RRGGBB”`, `“#ARGB”`, `“#AARRGGBB”`). Now, colors coming from Android resources (for instance, from `android.R.color`) are also accepted.

textColorHighLight

This attribute goes in the same direction of textColor and textColorHint. However, it did not work because it was wrongly named in the set of conditionals. The set of conditionals were expecting an attribute named textHighlightColor. This was easily solved by changing the name expected. Also, the enhancement done for the previous two attributes also apply to this one.

textSize

In spite of the fact that the setting of this attribute sort of worked, the resulting text size did not correspond to what was expected. It was much larger. For the majority of attributes related with sizes, their corresponding setting methods expect sizes in pixels. However, the method corresponding to textSize expects the size in Scaled-Independent Pixels (sp), and it was being given sizes in pixels.

3.3.4.2 Implicitly supported view attributes

Table 3.3 lists the attributes “explicitly” supported, i.e., the attributes that were in a set of conditionals inside `ViewInflater`. However, there were many attributes “implicitly” supported, inclduing scaleX, scaleY, rotation, and text (in the case of `TextViews`). These attributes were supported by using reflection. The `setDynamicProperty` method in `ViewInflater` was in charge of handling this.

The strategy to set the attribute of a view implicitly consisted in forming a string of the form `“set” + “AttributeName”`, and then use it with reflection to retrieve a method from the view with a name matching the string. This method is then invoked with the value given for the attribute. The following lines of code is aimed to clarify this strategy, but it must be taken only as a simple example.

```java
private void setDynamicProperty(View view, String attr, String value) {
   String name = “set” + toPascalCase(attr);
   Class<?>[] parameters = {CharSequence.class};  // A method may accept
                                                  // several parameters
   Method method = view.getClass().getMethod(name, parameters);
   method.invoke(view, value);
}
```

This strategy proves useful for many attributes but it is not a silver bullet, and a plethora of attributes do not get supported in this way. The following section will go through the work done in order to support more view attributes.

3.3.4.3 Extending the supported view attributes

As it was just mentioned, the strategy of `“set” + “AttributeName”` is useful to support many attributes in an implicit way but it has limitations. Firstly, important attributes like paddingLeft, paddingBottom, transformPivotX, among others do not match this pattern since these attributes have setting methods that are named differently.

Secondly, this strategy passes to the setting method the attribute value to be assigned as it is or with minor class casting. Nevertheless, many attributes need treatment for those values. For instance, some attributes accept a number with a valid measurement unit, but their setting methods expect an integer value that represents a number of pixels. Another example is with attributes that accept colors (either in the RGB model format or from a resource) but expect an integer value representing the color. For these and other cases, the value has to treated so that it is valid and consistent with the setting method.

Thirdly, the `“set” + “AttributeName”` strategy accepts class constants as values. By using reflection, based on the given attribute value the view's class is examined to find a matching constant field. If it is found the value of the constant is passed to the setting method. This strategy works as long as the class constant matches the value given in upper case, and the constant is a field of the same class of the view whose attribute is being set. For many attributes, this is not the case, so the value is not recognized as a valid class constant.

Lastly, some attributes are so complex to set that they require a special treatment. There is no possible way to treat the given value by a simple use of reflection. One example is the textSize attribute. For nearly every attribute that accepts values with units in their XML, its corresponding setting method expects an integer value in pixels. However, in the case of textSize, `setTextSize` expects a value in Scaled-Independent Pixels (sp), which breaks the pattern.

In spite of the fact that it does not support every single attribute, the strategy is still useful. In fact, since Android View classes count with a very large number of attributes it is basically impossible to treat all view attributes by employing only one strategy. The idea then was to implement different strategies and depending on each attribute using one or more.

In order to give support to the most critical view attributes, a thorough verification of the operation of the attributes was done for the following view classes: `View`, `ImageView`, `TextView`, `GridLayout`, `LinearLayout`, `RelativeLayout` and `TableLayout`. In each of these classes all the attributes were revised in order to known if they were correctly handled by the `“set” + “AttributeName”` strategy or they needed some special treatment. Of course there are many other view classes, however, there are way too many and these classes cover the most used views in Android.

After analyzing all the attributes for the aforementioned view classes and subclasses and taking into account the problems listed, the following are the identified cases in which it is necessary to apply a special treatment to an attribute in order to set its value correctly:

- The attribute's setting method is named differently to `“set” + “AttributeName”`.
- The given value needs some treatment. This value needs to be treated when the setting method expects a view id, a size in pixels or an integer matching a color.
- The given value corresponds to a class constant, but the name of the value in upper case does not match the class constant name or the constant is in a different class of the view whose attribute is being set. In many cases the class name do not match because it follows a different pattern. The identified patterns used for class constant names are (notice that they are in snake case): `“ATTRIBUTE_NAME” + “_” + “VALUE”`, `“SOME_PREFIX” + “_”+ “VALUE”`, `“SOME_PREFIX” + “ATTRIBUTE_NAME” + “_” + “VALUE”`.

And the following are the strategies that need to be applied in order to overcome the just mentioned problems:

- Provide the correct name so that it can be found using reflection.
- Transform the given value depending on its type. In order to know what transformation

hast to be applied, a set of helper flags are used.

- Provide the name of the class where the constants matching the values accepted are defined, and provide a string that serves as a prefix in case it is needed.

For many view attributes, applying one or more of the aforesaid strategies is enough to enable their correct setting by using reflection. Nevertheless, and as it was mentioned before, due to their complexity many attributes cannot be handled with any of these strategies. In these cases they need to be handled independently. Therefore, the more attributes that need to be handled this way, the longer the code and hence the more difficult its maintenance.

In order to attenuate this problem, it was necessary to apply an strategy that made the code readable and easily extensible. After several strategies used, the definitive one for this project was to use a `HashMap` object in `ViewInflater`, in which each key of the map is the name of the view attribute and the value is some information about how to handle this attribute. This `HashMap` is populated statically so that it is always available and does not have to be populated every time a `ViewInflater` object is created.

The question just after was, how to encode the information that each attribute will have? It was thought of enconding this information by using a JSON object passed as string that later would have to be decoded to extract all its attributes, each attribute being some information to handle the setting of the attribute correctly. The problem with this is that Java does not support JSON in a simple way, so encoding and decoding not only takes a long time but makes the code less natural.

Another option was to use `HashMaps` as values for the first `HashMap`, however, the concern was about the amount of memory that each `HashMap` would take, and taking into account the high number of attributes to be supported this way, the concern was even higher. The solution was to use a different Java implementation for maps called `EnumMap`. In this map all the keys come from a single `enum` type that is specified when the map is created. `EnumMap` objects are represented internally as arrays, which make them extremely efficient and compact.

Knowing this, the following `enum` was implemented:

```java
private enum AttributeInfo {
   HELPER_METHOD, ATTR_METHOD, VAL_MODIFIER, CONSTANT_CLASS, CONSTANT_PREFIX
}
```

With this, it is possible to create `EnumMap` objects in which the keys are the members of the `enum`, whilst the value can be any object. In this case, the values are strings that provide specific pieces of information about how the attribute has to be handled. For each attribute, an `EnumMap` object is used. On the other hand, not all the keys of an `EnumMap` object have to be used.

`ATTR_METHOD` is used to specify the correct name of an attribute setting method. `VAL_MODIFIER` is used to specify a modifier that must be used to transform the value given into something that is meaningful and coherent with the setting method. `CONSTANT_CLASS` is used to specify a class when an attribute accepts a set of values matching the constants of that class, and that class is not `View` nor a subclass of it. `CONSTANT_PREFIX` is related with class constants as well, but in this case it serves to specify a prefix because in many cases the class constants are named with the pattern `SOME_PREFIX + VALUE_IN_UPPERCASE`. Finally, `HELPER_METHOD` is used when the setting of the attribute is so complex that it needs to be handled by a method defined just for this purpose.

The following function was written to instantiate and populate `EnumMap` objects:

```
private static Map<AttributeInfo, String> mapAttrInfo(Object... attrInfo) {
   Map<AttributeInfo, String> infoMap = new EnumMap<>(AttributeInfo.class);
      for (int i = 0; i < attrInfo.length; i = i + 2) {
      infoMap.put((AttributeInfo) attrInfo[i], (String) attrInfo[i + 1]);
   }
   return infoMap;
}
```

For each attribute in the `HashMap` an `EnumMap` is created using this method. This shortens the definition of such `HashMap` and also makes it more readable. The following example, allows to see how some attributes are added:

```
static final Map<String, Map<AttributeInfo, String>> mXmlAttrs =
   new HashMap<>();

mXmlAttrs.put(“elevation”,
   mapAttrInfo(AttributeInfo.ATTR_MODIFIER, “dimension”));

mXmlAttrs.put(“nextFocusUp”,
   mapAttrInfo(AttributeInfo.ATTR_METHOD, “setNextFocusUpId”,
      AttributeInfo.ATTR_MODIFIER, “viewId”));

mXmlAttrs.put(“id”, mapAttrInfo(AttributeInfo.HELPER_METHOD, “setViewId”));
```

The way in which `mapAttrInfo` works consists in specifying the kind of information that will be specified, and then the corresponding string providing the information itself. We can see in the previous example that for the elevation view attribute a value modifier is used, “dimension” in this case. This information is used later to transform the value for elevation from dp, sp, etc, to an integer representing a number of pixels. For nextFocusUp two pieces of information are specified: the correct setting method name, and a modifier to retrieve an integer representing a view id from a string. The id attribute is complex to set so it needs to be handled separately, that is why a helper method is specified for it.

