TODO
2015-2016

TODO

836897 Simone Graziussi

TODO

Abstract

TODO

Contents

Part I

Introduction

1 Abstract

2 Testing

Testing is the process used in software development to assess the validity of functional and non-functional requirements of an application. Although this verification is able to guarantee the correctness of the tested components within the specific conditions described by the test cases, testing cannot assess the validity of the whole application in every situation, because this would require an unfeasible amount of detail. For this reason, testing is mainly used to discover software bugs in particular situations and to reach an acceptable confidence that the program works as expected under the most common conditions.

Test cases require a mechanism to actually determine the test outcome, i.e. to tell if the application behaves as expected during the validation process. This mechanism, called oracle, should ideally be complete but avoiding over-specification, while also being efficiently checkable [1]. Oracles can assume many forms, for example the behavior of the application is compared with the technical specifications (e.g. documentation), it is automatically checked by the system thanks to some constructs that allow the developer to specify the test conditions or it can even be manually evaluated by a human being.

Test cases can be designed from different points of view. In particular, we can have:

- White-box testing: the focus is on the internal structure of the application, i.e. tests are defined at the source code level (*how* the software behaves).

- Black-box testing: examines the external behavior of the application without considering the actual implementation, i.e. tests are defined at the user level (*what* the software does).

- Grey-box testing: combination of white-box and black-box testing. Tests are defined with a partial knowledge of the internal structure of the application (i.e. how the main components interact and the general algorithms used).

There are usually four levels on which test cases can be defined:

- Unit Testing: focuses on a specific unit of the program, for example a single function/method/class used in the source code. Usually, it follows a white-box testing approach and it is performed during development to build a program using units guaranteed to work.

- Integration Testing: tests interactions between components of the application, i.e. it usually puts together the units tested in the previous step to see if they work well together.

- System Testing: it considers the program as a whole to see if it meets all requirements and quality standards.

- Acceptance Testing: final step that decides if the application is complete and ready to be deployed (e.g. released to the public)

3 Android

Android is a operating system (OS) developed by Google. It is designed primarily for touchscreen mobile devices like smartphones and tablets, but recently it was extended to televisions (Android TV), cars (Android Auto) and smartwatches (Android Wear).

The OS is a multi-user Linux system in which each app is a different user. It is characterized by the so called sandboxing mechanism: each process runs in its own virtual machine (VM) and so every app runs in isolation from the other applications. This means that, by default, an application can access only a limited set of components and cannot access parts of the system for which it does not have permissions.

Android applications are developed in a Java language environment. The Android Software Development Kit (SDK) compiles code, data and resources into a package called APK, which is used by the devices to install the application.

4 Android App Components

Android applications are built by five main components, each with its specific purpose:

- Activity: an activity represents a single "action" that the user can take and, since almost all activities interact with the user, they provide a screen with a user interface. Each activity in the application is independent from the others, but it is of course possible to start an activity (for example when the user clicks on a button) from another to build the application flow.

- Fragment: this component was introduced in Android 3.0 (API level 11) to support dynamic and flexible UI on large screens, for example on tablets. A Fragment represents a "portion" of an Activity, with its own state and UI. Each Activity can contain multiple Fragments at a time, Fragments can be added/removed at runtime and each Fragment can be reused in more than one Activity. A Fragment can only be instantiated inside an Activity.

- Service: a service is a component that is executed in background and, as such, it provides no user interface. It is used to perform complex computations or to interact with an external API (e.g. via the network). The advantage of this approach is that another component (e.g. an Activity) can start and interact with a Service in order to avoid blocking its UI with computationally intensive operations.

- Content Provider: a content provider allows to store and retrieve data in some persistent storage location, for example a local SQL database or a remote repository. The provided data can be shared among different applications or private to a specific one.

- Broadcast Receiver: a broadcast receiver responds to global messages, i.e. messages received by all applications on the device. These events may be fired by the system (e.g. the device has just rebooted) or by a single application (e.g. some data is available), and then intercepted by the applications interested to them.

Activities, services and Broadcast Receivers are started asynchronously by messages called Intents. This allows not only an application to start its own components, but also to call on other applications. For example, a game may start its internal GameService to manage the game loop, but also send an intent to a social network application to share the game progress.

Part II
Android Testing State of the Art

Android tests are mainly based on JUnit [2], a testing framework for Java. It allows to create classes called test cases that contain methods annotated with `@Test`, each representing a test.

5 Unit Testing

White-box unit testing in Android can be

- Local: it runs on the local development machine (i.e. the computer where the application is coded). It has the advantage of being fast (avoids the overhead to load the application in a device/emulator), but can be exploited only if the tested unit has no dependencies or simple dependencies. This means that the test case should not use any device-specific features (e.g. expect a sensor input) or, if it does so, they should be minimal since they need to be mocked using for example tools like Mockito [3].

- Instrumented: it is executed in a physical device or on an emulator and so has access to instrumentation information, such as the Context (information about the application environment). It is slower than the previous case but it's more convenient if the unit dependencies are too complex to mock.

- Hybrid: the external library Robolectric [4] tries to take the advantages of the previous two approaches, i.e. it runs "instrumented" tests on the local machine, without mocking. As reported on the website, Robolectric allows a test style that is closer to black box testing, making the tests more effective for refactoring and allowing the tests to focus on the behavior of the application instead of the implementation of Android.

6   UI Testing

Android also provides a way to test User Interface (UI) to see if it behaves as expected. This type of testing can be defined as a grey-box approach: the application is tested at the user level without considering the actual implementation of the UI, but the definition of the test cases may require to know some information on the internal structure, for example the IDs of the buttons to be clicked.

- UI testing on a single app: the Espresso library provides APIs for writing UI tests to simulate user interactions. In general, defining a test case means building a series of `onView(Matcher).perform(ViewAction).check(ViewAssertion)` instructions, i.e. select a particular View that matches some description (e.g. a button with "Start" text), perform one or more actions on it (e.g. click) and check if some conditions are true (e.g. if after the click the button text changes). The main advantages of Espresso are that the test cases are easily readable and understood, and that it has automatic synchronization (before performing an action it waits for the previous ones to be completed, i.e. for the main thread to be idle).

- UI testing on multiple apps: the UI Automator library allows to test if the developed application interacts correctly with the system or other apps (e.g. the application may request an image, the Camera app is opened, the picture is taken and then the control goes back to the original application).

7   Runner Tools

In addition to the testing frameworks described in the previous sections, Android also offers some tools to run and test an application without accessing the source code (black-box testing):

- UI Exerciser Monkey: allows to run an application on a physical device or emulator generating a pseudo-random (but repeatable) streams of user events (e.g. clicks) and system-level events (e.g. start call). The developer can set several options like target package, probability of certain events, etc.

- monkeyrunner: controls a device or emulator from a workstation by sending specific commands and events defined as a Python program. It also allows to take screenshot during the test execution and store them on the workstation.

8   Assertions in Android

An assertion is a statement at a specific point in a program that enables to check an assumption. It is expected to be true, but if a bug is present the assertion will fail and the system will throw an error. Assertions are test oracles that specify what the application does rather than how.

In Java, the assertion statement is

`assert Expression1 : Expression2;`

where `Expression1` is the boolean expression to be checked and `Expression2` is an expression that has a value to better describe the error. If `Expression1` is false, an `AssertionError` exception is thrown.

Many frameworks, including JUnit itself, offer several utility methods to easily write more complex assertions without using the `assert` keyword. For example, the method `assertEquals(String message, Object expected, Object actual)` checks that the two given objects are equal and, if not, throws an `AssertionError` with the given message.

Since Android has its own virtual machine ART, which is not compatible

with JVM from Oracle, the `assert` keyword is not supported by default[1]. For this reason, in an Android application assertions are usually created exploiting JUnit-like methods that directly throw an `AssertionError` if the check fails. This means for example using the assertions provided by Android (via `Assert`, `MoreAsserts` and `ViewAssert` classes), the JUnit framework itself or other external libraries.

9 What's missing in Android testing (?)

?

Part III
Background

10 Introduction to Events

11 Background/Motivation

12 Three Levels of Testing

Part IV
Lifecycle Testing

13 Introduction

The focus this part of the thesis is on a specific type of events that are generated by Android applications: lifecycle changes. As it will be explained later in detail, handling the lifecycle of the app components like Activities and Fragments is a critical aspect of Android development, and so this type of events deserves a separate in-depth analysis.

In particular, two approaches for lifecycle testing will be proposed: static analysis and dynamic analysis. The former will try to analyze the source code of the application to detect possible issues related to the handling of some components in accordance to the host Activity/Fragment lifecycle, for

---

[1]It is however possible to manually enable assertions on a given device using for example the Android Debug Bridge command `adb shell setprop debug.assert 1` (assertions will be enabled for all applications until the device is rebooted), but the Android developers themselves discourage their use

example failing to release a resource that was previously acquired. The dynamic analysis approach, instead, will focus on pre-generated test cases that allow the developer to express some actions to be performed and conditions to be checked in common lifecycle events sequences, which will be used in several tests to assess the robustness of the application.

The following sections will present first a detailed explanation of the main concepts of lifecycle in Android applications, then the static approach will be analyzed and finally the dynamic approach will be described.

14 Lifecycle

14.1 Component Lifecycle

App components in Android such as Activities, Fragments and Services are characterized by their lifecycle, i.e. the current runtime state. In general each component is started and then destroyed, but some can also be paused and resumed, and go through several other states. When a lifecycle state transition happens, the Android system allows the developers to implement some callbacks (Java methods, usually overridden from the component class) in the component implementation to manage their behavior in those particular situations.

