Thomas Hirsch, BSc

Challenges in test automation in big Android applications

Master's Thesis
to achieve the university degree of
Master of Science
Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor
Dipl.-Ing. Dr.techn. Christian Schindler

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Graz, January 2020

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.
The LaTeX template is based on Karl Voits template around KOMA script
https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

Automated testing is a necessity for the success of complex long-term software systems. During continuous development the quality and performance of a suite of automated tests has a direct influence on the quality, performance and development velocity of the production software system. Test driven development (TDD) is a software development process that uses short incremental changes on tests to reflect the expected behavior and then changing production code to satisfy those tests. TDD relies on a fast and reliable testing framework as well as a stable and extensive test suite.

This thesis intends to shed some light on how to approach challenges in TDD, automated test strategies, and architecture that arise in big Android applications. The Android documentation and Google testing recommendations are overly simplistic and showcase only trivial and small examples, while the real challenges in testing big Android applications stay untouched. Test partitioning, classification of tests, and the application of the test pyramid to the Android platform are discussed. Besides a small exploratory study on the testing situation in a limited sample of open source Android applications, the consequences of a bad testing strategy and technical debt in a big and long-term Android project are shown in the case of Catrobat.

By Identifying anti patterns and code smells, general guidelines and patterns for testing big Android applications are established, as well as metrics for test code quality. This knowledge was applied to the Catrobat application, reducing technical debt, measurably increasing test code quality, and significantly increasing test suite performance. A condensed version of this thesis has been previously published in Hirsch et al., 2019.

Kurzfassung

Testautomatisierung ist eine Notwendigkeit für den Erfolg von komplexen, langlaufenden Softwareprojekten. Die Qualität und Performance von automatisierten Tests hat während der fortlaufenden Entwicklung einen direkten Einfluss auf die Qualität, Performance und Entwicklungsgeschwindigkeit des Softwareproduktes. Test Driven Development (TDD, testgetriebene Entwicklung) ist ein Softwareentwicklungsprozess bei dem kurze, inkrementelle Änderungen in Form von Tests die die Anforderungen abbilden vollzogen werden, während daraufhin der "Production Codeängepasst wird um diesen Tests und damit den Anforderungen zu entsprechen. TDD benötigt eine schnelle und zuverlässige Testumgebung und ein stabiles und umfassendes Set von bestehenden Tests.

Diese Diplomarbeit beleuchtet Herangehensweisen und Herausforderungen die sich in großen Android Applikationen bei der Umsetzung von TDD, Testautomatisierung, Teststrategie und Test-Architektur stellen. Die Android-Dokumentation sowie die Empfehlungen bezüglich Testautomatisierung von Google sind sehr simpel gehalten und beschränken sich auf triviale Beispiele, während die eigentlichen Herausforderungen die sich beim testen von großen Android-Applikationen stellen unangetastet bleiben. Im Zuge dieser Diplomarbeit wird die Einteilung und Klassifizierung von Test sowie die Anwendung der Testpyramide auf der Android-Plattform diskutiert. Eine explorative Studie bezüglich Teststrategie und Architektur in einer Anzahl von größerer Open-Source-Android Applikationen wurde unternommen. Weiters wurde eine eingehende Untersuchung der Catrobat Android Applikation vorgenommen, die mit dem Entwicklungsstand von Beginn 2016 die Konsequenzen schlechter oder nicht vorhandener Test Strategie und Architektur zeigt.

Die Identifizierung von Anti Patterns und Code Smells ermöglichte die Gründung und Festlegung von Richtlinien und Patterns zum Testen von großen Android-Applikationen. Weiters wurden Metriken für das Abbilden von Testcode-Qualität identifiziert und angewendet. Das Erworbene Wissen

wurde auf die Catrobat-Android-Applikation angewandt, dabei wurde Technical Debt reduziert, die Code-Qualität messbar verbessert, und die Testperformance erheblich verbessert.

Eine Kurzfassung der Ergebnisse und Erkenntnisse aus dieser Diplomarbeit wurde bereits in Hirsch u. a., 2019 veröffentlicht.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Catrobat

Catrobat is a visual programming language, inspired by MIT labs's Scratch programming language. All implementations of the Catrobat language are FOSS (free and open source) led by the International Catrobat Association at the Institute of Software Technology of the Technical University of Graz. It is developed mostly by students of the TU Graz with help of a handful employed developers, and a few contributors on GitHub. The Catrobat language, like Scratch, is designed for kids. While Scratch's main focus lies on desktop computers as a platform, Catrobat is developed for mobile platforms. Both languages share the same main goal, to offer a low threshold means to introduce kids to programming and teach them basic programming paradigms.

Programs are written not in text, but by visually stacking together blocks that provide certain functionalities, actions, and manipulations. This serves to maintain a very shallow learning curve, and provides an effective way for children to quickly write small programs with no programming experience required beforehand. Having mobile devices as a platform enables the user to write programs using the vast amount of sensors and interfaces offered by such a device.

Programming paradigm Catrobat is an event driven programming language, offering Sprite based graphics.

1.1.1 Nomenclature

Pocket Code The brand name / marketed name for implementations of the Catrobat language IDEs for the programming language.

Catroid The working title of the Android implementation of the Catrobat language and IDE. The app is marketed and can be found as "Pocket Code" in Google Play[1]. While there is an Apple IOs implementation (the already working HTML5 and Windows phone implementation were cancelled), Catroid was the first, and is currently also the most comprehensive implementation of the Catrobat language and IDE. Development started in 2010 and is ongoing.

Bricks Bricks are the smallest building blocks of Catrobat programs. Multiple Bricks stacked together form Scripts. Bricks contain atomic operations from setting variables, manipulations on the Sprite they belong to, up to starting services, for example, starting the camera. Bricks contained in the category "Control" are concerned with the program's control flow and bracket other Bricks, e.g., loops and if-then-else Bricks.

Scripts A Script is an aggregation of Bricks that have to start with a ScriptBrick. ScriptBricks are special Bricks that are triggered on certain events (eg. When program starts, When Touched) to then execute the Bricks in this Script.

Sprite A Sprite object in Catrobat is an object that holds one or more Looks defining its visual representation, Sounds, and Scripts that define its behavior.

Scene A Scene is a compound of multiple Sprites and a background Sprite, and constitutes a "screen" that is shown to the user when the program is run. Scenes and its contained Sprites can not interact with other Scenes, besides trigger a switch to another Scene.

Program / Project A Program in Catrobat is a container for all the Scenes. Programs cannot interact with other Programs. Sharing programs with other users is possible via the Pocket Code Share[2] web service. Downloaded Programs can be modified locally and re-uploaded as remixes.

[1] https://play.google.com/store/apps/details?id=org.catrobat.catroid
[2] https://share.catrob.at/pocketcode/

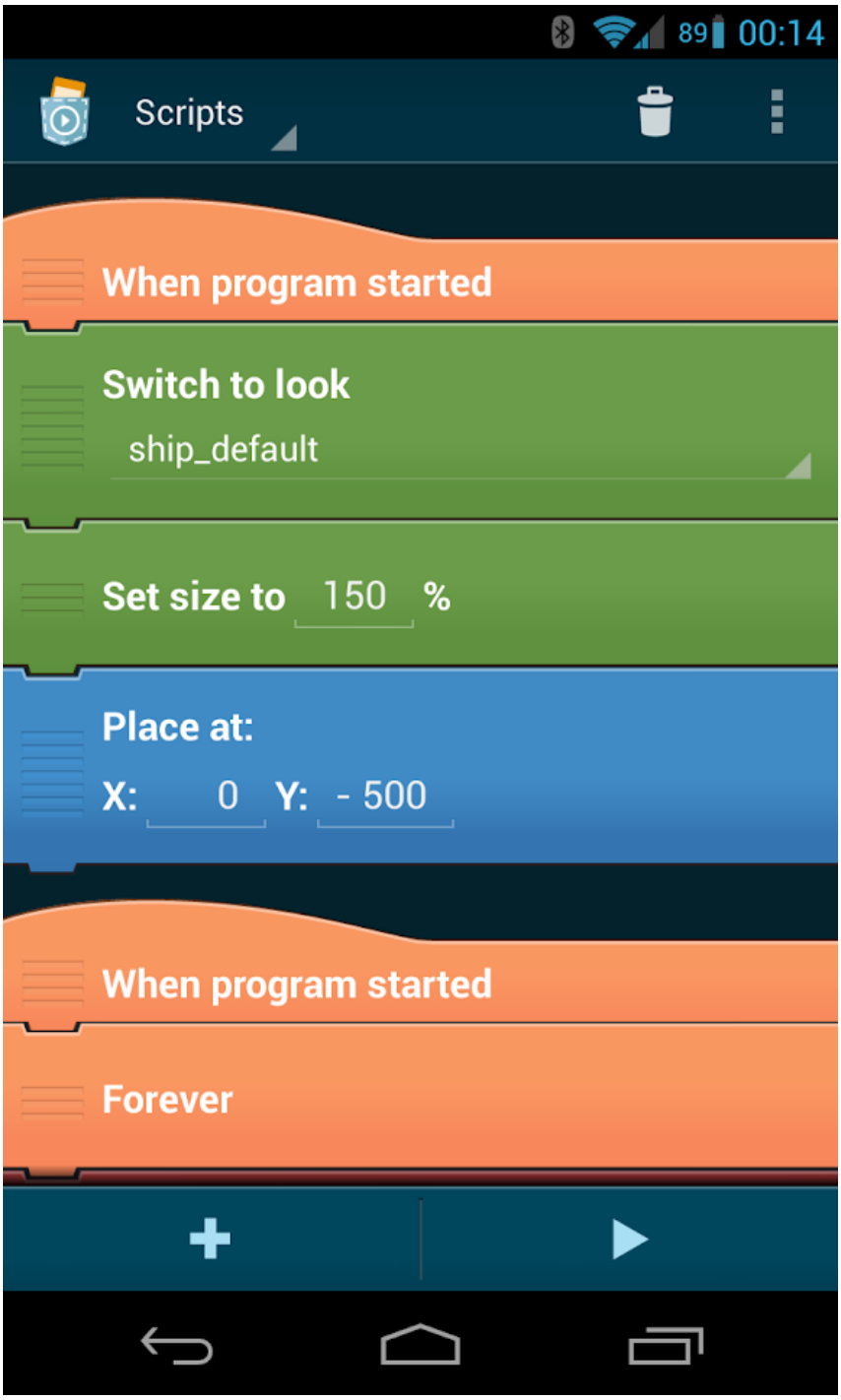

Figure 1.1: An example Catrobat script

Backpack The Backpack is Catroid's functionality for copying objects to other locations of the same Programs or other Programs.

1.1.2 Further implementations

- HTML5 player
  An interpreter for the Catrobat language build on HTML5 that can run Catrobat programs in a browser.
- IOs Pocket
  IOs implementation of the Catrobat interpreter and IDE.