The case of the helper method also takes advantage of reflection, it is used to retrieve the needed helper method having only a string with its name. In order to keep everything as organized as possible, a class inside `ViewInflater` was defined. This class has the task of holding all the helper methods and auxiliary functions that the helper methods use. This inner class is called `ViewAttributesHelper`.

Not only was this `HashMap` implemented, but also the `“set” + “AttributeName”` strategy was improved in order to support more attributes. Several attributes whose valid values have a corresponding class constant do not need to be specified in the `HashMap`. They can be easily solved by “parsing” the value given. The `“set” + “AttributeName”` strategy takes the given value and put it in upper case since many class constants match this pattern. However, when it was a string composed of several words in camel case, the uppercase version did not match. Converting from camel case to snake case and then to uppercase, gave support to several more attributes.

Table 3.4 lists all the attributes that got support by the previous strategies. The attributes that are not in this table, still rely on the `“set” + “AttributeName”` strategy for their setting. The attributes supported by these new strategies plus the ones supported by the `“set” + “AttributeName”` strategy, are in the vast majority of layouts used for Android applications. There is a still a long way to support all of the view attributes, but since supporting them represent a immeasurable amount work and they are hardly used, it was decided to stick to the ones supported so far.

Attribute	Problem(s) and solution
`View`	
accessibilityLiveRegion	Attribute values did not match class constants names. Solved by transforming the values from camel case to snake case.
accessibiltyTraversalAfter	Could not handle id as strings, only numbers. Solved with the `VAL_MODIFIER` for ids.
accessibiltyTraversalBefore	Could not handle id as strings, only numbers. Solved with the `VAL_MODIFIER` for ids.
backgroundTint	Very complex method. Needs to define a special kind of object to be passed to the setting function. `HELPER_METHOD` was implemented for it.
drawingCacheQuality	Attribute values did not match class constants names. Solved by transforming the value from camel case to snake case.
elevation	Accepts values with screen units. Got fully supported using the `VAL_MODIFIER` for units.
fadeScrollbars	The setting method name did not match `“set” + “AttributeName”`. Solved using the `ATTR_METHOD`. However, enabling the setting of this attribute is dangerous since if it set for a non scrollable view the

	whole application crashes. It was kept disabled.
fadingEdgeLength	Accepts values with screen units. Got fully supported using the `VAL_MODIFIER` for units.
importantForAccessibility	Attribute values did not match class constants names. Solved by transforming the value from camel case to snake case.
isScrollContainer	The setting method name did not match `“set” + “AttributeName”`. Solved using the `ATTR_METHOD`.
layoutDirection	Attribute values did not match class constants names. Solved by transforming the value from camel case to snake case.
minHeight	The setting method name did not match `“set” + “AttributeName”`. Moreover, this method accepts values with screen units. Solved using `ATTR_METHOD`. for the method name and `VAL_MODIFIER` for units.
minWidth	Same problems as minHeight.
paddingBottom	These attributes were quite complex since there is not method to set them individually. Therefore, `HELPER_METHOD` was used to handle the setting of each attribute using the general method `View.setPadding`.
paddingEnd	
paddingLeft	
paddingRight	
paddingStart	
paddingTop	
requiresFadingEdge	In order to set this attribute correctly, several methods need be used. Hence, it needed a specific method for this and `HELPER_METHOD` was used.
scrollbarDefaultDelayBeforeFade	The setting method name did not match `“set” + “AttributeName”`. Solved using the `ATTR_METHOD`.
scrollbarFadeDuration	
scrollbarSize	The setting method name did not match `“set” + “AttributeName”`. Moreover, this method accepts values with screen units. Solved using `ATTR_METHOD`. for the method name and `VAL_MODIFIER` for units.
scrollbarStyle	The setting method name did not match `“set” + “AttributeName”`, and attribute values did not match the class constants names. Solved using the `ATTR_METHOD` and `CONSTANT_PREFIX`.
textAlignment	Attribute values did not match class constants names. Solved by transforming the value from camel case to snake case.
textDirection	
transformPivotX	The setting method name did not match `“set” + “AttributeName”`. Moreover, this method accepts values with screen units. Solved using `ATTR_METHOD`.

	for the method name and `VAL_MODIFIER` for units.
transformPivotY	Same problems as transformPivotX
translationX	Accept values with screen units. Got fully supported using the `VAL_MODIFIER` for units.
translationY	
translationZ	
`ImageView`	
baseline	Accept values with screen units. Got fully supported using the `VAL_MODIFIER` for units.
maxHeight	
maxWidth	
tint	Complex attribute. A special kind of object has to be defined and then passed to the setting method. `HELPER_METHOD` was used to handle this.
`TextView`	
autoText	This attribute is complex to set since its setting conflicts with capitalize. Got fully supported with a `HELPER_METHOD`.
bufferType	This attribute is complex to set since its setting method requires two arguments. Got supported with `HELPER_METHOD`.
capitalize	This attribute is complex to set since its setting conflicts with autoText. Got fully supported with a `HELPER_METHOD`.
drawableBottom	Setting these attributes is complex since they are all set with the same function. Additionally, it is necessary to create a `Drawable` object based on the value given. Got fully supported with `HELPER_METHOD`.
drawableEnd	
drawableLeft	
drawablePadding	
drawableRight	
drawableStart	
drawableTop	
ellipsize	The “constant” values related to this attribute come from an `enum`, not from a class. Which makes it hard to use reflection. Got supported with a `HELPER_METHOD`.
fontFamily	The setting method name did not match `“set” + “AttributeName”`. Solved using the `ATTR_METHOD`.
imeActionId	The setting method needs information about imeActionLabel. Got supported with `HELPER_METHOD`.
imeActionLabel	The setting method needs information about imeActionId. Got supported with `HELPER_METHOD`.

lineSpacingExtra	The setting method requires information about lineSpacingMultiplier. Got supported with `HELPER_METHOD`.
lineSpacingMultiplier	The setting method requires information about lineSpacingExtra. Got supported with `HELPER_METHOD`.
maxLength	Complex attribute. A special kind of object has to be defined and then passed to the setting method. `HELPER_METHOD` was used to handle this.
scrollHorizontally	The setting method name did not match `"set" + "AttributeName"`. Solved using the `ATTR_METHOD`.
shadowColor	These attributes use the same setting method and thus all the attributes are setting at once. When setting one of these attributes, it is necessary not to overwrite the rest. Due to this complexity, it was necessary to use `HELPER_METHOD`.
shadowDx	
shadowDy	
shadowRadius	
textAllCaps	The setting method name did not match `"set" + "AttributeName"`. Solved using the `ATTR_METHOD`.
textColorLink	The setting method name did not match `"set" + "AttributeName"`. Solved using the `ATTR_METHOD`.
GridLayout class	
alignmentMode	Attribute values did not match class constants names. Solved by transforming the value from camel case to snake case.
`LinearLayout`	
divider	The setting method of this attribute is complex because it requires a `Drawable` object. Got supported with a `HELPER_METHOD`.
measureWithLargestChild	The setting method name did not match `"set" + "AttributeName"`. Solved using the `ATTR_METHOD`.
showDividers	Attribute valid values did not match class constants names. Solved using `CONSTANT_PREFIX`.
`RelativeLayout`	
ignoreGravity	Could not handle id as strings, only numbers. Solved with the `VAL_MODIFIER` for ids.
`TableLayout`	
collapseColumns	These attributes is closely related to the stretchColumns attribute that was explicitly supported. `HELPER_METHOD` is now used for all of them.
shrinkColumns	

Table 3.4: View attributes that got support for full screen tasks.

3.3.5 Extending UiFacade offered functions

In the previous section we saw how the layout inflation was improved by supporting a good deal of view attributes that were not supported before. In order to keep consistency with the update to Android Lollipop, four new functions were added to the UiFacade itself. These functions are:

- `fullSetToolbarTitle`
- `fullSetToolbarSubtitle`
- `fullSetToolbarTitleColor`
- `fullSetToolbarSubtitleColor`

These functions require an existing `Toolbar` object in a full screen task. In order to have a `Toolbar` object it is only necessary to define a `Toolbar` element in the XML layout passed to `fullShow`. After this, we only require the `Toolbar` object's id to be able to set its properties.