14.2 Activity Lifecycle

An Activity can be in three static states:

- Resumed: the Activity is visible and can receive user input

- Paused: the Activity is *partially* hidden by another visual component, for example a notification dialog, and has lost the focus. When the activity is paused it cannot receive any user input o execute code.

- Stopped: the Activity is completely hidden to the user, i.e. it is in the background. Like in the previous case, the activity cannot receive inputs or run code. In this state the Activity is still "alive": the state (e.g. member variables) is retained and the Activity can be later restarted.

An Activity can also be in two transient states:

- Created: the Activity has been instantiated and will soon become Started

- Started: the Activity has been initialized and will soon become Resumed

10

Finally, an Activity can be Destroyed. In this "state" the Activity instance is dead. An Activity can for example be destroyed by the OS when it is stopped and the device needs to free resources.

The Activity class provides several callbacks for lifecycle changes. These methods can be implemented by the developer to manage the state of the Activity: for example `onCreate()` can be used to initialize the UI components, `onStop()` to free resources for the other applications. Note that not all callbacks need to be explicitly implemented for each activity: actually simple Activities usually implement only `onCreate()`.

This image shows a simplified scheme of the Activity lifecycle transitions with the corresponding callbacks.
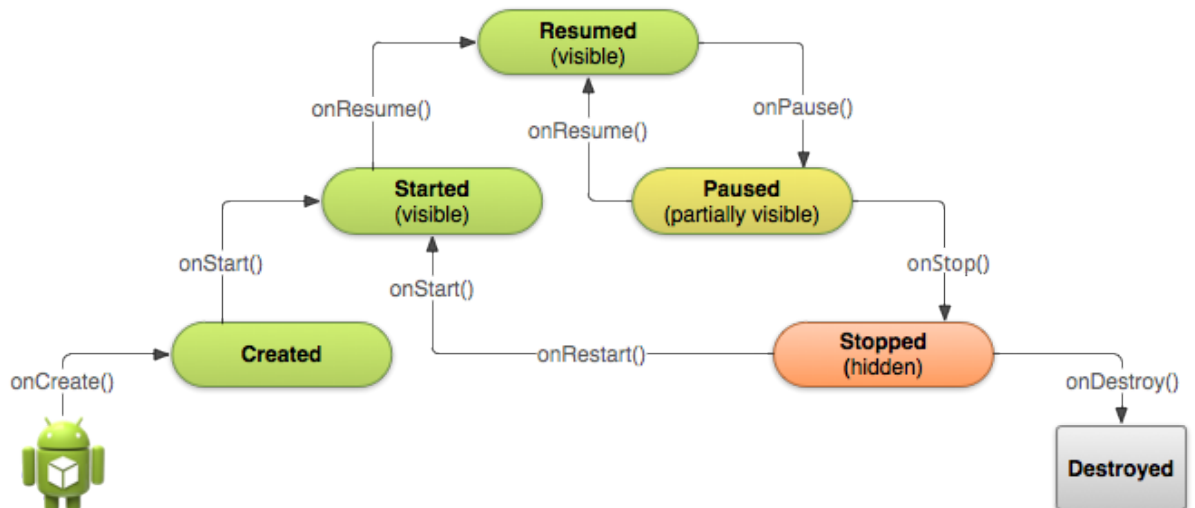
[TODO extend with more detailed explanations of callbacks, save instance, etc.? More detailed diagram?]

14.3 Fragment Lifecycle

The lifecycle of a Fragment is closely related to the lifecycle of the Activity that contains it: for example, when an activity is paused, all the contained Fragments are paused too. In addition to this, however, Fragments can go through lifecycle changes independently of their host Activity: in particular, since fragments can be dynamically added and removed at runtime, they can be created and destroyed while the Activity is running. Moreover, the developer also has the option to store removed fragments in the so-called backstack and be able to restore them later for example when the user presses the "back" button.

Like Activities, Fragments are characterized by three static states:

- Resumed: the Fragment is visible and can receive user input

- Paused: the Activity that contains this Fragment is *partially* hidden by another visual component and has lost the focus.

- Stopped: the Fragment is completely hidden to the user. This can happen if the host Activity is also not visible (i.e. in background) or if the Fragment has been stored in the backstack. Like Activities, stopped Fragments are still "alive" and their state is retained.

Lifecycle management for Fragments is very similar to the one for Activities, since all callbacks are the same. Fragments provide however some additional methods to manage the interaction with the host Activity: for example `onAttach()` is called when the Fragment is linked to an Activity, `onCreateView()` when the Fragment is ready to build its UI, etc.

14.4   Managing the Lifecycle

Handling the components lifecycle is a critical aspect in developing an Android application and it is often source of bugs or unexpected behaviors. For example, properly implementing Activity/Fragment lifecycle methods ensures that the app

- does not waste system resources (e.g. device sensors) while the user is not interacting with it

- stops its execution when the user leaves the application (for example a game should pause if the user receives a phone call)

- retains its state if the user leaves and then returns to the application (e.g. a messaging app must keep a partially written message even if the user puts the app in background for a moment)

- does not crash or loses user progress when lifecycle changes occur (e.g. an app that does not correctly manage lifecycle may crash with a NullPointerException if some internal component was destroyed during `onStop()` but not restored during `onStart()`)

- adapts to configuration changes (like a device rotation between landscape and portrait modes) without losing data or crashing

In general, for Activities and Fragments the developer should make sure to

- during `onCreate()`

    - set the layout resource defining the UI

- initialize View components (e.g. add click listeners to buttons)
  - initialize the component logic (e.g. class-scope variables)
  - restore the previous state (if any) saved during `onSaveInstanceState()` [this can be alternatively performed during `onRestoreInstanceState()`]

- during `onRestart()`

  - Requery raw Cursor objects previously deactivated during `onStop()`

- during `onStart()`

  - acquire "secondary" resources (e.g. Broadcast Receiver)
  - verify system features (e.g. GPS enabled), because they may change when the application is in background

- during `onResume()`

  - start animations and similar CPU-intensive operations
  - acquire "critical" resources (e.g. camera, sensors)
  - should *not* restart on-going operations that require user visibility (e.g. games, videos), but let the user decide when to do so

- during `onPause()`

  - commit unsaved changes made by the user (e.g. save a draft for an unfinished email), if it does not require too much time
  - stop animations and other things that may be consuming CPU
  - stop on-going operations that require user visibility (e.g. games, videos)
  - release "critical" resources (e.g. camera, sensors)
  - should *not* perform any long running operation, the call to this method must be very quick

- during `onStop()`

  - release "secondary" resources (e.g. Broadcast Receiver)
  - perform CPU-intensive shutdown operations (e.g. write unsaved data to database)

- during `onDestroy()`

  - stop background threads created during `onCreate()`
  - release long-running resources that could create memory leaks
  - should *not* save data, because the method may not be called in all situations

14.5   Existing Lifecycle Testing

Due to the importance of lifecycle handling, some approaches to test applications focusing on this aspect have been developed.

One example is THOR [5]. The idea of the tool is to run pre-existing UI test cases (defined in Robotium or Espresso) in adverse conditions to test their robustness. These adverse conditions are not however unusual events, but common expected behaviors of the application. In particular, THOR injects in the tests several neutral system events, i.e. events that are not expected to change the outcome of the test. These neutral events are mainly related to Activity lifecycle: for example Pause $\rightarrow$ Resume; Pause $\rightarrow$ Stop $\rightarrow$ Restart; Audio focus loss $\rightarrow$ Audio focus gain. Neutral events injected in an application that does not manage the lifecycle correctly can lead to the discovery of bugs, which are not necessarily crashes but also unexpected behaviors for the specific application.

The tool provides several interesting features like

- Neutral events are injected in suitable locations to avoid conflicts with the test case: in particular they are triggered when the event queue becomes empty and the execution of the remaining test is delayed.

- Multiple errors for each test: if a test fails after some neutral event injections, the test is rerun but injections are only performed after the previous failure point to maximize error detection

- Faults Localization: if a test fails, the tool tries to identify the exact causes (i.e. the neutral events responsible for the unexpected behavior) using a variant of delta debugging (scientific approach of hypothesis-trial-result loop), then displays this information to the user to allow further investigation

- Faults Classification: the errors that make the test cases fail are classified by importance and criticality

- Customization: the developer can select the set of tests to run, the set of neutral event sequences to take into account, and several other different variations

While very interesting and useful for bug detection, THOR is not a very user-friendly tool. First of all the tests are run via an external program (and so the developer is not able to simply run the tests via an IDE like Android Studio) that is only available for Linux and its installation is not immediate. Moreover, THOR only executes the test cases on an emulator running Android KitKat 4.4.3, which does not leave any choice to the developer on which version of Android to test.

15 Static Analysis

15.1 Introduction

This section will present static lifecycle checks more in detail. First a short introduction of the main concepts of static program analysis will be presented, then a detailed explanation of its application in lifecycle testing.

15.2 Static Program Analysis

As opposed to dynamic analysis that requires to actually run an application, static analysis only inspects the source code. One of the techniques for static analysis is to build the Abstract Syntax Tree (AST) of the application, i.e. the tree representation of the code structure, and visit it starting from the root. In the AST each node represents a syntax construct: for example in Java we may have class declaration nodes, method invocation nodes, branch nodes, etc.