1.2 Testing on Android

1.2.1 Test platforms

Tests are split into two distinct groups, based on the environment they have to run on. See Fig. 1.2. Either a local JVM, or as so called "instrumented" tests that are run on emulators (also called virtual device by google) or real devices. Both have their own advantages and disadvantages. While tests running on local JVM provide a lot of execution speed and stability, running tests on a real device provides the highest fidelity but lower execution speed.

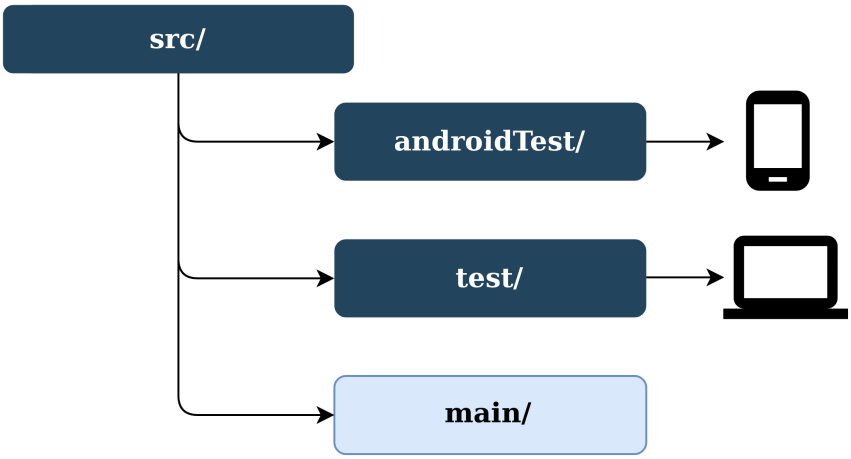

Figure 1.2: Android test packages.

1.2.2 Android architecture and testing

Google, 2019b pushes a model based architectural approach for writing apps, which eases a lot of problems introduced by lifecycle events and provides further means for achieving proper separation of concerns. This in turn will bring a project a step closer to testable code and enable developers to stick closer to the classic test pyramid while also performing a big portion of integration tests as local tests running on a regular JVM.

2 Related work

The following paragraph has been previously published in Hirsch et al., 2019. Numerous papers have been published investigating the testing approach of Android apps quantifying how and why developers test their apps. Silva et al., 2016 performed an exploratory study on open source Android applications in regard to automated testing approaches, framework usage, and size comparison of test suites. Linares-Vásquez et al., 2017 further conducted a study on Android developers to identify and analyze testing practices, tools used, frameworks, and metrics. Coppola, Morisio and Torchiano, 2017 published a study on Android UI (User Interface) test fragility examining test suite evolution over time on a significant sample of open source Android applications. Kochhar et al., 2015 performed a survey on open source mobile applications, analyzing testing practices, testing tool and framework usage, as well as applied metrics, performed by both code analysis and interviews. Yet, little research has been done on the specifics of testing big Android apps.

Google's Android documentation and testing recommendations (Google, 2018a; Google, 2018c; Google, 2019a) are overly simplistic and only contain trivial examples. The actual challenges in testing bigger Android apps are left out completely.

There is a plethora of scientific literature on test automation and test strategy. Most notably Beck, 2003 in his book "Test-driven Development: By Example', Meszaros, 2006 in "XUnit Test Patterns: Refactoring Test Code", and Cohn, 2009 in "Succeeding with agile: software development using Scrum". Alégroth, Steiner and Martini, 2016 in their industrial case study on technical debt in GUI (Graphical User Interface) test suites and their empirical study on GUI testing with a focus on maintenance costs (Alégroth, Feldt and Kolström, 2016). Mar and James, 2006 in their work "Technical Debt and Design Death" discussed technical debt, metrics, and coping strategies for legacy systems. Chen and Wang, 2012 identified GUI test smells and proposed refactoring methods to remove them. Persson and

Yilmazturk, 2004 in their industry experience report on establishing test automation in an existing project, discussing pitfalls, and effects of such an endeavour. Scott, 2019, Cohn, 2019, Fowler, 2012, and Dijkstra, 2014 in technical blog posts discussed the test pyramid.

3 Problem statement

The following subsection has been previously published in Hirsch et al., 2019.

3.1 Catrobat situation in late 2015

The project's tests were heavily relying on the Robotium framework, representing an ice cream cone testing structure as defined by Fowler, 2012 and Scott, 2019. All tests were designed as instrumented tests which required an emulator or a real device for execution. Local unit tests did not exist and most of the tools in use had not been updated to the latest stable version for a long time.

3.1.1 Execution time

Robotium based UI tests are rather slow in execution since Robotium does not provide any useful means for synchronization with the Activity under test. As a result, it is required to introduce additional wait times in the test code. With a rather big number of UI tests in Catrobat's code base, the *sleep* statements in the tests were responsible for an increased test instability and flakiness. The *sleep* statements and the automatic reruns of unstable tests in combination led to test execution times which became unmanageable (e.g., 16 hours as of summer 2016).

3.1.2 Flaky and fragile tests

UI tests are inherently fragile (Coppola, Morisio and Torchiano, 2018; Coppola, Morisio and Torchiano, 2017) and bring some flakiness (Micco and Memon, 2017) with them that has to be dealt with on the Continuous Integration (CI) side. With a UI test suite as big as in the Catrobat project one

has the problem that even in the case of one testrun reaching completion, no assertion about the app's stability can be made due to a high percentage of failing tests in each single testrun. Only with multiple reruns it is possible to identify flaky tests and isolate tests rightfully failing due to regression. Testing stability and consistency ultimately influences the trust in the app's quality and hence the development of new features.

3.1.3 Architectural flaws

Due to the low technical and project specific experience of the developers the practiced TDD (Test Driven Development) approach was based on UI tests rather than unit tests. This was amplified by the lack of experience with mocking frameworks. The high level approach to TDD relying mainly on UI tests introduced a plethora of architectural flaws that stayed undetected for a long period of time. Improper encapsulation, strong binding of components, reliance on background threads and platform services, significantly increased the need for *sleep* statements in the test code.

3.1.4 Test code quality

Since the developers did not focus on test code quality in the past, it was very low at the first point of consideration for this analysis. Although test code quality has been intensively researched and well documented by multiple researchers (e.g., Martin, 2008; Persson and Yilmazturk, 2004; Alégroth, Steiner and Martini, 2016; Alégroth, Feldt and Kolström, 2016), it was common practice in the project that new code had to have tests to reach a high code coverage. This is not a bad approach per se but completely without test development guidelines and de facto no code reviews the test code base started to deteriorate showing various known smells and anti patterns lined out in Section 5.7.1.

3.1.5 Death of a test suite

All of the above points, further amplified through stability issues with the Polidea testrunner, led to a situation in which it was practically impossible to get a run of the test suite to completion due to random crashes. Paired with

the extremely high execution times, making it infeasible to do reruns, and often occurring problems on the Jenkins instance related to uncoordinated updates and changes of the emulators, plugins, OS and drivers, the test suite was disregarded by developers. By that time a workaround to deal with Instrumentation exceptions was implemented on the project's Jenkins instance (introduced about one year later in summer 2016). About 20% of the tests were constantly failing due to the fact that the Catrobat application was modified and further extended. Further, developers still faced considerable flakiness with about another 10% of tests failing at random every run and still 16 hours of runtime for the full test suite. As a result, the Robotium tests were finally abandoned.

3.2 Vital issues summary

- A proper test strategy has to be developed.
- Technical debt in the form of outdated frameworks has to be dealt with by replacing those frameworks and refactoring adjacent code parts.
- Production code testability has to be improved by refactoring.
- Test code quality has to be improved by refactoring.

4 Approach

4.1 Analysis of Android test strategies

This section analyzing Android test strategies has been previously published in Hirsch et al., 2019.

4.1.1 Test strategies for Android in developer documentation and literature

The test pyramid

The test pyramid, as illustrated in Fig. 4.1, is a well established rule of thumb on how to balance different types and layers of tests in a project, which first reached a broader audience through Cohn's 2009 book "Succeeding with agile: software development using Scrum'. Since then the test pyramid has been discussed and promoted by a huge number of researchers, coaches, and developers (such as Fowler, 2012; Wacker, 2015; Dijkstra, 2014; Google, 2019a; Cohn, 2019; Scott, 2019).

Over time slight changes in nomenclature emerged. While Cohn's 2009 test pyramid had the tip of the pyramid labeled as "UI" and the middle layer as "Service", some sources (e.g., Scott, 2019; Wacker, 2015; Google, 2019a) renamed those layers to "System" or "End to End" for the tip and "Integration" for the middle layer. Google, 2019a within its guidelines suggests a 70% - 20% - 10% split for Unit, Integration, and System tests.

Test classification

Unit tests are tests on method or class level (Fowler, 2012; Dijkstra, 2014). Integration tests are defined as testing the full system, or portions of it through a service (Fowler, 2012; Fowler, 2011) or API level (Dijkstra, 2014).

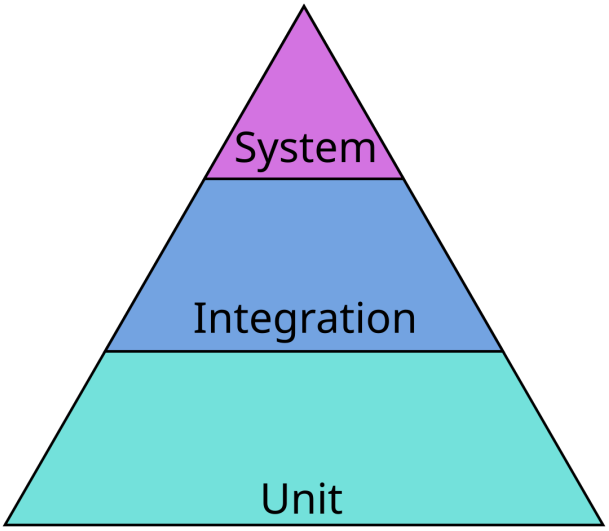

Figure 4.1: The test pyramid according to Cohn, 2009; Fowler, 2012; Wacker, 2015; Dijkstra, 2014.

The border between "unit" and "integration" layers of the test pyramid becomes blurred in practice and its definition is left to the project. This leads to different interpretations of unit and integration tests, which has been observed independently by Fowler, 2014 and Stewart, 2010. Google internally established three categories to differentiate more clearly and to be able to enforce restrictions (Google, 2019a):

- @SmallTest - Single thread, no access to network, file system, external systems, system properties, short runtime.
- @MediumTest - Access to external systems is discouraged, and network limited to localhost.
- @LargeTest - No restrictions, long runtime.