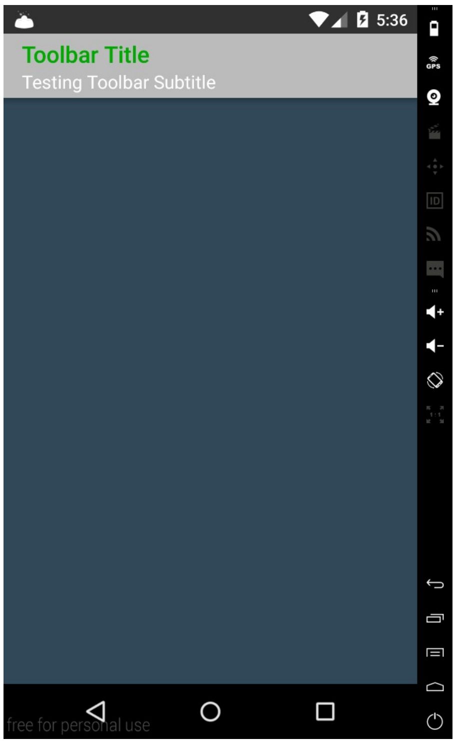


Figure 3.14: New supported `Toolbar` properties

This is roughly the changes implemented in SL4A for this work. Due to the nature of SL4A, there are still many things to do as it can grow along Android's growth. This will be touched in more detail in chapter 5. The following chapter reviews the tests performed to verify that the modifications did not introduce bugs that jeopardize SL4A's use.

Chapter 4
Experiments and Validation

As it has been stated in previous chapters, the Scripting Layer for Android is a framework that allows a developer to execute code written in different scripting languages on an Android device. However, this project has been abandoned for a long time and despite that there are some new forks that are working on it, the progress have been of minor consideration.

In chapter 3 all the carried out changes in SL4A were reviewed, and also it was examined how these changes brought SL4A up to date in several aspects. All of these changes and introduced functionalities were tested along the development cycle. However, with the purpose of having better readability in this text, the development and testing were separated into two different chapters. This chapter is dedicated to test and validate all of what was reviewed in chapter 3.

4.1 Validating UI changes

As it was seen in the chapter 3, SL4A went under a restyle with the purpose of making the application more according to current trends. It now follows the Google's Material Design principles that all Android applications are supposed to follow after the release of Android Lollipop.

In order to validate this facelift, it was necessary to check that none of the functionalities present with the old user interface went missing or did not work accordingly. The following use cases were tested in both the older and newer version to check that the behavior was identical:

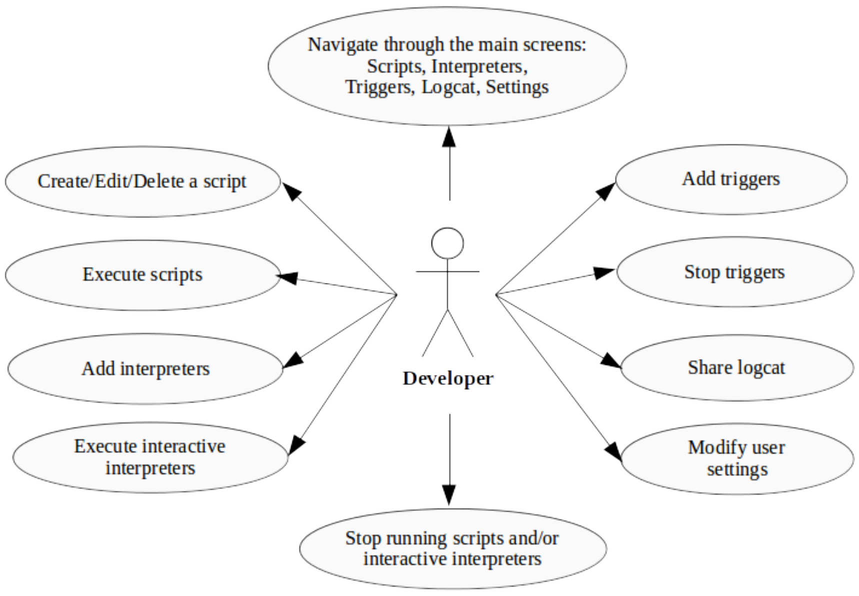

Navigate through the main screens: Scripts, Interpreters, Triggers, Logcat, Settings
Create/Edit/Delete a script
Execute scripts
Add interpreters
Execute interactive interpreters
Add triggers
Stop triggers
Share logcat
Modify user settings
Developer
Stop running scripts and/or interactive interpreters

Figure 4.1: Use cases tested after graphical user interface renewal.

4.2 Testing the UiFacade

As it was seen in chapter 3, table 3.2 lists all the functions offered by the UiFacade and the state in which they were before carrying out this work. In order to check the functionality of these functions, there was no better choice than writing scripts in order to evaluate their behavior. The following subsections will show the scripts used to test along with the results obtained.

In spite of the fact that some of the scripts test several functions of the facade at once which helps compact and not repetitive tests, the number of functions offered by the facade is large so this text only shows the most representative tests. If the reader wants to check all the test scripts, please go to the corresponding folder in the repository[14]. All of these scripts are written in the Python Language.

4.2.1 Contextual menu

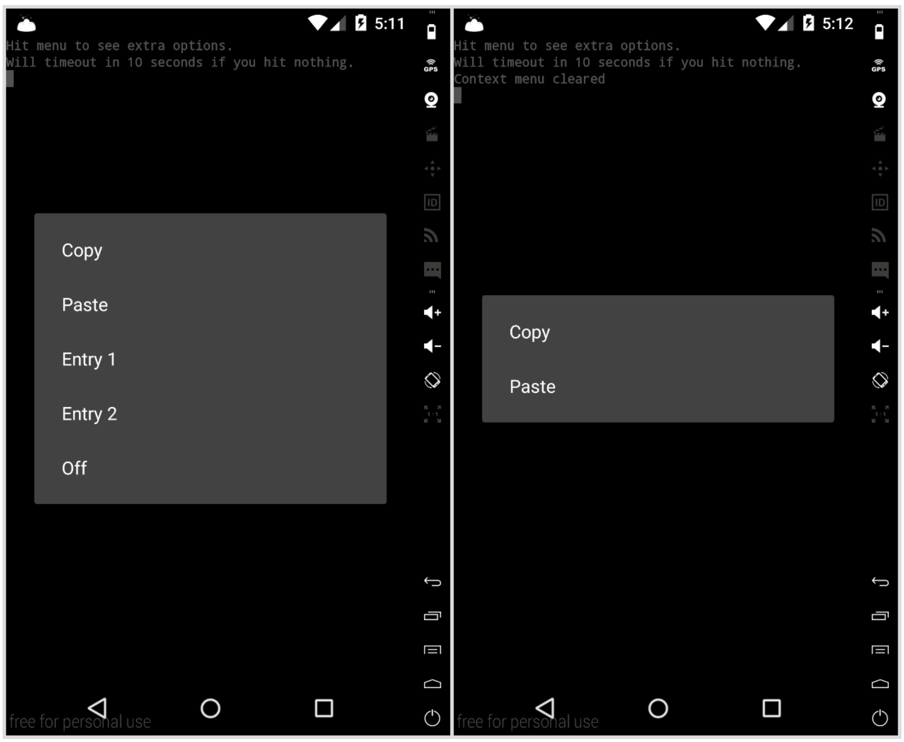
This section is dedicated to the functions `addContextMenuItem` and `clearContextMenu` of the UiFacade. The first function allows us to add extra items to the context menu, which is displayed when the user taps and holds for about one second. The script for `addOptionsMenuItem` and `clearOptionsMenu` is very similar.

```python
import android
import time

droid = android.Android()

droid.addContextMenuItem("Entry 1", "entry1", None)
droid.addContextMenuItem("Entry 2", "entry2", "Tapped 2.")
droid.addContextMenuItem("Off", "off", None)

print "Hit menu to see extra options."
print "Will timeout in 10 seconds if you hit nothing."

while True:  # Wait for events from the menu.
   response = droid.eventWait(10000).result
   if response is None:
      break
   print response
   if response["name"] == "off":
      print "You tapped \"Off\""
      break

droid.clearContextMenu()

print "Context menu cleared"
```

14 https://github.com/miguelpalacio/sl4a/tree/UpdateToApi22/test-scripts/UiFacade

```
time.sleep(5)
print "And done."
```

Figure 4.2 shows the results. Firstly, with the contextual menu populated with the entries we added with `addContextualMenuItem`, and then the contextual menu after removing the entries previously added with `clearContextMenu`. When “Off” is tapped the contextual menu is cleared before the timeout and the script finishes.



```
Hit menu to see extra options.
Will timeout in 10 seconds if you hit nothing.
```

```
Hit menu to see extra options.
Will timeout in 10 seconds if you hit nothing.
Context menu cleared
```