An example of static analysis tool for Java is the one integrated in IntelliJ IDEA (on which Android Studio is based) that allows for example to

- find probable bugs (e.g. using an object that may be null)

- locate "dead" code (unused/unreachable code that only makes the application heavier)

- detect performance issues (e.g. suggest ways to implement "tail recursion" in a recursive method)

- improve code structure and maintainability (code dependencies)

- conform to coding guidelines, standards and specifications

15.3 Static Lifecycle Checks

This part of the thesis focuses on possible lifecycle checks using a static analysis approach, provided by Android Lint. The idea is to consider some components used in the applications that require special attention, either because of their own lifecycle or because they need to be carefully used in accordance to the host component lifecycle.

As previously mentioned in section 14.4, acquiring and releasing components in accordance to the host Activity/Fragment lifecycle is a critical problem to avoid unexpected behavior or resource waste. An unexpected behavior can be for example the loss of user data: if the developer does not react correctly to an activity being destroyed in a note-taking app the currently written text may be lost. An example of resource waste can be requesting GPS location updates every few seconds in an application that uses maps

and failing to cancel them when the application is paused/stopped: the system will continue querying the device sensors even if the user is not currently looking at the map.

Static program analysis can help detecting this type of problems. For example, if a method call to acquire a certain component is detected but no call to release it is found in the code, then a warning can be shown to the user. As it can be expected, static analysis may produce false positives or skip some situations where the lifecycle is wrongly managed, but it may help to detect some issues in the early stages of development.

15.4 Target Components

In this section a more detailed explanation of the target components that may be useful to check statically is reported. Note that however not all of these static checks have been actually implemented.

Each component will be analyzed in terms of

- Release: should the component be always released after it is acquired? If so, the static analysis can check if the call to release is always performed.

- Best Practices: in which states of the host Activity/Fragment lifecycle is recommended/usual to acquire and release the component? If there are best practices, the analysis can check if the calls follow them.

- Double Instantiation: can acquiring the same component more than once cause some unexpected behavior or waste computational power? If so, the static analysis can check if the resource may be acquired multiple times during the execution.

Examples of Android components that may be checked with static analysis are:

- Broadcast Receiver: it allows to receive certain types of messages from the system or another application component

  - Release: the official documentation states that a Broadcast Receiver, if registered with an Activity context (as it is usually the case), should ideally be unregistered before the Activity is done being destroyed, but if it not so the system will clean up the leaked registration anyway and only log an error. In case the receiver is registered using the Application context, however, it will be never unregistered by the system, so missing a call to

16

the unregister method may lead to serious leaks[2]. Moreover, it is reported that the developer should not unregister during an Activity `onSaveInstanceState()` method, because it won't be called if the user moves back in the history stack[3].

- Best Practices: the only official recommendation on how to handle a Broadcast Receiver in accordance to the lifecycle of the Activity/Fragment that uses it is to unregister it during `onPause()` if it is registered during `onResume()`[4]. The common usage of the receiver is to register during `onResume()` or `onStart()` and to unregister during `onPause()` or `onStop()` respectively.

- Double Instantiation: although not reported in the documentation, registering twice the same Broadcast Receiver leads to receiving each event twice and so it is usually to be avoided.

- Google API Client: the `GoogleApiClient` class allows to connect to the Google Play services for several APIs, like Google+, Google Drive, wearables, etc. The developer can either manage the connection manually or leave it to the system. The following specifications are of course valid for the former case.

  - Release: the API Client should always be disconnected when the application is done using it.

  - Best Practices: the official recommendation is to connect during `onStart()` and to disconnect during `onStop()`[5].

  - Double Instantiation: it is not a problem since the call to `connect()` returns immediately if the client is already connected or connecting[6].

- Fused Location Provider API: the `FusedLocationProviderApi` class allows to query information about the current location. In particular, we are interested in the functionality that allows to receive periodic updates.

  - Release: the location updates should be removed by calling the `removeLocationUpdates()` method.

  - Best Practices: the official documentation encourages the developer to think whether it may be useful to stop the location up-

---

[2]https://developer.android.com/reference/android/content/Context.html#getApplicationContext()

[3]https://developer.android.com/reference/android/content/BroadcastReceiver.html

[4]https://developer.android.com/reference/android/content/BroadcastReceiver.html

[5]https://developers.google.com/android/guides/api-client#start_a_manually_managed_connection

[6]https://developers.google.com/android/reference/com/google/android/gms/common/api/GoogleApiClient#public methods

dates when the Activity is no longer in focus, to reduce power consumption while the app is in background[7].

– Double Instantiation: does not need to be considered since any previous location updates are replaced by the call to `requestLocationUpdates()`[8].

- Camera:

  – `Camera` class: this is the older API to control the device camera, deprecated in API level 21 (Lollipop).

    * Release: releasing the camera is fundamental. Failing to do so means that all the applications on the device will be unable to use it[9].

    * Best Practices: the best practice is to acquire the camera during `onResume()` and release it during `onPause()`[10].

    * Double Instantiation: if the developer tries to acquire the camera twice, the system will throw a runtime exception[11].

  – `camera2` package: this is the new API introduced in Android Lollipop to manage the device camera

    * Release: although the documentation does not clearly state that the developer *must* release the components, the objects used to manage the camera such as `ImageReader`[12], `CameraDevice`[13], `CameraCaptureSession`[14], etc. all provide a `close()` method to release the resource. In the sample application provided by Google[15] these methods are all called during `onPause()` and a `Semaphore` is used "to prevent the app from exiting before closing the camera", so it is safe to assume that releasing these resources is required.

    * Double Instantiation: methods to acquire the components like `setRepeatingRequest()`[16], `createCaptureSession()`[17], etc. replace the previous

---

[7]https://developer.android.com/training/location/receive-location-updates.html#stop-updates

[8]https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderApi#methods

[9]https://developer.android.com/reference/android/hardware/Camera.html#release()

[10]https://developer.android.com/reference/android/hardware/Camera.html

[11]https://developer.android.com/reference/android/hardware/Camera.html#open(int)

[12]https://developer.android.com/reference/android/media/ImageReader.html#close()

[13]https://developer.android.com/reference/android/hardware/camera2/CameraDevice.html#close()

[14]https://developer.android.com/reference/android/hardware/camera2/CameraCaptureSession.html#close()

[15]https://github.com/googlesamples/android-Camera2Basic/blob/master/Application/src/main/java/com/example/

[16]https://developer.android.com/reference/android/hardware/camera2/CameraCaptureSession.html#setRepeatingR android.hardware.camera2.CameraCaptureSession.CaptureCallback, android.os.Handler)

[17]https://developer.android.com/reference/android/hardware/camera2/CameraDevice.html#createCaptureSession(j android.hardware.camera2.CameraCaptureSession.StateCallback, android.os.Handler)

calls. For this reason, there's no need to check double instantiation for correctness, but since most of these calls can take time to complete (`createCaptureSession()` "can take several hundred milliseconds") it can be checked for performance reasons.

- Ads View: the `AdView` class is a View used to display banner advertisements.

  - Best Practices: the class provides methods like `destroy()`, `pause()`, etc. that should be called in the Activity/Fragment lifecycle methods with similar name (`onDestroy()`, `onPause()`, etc.)[18].

- TODO find more?

Other components such as Services or Threads should also be carefully managed according to the lifecycle. However, since there are too many different use cases (e.g. a Service may be active during several activities, work when the application is in background, etc.) it is difficult to extract a list of best practices and, as a consequence, to statically check the code for misuse.

TODO maybe add checks about the actions during onPause etc. like animations, view manipulation, etc.?

15.5 Design

TODO needed? Not much to say...

15.6 Implementation

15.6.1 Introduction

Not all the static checks listed in the previous section have been implemented, but just a selected few to show what can be achieved with static analysis (?). TODO check, expand

15.6.2 Android Lint

Android Lint [6] is a static analysis tool that scans Android projects for potential bugs, performance, security, usability, accessibility and internationalization issues, and more. It is an IDE-independent analyzer, at the moment integrated with Eclipse and Android Studio.

Some of the checked issues are run directly when the user is writing the code and shown via an in-line warning. A more complete analysis can be performed by explicitly running the tool on the whole project, via terminal

---

[18]https://developers.google.com/android/reference/com/google/android/gms/ads/AdView

command or directly from the IDE options. The results are then presented in a list with a description message and a severity level, so that the developer can easily identify the most critical problems and understand their causes. It is also possible to customize the checker for example by specifying the minimum severity level of the detected issues and by suppressing some specific checks if the developer is not interested.

Android Lint provides more than 100 built-in checks. Some examples:

- MissingPermission (correctness issue): lint detects that a call to some method (e.g. store a file on the SD card) requires a specific system permission (e.g. access to external storage) that has not been included in the application manifest

- WrongThread (correctness issue): checks that the methods that must run on the UI thread (e.g. manipulation of a View component) are actually called there

- SecureRandom (security issue): detects random numbers generated by fixed seeds, usually used only during debugging

- UnusedResources (performance issue): a resource like an image, a string, etc. is not used in the application and so it can be deleted to free space

- UselessParent (performance issue): a layout file contains a View component that is of no use and can be removed for a more efficient layout hierarchy

- ButtonOrder (usability issue): makes sure that "cancel" buttons are placed on the left of the UI component, to follow the Android Design Guideline

- ContentDescription (accessibility issues): checks that important visual elements like image-buttons have a textual description to allow the system accessibility tools to describe their purpose

- HardcodedText (internationalization issue): makes sure that text strings displayed to the user are placed on the appropriate xml files to allow translation and not hard-coded in Java

15.6.3   Custom Android Lint Checks

Important note: the Lint API is not final and mostly undocumented. This means that what is reported below is based only on a few examples/tutorials found on the Internet, on the built-in checks implementations and on source code inspection, and not on official documentation. Moreover, since it is not final, future releases of the tool may invalidate the following statements and

the custom implementations.