4.1.2 Exploratory study on test strategies in other open source Android applications

A small sample of eight open source Android apps, listed in Table 4.1, has been selected for analysis based on overall size in LoC (Lines of Code),

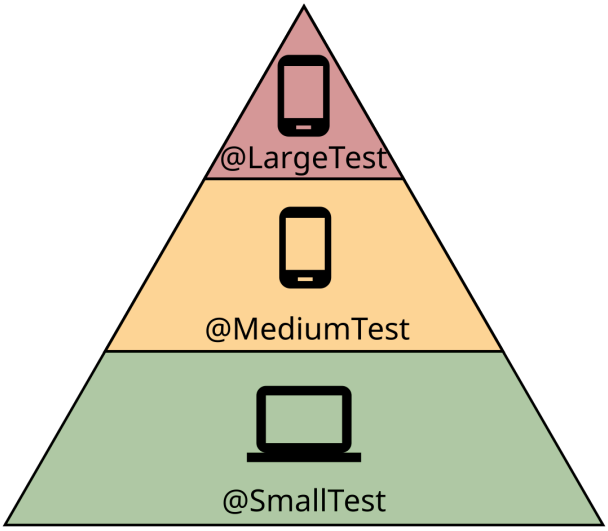

Figure 4.2: The test pyramid for Android as provided by the Android documentation (Google, 2019a).

popularity, and the developers reputation for applying best practice. The test suites are investigated in regard to their size and partitioning.

Sampled Android applications

All analyzed apps' code repositories have been checked out (cloned) for the analysis process in January 2019. For Catrobat, serving as main case of this work, two commits (Catrobat-2015 from September 2015, and Catrobat-2019 from March 2019) have been analyzed. These are also used for the in-depth analysis in Section 3.1 outlining the evolution of a testing strategy and therewith connected guidelines for practice.

In Table 4.1 we list the actual size of production code and test code in LoC of the analyzed applications. Fig. 4.3 shows the normalized proportions of production code and test code of those analyzed applications.

Table 4.2 lists the number of test methods found in the applications' code base split into the categories introduced in the previous Section 4.1.3, i.e., local JVM tests, instrumented tests, and UI tests. In Fig. 4.4 we show the normalized proportions of those categories.

Due to the total lack of tests, the projects Telegram and Timber are

Table 4.1: Production and test code LoC in the analyzed projects

Android Application	**Loc**	
	Production code	***Test code***
AmazeFileManager	30,795	2,533
Catrobat-2015	72,434	36,572
Catrobat-2019	66,983	47,653
ioSched	11,600	4,177
Signal	86,164	603
Telegram	308,100	0
Timber	20,653	0
Wikipedia	53,442	6,725
Wordpress	103,456	4,081

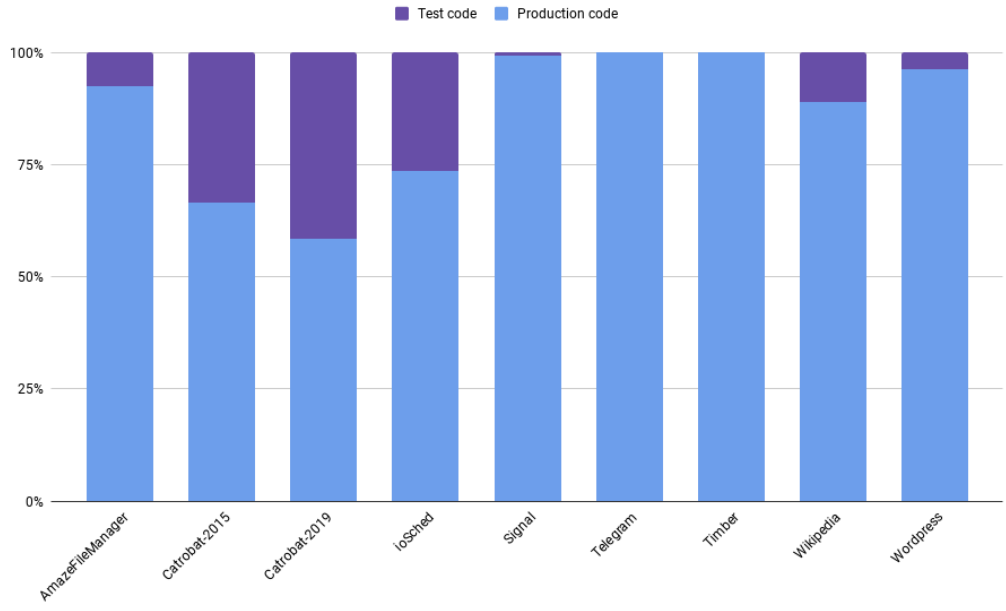

Figure 4.3: Production and test code LoC distribution in the analyzed projects.

Table 4.2: Number of test methods grouped by underlying technology and platform

Android Application	**Number of test methods**		
	JVM unit tests	***Instrumented Unit/Integration tests***	***Ui tests***
AmazeFileManager	119	25	2
Catrobat-2015	0	550	597
Catrobat-2019	682	507	397
ioSched	187	0	9
Signal	31	0	0
Telegram	0	0	0
Timber	0	0	0
Wikipedia	403	5	7
Wordpress	145	121	0

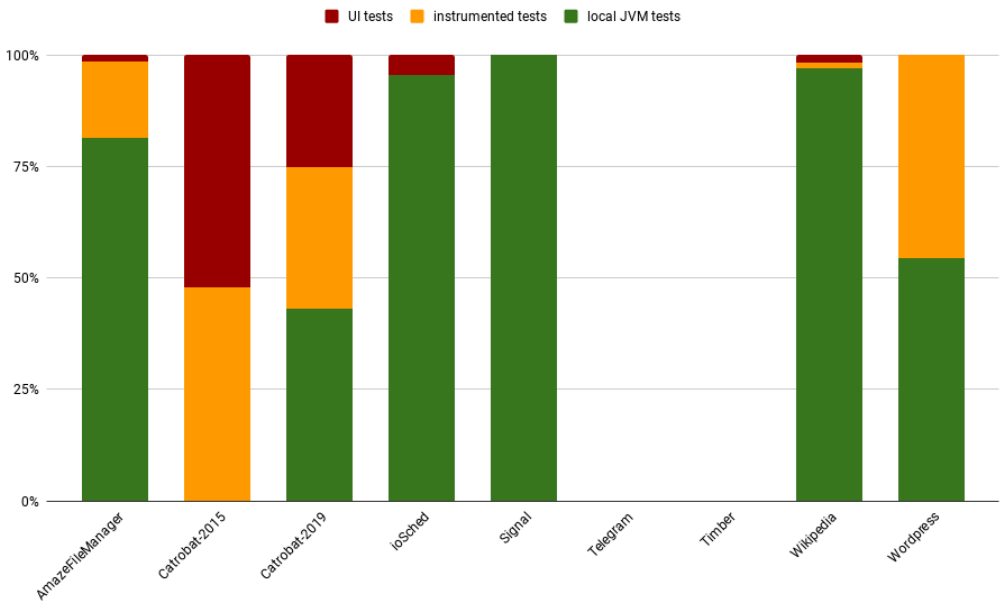

Figure 4.4: Test methods grouped by underlying technology and platform.

excluded from the following discussion. The evolution of Catrobat is investigated and discussed in detail in the sections hereafter. The analysis of the remaining projects shows that four out of five projects heavily rely on local JVM tests, two do not have any UI tests, three use the Robolectric testing framework, and only one uses PowerMock. From the analyzed projects, only two contain parameterized tests to a very limited extent. None of the analyzed projects consistently uses the @SmallTest, @MediumTest, and @LargeTest annotations. In the Wordpress project only one out of 30 test classes, ioSched one out of 42 test classes, AmazefileFileManager two out of 20 test classes, and Wikipedia three out of 79 test classes, were annotated.

This shows that the test pyramid as suggested by the Android developer documentation is not used in practice. Furthermore, it turns out that the majority of the analyzed projects is mainly tested through local JVM tests. Most of these tests would be considered integration and system tests according to the original test pyramid. The use of Robolectric further strengthens this phenomenon. This backs the proposed extended Android test pyramid we introduced in Section 4.1.3.

4.1.3 Proposed strategy - The Android test pyramid

Tests for Android applications are split into two groups (Google, 2018c). "Local unit tests" that reside in src/test and run on a local JVM (Java Virtual Machine) and "Instrumented tests" that reside in src/androidTest and run on an emulator or a connected device. This poses a challenge on how to integrate this split into the well established test pyramid.

The Android Developer guide in the best practice article "Fundamentals of Testing" (Google, 2019a) defines a specific test pyramid for apps, as depicted in Fig. 4.2 with a slightly different approach by categorizing tests into @SmallTest, @MediumTest, and @LargeTest:

- @SmallTest - Unit tests running on a local JVM (tests located in src/test).
- @MediumTest - Instrumented integration tests (tests located in src/androidTest).
- @LargeTest - Instrumented UI tests (tests located in src/androidTest and use a UI test framework, like Espresso, UI Automator, or Robotium).

The split that occurs due to the two different test targets as discussed above is put on the border in between @SmallTest and @MediumTest. At first glance this falls in line with the intent and restrictions of Google's test categories.

Whereas the original definition of @SmallTest is already broad, within the Android context it is further widened and now encompasses significantly more than the definition of *unit* in the original test pyramid. This is amplified by frameworks such as Robolectric and PowerMock that enable developers to execute more tests on a local JVM. Robolectric simulates an Android runtime inside a local JVM. PowerMock enables mocking of constructors, static methods, static initializers, final classes and methods, as well as private methods. In a big Android application with good architecture and a proper separation of concerns the middle layer (@MediumTest) will likely disappear completely, with the majority of tests being located in the @SmallTest and very few in the @LargeTest layer, which also stands in contrast to the 70% - 20% - 10% rule. Integration and system tests, according to the definitions from the original test pyramid, will effectively be categorized into the @SmallTest category as well. We were also able to observe this fact in some open source Android projects investigated in Section 4.1.2.

While the initial intent for those categories was to establish a naming convention to prevent ambiguities, especially in the definition of unit and integration, they lost expressiveness in the Android context.

To achieve a meaningful application of the test pyramid in an Android project, a need for an additional dimension arises. The classical test pyramid is enhanced by additional layers along this dimension representing the test targets and employed technologies. To avoid further confusion, instead of using Google's @SmallTest, @MediumTest, and @LargeTest, following categories are proposed:

- Local JVM tests - Tests running on a local JVM, located in src/test.
- Instrumented tests - Instrumented headless (i.e. without a GUI) tests, located in src/androidTest.
- UI tests - Instrumented UI tests, located in src/androidTest and use a UI test framework, like Espresso, UI Automator, or Robotium.