Figure 4.2: Testing the operation of `addContextualMenuItem`

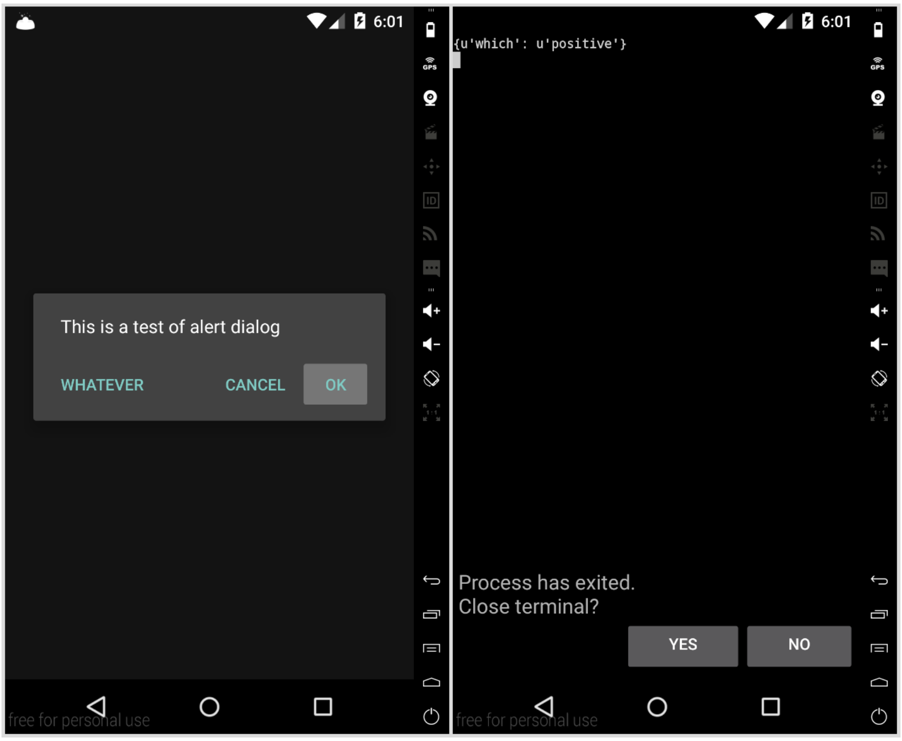
4.2.2 Dialogs

When talking about the number of functions offered, most of the UiFacade functions are related to dialogs. And this is fine, since dialogs provide a very simple but effective way to display information to the user and also receive input from her in order to program scripts. The following subsections will review tests for some of the dialogs offered by SL4A's UiFacade.

4.2.2.1 Alert dialogs

This is the simplest of the dialogs but it is very useful to provide information or for instance, to ask “yes” or “no” questions. The following script tests some of the functions offered for dialogs in general, including setting message and button labels. It also fetches the user answer as it can be seen in figure 4.3.

```python
import android

droid = android.Android()

message = "This is a test of alert dialog"

positive = "OK"
negative = "Cancel"
neutral = "Whatever"

droid.dialogCreateAlert("", message)

droid.dialogSetNegativeButtonText(negative)
droid.dialogSetPositiveButtonText(positive)
droid.dialogSetNeutralButtonText(neutral)

droid.dialogShow()

print droid.dialogGetResponse().result
```


This is a test of alert dialog

```
{u'which': u'positive'}
```

Process has exited.
Close terminal?

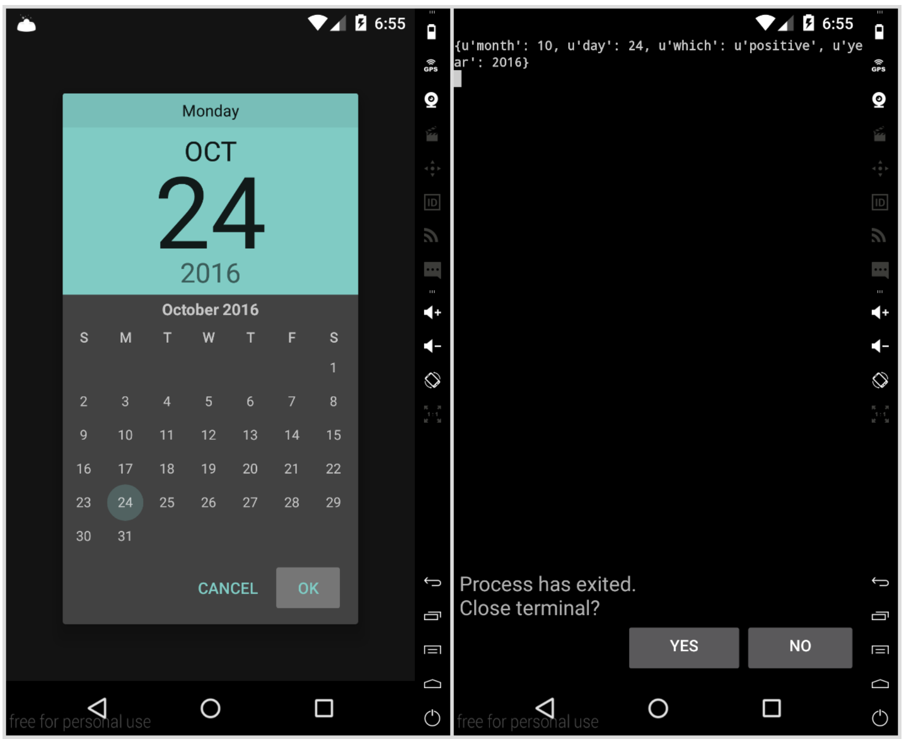
Figure 4.3: Testing alert dialogs and related functions of the UiFacade.

4.2.2.2 Date and time picker dialogs

SL4A also provides date and time picker dialogs. These types of dialogs provide very useful data since it allows the user to specify dates and times that can be used in many ways by scripts.

The following script tests the date picker dialog and also fetches the user response to be used later. Figure 4.4 shows the resulting dialog and the result of the selection of date, please note that the result in this case also provides information about which button the user clicked on. As it was mentioned in chapter 3, this dialog did not dismiss automatically after a date was chosen, so `dialogDismiss` needed to be called explicitly. This problem is now solved and the dialog is dismissed once the user taps on one of the buttons.

```python
import android

droid = android.Android()

droid.dialogCreateDatePicker(2016, 10, 1)
droid.dialogShow()

response = droid.dialogGetResponse().result

print response
```



```
{u'month': 10, u'day': 24, u'which': u'positive', u'year': 2016}
```

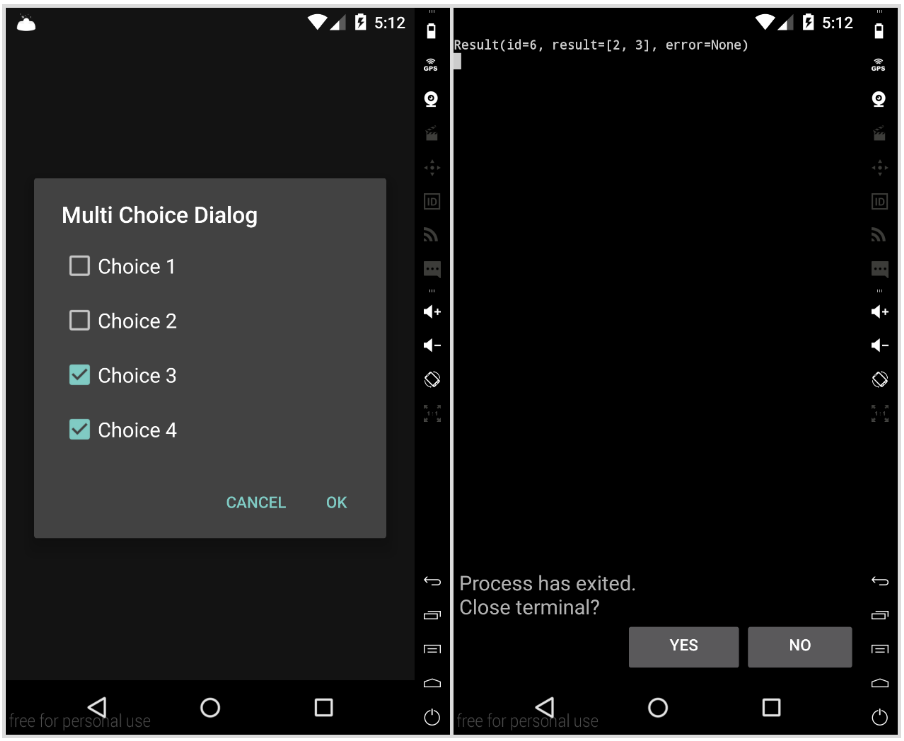
Process has exited.
Close terminal?

Figure 4.4: Testing the date picker dialog function offered by the UiFacade

4.2.2.3 Multi choice dialogs

SL4A provides single and multi choice dialogs. These type of dialogs are very useful to present to the user a series of options from which she can choose one or more. These dialogs are very useful for scripts as well. The following script test this dialog and fetches the response received can be use later for many purposes. Figure 4.5 shows the dialog with the selections made, and also the format of the response later.

```python
import android

droid = android.Android()

title = "Multi Choice Dialog"
droid.dialogCreateAlert(title, "")

droid.dialogSetNegativeButtonText("cancel")
droid.dialogSetPositiveButtonText("ok")

items = ["Choice 1", "Choice 2", "Choice 3", "Choice 4"]
selected = [0, 2]  # Default selection

droid.dialogSetMultiChoiceItems(items, selected)

droid.dialogShow()

selection = droid.dialogGetSelectedItems()
print selection
```



```
Result(id=6, result=[2, 3], error=None)
```

Process has exited.
Close terminal?