The open source Android Lint API allows to build custom rules for the static analyzer.

Implementing a custom Lint rule means building four components:

- Issue: a problem detected by the rule, characterized by properties like ID, explanation, category, priority, etc. For example the "Missing-Permission" issue has "9/10" priority, "Error" severity, "Correctness" category and an explanation describing the problem to the developer.

- Detector: a detector is in charge of identifying one or more issues (if more than one, they are independent but logically related). For example the built-in `ButtonDetector` identifies the "Order" (cancel button on the left), "Style" (button borders), "Back" (avoid custom back buttons) and "Case" (capitalization of "OK" and "Cancel" labels on buttons) issues.

- Implementation: links an issue to a detector and specifies the rule scope.

- Registry: it is simply in charge of registering the issues to allow the Lint tool to identify the custom rules.

More in detail, a custom detector extends the `Detector` class and implements one (or more in special cases) of these interfaces:

- `XmlScanner` if it needs to analyze XML files (such as the application manifest or layout descriptions)

- `JavaScanner` if it needs to analyze Java files (source code for classes)

- `ClassScanner` if it needs to analyze Class files (compiled Java files)

The extended class and the interfaces provide some utility methods that can be overridden by the developer to filter applicable files (e.g. name contains a given substring), nodes (e.g. only variable declarations), elements (e.g. an XML scanner only wants to read `TextView` elements), etc. to focus the rule attention only on particular situations and improve performance.
In particular, a detector that implements `JavaScanner` is able to visit the Java files via an Abstract Syntax Tree, represented with the `lombok.ast` API. The developer has two options:

- Use the `Detector` callback methods to visit the nodes using the default AST Visitor. First of all, if the detector is interested only in specific types of nodes in the AST (as it is usually the case) the developer should override the `getApplicableNodeTypes()` method and

return a list of types (e.g. `ClassDeclatation`, `MethodInvocation`, etc.). The class also provides several other methods to focus the search, like `getApplicableMethods()` (return list of method names), `applicableSuperClasses()` (return list of super-classes names), etc. Once this is done, the developer can override the detector methods like `visitMethod()` to receive all matching method invocations found in the tree, `checkClass()` to receive the matching class declarations, etc. to implement the rule logic.

- Return in `createJavaVisitor()` method a custom implementation of the AST Visitor (subclass of `AstVisitor`) that implements the rule logic. Inside the AST Visitor the developer can override one or more methods that allow to visit every type of node in the tree such as `visitMethodDeclaration()`, `visitVariableDeclaration()`, `visitAnnotation()`, `visitWhile()`, etc.

If a problem is found during one of the callbacks, the detector can call the `report()` method to pass the issue, the location (i.e. file and line) and a message in order to show the warning to the user.

Note that some detectors can just visit the nodes and immediately recognize and report an error (e.g. if an attribute of a given component is not set), but others may require to do more expensive computation (even across multiple files). If this is the case, the developer can use the detector's `afterProject()` hook that is called when the whole project has been analyzed. In order to decide if a problem is present or not, the developer must of course be able to store the state of the computation: variables like boolean flags can be easily stored in the detector object, but the problem is with code locations. Computing the location of an issue is an expensive operation and, especially if the probability of error is very low, we may have performance issues (e.g. a detector that finds unused resources cannot store the location of each of them before finding out that they are actually used somewhere). To solve this issue we can store location handles (lightweight representations of locations that can later be fully resolved if the problem is actually present) or request a second pass to the Lint tool (i.e. in the first project analysis one only sets some flags to detect problems, then if that is the case the project is scanned again to gather the actual locations of the issues).

As a side note, we can also mention the `ControlFlowGraph` class available during detection. It allows to build a low-level Control Flow Graph (all paths that may be traversed) containing the *insns* nodes of a method, i.e. the RTL (Register Transfer Language) representation of the code where each *insns* node is a bytecode instruction (e.g. jump). This Control Flow Graph can be useful for example to analyze if some component is always released, e.g. the built-in `WakelockDetector` uses it to see if, when the Wake-Lock

(a lock to keep the device awake) is acquired in a method, it is released afterwards in every possible path.

Issues are usually defined as public, final and static fields inside their detector class. Issues are simply objects of the `Issue` class that are instantiated calling `Issue.create()` with some parameters like ID, category, severity, etc.

Implementations are objects of the `Implementation` class, whose constructor requires the detector class and the scope (e.g. `Scope.JAVA_FILE_SCOPE`). The implementation is passed as the last parameter of the `Issue.create()` method to link the issue with the detector.

Finally, registers are sub-classes of `IssueRegistry` that usually only override the `getIssues()` method to return the list of custom issues. The registry must be referenced in the `build.gradle` file (in Android Studio) or in a manifest file (in Eclipse) to allow Lint to find it.

Once every component has been implemented, to actually include the Lint check in the list of rules enforced by the tool one must copy the generated JAR file of the library in the $\sim$/`.android/lint/` directory. Using Gradle in Android Studio, one can write a snippet to automatically copy the JAR after each *install* command.

15.6.4 Code Structure

Given the components listed in the previous section, the code structure of the static lifecycle checks is straightforward. TODO continue

15.7 Evaluation

16 Dynamic Analysis

16.1 Lifecycle Test Cases

In this section a dynamic approach to lifecycle testing will be presented, as opposed to the static technique described previously. In particular, the idea is to provide to the developer pre-generated test cases that allow to explore the most common lifecycle changes.

16.2 Design

To allow custom actions and checks during the pre-defined lifecycle tests the system uses several callbacks, for example the *PauseCallback* allows the developer to specify some actions and checks before the component is paused, some checks while it is paused and some other actions and checks

when the component is resumed again. These callbacks are defined by the developer inside a test case and then used by the system during the actual lifecycle tests in the appropriate moments.

The pre-generated lifecycle tests will be defined for two frameworks:

- ActivityTestRule tests: `ActivityTestRule` objects define an activity to test inside a test case. It is used for

  - Instrumented unit testing
  - Espresso UI testing

- Robolectric tests: "hybrid" unit testing

Which cover the most used modern testing mechanisms in Android.

The defined tests are:

- Pause: simulates the component being partially hidden, i.e. paused and then resumed. It is useful for example to see if the component correctly frees/stops and reacquires/starts "critical" resources and CPU-intensive operations.

  - `onCreate()`
  - `onStart()`
  - `onResume()`
  - *Actions and assertions before the pause*
  - `onPause()`
  - *Assertions while paused*
  - `onResume()`
  - *Actions and assertions after the pause*

- Stop: simulates the component being completely hidden (in background), i.e. stopped and restarted. The developer can for example check if all resources used by the application are correctly released while the component is in background, to avoid wasting computational power.

  - `onCreate()`
  - `onStart()`
  - `onResume()`
  - *Actions and assertions before the stop*
  - `onPause()`
  - `onStop()`

- *Assertions while stopped*
- `onRestart()`
- `onStart()`
- `onResume()`
- *Actions and assertions after the stop*

- Destruction: simulates the component being closed (e.g. back button pressed or the application is killed by the system). In this case two different tests are defined, one where `onDestroy()` is called and another one where it is not, since in a real application this call is not guaranteed[19]. This test can be used for example to see if the component stops all computations and, if needed, stores unsaved data in memory.

  - `onCreate()`
  - `onStart()`
  - `onResume()`
  - *Actions and assertions before the destruction*
  - `onPause()`
  - `onStop()`
  - [`onDestroy()`]
  - *Assertions after the destruction*

- Recreation: simulates the component being recreated. This can happen for example when a device configuration change happens (e.g. device is rotated) or an application in background is killed by Android to free resources and then restarted. This is usually the most critical lifecycle transition: since a completely new instance of the component is created (with a different reference), all variables that are not passed in the saved instance state `Bundle` are lost, other components (e.g. `AsyncTask`) may not be able to commit their results to the new component, etc.

  - `onCreate()`
  - `onStart()`
  - `onResume()`
  - *Actions and assertions before the recreation*
  - `onPause()`
  - `onStop()`

---

[19]https://developer.android.com/reference/android/app/Activity.html#onDestroy()

- onDestroy()
- onCreate()
- onStart()
- onResume()
- *Actions and assertions after the recreation*

The test also provides a way to test the actual contents of the saved instance state `Bundle` to see if it contains the correct data passed from the old to the new instance of the component.

- Rotation: simulates the device being rotated, from portrait to landscape modes or vice-versa. By default, from the lifecycle point of view this has the same effect of a recreation, but the developer can specify a different behavior changing the `configChanges` attribute in the application manifest. This test can be useful to see if the component dynamically adapts to the new rotation, e.g. providing a different UI layout.

  - onCreate()
  - onStart()
  - onResume()
  - *Actions and assertions before the rotation*
  - [Possible lifecycle changes, by default recreation]
  - *Actions and assertions after the rotation*

16.3 Implementation

TODO DIAGRAM HERE The system is split in three modules. The reason for this is the Android system of including test libraries: using Gradle, the developer can *testCompile* the libraries used for local tests (i.e. that run on the development computer) and *androidTestCompile* the libraries used for instrumented tests (i.e. that run on real mobile devices or emulators). Since Robolectric is local and ActivityTestRule is instrumented, two distinct modules `RobolectricLifecycleTesting` and `ActivityTestRuleLifecycleTesting` allow the developer to correctly include them without creating useless dependencies. The module `LifecycleTesting` includes the common parts of the other two.