These categories, shown as the vertical plane in Fig. 4.5, are related to the @SmallTest, @MediumTest, and @LargeTest as defined in the Android documentation but with an emphasis on the test target and frameworks they

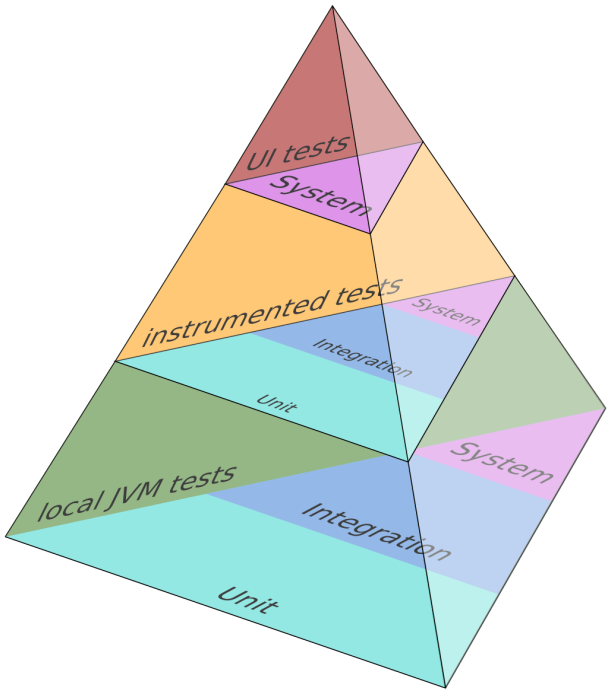

Figure 4.5: Suggested extended Android test pyramid.

use and run on. Further classification of tests contained in those categories is achieved by applying the original test pyramid's categories on each horizontal plane.

The experiences gained in Section 3.1 suggest that a large amount of tests in the middle and top layer, instrumented and UI tests, can be a symptom of insufficient separation of concerns and bad architecture. This further leads to the question if the 70% - 20% - 10% split suggested for the original test pyramid is still applicable in this context and if the partitions' relations also differ in the two horizontal planes. These are questions for future research and need further investigation. However, practice implies that especially in an Android context this illustrated multi-dimensional approach may be a powerful guideline to test big mobile applications.

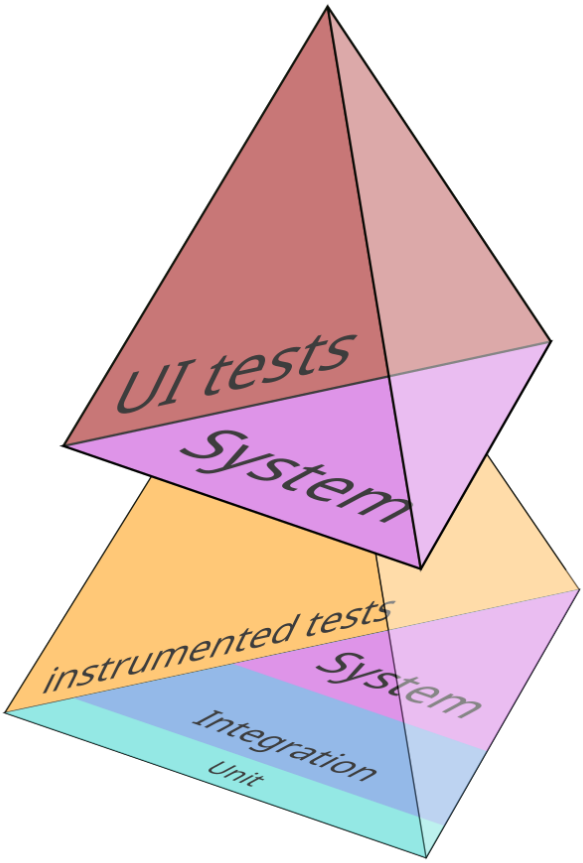

Figure 4.6: Catrobat 2015 test code base.

4.1.4 Status end of 2015 visualized

The status of the test codebase visualized through the proposed Android test pyramid, makes the deviation from the envisioned good test suite structure as displayed in Fig. 4.6 apparent.

4.2 Analysis of UI test frameworks

Only open source Android UI test frameworks are considered for use in the project.

4.2.1 Robotium

Robotium is an open source UI automation testing framework for Android. Version 1.0.0 was released in January of 2010, the last Version 5.6.3 was released in September 2016. It uses Android Instrumentation to interact with the activity under test and it can only interact with the application under test. At its peak robotium was one of the most widespread UI automation testing frameworks for Android (Kochhar et al., 2015; Lämsä, 2017), but has since given way to Appium and Espresso, the latter being the official UI automation testing framework offered and recommended by Google, 2018c.

Execution time Robotium does not provide synchronization measures beyond sleep-retry mechanism to the developers. Robotium internally also operates on this sleep-retry basis, making it noticeably slower than Espresso to begin with, and brings a certain inherent flakiness with it. This leads to developers introducing additional wait times in test code to reduce flakiness. With the large number of UI tests in Catroid this accumulates to run times for test suites that are unmanageable (16 hours as of summer 2016).

Being a 3rd party tool Robotium is a 3rd party framework, and after development went stagnant the choice fell on Espresso, because it is provided by Google and the official recommended framework for Android UI testing (Google, 2018c).

Flakiness Robotium tests show a higher flakiness than Espresso tests to begin with, which has also been shown in other projects (Lämsä, 2017). With a test suite as big as in Catroid this poses the problem that even in the case of a testrun reaching completion the test results are tainted and without multiple reruns no reliable data is provided on the stability of the app.

4.2.2 Espresso

Espresso is an open source UI automation testing framework for Android. In the current version of the Android developer documentation it is the recommended single application UI test framework. It is part of the Android Testing Support Library. Similar to Robotium it uses Android Instrumentation to interact with the activity under test, and can only interact with the application under test. Espresso was envisioned to replace Robotium (Zakharov, 2014). Version 1.1 was released in January 2014, the current stable Version is 3.0.1 released in December 2017.

Synchronization Capabilities The main advantage over Robotium lies in the synchronization capabilities of Espresso. This is achieved by watching the message queue of the UI thread and the async task pool, while also offering means to the developer to lever on its synchronization by introducing the Espresso IdlingResource (Google, 2018a). This provides more stability, speed, and elegant synchronization methods to the developer compared to *sleep* statements in test code.

Extensibility The Espresso API is mostly consisting of

- Espresso entry commands
- ViewMatchers
- ViewActions
- ViewAssertions
- ViewInteractions
- DataInteractions

Espresso was developed as a white box testing tool. It is open for custom extensions. However, it can also be used in a black box testing manner for simple examples. The AndroidStudio test recorder plugin produces such black box tests. ViewMatchers, ViewAction, and ViewAssertions are easily extendable, with Google offering good documentation on those. ViewInteraction and DataInteraction, on the contrary were made package private in Espresso and cannot be extended, which poses significant drawbacks when working with complex UI interactions.

4.2.3 UIAutomator

UIAutomator[1] is an open source UI testing framework for Android, and it is part of the Android Testing Support Library. From version 2.0 on it is based on Android Instrumentation, the latest version is 2.2.0. UIAutomator supports cross application testing, including system applications, but does not support access to internals of the application under test, therefore it can be only used for black box testing. Android documentation suggests using Espresso for single application testing and UIAutomator for multi / cross application testing.

4.2.4 Selendroid

Selendroid[2] is an open source Android UI test framework that brings the Selenium web application test framework to Android. It achieves this by using Android Instrumentation to interact with the emulator or device and offers the developer a Selenium client API for writing tests. The current version is 0.17.0 which was released in 2015, the first release on Github was in 2013. Selendroid only supportsAndroid API levels 10(Gingerbread Android version 2.3.3 - 2.3.7) - 19(KitKat Android version 4.4 - 4.4.4), and allows only black box testing.

4.2.5 Appium

Appium[3] is a cross platform open source UI test framework. Android, IOs, Windows and Mac Desktop are supported. As a driver for Android testing, Selenium is utilized for Android target API 10 (Gingerbread Android version 2.3.3 - 2.3.7) - 16 (Jelly Bean Android version 4.1.x), while UIAutomator2 is used for target APIs above. Tests are written in the WebDriver protocol, and it therefore supports multiple languages for writing tests. The current version is 1.14.0, released in 2019, the first release on GitHub was in 2013. Appium only allows black box testing on Android.

[1]https://developer.android.com/training/testing/ui-testing
[2]http://selenium.googlecode.com/ http://selendroid.io/
[3]http://appium.io/

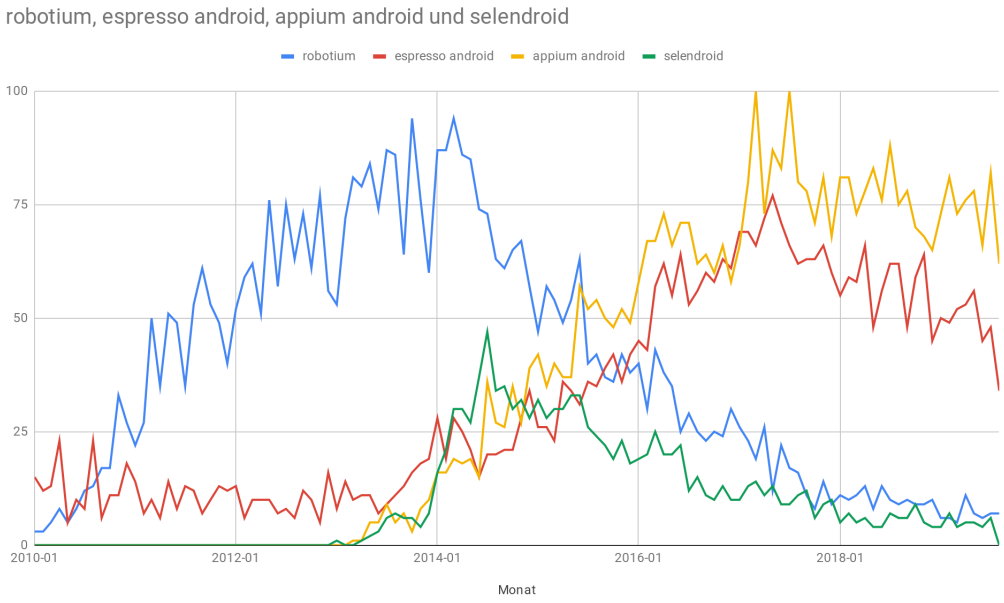

Figure 4.7: Google trends on Android test frameworks.

4.2.6 Other UI test frameworks for Android

Ranorex Android is a proprietary test framework [4].

4.2.7 Spread of UI test frameworks

Analyzing Google trends[5], shows that in 2016 the decline of Robotium's popularity was already apparent, while Appium and Espresso were on the rise as can be seen in Fig.4.7.

4.2.8 Proposed UI test frameworks to replace robotium

Espresso UI test framework was chosen to replace the outdated Robotium framework in the Catroid project. Good benchmarks due to its synchro-

[4]https://www.ranorex.com/de/android-testautomatisierung/
[5]https://trends.google.com

nization capabilities (Lämsä, 2017), wide spread, good documentation, extensibility, and the capability to perform white box testing made it the first choice (Google, 2018a). Google's Android developer guidelines and recommendations also weighed heavily in this decision (Google, 2019a).