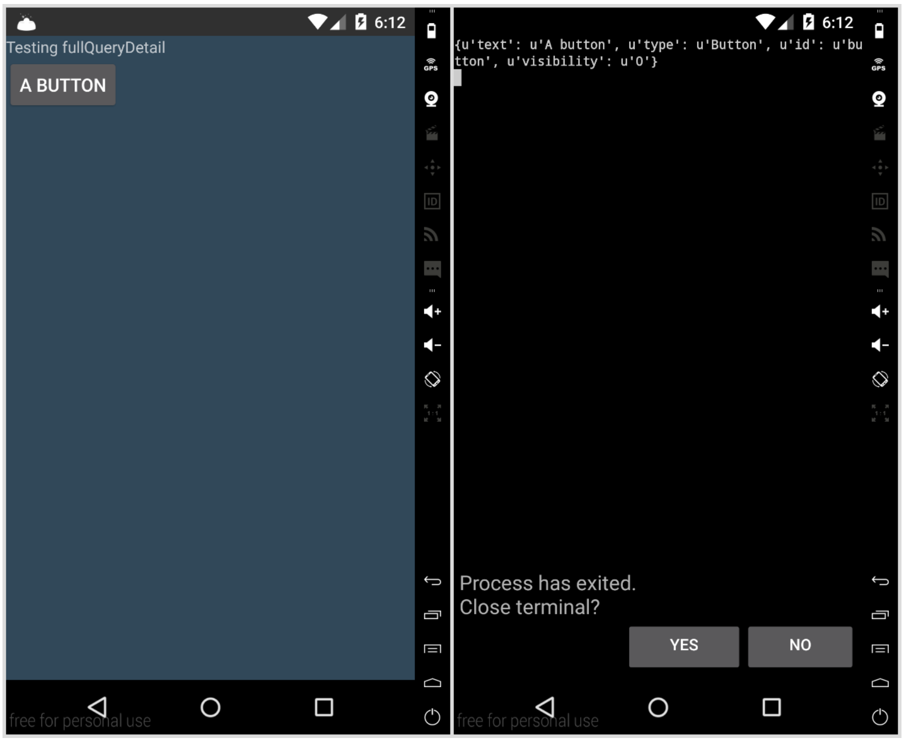
Figure 4.5: Testing the multi choice dialog and `dialogGetSelectedItems` functions offered by the UiFacade

As it was stated in chapter 3, there was a problem with `dialogGetSelectedItems` since it was not returning the user's choices, only whatever thing it was selected by default. This was corrected and now the user input is correctly retrieved.

4.2.3 fullQuery and fullQueryDetail

These functions are used along with `fullShow` (which will be the main topic in section 4.3) and serve to retrieve information about the layout being displayed or a specific item in the same. `fullQuery` retrieves the information of the whole layout, whilst `fullQueryDetail` retrieves the information regarding a specific view. The following test script loads a simple layout with a `TextView` and a `Button` and retrieves the information about the button. Figure 4.6 shows the information retrieved.

```python
import android
import os.path
import time
from string import join

# Get XML layout as a string
f_path = os.path.dirname(os.path.realpath(__file__)) + "/" + "layout.xml"
f = open(f_path)
xml = join(f.readlines())
f.close()

droid = android.Android()

droid.fullShow(xml)

query = droid.fullQueryDetail("button")  # Get view info by passing its id

time.sleep(2)

droid.fullDismiss()

print query.result
```



```
{u'text': u'A button', u'type': u'Button', u'id': u'bu
tton', u'visibility': u'0'}
```

Process has exited.
Close terminal?

Figure 4.6: Testing the `fullQueryDetail` function offered by the UiFacade

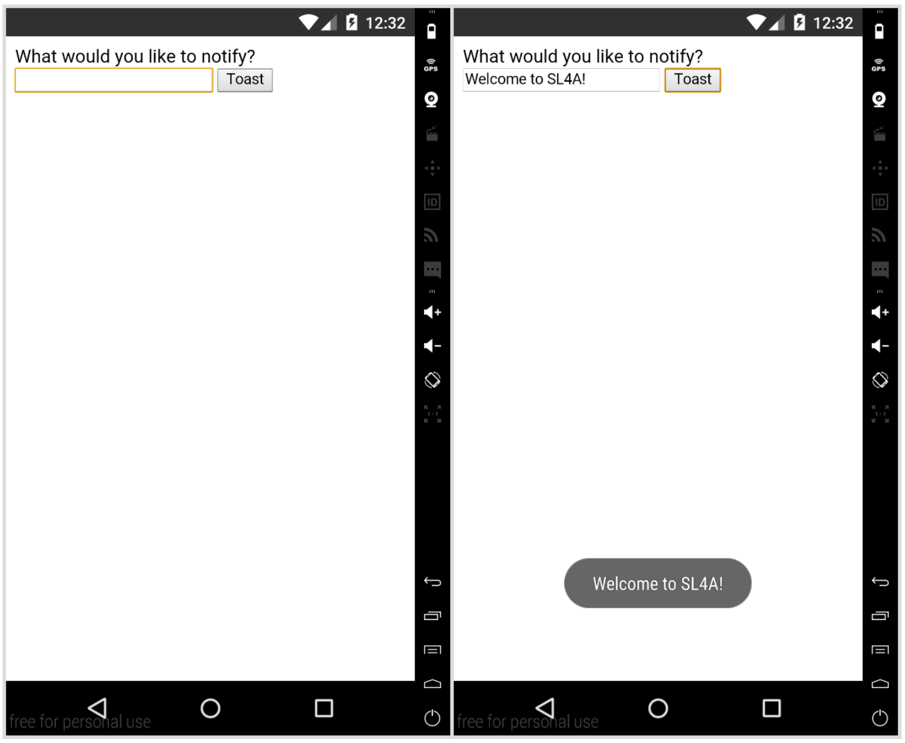
4.2.4 webViewShow

This function is special since it is possible to execute JavaScript code that can make use of the API offered by SL4A. Due to the changes that have experienced Android and that were mentioned in section 3.3.3.4, this function did not work. The following script tests the operation of `webViewShow` with a local HTML file. Please remember that external URLs are now handled by a external browser.

```python
import android

droid = android.Android()

droid.webViewShow("file:///sdcard/sl4a/scripts/Tests/
    UiFacade/webViewShow_test.html")
```

The following code shows the HTML used by the previous script, as well as the JavaScript code that is executed. Figure 4.7 shows the resulting screen after loading the HTML file using `webViewShow`, and then clicking at the button to display a toast with a custom message.

```html
<html>
   <head>
      <title>Toast Notifications</title>
      <script>
         var droid = new Android();
         var toast = function() {
            droid.makeToast(document.getElementById("notify").value);
         }
      </script>
   </head>
   <body>
      <form onsubmit="toast(); return false;">
         <label for="notify">What would you like to notify?</label>
         <input type="text" id="notify" />
         <input type="submit" value="Toast" />
      </form>
   </body>
</html>
```


What would you like to notify?

Toast

What would you like to notify?

Welcome to SL4A!

Toast

Welcome to SL4A!

Figure 4.7: Testing the `webViewShow` function offered by the UiFacade as well as some scripting with JavaScript

4.3 fullSetProperty

This function offered by the UiFacade deserves its own section since it had a great part of the focus of this work and most of the scripts written to test the functionality of this facade were done specifically to test `fullSetProperty`. There are as many scripts for this function as there are view attributes whose correct setting had to be tested. As it happened with the rest of the functions offered by the façade, it is really unthinkable to show in this document all the scripts that test how `fullSetProperty` sets values for all the supported view attributes, so it is recommended to visit the project repository to check a larger number of test scripts.

In the following subsections some of the test scripts will be shown and with them their respective results. These subsections will be divided into the `View` class and some of its subclasses. But before going to the subsections, let us review the following code:

```python
import os.path
from string import join

cur_dir = os.path.dirname(os.path.realpath(__file__)) + "/"

def getXML(filename):
   fin = open(cur_dir + filename, "r")
   lines = fin.readlines()
   fin.close()
   return join(lines)
```

This code is used to load an XML layout and convert it into a string, that then is used by `fullShow` to load and display such layout. Since this functionality will be used heavily by the test scripts, it is better to define a module to use the function. The file corresponding to this module (`xml_loader.py`) must be in the same directory that the scripts that use the `getXML` function.

4.3.1 View class attributes tests

It is impossible to write isolated tests for each attribute. Not only is it unpractical, but also it is necessary to use some attributes in order to test others. Therefore, in this and the following sections the test scripts shown are aimed to test a set of attributes at once.

Since the amount of attributes supported by the View class is huge, only a few of them will be tested by the next script. The following is the layout that will be used. This layout contains one `TextView` and two `Button` widgets.

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:background="#BCA1C4"
   xmlns:android="http://schemas.android.com/apk/res/android">

   <Button
      android:id="@+id/buttonA"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Button A"
      android:textSize="16sp"
      android:gravity="center" />

   <Button
      android:id="@+id/buttonB"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Button B"
      android:textSize="16sp"
      android:gravity="center" />

   <TextView
      android:id="@+id/textView"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Testing View Class"
      android:textSize="14sp"
      android:gravity="center" />
</LinearLayout>
```

The script is centered in testing some of the attributes of the `View` class, however, it can be seen in figure 4.8 that the attributes set for both `TextView` and `Button` classes are correctly handled as well (and thus implicitly testing some `TextView` attributes).