The `LifecycleTest` class inside the `LifecycleTesting` module is the main class of the system. It defines:

- Abstract methods implemented by the sub-modules that allow to specify the means to control the component lifecycle, according to their structure.

- Abstract methods implemented by the end-developer that allow to pass the callbacks for each lifecycle test.

- The actual tests that use the previous two sets of methods to drive the lifecycle and to perform actions and checks in between.

Inside the module, several interfaces for the callbacks are also defined.

The `RobolectricLifecycleTest` class inside the `RobolectricLifecycleTesting` module is the implementation of the lifecycle tests for the Robolectric framework. Besides implementing the methods to drive the component lifecycle using the Robolectric `ActivityController` class, it also provides the abstract method `getActivityClass()` to allow the developer to define the activity under test and the utility `getActivity()` that allows the developer to retrieve the activity during a callback.

The class for ActivityTestRule tests is provided by the `ActivityRuleLifecycleTest` class inside the `ActivityTestRuleLifecycleTesting` module. It implements the lifecycle methods using the Android `Instrumentation` class and provides the abstract method `getActivityTestRule()` to allow the developer to define the ActivityTestRule under test.

16.4   Evaluation

Part V
Event-based Testing

17   Introduction

In this part of the thesis a system to test Android applications from the point of view of the generated events will be presented. In particular, the idea is to allow the developer to specify some conditions on the event stream (for example that an event of a certain type always comes after another event of a certain type) that will later be verified at runtime. The implementation of this event-based testing approach will exploit the ReactiveX library, an innovative system of industrial interest (it is employed for example by Microsoft and Netflix) that allows to observe, transform and listen to events.

The following sections will present first an introduction to event-based systems, then a formalization of the temporal assertions language used to define the consistency checks on the event stream, followed by a detailed explanation of the design, implementation and evaluation of the system.

18 Event-based Systems

Event-Driven Architecture is a software pattern where the focus is on generation and reaction to events. An event can be defined as a message generated by a producer that represents a change of state or a relevant action performed by some component/actor (e.g. the user). Once generated, events are sent via event channels to all the consumers that are interested to them. Event-Driven Architectures are

- extremely loosely coupled: the producer does not know about the consequences of its events. It just generates them and then it's up to the consumers to manage everything else.

- well distributed: an event can be anything and exist almost anywhere

18.1 Android as an Event-Driven Architecture

Android is implemented as an event-based model. This is because a device must manage several events, like clicks on the touchscreen, sensor data, network requests/responses, etc.

From an application point of view, events can be generated internally (e.g. from a service/thread created by the application itself) or externally (e.g. sensor data). Each application is composed of several threads, a subset of which, called Looper threads, are in charge of processing events by invoking an appropriate event Handler for each of them.

Events from a single thread are atomic, i.e. they are placed on a FIFO event queue and processes one by one. However, events produced by several threads are processed concurrently and not guaranteed to be ordered or atomic. For example the user may click on a button while the application receives a network response and the device broadcasts some sensor data.

Examples of events that can be observed in an Android application:

- User input events can be observed attaching listeners to View components. An event listener is an interface whose methods are called by the Android framework when the View is triggered by user interaction, for example the `onClick()` callback of a listener attached to a Button will be called when the user clicks on that button. The developer can then implement the callback to perform any action in response to the input event generation, e.g. starting another activity.

- Lifecycle events are triggered when there is a state transition of some component, like an Activity. For example the `onPause()` callback is invoked when an Activity is paused.

- Broadcast events like system messages (e.g. device boot completed, alarm goes off, etc.) or application-specific messages are received by BroadcastReceivers in the `onReceive()` callback.

- Asynchronous messages from services external to the application (e.g. sensor data, network requests, etc.) also have their specific callbacks

And many more.

18.2 Event Concurrency Errors

The concurrency among events inside a single application can lead to unexpected behaviors or crashes.

When the Android developer decides to explicitly create a multi-threaded application we may have the classical problems of concurrency like synchronization, deadlocks and starvation, but problems arise even if he/she does not manually create threads, due to the aforementioned concurrency of events.

The most relevant problem in an Android application is race conditions: two or more events do not happen in the order the programmer intended. A simple example of this issue is when the developer asynchronously starts an AsyncTask (an utility class that allows to perform short operations in the background without blocking the UI thread) to download an image from the network. The AsyncTask, when the download is completed, expects the Activity that started it to be still there to receive the result, but maybe the user already left the application and so we encounter for example a NullPointerException.

18.3 Existing Event Testing

Due to the relevance of the race conditions problem, several research studies tried to provide a way to detect them.

For example, CAFA [7] is a tool that allows to detect use-free races. The authors note that thousands of events may get executed every second in a mobile system and that, even if they are processed sequentially in one thread, most of them are logically concurrent. These concurrent events could be commutative with respect to each other, i.e. the result is the same even if they are executed in a different order. The tool tries to determine if two events are commutative or not, restricting the focus on use-after-free violations (a reference is used after it has been freed, i.e. it does not point to an object anymore). If two events where one uses and the other frees a reference are logically concurrent, they must be non-commutative. To detect possible racy conditions the tool analyzes the traces of the low-level read and write operations, as well as certain branch instructions. While CAFA was a good starting point of race condition analysis, its main problems are that it's quite slow (it may take hours to analyze an application), only focuses on a very specific type of race condition and the tool is not publicly available.

Android EventRacer [8] is an improvement of CAFA, defined by the authors as the first scalable analysis system for finding harmful data races in real-world Android applications. EventRacer not only analyzes use-free races, but also other types of race conditions like:

- Data Races Caused by Object Reuse: list components in Android, like ListView, usually reuse rows while the user scrolls to improve performance (no need to create as many rows as the number of data values). If data for each row is loaded asynchronously, it may happen that the wrong data is loaded in a row (the user scrolled the list in the meanwhile and the row has already been reused).

- Data Races Caused by Invalidation: these are the races similar to the first example, i.e. the AsyncTask completing when the Activity no longer exists

- Callback Races: different listener callbacks may be invoked in any order. For example the developer may asynchronously create a GoogleMap object and request location updates from the device GPS. He/she may expect the `onMapReady()` callback to be executed before the first `onLocationChanged()` callback (and so use the map to place a marker at the received location), but it might not be the case.

Moreover, EventRacer analyzes an application in much less time than CAFA: building the Happens-Before relationship graph has $\mathcal{O}(n^2)$ time complexity instead of $\mathcal{O}(n^3)$. The tool is also publicly available as an online tester or an offline application (only available for Linux systems). One disadvantage of the EventRacer approach is that the implementation requires to modify the Android framework in order to access low-level information and, for this reason, it is only available for Android 4.4, which limits the options of the developer for testing other versions of the OS.

A completely different approach for race detection is provided by DEvA [9]. The idea of the tool is to base its search on static code analysis rather than dynamic analysis (CAFA and EventRacer run the application on emulators, interact with it via "robots" and then retrieve the traces to be analyzed): this approach of course guarantees more code coverage and completeness. DEvA focuses on a specific type of problem called Event Anomalies: processing of two or more events results in accesses to the same memory location and at least one of those is a write access (note that the use-free races analyzed by CAFA are a subclass of this type of issue). The idea of the tool is to identify variables that may be modified as a result of receiving an event (i.e. a potential Event Anomaly) using the Control Flow Graph of the application (i.e. all paths that may be traversed in the application during its execution). The tools receives as input from the developer

- the list of all methods used as event handlers (callbacks that use the events)

- the base class used to implement events in the system

- the set of methods used as consumed event revealing statements (i.e. methods that retrieve information stored in an event without modifying the event's attributes), used to tell apart events when a general parameter is passed to a callback

DEvA allows a very fast analysis (usually 1 or 2 minutes) and guarantees complete code coverage, but static analysis may report false positives or be unable to detect some anomalies.

19 Temporal Assertions Language

19.1 Consistency Checks

An approach to event-based testing is to define consistency checks, i.e. conditions that should be verified in the event stream. For example we can define the happens-before relationship between two or more events, specify the exact number of events of a certain type that can appear during a particular execution and so on.

The idea of event-based testing described in this part of the thesis is based on a temporal assertions language: a language that allows to express consistency checks on some of the events generated during the application execution and, in case they are not satisfied, the assertion error is notified to the developer. A detailed explanation of the language to express checks is reported in the next sections.

Components and abbreviations:

- Check (c, $c1$, $c2$,...): final output of a specification

- Event (e, $e1$, $e2$,...): an event of the stream

- Matcher (m, $m1$, $m2$,...): a specification that describes one or more events in the stream. For example a matcher "is a text change event" matches all events of the stream that represent a text change, "is a click on myButton" matches all click events on that specific button.