4.3 Analysis of Unit test framework and tools

4.3.1 Test runner

PolideaTestRunner[6] was introduced in 2011 to provide XML output which is required for integration with Jenkins CI. At that time the AndroidJunitRunner was not able to create the test results as XML. Unfortunately updates for the PolideaTestRunner ceased in 2013. Furthermore it only supports JUnit3.*, while the AndroidJunitRunner was significantly enhanced in the meantime and is still maintained, now providing proper functionality to be used with Jenkins CI, as well as JUnit4 support.

4.3.2 JUnit version

JUnit4[7] introduced a number of changes compared to JUnit3. Most apparent is the heavy use of Annotations in JUnit4 over the strict naming conventions in JUnit3. Examples of this would be setUp and tearDown methods that now can have arbitrary names and are marked using the @Before and @After annotations. Similar change with test methods, while in JUnit3 the method name had to start with "test", this is now achieved with @Test annotations. Test cases now do not extend TestCase anymore and do not have a common base class. JUnit Rules are now used to capture common behavior of tests, instead of subclassing tests, while at the same time adding more fine grained access than simple setUp and tearDown on the stages of test execution on Android. JUnit4 also offers better support for testing exceptions through Rules and annotations. JUnit4 requires Java 1.5 or higher.

[6]https://github.com/Polidea/the-missing-android-xml-junit-test-runner
[7]https://junit.org/junit4/

4.3.3 PowerMock

PowerMock[8] is an open source JUnit mocking framework extending other mocking frameworks. APIs for extension of either EasyMock, Mockito, or Mockito2 are available, and PowerMock depends on one of those frameworks to be available in the project. Since version 2.0 only JUnit4 and above are supported. PowerMock enables mocking of constructors, static methods, static initializers, final classes and methods, as well as private methods.

4.3.4 Robolectric

Robolectric[9] is an open source testing framework for Android that simulates an Android runtime inside a local JVM, enabling the developer to run tests on a local JVM that normally would require an Emulator or real device. This is useful when mocking is not possible or desired, while still gaining the performance increase of running tests on a local JVM instead of an emulator or real device. Use of Robolectric is endorsed in the Android Developer documentation (Google, 2019a). Version 1.0 was released in 2011, current version is 4.3.

[8]https://github.com/powermock/powermock
[9]http://robolectric.org/

5 Implemented short-term measures

5.1 Switch from Robotium to Espresso

This subsection has been previously published in Hirsch et al., 2019.

All Robotium tests and dependencies have been removed and partially re-implemented in Espresso to get some coverage back that still could not be achieved with unit and integration tests due to the hard to test architecture of the production code.

5.1.1 An object oriented approach to UI testing

A more object oriented approach on UI testing as suggested by various authors (see Leotta et al., 2013; Alégroth, Steiner and Martini, 2016; Chen and Wang, 2012) was applied to deal with test fragility in a more effective way in the future. While Espresso is very open to customization and for extension through custom Actions and Matchers, it is not possible to customize or extend upon its interaction objects (ViewInteraction, DataInteraction) to add functionality to those test abstractions of UI elements. Therefore, wrappers around those Espresso interactions were implemented. The current version of Espresso used in the project (3.0.1) unfortunately still lacks decent support of certain Android UI elements like RecyclerViews. RecyclerViews had to be introduced in a refactoring effort to modernize the UI in 2017. According to the Android developer documentation the more performant RecyclerViews are to be favored over ListViews. Custom Espresso extensions had to be implemented to properly test components using RecyclerViews.

5.1.2 Espresso in action

Espresso relies heavily on Hamcrest Matchers for specifying the targeted view or data. The matched view or data is presented in the form of an Espresso ViewInteraction or DataInteraction, on which assertions or actions can be performed.

Espresso entry points

There are mainly two Espresso entry points to provide the developer with a handle to perform actions and assertions:

`static ViewInteraction onView(Matcher<View> viewMatcher)` Returns a ViewInteraction that fits the given Matcher. This can be used for any kind of view that is currently visible, although for elements inside AdapterViews the usage of onData is encouraged.

```
onView(withText("Continue"))
```

`DatawInteraction onData(Matcher<? extends Object> dataMatcher)`

Returns a DataInteraction for the AdapterView of which the underlying Data fits the given Matcher. This provides a stable access point for elements inside AdapterViews, eg. ListViews, where not all contained objects are currently displayed on the screen. When using onData the developer does not have to deal with scrolling and or making sure that the object is currently displayed when having to perform some action or assertion on it, Espresso takes care of this.

```
onData(instanceOf(Brick.Class))
```

ViewInteraction

ViewInteractions are used to perform actions or assertions on the associated view. The most important methods are:

- `ViewInteraction check(ViewAssertion assertion)`
  To perform an assertion on the view.
- `ViewInteraction perform(ViewAction... action)`
  To perform an action on the view.

Methods of ViewInteraction return again ViewInteraction, to make it possible to stack actions or assertions performed on the same view in succession.

```
onView(withId(R.id.SomeEditText))
    .perform(click())
    .perform(typeText("FooBar"))
    .perform(closeSoftKeyboard())
    .check(matches(withText(FooBar)));
```

Which could be shortened even further since perform takes varargs:

```
onView(withId(R.id.SomeEditText))
    .perform(
    click(),
    typeText("FooBar"),
    closeSoftKeyboard())
    .check(matches(withText(FooBar)));
```

DataInteraction

Most of the time it is necessary to further narrow down the desired View. The most important methods for achieving this:

- `DataInteraction inAdapterView(Matcher<View> adapterMatcher)`
  To Narrow down a specific AdapterView if there are multiple that fit the Matcher in the initial onData.
- `DataInteraction atPosition(Integer atPosition)`
  If there are multiple matching objects contained in the AdapterView. Example: onData(instanceOf(Brick.Class)) .atPosition(3)
- `DataInteraction onChildView(Matcher<View> childMatcher)`
  To handle specific views contained in the AdapterView. Example: onData(instanceOf(Brick.Class)) .atPosition(3) .onChildView(withText(”FooBar”))
  DataInteraction of course also offers the same methods for performing actions or assertions as ViewInteraction:
- `ViewInteraction check(ViewAssertion assertion)`
  To perform an assertion on the view.
- `ViewInteraction perform(ViewAction... action)`
  To perform an action on the view.

```
onData(instanceOf(Brick.Class))
    .atPosition(3)
    .onChildView(withId(R.id.SomeTextView))
    .check(matches(withText("FooBar")))
    .perform(click())
```

DataInteractions can become rather verbose and hard to read.

```
onData(instanceOf(Brick.class))
    .inAdapterView(
        ScriptListMatchers
        .isScriptListView())
    .atPosition(5))
    .onChildView(withId(R.id.SomeSpinner))
    .perform(click())
```

ViewMatcher

Espresso offers a big variety of Hamcrest Matchers that match views, just to list a few simple examples:

- `static Matcher<View> withId(int id)`
  Matches views based on the given resource id.
- `static Matcher<View> withText(String Text)`
  Matches TextViews containing the given text.
- `static Matcher<View> isDisplayed()`
- `static Matcher<View> hasFocus()`

ViewAction

Again a big variety of interactions with the views and/or data acquired with onView and onData, just a few simple examles:

- `static ViewAction click()`
- `static ViewAction longClick()`
- `static ViewAction scrollTo()`
- `static ViewAction typeText(String stringToBeTyped)`
- `...`

ViewAssertion

Providing the means to assert properties of the acquired view:

- `static ViewAssertion doesNotExist()`
  To assert that the view or data matched with the entry point is not existing.
- `static ViewAssertion selectedDescendantsMatch(Matcher<View> selector, Matcher<View> matcher)`
  To bulk assert that all descendents matching selector are also matching the given Matcher.
- `static ViewAssertion matches(Matcher<? super View> viewMatcher)`
  Probably the most important, making it possible to use the vast offer of ViewMatchers as assertions.

```
onView(withId(R.id.SomeEditText))
    .check(matches(withText(FooBar)))
    .check(matches(hasFocus()));
```

An examplary (simple) Espresso test

```
public void newProject() {
  onView(withText(R.string.main_menu_new))
          .perform(click());

  //check if dialog title is displayed
  onView(withText(R.string.new_project_dialog_title))
          .check(matches(isDisplayed()));

  //enter new project name
  onView(withClassName(is(
      "android.support.design.widget.TextInputEditText")))
          .perform(
              typeText("TestProject"),
              closeSoftKeyboard());

  onView(withText(R.string.ok))
          .perform(click());

  //check if orientation dialog is displayed
  onView(withText(R.string.project_orientation_title))
          .check(matches(isDisplayed()));

  onView(withText(R.string.ok))
          .perform(click());

  //open scene
  onView(withText(R.string.default_scene_name))
          .perform(click());

  //add sprite
  onView(withId(R.id.button_add))
          .perform(click());

  //check if new object dialog is displayed
  onView(withText(R.string.new_look_dialog_title))
          .check(matches(isDisplayed()));
}
```

5.1.3 ActivityTestRule

ActivityTestRule is replacing the now deprecated ActivityInstrumentationTestCase2 and is part of the Android testing support library [1]. ActivityTestRule provides the developer with the utilities necessary for testing activities as did the now deprecated ActivityInstrumentationTestCase2 before. However, instead of an inheritance based approach as with ActivityInstrumentationTestCase2, where every test class had to extend upon it ("is a"-approach), ActivityTestRule is a JUnit4 test rule that the test class has as a member ("has a"-approach). This helps pushing further away from inheritance based test structure, as subclassing tests is by now considered an anti pattern (Koskela, 2013; King, 2018; Kainulainen, 2014).

ActivityTestRule provides better and more fine grained set up and tear down methods in relation to the Android Activity lifecycle of the Activity under test. When using the default parameters for the rule on instance creation, the Activity will be started before any methods of the test class are run, that includes set up methods. This is very similar to the lifecycle of ActivityInstrumentationTestCase2, see Fig. 5.1. However, Activity test rule also allows to postpone the Activity start, to enable the developer to do set up or test code before the Activity is created, and then starting the Activity under test by calling ActivityTestRule.launchActivity(Intent intent), see Fig. 5.2.

5.1.4 Espresso code style rules

With a high number of developers writing tests for the Catroid project, to keep things simple, the test base stable, and the readability high, following rules especially for Espresso tests were established.

Separate acquisition and the action on Views

Acquisition of the desired view and action on the view should always be visually separated by a line break. When complexity is growing this is absolutely necessary to keep tests from getting an unreadable mess.