```python
import android
import time
from xml_loader import getXML

xml = getXML("layout_view.xml")
droid = android.Android()

droid.fullShow(xml)

# Short delay to observe changes
time.sleep(3)

droid.fullSetProperty("buttonA", "elevation", "32dp")
droid.fullSetProperty("buttonA", "paddingLeft", "24dp")
droid.fullSetProperty("buttonA", "paddingTop", "24dp")

droid.fullSetProperty("buttonB", "transformPivotX", "16dp")
droid.fullSetProperty("buttonB", "transformPivotY", "16dp")
droid.fullSetProperty("buttonB", "rotation", "10")

droid.fullSetProperty("textView", "visibility", "invisible")

time.sleep(3)
```

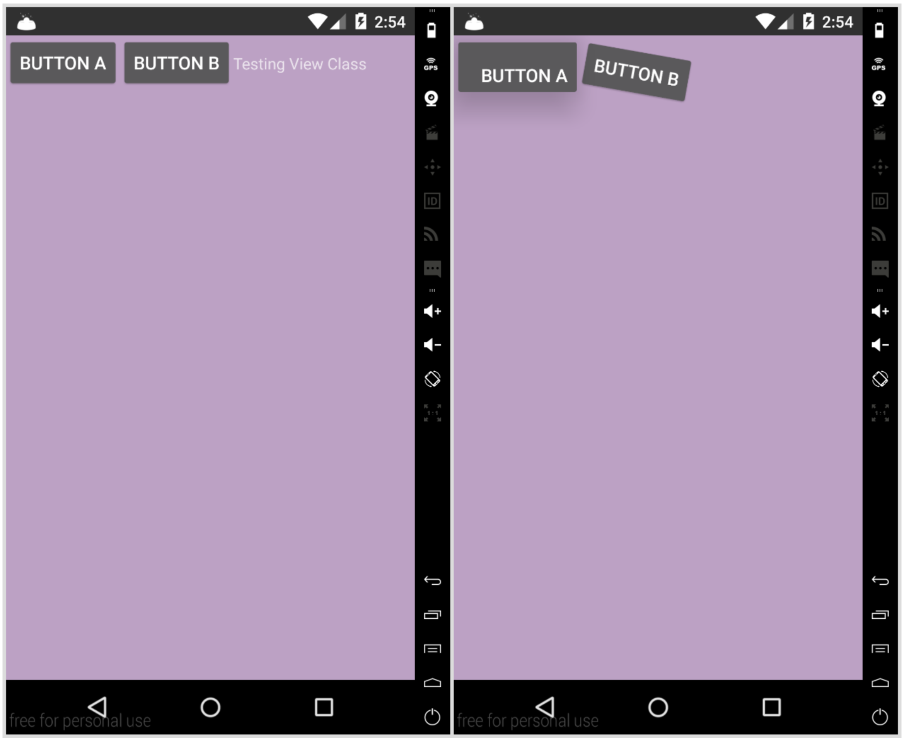


Figure 4.8: Testing attributes for the View class.

As it can be seen in figure 4.8 in the picture to the right, the setting of elevation, paddingLeft, paddingBotton, transformPivotX, transformPivotY, rotation and visibility are correctly handled in run time. The setting of attributes such as background, id, etc, are handled implicitly by `fullShow`. However, if something cannot be handled by `fullShow`, it cannot be handle by `fullSetProperty` either.

4.3.2 TextView class attributes tests

As with the `View` class, a script testing only a few of the `TextView` attributes can be shown not to make this reading too long. In the same way as with the `View` class, there are other attributes that had to be set and that were not part of the `TextView` class in order to perform a clear test. The following code corresponds to the layout file used for this test. Notice that the `EditText` class is a subclass of `TextView`.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:background="#A590AB"
   android:orientation="vertical"
   xmlns:android="http://schemas.android.com/apk/res/android">
   <TextView
      android:id="@+id/textViewA"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="SL4A demo"
      android:textSize="16sp"
      android:gravity="center" />

   <EditText
      android:id="@+id/editText"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Default input"
      android:textSize="16sp"
      android:gravity="center" />

   <TextView
      android:id="@+id/textViewB"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="TextView test"
      android:textSize="16sp"
      android:gravity="center" />
</LinearLayout>
```

The following code corresponds to the script used to test the `TextView` class. Each `TextView` widget of the previous layout was set different attributes. The result can be seen in figure 4.9.

```
import android

import time
from xml_loader import getXML

xml = getXML("layout_text_view.xml")
droid = android.Android()

droid.fullShow(xml)

# Short delay to observe changes
time.sleep(3)

droid.fullSetProperty("textViewA", "textColor", "#ff0")
droid.fullSetProperty("textViewA", "textSize", "13sp")
droid.fullSetProperty("textViewA", "textStyle", "italic")
droid.fullSetProperty("textViewA", "typeface", "monospace")

droid.fullSetProperty("editText", "drawableLeft",
    "@android:drawable/ic_dialog_alert")
droid.fullSetProperty("editText", "inputType", "textPassword")
droid.fullSetProperty("textViewB", "shadowColor",
    "@android:color/holo_red_dark")
droid.fullSetProperty("textViewB", "shadowDx", "45")
droid.fullSetProperty("textViewB", "shadowDy", "45")
droid.fullSetProperty("textViewB", "shadowRadius", "5")

time.sleep(3)
```

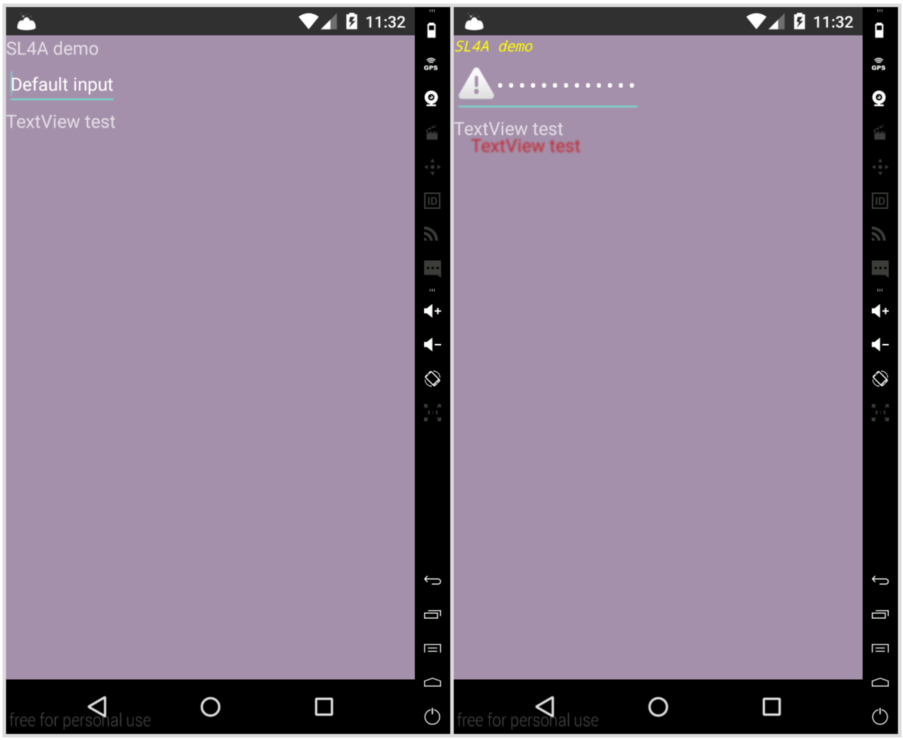


Figure 4.9: Testing attributes for the `TextView` class

4.3.3 Tests for ViewGroup class its subclasses

`ViewGroup` is one of the most important classes since it is the super class of all the views that hosts other views, i.e., the layout views. The most common group view is `LinearLayout`, and it has been the one used for the previous tests, so it can be safely assumed that it has been well tested. `LinearLayout` does not have many attributes by itself to be tested, however, it is worth mentioning that for the `TextView` test the layout orientation changed to vertical.

Within `ViewGroup` there is an inner class that defines the layout_height and layout_width properties. As it can be seen in the previous tests, these two attributes are used by all the views in order to set their size. However, all the tests have been using wrap_content as the special value for both attributes. The following layout uses a richer set of values in order to test their correct operation.