- Number (n, i): a non-negative integer value

Each check will be formalized by:

- Code structure: abstract description of the check specification. It will have the same structure of the final code but use the aforementioned abbreviations to tell which components are expected.
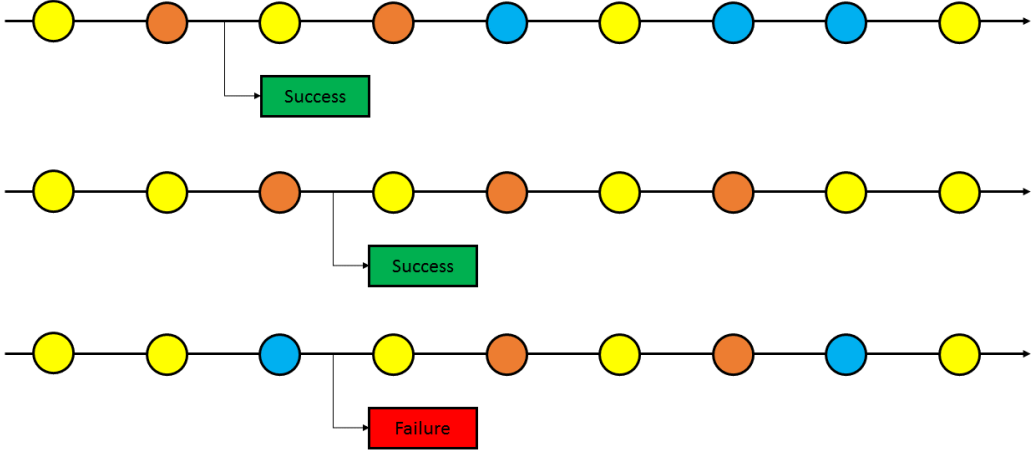
- Description: a brief explanation of the check logic.

- First Order Logic (FOL) formalization of the check meaning, expressed using the following logic relationships:

  - $match(e, m)$ means that the matcher m matches the event e.
  - $before(e1, e2)$ means that the event $e1$ is generated in the sequence before $e2$.
  - $between(e2, e1, e3) \equiv before(e2, e1) \wedge before(e1, e3)$ means that the event $e1$ is generated in the sequence after $e2$ and before $e3$.

- Visual representation: shows some examples of event streams and the corresponding check outcome (success or failure). The outcome will be placed in the spot where it will be actually produced by the system: some checks may not need to explore the whole stream, if the condition is fulfilled or violated after an intermediate event the check "short-circuits".

- Actual code example: an example of real call as implemented by the system. The structure of the Java classes and objects will be clear later when the implementation will be explained in detail.

19.2 Checks on Single Events

The first type of checks focuses on single events at a time: for example we may say that an event of a certain type can be detected in the stream only after another event of a certain type.

19.2.1 Can Happen Only After

Structure	`anEventThat(m1).canHappenOnlyAfter(anEventThat(m2))`
Description	Checks that the events that match `m1` happen only after any event that matches `m2`, i.e. there cannot be an event that matches `m1` before the first event that matches `m2`.

FOL	$\forall e1 \Big(match(e1,\ m1) \ \Rightarrow \ \exists e2 \Big(match(e2,\ m2) \ \wedge \ before(e2,\ e1) \Big) \Big)$
Visual	anEventThat(🔵).canHappenOnlyAfter(anEventThat(🟠)) Note in particular that in the second case the checks succeeds even if no blue event occurs: we are just saying that blue events *can* happen after an orange one.
Code Example	``` anEventThat(isMarkerPlacement()) .canHappenOnlyAfter(anEventThat(isMapReady())) ```

19.2.2 Can Happen Only Before

`anEventThat(m1).canHappenOnlyBefore(anEventThat(m2))`

Checks that the events that match `m1` happen only before any event that matches `m2`, i.e. there cannot be an event that matches `m1` after the last event that matches `m2`.

$$\forall e1 \Big(match(e1, \ m1) \ \Rightarrow \ \exists e2 \big(match(e2, \ m2) \ \wedge \ before(e1, \ e2) \big) \Big)$$

19.2.3 Can Happen Only Between

```
anEventThat(m1).canHappenOnlyBetween(anEventThat(m2),
    anEventThat(m3))
```

Checks that the events that match `m1` happen only between any pair of events that match `m2` and `m3` respectively, i.e. there cannot be an event that matches `m1` before the first event that matches `m2` or after the last event that matches `m3`.

$$\forall e1 \Big(match(e1, \ m1) \ \Rightarrow \ \exists e2, \ e3 \big(match(e2, \ m2) \ \wedge \ match(e3, \ m3)$$
$$\wedge \ between(e2, \ e1, \ e3) \big) \Big)$$

19.3 Checks on Sets of Events

These checks work on sets of events: for example we may say that a certain event generates a set of n other events. The cardinality of each set is specified by the quantifiers

- exactly
- at most
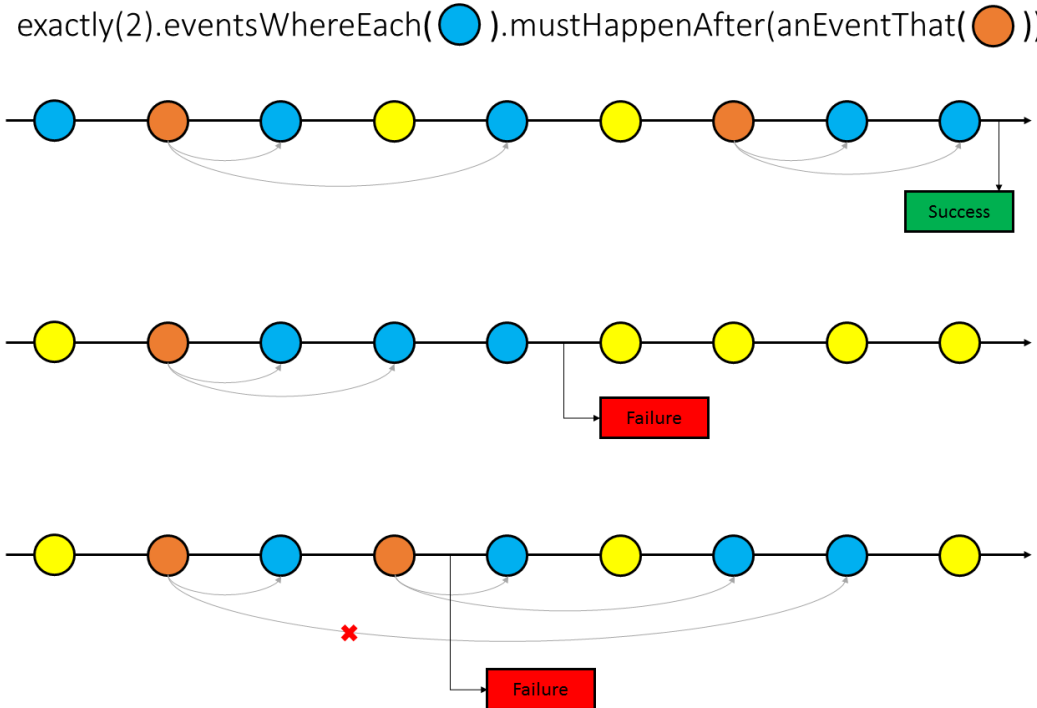- at least

Only the formulas for "exactly" will be written since the others can be derived by analogy.

First Order Logic specifications use the counting quantifiers [10] as a notational shorthand, i.e. $\exists_{=n} x$ means that there exist exactly n elements x.

19.3.1 Must Happen After

Structure	
	`exactly(n).eventsWhereEach(m1).mustHappenAfter(anEventThat(m2))`

Description	Checks that exactly `n` events that match `m1` happen exclusively after every event that matches `m2`. "Exclusively" means that there cannot be another event that matches `m2` before the sequence of `n` events is completed.
FOL	$$\forall e2\Bigg(match(e2,\ m2)\ \Rightarrow\ \exists_{=n}e1\Bigg(match(e1,\ m1)\ \wedge\ before(e2,\ e1)$$ $$\wedge\ \neg\exists e2'\Big(match(e2',\ m2)\ \wedge\ between(e2,\ e2',\ e1)\Big)\Bigg)\Bigg)$$
Visual	exactly(2).eventsWhereEach(🔵).mustHappenAfter(anEventThat(🟠)) Note in particular that the third case shows the "exclusively" constraint mentioned before: the check fails because we do not have two blue events after the first orange but before the second one (i.e. the first orange event does *not* match one of the three blue events that follow the second orange event)

```
exactly(1).eventsWhereEach(isTextChangeFrom(locationTextView))
    .mustHappenAfter(anEventThat(isLocationChange()))
```

19.3.2   Must Happen Before

```
exactly(n).eventsWhereEach(m1).mustHappenBefore(anEventThat(m2))
```

Checks that exactly `n` events that match `m1` happen exclusively before every event that matches `m2`. "Exclusively" means that the sequence of `n` events must be after the previous (if any) event that matches `m2`.

$$
\forall e2 \bigg(match(e2,\ m2) \ \Rightarrow\ \exists_{=n}e1 \Big(match(e1,\ m1)\ \wedge\ before(e1,\ e2)
$$

$$
\wedge\ \neg \exists e2' \Big(match(e2',\ m2)\ \wedge\ between(e1,\ e2',\ e2) \Big) \Big) \bigg)
$$

19.3.3   Must Happen Between

```
exactly(n).eventsWhereEach(m1).mustHappenBetween(AnEventThat(m2),
    AnEventThat(m3))
```

Checks that exactly `n` events that match `m1` happen exclusively between every pair of events that match `m2` and `m3` respectively. "Exclusively" means that the sequence of `n` events cannot be "shared" among overlapping pairs.

$$
\forall e2 \bigg(match(e2,\ m2) \ \Rightarrow\ \exists e3 \Big(\big(match(e3,\ m3)\ \wedge\ before(e2,\ e3)
$$

$$
\wedge\ \neg \exists e2',\ e3' \big(match(e2',\ m2)\ \wedge\ between(e2,\ e2',\ e3)\ \vee\ match(e3',\ m3)
$$

$$
\wedge\ between(e2,\ e3',\ e3) \big) \big)\ \iff\ \exists_{=n}e1 \Big(match(e1,\ m1)\ \wedge\ between(e2,\ e1,\ e3) \Big) \Big) \bigg)
$$

19.4   Checks on the Whole Stream

The third type of checks concerns the entire stream of events: they allow for example to specify the number of all the events of a certain type in the whole sequence generated by a particular execution.