[1] https://developer.android.com/reference/android/test/ActivityInstrumentationTestCase2

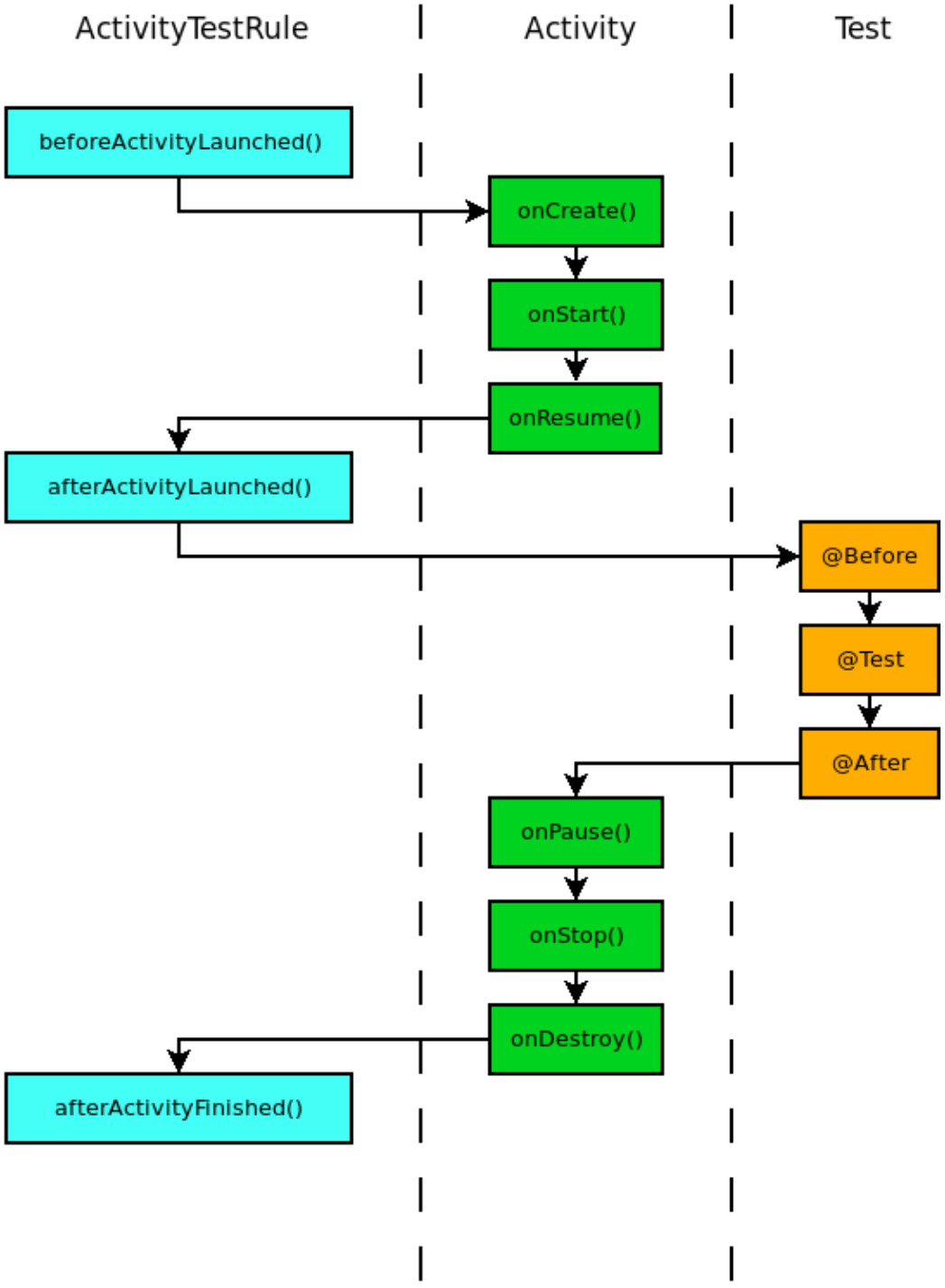

Figure 5.1: ActivityTestRule lifecycle with default parameters.

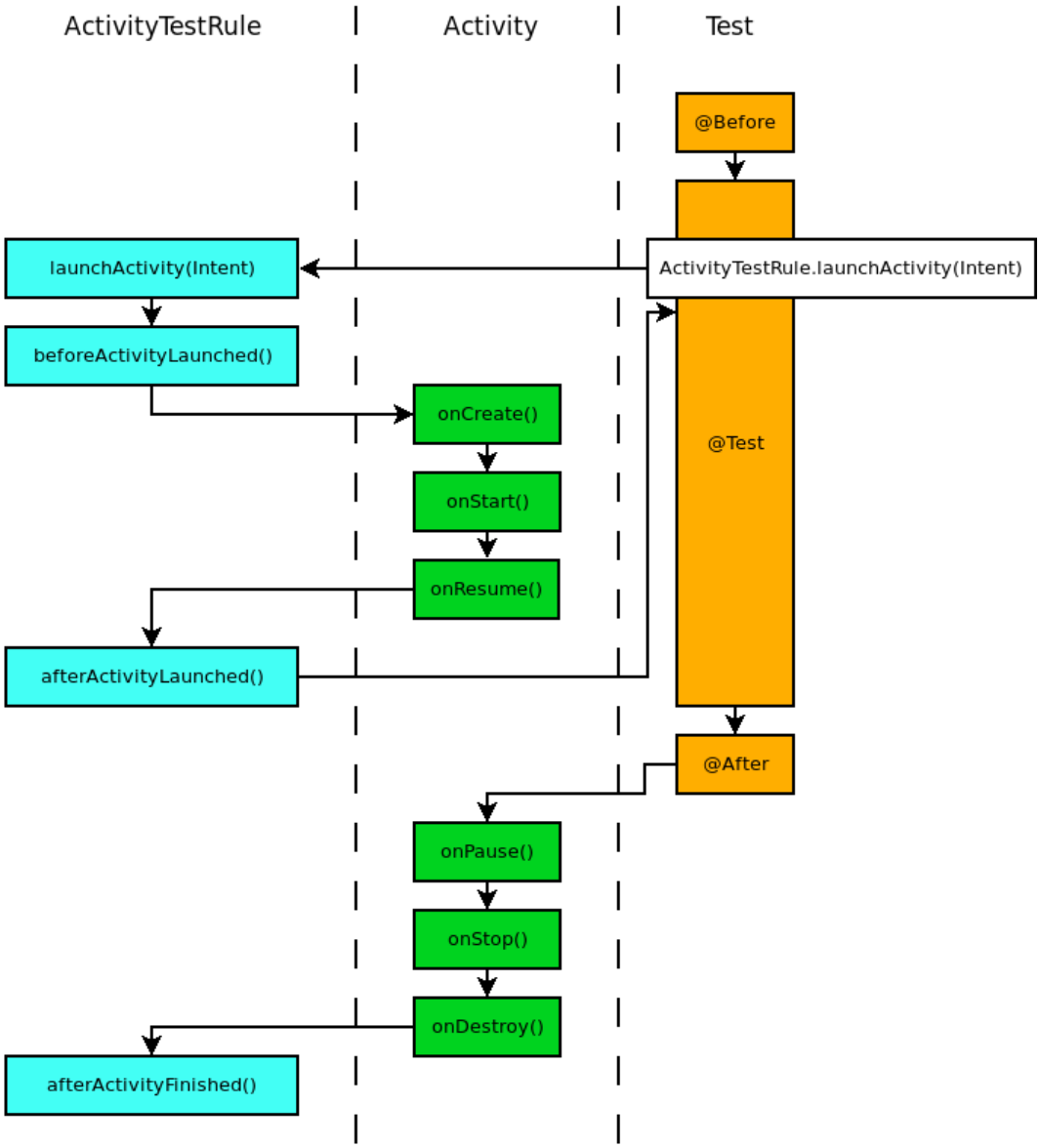

Figure 5.2: ActivityTestRule lifecycle with manual Activity launch.

```
GET_YOUR_DESIRED_VIEW_OR_OBJECT
    .PERFORM_SOMETHING_ON_IT
```

A simple example:

```
onView(withText(SomeString))
    .check(isDisplayed);
```

A more complex example:

```
onData(instanceOf(Brick.class))
    .inAdapterView(ScriptListMatchers
        .isScriptListView())
    .atPosition(5))
    .onChildView(withId(R.id.SomeEditText))
    .perform(click())
    .perform(typeText("FooBar"))
    .perform(closeSoftKeyboard())
    .check(matches(withText(FooBar)));
```

‘
or

```
onData(instanceOf(Brick.class))
    .inAdapterView(ScriptListMatchers
        .isScriptListView())
    .atPosition(5))
    .onChildView(withId(R.id.SomeEditText))
    .perform(click(),
      typeText("FooBar"),
      closeSoftKeyboard())
    .check(matches(withText(FooBar)));
```

Order of methods inside test classes

Following order of methods should be applied:

- `/@Rule`
- `/@Before` and functions only called from here
- `/@Test`
- `/@After` and functions only called from here
- other functions

5.1.5 Espresso wrappers

As already mentioned, Espresso can become rather verbose, especially when using onData and accessing Views inside complex Adapters. Have a look at the following example, where a value is selected from a spinner:

```
onData(instanceOf(Brick.class))
    .inAdapterView(
    ScriptListMatchers.isScriptListView())
    .atPosition(5)
    .onChildView(withId(R.id.SomeSpinner))
    .perform(click())
onData(
    allOf(is(instanceOf(String.class)),
    is(stringToBeSelected)))
    .perform(click());
```

Entry points

The trivial approach of reducing the amount of boiler plate code is by creating methods that serve as entry points.

```
public DataInteraction
      onBrickAtPosition(int position) {
  return onData(instanceOf(Brick.class))
      .inAdapterView(
          ScriptListMatchers.isScriptListView())
      .atPosition(position))
}
```

The example from above would be then reduced to:

```
onBrickAtPosition(5)
    .onChildView(withId(R.id.SomeSpinner))
    .perform(click())
onData(allOf(is(instanceOf(String.class)),
    is(stringToBeSelected)))
    .perform(click());
```

Encapsulating more complex user actions

Custom entry points increase readability quite significantly. However, it is not possible to properly encapsulate more complex interactions in this way. If required often these create a lot of unreadable boiler plate code. There is of course the obvious way of encapsulating such complex behavior in utility methods. This leads to poor readability due to the rapidly growing number of method parameters, and the poor expressiveness of a method name. Above example could be encapsulated in a method like the following:

```
selectItemFromSpinnerOnBrickAtPosition(
      stringToBeSelected, R.id.SomeSpinner, 5)
```

This is not only hard to read, it conflicts with multiple points in Robert C. Martins Clean Code book. (Robert C. Martin - Clean Code: A Handbook of Agile Software Craftsmanship)

- Do one thing
- Number of arguments
- and of course the function name

So there is a need to find a better way for encapsulating complex behavior than in utility methods. Something that is overlooked by many rookie test authors is that tests and their utility methods are not bound to pure imperative programming. An object oriented approach to this problem is not only possible, but beneficial in many cases.