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:background="#A590AB"
   android:orientation="vertical"
   xmlns:android="http://schemas.android.com/apk/res/android">

   <Button
      android:layout_width="150dp"
      android:layout_height="wrap_content"
      android:text="Button A"
      android:textSize="16sp"
      android:layout_marginLeft="32dp"
      android:gravity="center" />

   <Button
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="Button 3"
      android:textSize="16sp"
      android:gravity="center" />

   <Button
      android:layout_width="200dp"
      android:layout_height="200dp"
      android:text="Button C"
      android:textSize="16sp"
      android:layout_gravity="center"
      android:gravity="center" />
</LinearLayout>
```

Also notice that there are a couple of more attributes that have not been tested before. Such attributes are layout_gravity, layout_marginLeft (also applies for layout_marginRight, layout_marginBottom, etc). The result can be seen in figure 4.10 after the layout is loaded using `fullShow`.

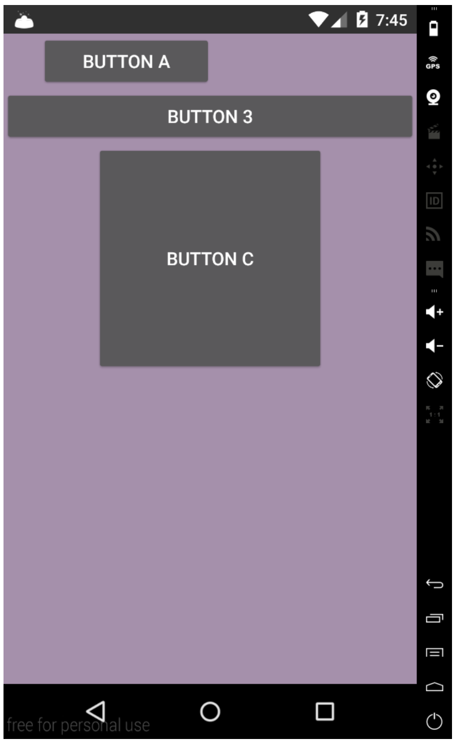


Figure 4.10: Testing some attributes defined in the `ViewGroup` class

Continuing with another `ViewGroup` subclass, it is now the turn for `TableLayout`. This kind of layout is very useful to organize a series of widgets in a table manner. The following lines of code represent a simple table layout:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout
   android:id="@+id/tableLayout"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:background="#A590AB"
   android:orientation="vertical"
   android:shrinkColumns="*"
   xmlns:android="http://schemas.android.com/apk/res/android">

   <TableRow
      android:layout_width="match_parent"
      android:layout_height="wrap_content" >

      <Button
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="r1c1"
         android:textSize="16sp"
         android:gravity="center" />
      <Button
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="r1c2"
         android:textSize="16sp"
         android:gravity="center" />
   </TableRow>

   <TableRow
      android:layout_width="match_parent"
      android:layout_height="wrap_content" >

      <Button
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="r2c1"
         android:textSize="16sp"
         android:gravity="center" />

      <Button
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="r2c2"
         android:textSize="16sp"
         android:gravity="center" />

      <Button
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="r2c3"
         android:textSize="16sp"
         android:gravity="center" />
   </TableRow>
</TableLayout>
```

This layout has all its columns shrunk, as it can be seen in the `TableLayout` definition where the property shrinkColumns is set to `“*”`, meaning that this property applies to all the columns of the table layout. This can also be done via scripting by calling `fullSetProperty(“tableLayout”, “shrinkColumns”, “*”)`. In fact, figure 4.11 shows the layout just after it is loaded and after `fullSetProperty(“tableLayout”, “stretchColumns”, “0,2”)` is called.

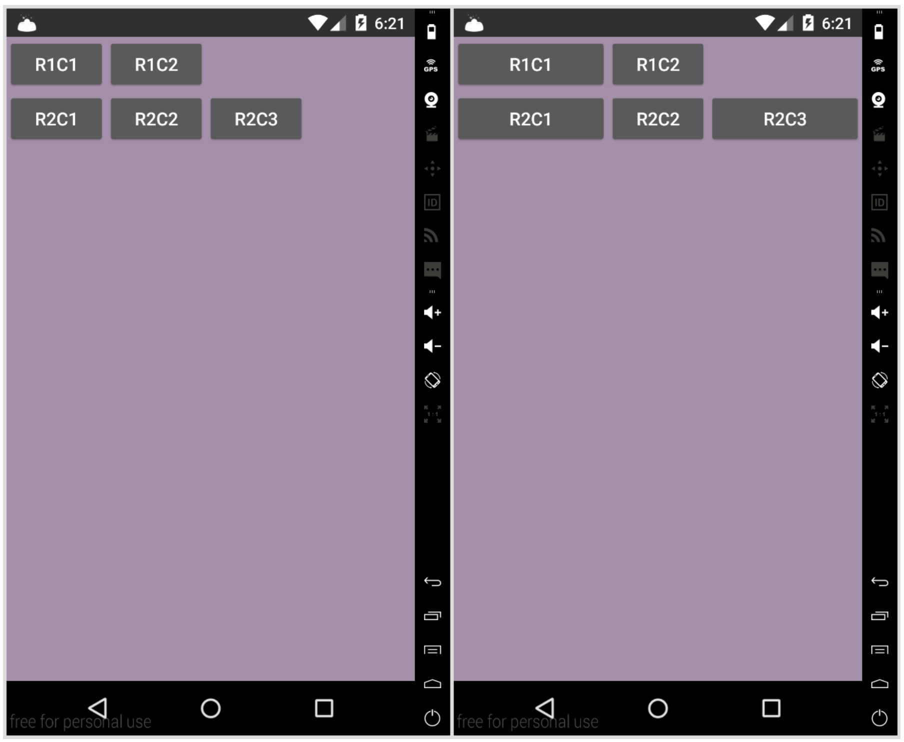


Figure 4.11: Testing attributes of the `TableLayout` class. A `ViewGroup` subclass.

The last test of this section is reserved for `RelativeLayout`. As `LinearLayout`, `RelativeLayout` is also a very important and widely used subclass of `ViewGroup`. In this kind of layout, the position of the views is relative to each other. The following lines of code represent a simple relative layout:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:background="#A590AB"
   xmlns:android="http://schemas.android.com/apk/res/android">

   <ImageView
      android:layout_width="200dp"
      android:layout_height="200dp"
      android:layout_alignParentTop="true"
      android:layout_centerHorizontal="true"
      android:src="file:///sdcard/download/polimi_logo.png" />

   <TextView
      android:id="@+id/textView"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_centerVertical="true"
      android:layout_alignParentStart="true"
      android:text="RelativeLayout Test"
      android:textSize="16sp" />

   <EditText
      android:id="@+id/editText"
      android:layout_width="180dp"
      android:layout_height="wrap_content"
      android:hint="Please type here"
      android:layout_below="@id/textView"
      android:layout_toEndOf="@id/textView"
      android:textSize="16sp" />

   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_alignParentBottom="true"
      android:layout_toStartOf="@id/editText"
      android:text="Button"
      android:textSize="16sp" />
</RelativeLayout>
```

The result after it is inflated can be observed in figure 4.12. Please notice that the problems mentioned in section 3.3.4.1 are now gone and there are no issues by setting attributes such as layout_alignParentBottom.

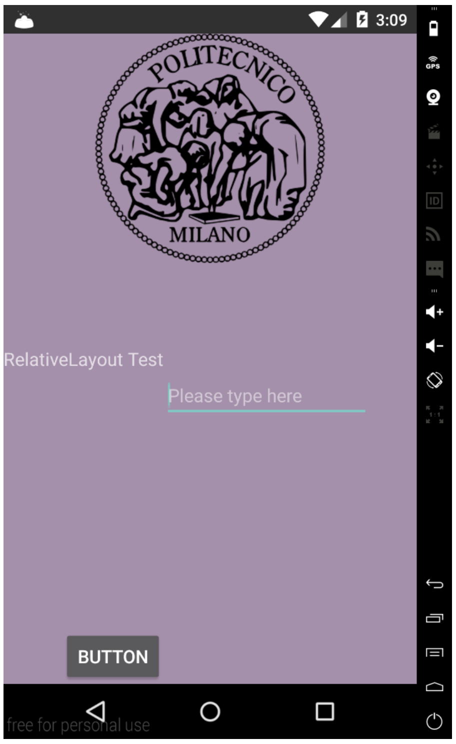


Figure 4.12: Testing attributes of the `RelativeLayout` class.

4.4 Toolbar

This last section shows the script used in order to test the `Toolbar` related functions that were added to the UiFacade. The following is the layout used for testing these functions.

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:background="#ff314859"
   android:orientation="vertical"
   xmlns:android="http://schemas.android.com/apk/res/android">

   <android.support.v7.widget.Toolbar
      android:id="@+id/toolbar"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:background="#BBB"
      android:elevation="4dp" />
</LinearLayout>
```

Please observe the way in which the `Toolbar` widget is used. This is the actual way in which it is used in standard Android projects. The following script is in charge of testing the new `Toolbar` functions. The result is the one seen in figure 3.14.

```python
import android
from xml_loader import getXML

xml = getXML("layout_toolbar.xml")
droid = android.Android()

droid.fullShow(xml)

droid.fullSetToolbarTitle("toolbar", "Toolbar Title")
droid.fullSetToolbarSubtitle("toolbar", "I'm a subtitle!")
droid.fullSetToolbarTitleColor("toolbar", "#0A0")
droid.fullSetToolbarSubtitleColor("toolbar", "#DDD")
```

And with this test this chapter finalizes. There could be a plethora of other tests but as it was aforesaid, there is no extra benefit in putting all those tests in this document.

The following chapter is aimed at looking at the future work that has to be carried out in order to keep SL4A alive and to make it grow so that it becomes more attractive for a higher number of Android developers. Moreover, some conclusions about the impact of this work will are reviewed.

Chapter 5
Conclusions and Future Development

As it has been seen throughout the chapters of this text, the Scripting Layer for Android development got stalled after the initial work of Damon Kohler. This project aimed to bring SL4A back to life again by updating the project to support the latest versions of the Android platform, and this objective was achieved.