19.4.1 Match In Order

`allEventsWhereEach(m).matchInOrder(m$_1$, m$_2$, ... , m$_n$)`

Checks that *all* the events (in the whole sequence) that match `m` match in order the given matchers.

$$\forall e\Big(match(e,\ m)\ \Rightarrow\ \exists i\Big(indexOf(e,\ m,\ i)\ \wedge\ match(e,\ m_i)\Big)\Big)$$

$$indexOf(e,\ m,\ i)\ \equiv\ \exists_{=i}e'\Big(match(e',\ m)\ \wedge\ before(e',\ e)\Big)$$

19.4.2 Are Ordered

`allEventsWhereEach(m).areOrdered(f)`

Checks that *all* the events (in the whole sequence) that match `m` are in the order defined by the comparator function f (receives two events and returns an integer < 0 if the first is less than the second, 0 if they are equal, and > 0 if the first is greater than the second).

$$\forall e,\ e'\Big(match(e,\ m)\ \wedge\ match(e',\ m)\ \wedge\ before(e,\ e')\wedge$$

$$\neg\exists e''\Big(match(e'',\ m)\ \wedge\ between(e,\ e'',\ e')\Big)\ \Rightarrow\ ordered(f,\ e,\ e')\Big)$$

$$ordered(com,\ e1,\ e2)\ \equiv\ \Big(\Big(com\ <\ 0\ \Rightarrow\ before(e1,\ e2)\Big)\vee$$

$$\Big(com\ =\ 0\ \Rightarrow\ before(e1,\ e2)\ \vee\ before(e2,\ e1)\Big)\vee$$

$$\Big(com\ >\ 0\ \Rightarrow\ before(e2,\ e1)\Big)\Big)$$

19.4.3 Exists An Event

`existsAnEventThat(m)`

Checks that at least one event that matches the matcher `m` exists anywhere in the sequence.

$$\exists e\Big(match(e,\ m)\Big)$$

19.4.4 Exist Events

`exist(exactly(n)).eventsWhereEach(m)`

Checks that *all* the events (in the whole sequence) that match `m` are exactly `n`.

$$\exists_{=n} e \Big(match(e, \ m) \Big)$$

In the same way seen in the previous section, the checks for "at most" and "at least" can be derived by analogy.

19.5 Connectives between Checks

These constructs allow to specify the standard logic connectives between one or more of the previously defined consistency checks.

19.5.1 And

Structure	`allHold(c1, c2,...)`
Description	All sub-checks *c1*, *c2*, etc. must succeed.
FOL	$c1 \ \wedge \ c2 \ \wedge \ ...$
Visual	

```
allHold(
    exactly(1).eventsWhereEach(isDownloadImage())
        .mustHappenAfter(anEventThat(isClickOn(downloadButton))),
    anEventThat(isEnlargeImage())
        .canHappenOnlyAfter(anEventThat(isDownloadImage)))
```

19.5.2 Or

`anyHolds(c1, c2,...)`

At least one sub-check must succeed.

$$c1 \ \lor \ c2 \ \lor \ ...$$

19.5.3 Not

`notTrueThat(c)`

Inverts the outcome of the sub-check.

$$\neg c$$

19.5.4 Single Implication

`providedThat(c1).then(c2)`

c2 is checked only if c1 succeeds.

$$c1 \ \Rightarrow \ c2$$

19.5.5 Double Implication

`weHaveThat(c1).ifAndOnlyIf(c2)`

Succeeds only if both sub-checks fail or both succeed.

$$c1 \ \iff \ c2$$

20 Design

To implement the event-based testing approach described so far we need the following main components:

- Events: objects that represent events in the application

- Event Generators: structures that actually generate the events whenever something happens in the application

- Checks: ways to specify the consistency rules defined in the previous section

- Results: the outcomes of the checks

- Event Monitor: the main interface of the system, receives the events, applies the checks and produces the results

The following UML class diagram shows an abstract representation of the system:
TODO PLACE UML HERE
The *EventMonitor* offers methods to add one or more *EventGenerator* objects to define the event stream, to add one or more *Check* objects to be verified and handles the *Result* objects in some way (e.g. makes a test case fail if the *Result* is a failure).

Each *Check* contains the logic of a consistency check: it will receive each *Event* of the stream in order and act accordingly. A *Check* has the possibility to short-circuit its behavior (i.e. interrupt the stream of events if the consistency check succeeds or fails before the end of the sequence). In any case, when it short-circuits or the stream comes to an end, the *Check* produces one and only one *Result* object.

Each *Result* contains an *Outcome* and a message describing it. In particular, an *Outcome* can be a *Success*, a *Failure* and a *Warning*. A *Warning* outcome has not been included in the formal specifications of the checks in the previous section for simplicity: it's meaning is that the *Check* succeeded with some minor error or in a particular situations (for example a check that specifies that an event A always generates exactly one event B may return *Warning* if no event A has been found in the sequence).

A *Descriptor* allows to specify a *Check*. It describes one or more events of the stream and provides some methods to express a condition on those events. In particular, *SingleEvent* describes a single event of a certain type at a time, *SetOfEvents* describes a set of events of a certain type at a time and *AllEvents* describes all the events of a certain type in the whole stream. To determine the type of the events identified by the *Descriptor*, a *Matcher* object is used.

The cardinality of the descriptor *SetOfEvents* is defined by a *Quantifier* that has an integer values as input. A quantifier can be*Exactly, AtMost* or *AtLeast*.

Finally, a *CheckConnective* allows to transform one or more *Check* objects into a single *Check*, i.e. it specifies one of the standard logic connectives *Not, And, Or, SingleImplication* and *DoubleImplication*.

21   Implementation

For the implementation of event-based testing, the ReactiveX library was chosen. This innovative reactive programming paradigm focuses on data flow: this means generating (statically or dynamically) a flow of events, propagate them and allow interested components to react to them. The idea of reactive programming is fire-and-forget messaging: send a request and asynchronously wait for the response to be ready or, even more importantly, if we are dealing with response sets wait for individual results to be forwarded (without waiting for the whole set to be computed).

In the next sections a more detailed introduction to the ReactiveX library will be presented, followed by the implementation of the actual event-based testing system.

21.1   ReactiveX

The Reactive Extensions (ReactiveX or Rx) are a reactive programming library to compose asynchronous and event-based programs. As defined by its authors, ReactiveX is a combination of the best ideas from

- the Observer pattern (design pattern where a subject automatically notifies the so-called observers of its state changes)

- the Iterator pattern (design pattern where an iterator is used to traverse a collection of elements, like an array)

- functional programming (declarative programming paradigm where computation is performed via mathematical functions that are not allowed to change the state of the system)

This approach is asynchronous because many instructions may execute in parallel and their results (events) are later captured, in any order, by the listeners. For this reason, the main idea to perform a computation is not to call methods like in classic sequential programming but to define a mechanism to react to results when they are ready.

ReactiveX programming paradigm is essentially based on three steps:

- Create: the Observable components are used to generate event or data streams

- Transform: Operators allow to modify the event streams (e.g. change each event or filter out some of them) and compose them (e.g. join two streams)

- Listen: Subscriber components can listen to event streams and receive their elements one by one, to perform some computation

An Observable is in charge of emitting events: it generates zero, one or more than one events (depending on the specific implementation), and then terminates either by successfully completing or with an error. Observables can be "hot" (emit events even if no Subscriber is listening) or "cold" (emit events only after a Subscriber is registered).

An Observable can be modified by an Operator: most operators can be applied on an event stream generated by an Observable to return a new modified event stream. For example the operator `map(function)` allows to apply a function to each event in the stream (e.g. a stream of numbers modified by a map with a summing function may become a stream where all the original numbers have been increased by 1). Since the result of an operator is an Observable, operators can be applied in chain (i.e. apply an operator on the result of another operator) to achieve complex modifications.

A Subscriber consumes the events emitted by the Observables (that can either be "original" or the result of one or more Operators). Subscribers allow to react to asynchronous results: for example, an Observer may send a network response whenever it is ready, the Subscriber receives it and shows the information to the user.

The advantages of the Rx paradigm are:

- Cross-Platform: it is available in many programming languages, like RxJava, RxSwift, RxJS, RxPHP, etc.

- Can be used for any application, from Front-End (e.g. UI events and API responses) to Back-End (e.g. asynchronicity and concurrency)

- Operators usually make computations less verbose and more readable

- Error handling: if an error occurs in one of the steps of the computation the exception is automatically intercepted by Rx and forwarded to the user via appropriate callbacks

- Easy concurrency: Rx allows to easily specify in which threads the components should be run, without worrying about implementation details

- Extensible: a developer can define custom Observables, Operators and Subscribers to achieve anything an application may require

The characteristics of ReactiveX make this programming paradigm very suitable for many applications. It is successfully employed in industry: examples of users of Rx are Microsoft, Netflix and GitHub.