Interaction wrappers

A more object oriented approach to testing is already well established practice in Web UI testing and known as the PageObject Pattern. It has been discussed (see Leotta et al., 2013; Alégroth, Steiner and Martini, 2016; Chen and Wang, 2012) and promoted by multiple researchers and industry specialists (see Fowler, 2013). Espresso can be extended with custom actions and wrappers quite easily, but it is mostly closed for modification. DataInteraction and ViewInteraction are *package private* in Espresso and one cannot extend their functionality through inheritance. As a workaround wrapper classes were introduced, which mimic DataInteraction and ViewInteraction by proxying their existing methods, while offering the possibility to extend

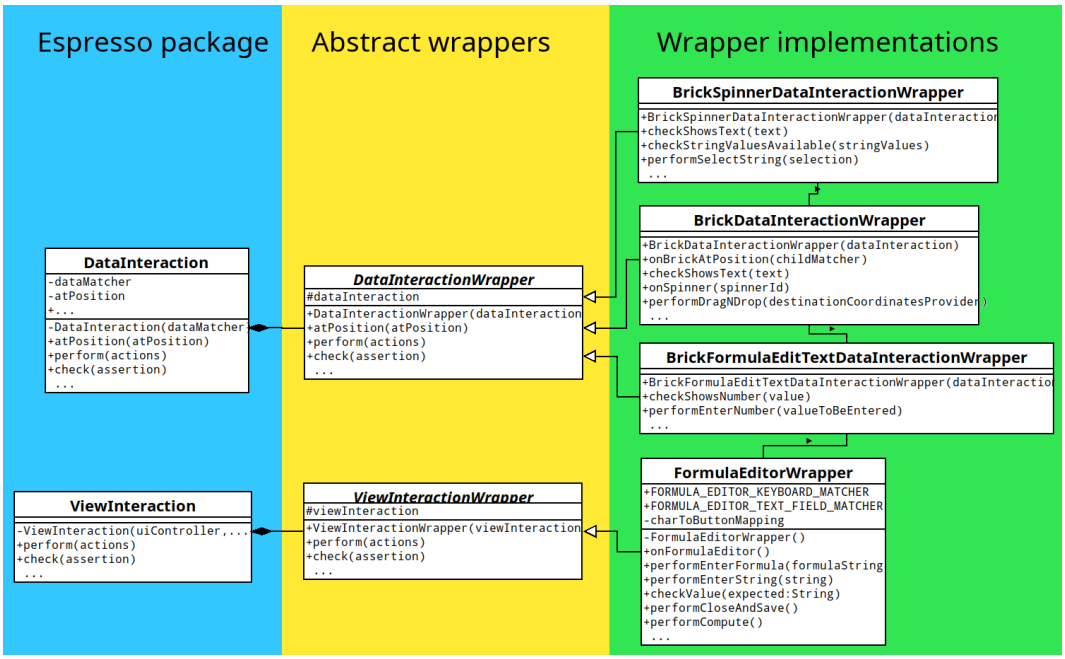

Figure 5.3: InteractionWrapper class diagram.

them. As an example, the BrickDataInteractionWrapper includes the custom entry point, but in contrast to the last example, the entry point returns a BrickDataInteractionWrapper, and therefore acts as a static factory method. This pattern of entry points as static factory methods in combination with wrapper methods creating and returning another object's wrapper enable modelling complex UI models while retaining the simple Espresso call style. A small example class diagram of such a wrapper structure can be seen in Fig. 5.3.

With all of those wrapper classes in place the above example of selecting a spinner item that resides on a Brick will look like this:

```
onBrickAtPosition(1)
    .onSpinner(R.id.SomeSpinner)
    .performSelect(stringToBeSelected)
```

onBrickAtPosition(1) will return a BrickDataInteractionWrapper that represents the Espresso DataInteraction of the highlighted view, seen in Fig. 5.4.

onSpinner(R.id.SomeSpinner) on the above wrapper will return a Brick-

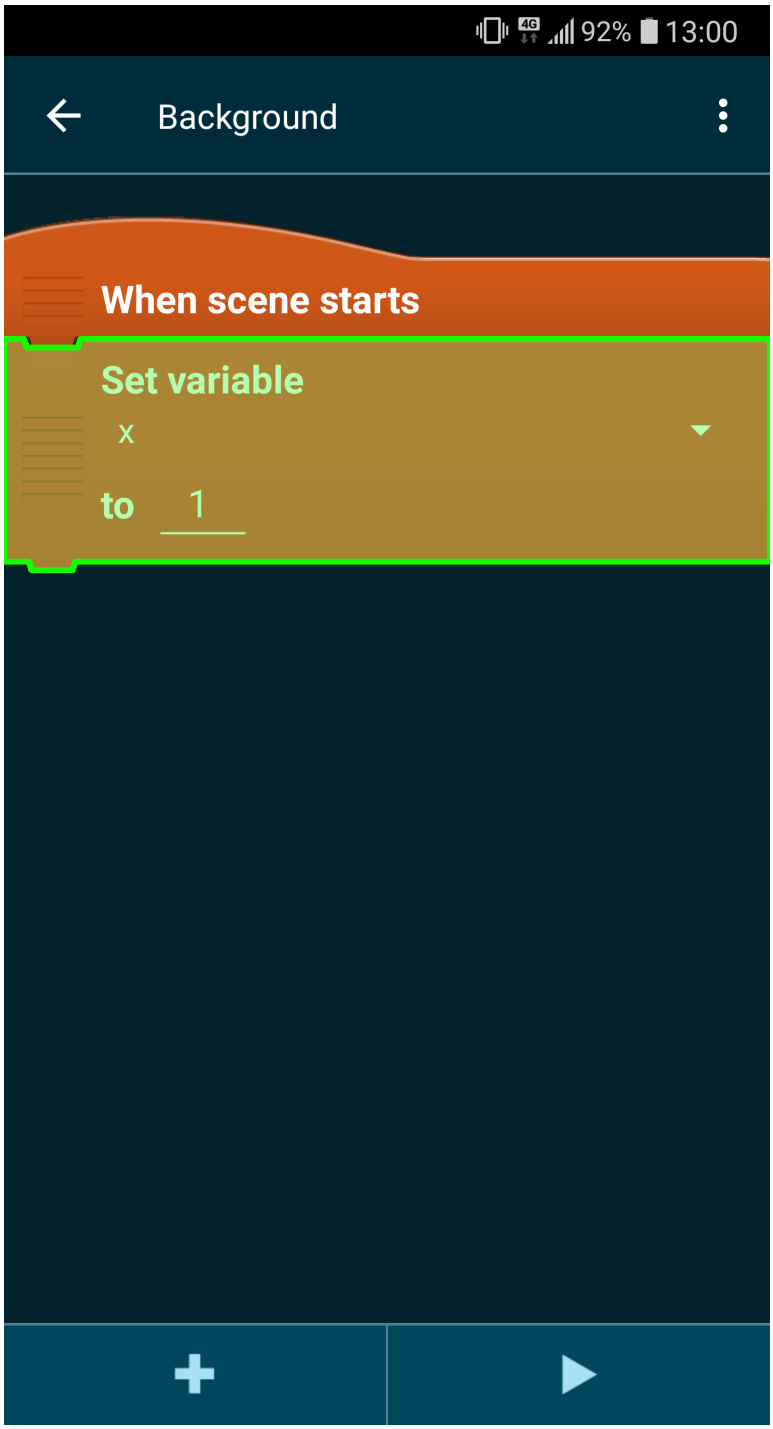

Figure 5.4: BrickDataInteractiongWrapper, onBrickAtPosition(1)

SpinnerDataInteractionWrapper representing the DataInteraction of the Spinner highlighted in Fig. 5.5.

The method performSelect(stringToBeSelected) is a shortcut for clicking the spinner and selecting the desired item from the dropdown. This brings a lot of advantages, first off this is very easy to read and comprehend. The Espresso code style for achieving something by first stating a destination followed by an action stays intact, and the boilerplate code is hidden away. All of which happens in an easily maintainable method that does only one thing and takes only one parameter in most cases. This can be pushed further by wrapping away longer and more complex view interactions, as with the following example of deleting a Brick:

```
public void performDeleteBrick() {
  dataInteraction.perform(
      new GeneralClickAction(
    Tap.SINGLE,
    BrickCoordinatesProvider.UPPER_LEFT_CORNER,
    Press.FINGER));
  onView(anyOf(
      withText(
    R.string.brick_context_dialog_delete_brick),
      withText(
    R.string.brick_context_dialog_delete_script)))
      .perform(click());
  onView(withText(R.string.yes))
      .perform(click());
}
```

The PageObject pattern approach used here is also a useful tool to counteract UI test fragility. By encapsulating the details of the UI objects required for testing in their wrapper classes, it provides a single point for changes if the layout is to be changed in the future. A good example in the Catroid codebase is the FormulaEditorWrapper encapsulating all layout resource IDs. A class diagram can be seen in Fig. 5.6.

5.1.6 RecyclerView testing in Espresso

Although RecyclerView has been promoted as a better alternative to ListView in the Android documentation (Google, 2018b) for a long time, Espresso still lacks proper testing support. The DataInteractions provided by the onData