The purpose of this chapter is to summarize what was accomplished with this project, and also to show the conclusions that were reached after the execution of it. Nevertheless, since SL4A is far from being complete, some ways in which this project can become a greater alternative for developers will be analyzed.

5.1 Work summary

- This work started by carrying out a research about the current status of the project. Due to the closure of Google Code where SL4A was initially hosted, the codebase was migrated towards Github. In the Github repository, different branches started and after an analysis in the commit history and activity of those branches, it was decided to fork kuri65536's branch. This user had already migrated the codebase from Eclipse to Android Studio, and given the size and complexity of the project, that was a huge time saving.

- Afterwards, the focus was to bring SL4A to the last version of Android by then: Android Lollipop. This required to replace many lines of code, that not only were obsolete but also were breaking the correct operation of the application.

- However, after the update of the target API to Android Lollipop, the interaction with the application was clumsy due to the outdated interface. One of the most highlighted features of Lollipop was the introduction of a new UI paradigm called Material Design. The work went in that direction in order to make SL4A be a more standard application, increasing the ease of use and familiarity for new users.

- After the application backbone and interface were updated to Android Lollipop and followed its guidelines, all the work was focused in updating the SL4A API, which was the main objective of this thesis. Initially, the idea was to carry out a review of all the functions offered by SL4A; however, due to the enormous amount of functions offered by the vast number of different façades, all the work was done in the UiFacade. The UiFacade was thoroughly reviewed, several of the functions offered were not working properly and they were corrected. The functions that got the most attention were `fullShow` and `fullSetProperty` due to its complexity and the possibilities they provide: creating applications with complex user interfaces.

5.2 Conclusions

- The longer the time a project has been stalled, the more difficult it is to bring it back to life. This is specially true in mobile projects were the pace of development is really fast. With tons of new features in each release of the Android platform, the Scripting Layer for Android was very outdated after approximately three years of no development. It required, and still requires a lot of effort to make it fully up do date with current versions of the Android platform.

- The more devices an Android project supports, the larger its codebase and also the greater amount of obsolete code that has to be kept. Supporting a great number of versions of the Android API is really good since the number of possible users increase; nevertheless, it also makes it difficult to maintain a project. Moreover, in most occasions supporting a great number of Android versions is not practical since most users are also up to date in terms of the Android platform. A nice way to determine which platforms to support in a project is to visit the Android Dashboards utility that Google provides on its page for Android developers.

- Due to the characteristics of the operation of the Scripting Layer for Android, everything regarding the scripts executed must be done in runtime. This is very different to the way in which Android normally builds applications (statically via compilation). Performing everything in runtime is a clear challenge when compared with static analysis. Not everything is possible to be done in runtime and in many times advanced techniques such as type introspection and reflection need to be used in order to keep the code neat and compact.

- Considering the amount of functions that must be implemented in SL4A in order to provide a rich API, it is necessary to reuse as much code as possible otherwise the code can grow really quickly rendering it unmaintainable. This can be seen in `ViewInflater` where the amount of code increased a lot in spite of the efforts done to reuse as much code as possible. Reflection and type introspection were used in order to keep the code minimal, but it was not enough to prevent its growth. Other techniques ought to be sought to achieve a better code reuse.

- Giving support to all (or at least most) Android features require a very long and tiring review of the whole Android documentation. This increases the difficulty of the project since the functionalities provided by Android are naturally many. Each of them must be studied carefully in order to understand and implement it correctly. This makes this project extremely difficult since the amount of work is immense and the number of contributors is rather limited.

- The use of this application is to be limited to developers only. Its use may lead to stability and security problems, so only people with certain degree of knowledge should use it. Due to this nature it is very unlikely that SL4A can be distributed from Google Play. Fortunately, installing applications on Android that are not distributed officially is as simple as downloading an installer from the Internet and execute it. The distribution can be done from the project's repository as it has been done so far. Once the project gets some exposure it can reach more developers.

- Understanding the whole codebase of this project would require a lot of effort, and it is very unlikely that without being a paid contributor someone can achieve this in a sensible amount of time. That is why it is better to understand only some pieces of it and tackle them. This is precisely what it was done in this work: getting certain grasp of the way in which the user interface work in order to modify, and afterwards getting familiar with the RPC protocol used and the façades with the purpose of enhancing the UiFacade.

- From it was said in the previous point, understanding the whole codebase of the project is a daunting task. It is better to align efforts with other developers in order to distribute the work and improving the project from different flanks. Unfortunately, this is something too ideal. It is normal very difficult to achieve this sort of synergy since people have their own goals. Bigger projects have suffered this a lot and that is why we can see many projects with many forks attempting to achieve similar things. For this reason it was decided to keep this thesis isolated at least during its execution.

5.3 Future development

The Scripting Layer for Android is far from being complete. In fact, it will never be as long as the development of the Android platform continues.

Regarding to what was done in this project, there is still much work to be done in the UiFacade. There are still many view attributes that can be supported, and many others are supported but their support can be improved. Moreover, the UiFacade is not only about view attributes. It was already seen that dialogs are quite well supported as well. In this thesis a new feature was added to support more complex user interfaces: the `Toolbar` widget.

Additionally, there are other widgets that can be offered in SL4A and will provide a way to make complex applications. Tabs and the navigation drawer are among those widgets; however, a lot of work has to be done in order to provide a decent support of them. Furthermore, the `Toolbar` support is far from be complete, it is necessary to enable the use of its pop-up menu to make it more than a simple decoration.

As it was done with the UiFacade, it is necessary to perform a complete review of the other façades that conform the SL4A API. Such façades must have obsolete code, and many new functionalities and features for new sensors provided by the Android platform are not supported by them. For instance, Android Beam for near field communication (NFC) was introduced with Android Ice Cream Sandwich and it is not supported by SL4A.

Due to time constraints, it was no possible to extend the support to the new Android Marshmallow. The main issue with Android Marshmallow is the way in which permissions are managed. Not only are they handled in the Android Manifest as it has been traditionally done, but now they must be handled in runtime as well. Permissions are disabled by default despite the fact that users grant permissions upon installing applications. These permissions must be enabled by the user in runtime, once the application requires them and asks the user to enable them. In order to achieve this, it is required to have a very profound knowledge of the whole codebase of the project in order to know where this permissions must be asked to the user.

Since the number of features offered by SL4A is large, the number of permissions required is also large. And since the codebase is huge, finding where these permissions must be asked is a exhausting task. Nonetheless, this job needs to be done in order to keep the project alive, so it is of utter importance to make the application compliant with the Android Marshmallow guidelines.

Regarding the scripting languages that SL4A supports, theoretically many are supported but only very few are working. Thanks to the nature of SL4A, each scripting language engine for Android is a work that should be carried out independently from it. In order to truly support the languages offered by SL4A, each of script engine needs to be tackled independently in order to make them work with Android itself.

Despite the fact that aligning efforts with other developers is rather complex, it is strictly necessary in order to maintain the Scripting Layer for Android alive. As it has been stated, the size of this project is enormous and there is no way in which a single developer can cope with all the work that needs to be done, even if paid. Therefore, it is of utter importance to create a community of developers who want to make contributions to the project.

It is critical to get more visibility in the Android developer scene once the project is more robust. This can be achieved by developing all sorts of sample applications and writing about the possibilities given by the Scripting Layer for Android. Moreover, the bigger the community of developers, the higher the likelihood of the growth of SL4A itself since the number of possible contributors will also increase.

References

Lucas Jordan, and Pieter Greyling. Practical Android Projects, Chapter 5: Introducing SL4A: The Scripting Layer for Android. 2011th Edition. New York: Apress, 2011.

Paul Ferrill. Pro Android Python with SL4A. 2011th Edition. New York: Apress, 2011.

Carl Smith. “Python for Android: The Scripting Layer (SL4A)”. Online posting. December 5, 2013. Python Central.
<http://pythoncentral.io/python-for-android-the-scripting-layer-sl4a/>

Damon Kohler. “Introducing Android Scripting Environment”. Online posting. June 8, 2009. Google Open Source Blog.
<http://google-opensource.blogspot.com.co/2009/06/introducing-android-scripting.html>

Paul Barry. “Python for Android”. Online posting. April 30, 2011. Linux Journal.
<http://www.linuxjournal.com/article/10940>

Android Developers site. Online Android Reference.
<https://developer.android.com/reference/packages.html>

Android Developers site. API Guides – API levels.
<https://developer.android.com/guide/topics/manifest/uses-sdk-element.html>

The Python Language Reference. Data model – The standard type hierarchy.
<https://docs.python.org/2/reference/datamodel.html#the-standard-type-hierarchy>

Mark Friedman. “App Inventor for Android”. Online posting. July 12, 2010. Google Official Blog.
<https://googleblog.blogspot.com.co/2010/07/app-inventor-for-android.html>

Qt Documentation. Qt for Android.
<http://doc.qt.io/qt-5/android-support.html>

"Announcing RubyMotion 3.0: Android Platform, WatchKit Apps, and More". Online posting. December 11, 2014. RubyMotion News.

<http://www.rubymotion.com/news/2014/12/11/announcing-rubymotion-3.html>