21.2   RxJava and RxAndroid

RxJava [11] is the Java implementation of ReactiveX. It is an open source project initially developed by Netflix for server-side concurrency. The main reason for its adoption and development was to avoid Java Futures (results of asynchronous computation) and callbacks because both are expensive when composed, especially if nested.

In addition to implementing all functionalities of ReactiveX paradigm (Observables, Subscribers, Operators, etc.), RxJava also has the advantages of being

- Lightweight: zero dependencies and single small JAR to contain the whole library

- Polyglot: supports Java 6 or higher and JVM-based languages such as Groovy, Clojure, JRuby, Kotlin and Scala

- Composable: several RxJava Libraries are available to developers to manage common use cases

RxAndroid [12] is a RxJava module that provides specific bindings for the Android platform, for example to easily specify the main (UI) thread as the observing thread or, more in general, a custom Looper. RxAndroid can in turn be extended by other modules, some of which are listed in the next section.

21.3   Events Observable in Android

Several modules allow Android developers to observe many events inside an application. For example, we can observe:

- UI Widgets: RxBinding [13] module allows to observe user inputs or changes on UI widgets like TextView (e.g. clicks, text change), app bar menu (e.g. option selected), etc.

- Settings: RxPreferences [14] module allows to receive events from the Shared Preferences system (storage provided by Android to store the app settings)

- Files: RxFileObserver [15] fires events for file accesses or changes

- Database: StorIO [16] allows to manage and observe an SQLite database

- Network: ReactiveNetwork [17] detects network changes (e.g. WiFi or mobile connection)

- External API Requests: Retrofit [18] offers observables to receive network responses (REST client)

- Broadcasts: RxBroadcast [19] builds an event stream from a Broadcast Receiver results

- Location: ReactiveLocation [20] allows to observe location changes

- Sensors: ReactiveSensors [21] fires events from hardware sensors

- Permissions: RxPermissions [22] allows to receive events from the permissions manager

- Google Maps: RxGoogleMaps [23] fires events related to Google Maps

- Google Wear: RxWear [24] allows to observe messages to and from a connected smartwatch

And many others.

The utility module RxLifecycle [25] allows to bind the listed observables to the lifecycle of an Activity or a Fragment to avoid leaks. For example, without any binding to the lifecycle, an observable that emits text change events on a TextView never ends (i.e. never calls `onCompleted()` or `onError()`) because it has no way of understanding when the text will stop changing. In that situation the observable will run even after the Activity has been removed and, keeping its reference, won't allow the garbage collector to delete the instance (memory leak). Thanks to RxLifecycle the developer can bind the observable until a lifecycle event occurs (e.g. activity is stopped), allowing it to terminate correctly.

21.4 The System

The structure of the actual implementation of event-based testing is similar to what has been designed in section 20. A class diagram of the system will be proposed, followed by more detailed explanations of the main components.

[TODO PUT UML DIAGRAM HERE]

Before beginning, note that many classes like the descriptors have a private constructor and a static method to create an instance. Similarly to

the standard way of building assertions in JUnit, this is just syntactic sugar that allows to statically import the methods and to avoid writing the `new` keyword.

21.5 Event Monitor

The `EventMonitor` is a singleton class that is set up via the `initialize()` method. Once this is done, the developer can add observables via `observe(Observable)` to build the event stream and checks via `checkThat(String, Check)`.

At this point a call to `startVerification(Subscriber<Event>, Subscriber<Result>)` is performed to start the validation process on the defined stream: the two subscribers are used by the developer to receive all the events of the stream in order and all the results of the checks respectively. The class provides some static methods to get simple pre-generated subscribers for this purpose, e.g. passing `EventMonitor.getAssertionErrorResultsSubscriber()` as the second parameter the app will crash (or make the test case fail) if a failure result is found.

When the developer calls `stopVerification()` the stream will be interrupted and all the remaining check results will be generated.

Finally, the class also provides the utility method `fireCustomEvent(Event)` that allows to directly generate an `Event` without creating an `Observable`.

Note that the `EventMonitor` observes the events on the thread specified by each observable (usually the main thread) and runs the checks in a separate thread to avoid possible slowdowns of the UI thread.

The `EventMonitor` can be used as

- runtime monitor during the manual debugging phase of the application development: just like the standard assertions placed in production code, the developer can use the monitor for example inside an activity to log all events and the results of the checks

- testing mechanism: the developer can start the monitor at the beginning of a test and add some specific checks, then execute the test actions as usual and finally stop the monitor at the end. The monitor will work as a standard assertion mechanism, making the test fail in case a consistency check is not successful. Note that this mechanism, since it's not framework-specific, can be used for any type of test, from unit to UI, and with any library, provided that it is able to observe all the required events in the application.

21.6 Event Generators

As introduced, the event generators are implemented as `Observable` objects provided by RxJava. Each observable added to the `EventMonitor` will build the event stream: in particular, all the observables are combined using the RxJava operator `merge()`, which transforms a set of observables into a single one that emits all the events in their respective order.

The events generated by these observables are subclasses of `Event`, of which just two have been reported in the diagram since there could be hundreds of them. An `Event` class can also provide some static methods to create one or more matchers.

21.7 Checks

A `Check` implements is logic using a `CheckSubscriber`, a subclass of the RxJava `Subscriber`. This component receives all the events of the stream via `onNext(Event)` and performs some internal computation, possibly short-circuiting the validation calling `endCheck()`. The callback `getFinalResult()` will be called to return the unique `Result` of the check.

The `EventMonitor` applies a check on the event stream by calling the operator `EnforceCheck` on it, which is in charge of transforming a stream of `Event` objects into a stream containing a single `Result`.

Connectives between checks are implementations of `CheckConnective`, which is in turn a subclass of `Check`. Each of them implements a `ResultsSubscriber` that, analogously to a `CheckSubscriber`, receives the results of its terms, performs some computation and then produces a single `Result` as output. `Not`, `AllHold` and `AnyHolds` (implementations of the logic *negation*, *and* and *or* respectively) provide static methods as constructors (e.g. `allHold(check1, check2, check3)`), while `IfThen` and `IfAndOnlyIf` (implementations of single and double implication) are built starting from the `ProvidedThat` and `IsSatisfied` classes respectively (e.g. `providedThat(check1).then(check2)`).

21.8 Descriptors

The descriptors are subclasses of `AbstractEventDescriptor`. `AnEventThat` (single event) and `AllEventsWhereEach` (all the events in the whole stream) provide static constructors, while `EventsWhereEach` descriptors (sets of events) do not provide one because they are built starting from a quantifier, e.g. `exactly(10).eventsWhereEach(...)`.

Quantifiers are subclasses of `AbstractQuantifier`. Each of them uses an internal counter and overrides several methods like `isConditionMet()` and `canStopCurrentComputation()` to implement its logic. All of them have a

private constructor and a static method like the descriptors.

The matchers that describe the type of events identified by each descriptor are implemented using the external library JavaHamcrest [26], a well known way to describe objects. The library provides several matchers on the most common objects (like Strings, e.g. `isEmptyString()`, `startsWith(String)`, etc.), some connectives to compose matchers (e.g. `allOf(Matcher...)`) and the means to implement custom matchers, which have been used to define matchers for the events. TODO ADD EXIST

22   Evaluation

Part VI
Conclusion

23   Recap

24   Future Work

References

[1] Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. http://www.cs.uoregon.edu/~michal/pubs/oracles.html.

[2] JUnit. http://junit.org.

[3] Mockito. http://mockito.org/.

[4] Robolectric. http://robolectric.org/.

[5] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of android test suites in adverse conditions. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, pages 83–93, New York, NY, USA, 2015. ACM.

[6] Lint. http://tools.android.com/tips/lint.

[7] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. Race detection for event-driven mobile applications. SIGPLAN Not., 49(6):326–336, June 2014.

[8] Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable race detection for android applications. SIGPLAN Not., 50(10):332–348, October 2015.

[9] Gholamreza Safi, Arman Shahbazian, William G. J. Halfond, and Nenad Medvidovic. Detecting event anomalies in event-based systems. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pages 25–37, New York, NY, USA, 2015. ACM.

[10] Ian Pratt-Hartmann. Logics with counting and equivalence. In Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, pages 76:1–76:10, New York, NY, USA, 2014. ACM.

[11] RxJava. https://github.com/ReactiveX/RxJava.

[12] RxAndroid. https://github.com/ReactiveX/RxAndroid.

[13] RxBinding. https://github.com/JakeWharton/RxBinding.

[14] RxPreferences. https://github.com/f2prateek/rx-preferences.

[15] RxFileObserver. https://github.com/phajduk/RxFileObserver.

[16] StorIO. https://github.com/pushtorefresh/storio.

[17] ReactiveNetwork. https://github.com/pwittchen/ReactiveNetwork.

[18] Retrofit. http://square.github.io/retrofit/.

[19] RxBroadcast. https://github.com/cantrowitz/RxBroadcast.

[20] ReactiveLocation. https://github.com/mcharmas/Android-ReactiveLocation.

[21] ReactiveSensors. https://github.com/pwittchen/ReactiveSensors.

[22] RxPermissions. https://github.com/tbruyelle/RxPermissions.

[23] RxGoogleMaps. https://github.com/sdoward/RxGoogleMaps.

[24] RxWear. https://github.com/patloew/RxWear.

[25] RxLifecycle. https://github.com/trello/RxLifecycle.

[26] JavaHamcrest. http://hamcrest.org/JavaHamcrest/.