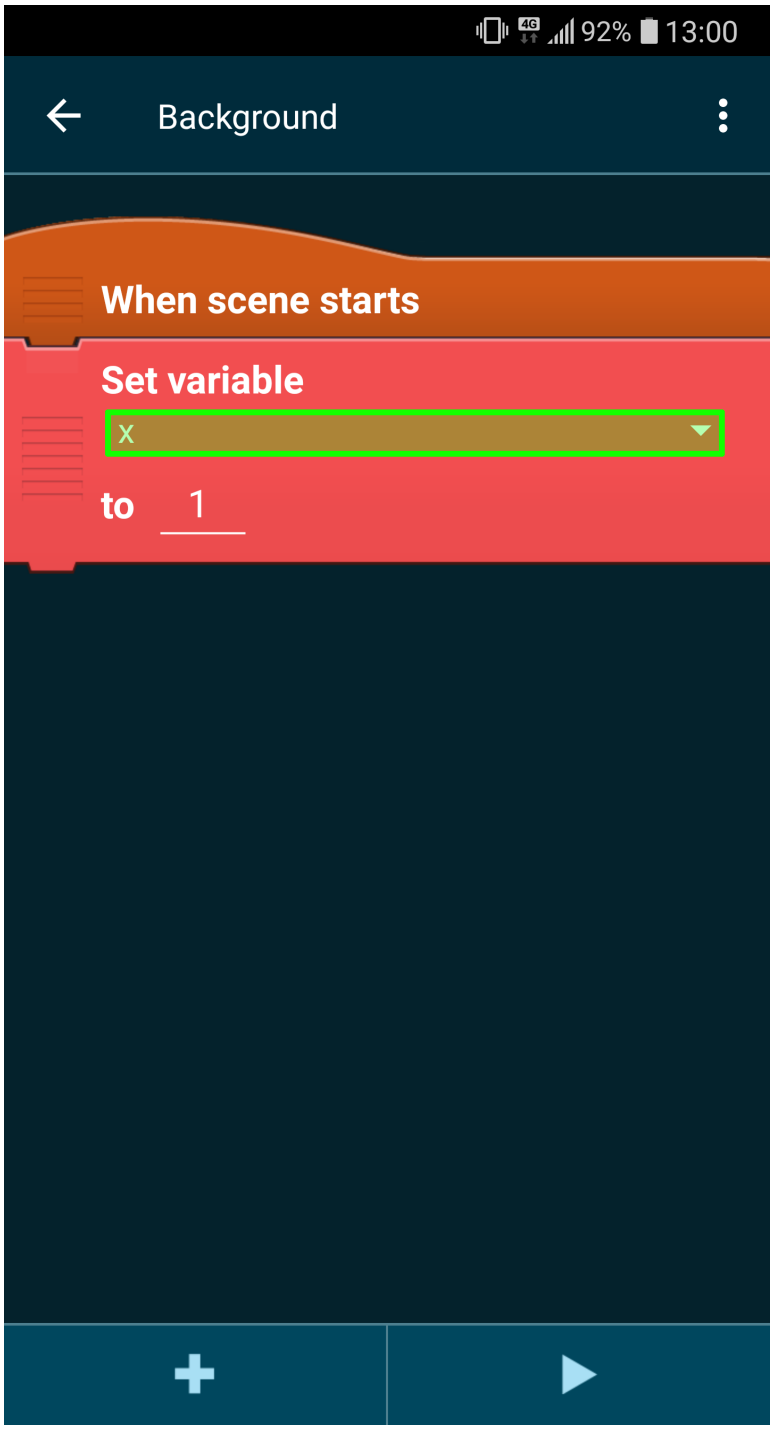

Figure 5.5: BrickSpinnerDataInteractionWrapper, onSpinner(R.id.SomeSpinner)

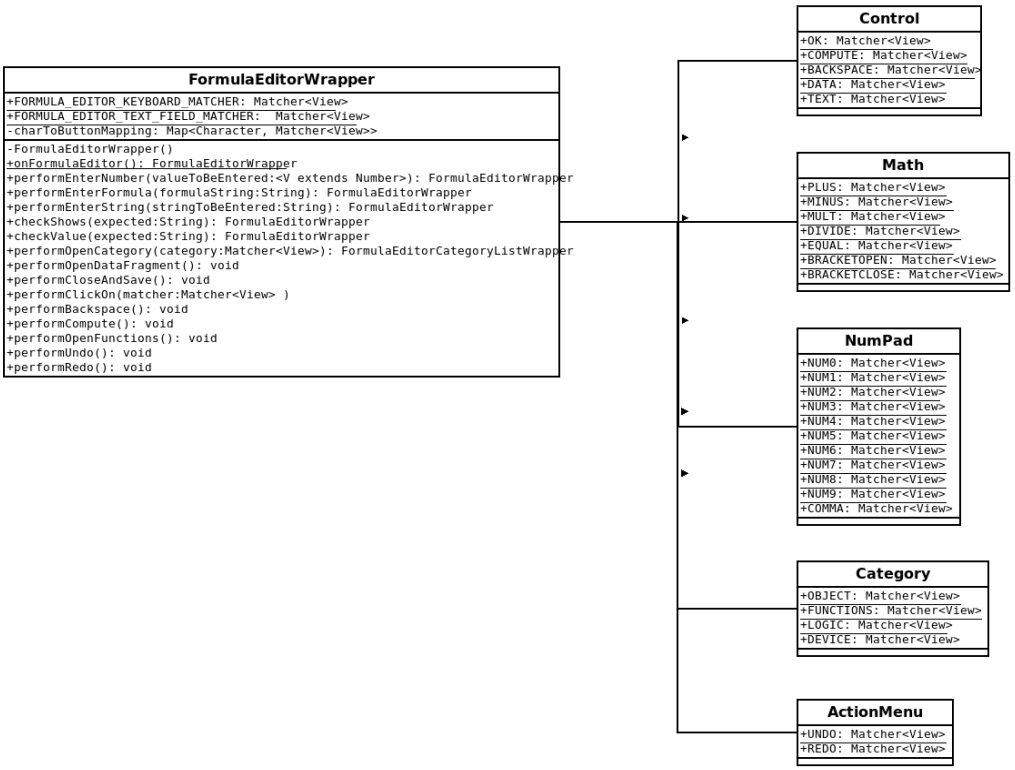

Figure 5.6: FormulaEditorWrapper.

method, while very powerful and well thought out, are not applicable when dealing with RecyclerViews.

Shipped support

The only support for testing RecyclerViews comes in the form of the RecyclerViewActions package [2] from Espresso-contrib. However, it lacks some major functionalities like performing any kind of ViewAssertions on items or their content. Furthermore, it breaks with the Espresso code style of separating definition of a target and the action on it, and offers no support for dealing with more complex ViewHolders that are parent to multiple child Views. ()

The following functionality is supplied:

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

```
static <VH extends RecyclerView.ViewHolder>
    RecyclerViewActions.PositionableRecyclerViewAction
      actionOnHolderItem(
        Matcher<VH> viewHolderMatcher,
        ViewAction viewAction)
//Performs a ViewAction on a view
//matched by viewHolderMatcher.

static <VH extends RecyclerView.ViewHolder>
    RecyclerViewActions.PositionableRecyclerViewAction
      actionOnItem(
        Matcher<View> itemViewMatcher,
        ViewAction viewAction)
//Performs a ViewAction on a view
//matched by viewHolderMatcher.

static <VH extends RecyclerView.ViewHolder>
    ViewAction actionOnItemAtPosition(
      int position,
      ViewAction viewAction)
//Performs a ViewAction on a view at position.

static <VH extends RecyclerView.ViewHolder>
    RecyclerViewActions.PositionableRecyclerViewAction
      scrollTo(Matcher<View> itemViewMatcher)
//Returns a ViewAction which scrolls RecyclerView
//to the view matched by itemViewMatcher.

static <VH extends RecyclerView.ViewHolder>
  RecyclerViewActions.PositionableRecyclerViewAction
    scrollToHolder(Matcher<VH> viewHolderMatcher)
//Returns a ViewAction which scrolls RecyclerView
//to the view matched by viewHolderMatcher.

static <VH extends RecyclerView.ViewHolder>
  ViewAction scrollToPosition(int position)
//Returns a ViewAction which scrolls RecyclerView
//to a position.
```

Espresso code style of separating target matching and action on target RecyclerViewActions obvuscate this separation by having all methods con-

tain the Matcher and the action on the item.

```
onView(ViewMatchers.withId(R.id.recyclerView))
  .perform(
    RecyclerViewActions.actionOnItem(
          withId(R.id.item_edit_text),
          click()));
```

Once the ViewHolder consists of anything more complex than a single View, performing an action on a child View is only possible through creating a custom ViewAction that matches the desired child View. This leads to further mixing responsibilities since one has to write ViewActions that also do matching beforehand.

ViewAssertions

As the name already implies, there is simply no functionality included in RecyclerViewActions to perform any ViewAssertions on the ViewHolder or any child views. So besides performing a scrollTo or scrollToHolder and then using ViewInteraction on the examined View, performing any assertion is not possible. However, using scrollTo is not always feasible and limited to few use cases. For example trying to assert that ”Hello” is displayed exactly on the third item in the RecyclerView is impossible this way.

RecyclerView handling in Catroid tests

To get around the problems and shortcomings of RecyclerViewActions discussed above, a set of custom ViewActions ViewMatchers and ViewInteractionWrappers have been implemented.

RecyclerViewInteractionWrapper extends ViewInteractionWrapper, to present the tester with an interface similar to Espresso’s onData(), but for an underlying RecyclerView. This is limited to accessing items via their position in the RecyclerView. However, by adding another Wrapper for the ViewHolder it enables us to access child Views inside the ViewHolders and to perform any kind of ViewAction or ViewAssertion on them.

The complexity is now encapsulated in those Wrapper and ViewMatcher classes, giving the developer a clear and concise interface for testing, that

again sticks to Espresso's separation of responsibilities and resembles testing lists using onData().

```
onRecyclerView().atPosition(3)
      .onChildView(R.id.some_edit_text)
      .check(matches(
      withText(
        R.string.new_item_name)))
      .perform(click());
```

To match the view inside the RecylcerView at a certain position, as listed in the example above, a custom Matcher had to be implemented. Using Matchers to achieve this enables the developer to continue applying Espresso interactions on the matched view.

```
public boolean matchesSafely(View view) {
  RecyclerView recyclerView =
              (RecyclerView) view.getRootView().findViewById(
                 recyclerViewId);

  return recyclerView != null
              && recyclerView.getId() == recyclerViewId
        && recyclerView.findViewHolderForAdapterPosition(
           position) != null
        && view == recyclerView.
           findViewHolderForAdapterPosition(position).
           itemView;
}
```

Custom Hamcrest Matchers

A plethora of custom Matchers was implemented to either deal with the shortcomings of Espresso or to enable developers to match classes and data from Catroid. Examples of the first category are Matchers for Bundles, RecyclerViews (as discussed above), Toasts, Spinners, and Intents. Examples of the second category of Matchers are UserVariables and UserLists, LibGdx Stage, and various ListAdapters.

Custom Actions

Similar to the custom Matchers above, custom ViewActions had to be implemented due to the same reasons. Espresso does not bring any support for drag and drop handling or swipe actions. Therefore, those were implemented as custom actions, as well as actions for interaction with the LibGdx Stage.

Intent handling in Espresso tests

Espresso does support testing Intents through Espresso.Intents. Espresso.Intents enable the developer to perform assertions on outgoing Intents, as well as catching and returning subbed Intents. This is particularly useful to prevent foreign activities or services to be started in a test environment, while providing the means to perform assertions on outgoing Intents. However, the Intent Matchers provided by Espresso lack some basic functionality that again had to be implemented as custom Matchers.

5.2 Refactoring portions of production code

Portions of production code were refactored with a focus on proper separation of concerns and dependency injection was introduced in some classes that were otherwise difficult to test. This increased testability enabled developers to perform the best practice approach of subcutaneous testing as described in Meszaros, 2006, while simultaneously having the positive side effect of breaking up big classes into smaller ones.

5.3 Testsuites and flaky tests

Introducing a set of custom JUnit rules and splitting UI tests into multiple test suites enabled the Jenkins CI instance to properly deal with test flakiness. These measures limit the number of reruns since only the flaky tests are repeated and prevent crashes of the whole instrumentation process.

5.3.1 FlakyTestRule

JUnit does not provide any means to automatically retry failing tests out of the box. There are various Jenkins plugins, such as the FlakyTestHandler Plugin[3], that retrofit this behavior. Most of those solutions use Java annotations that include the number of maximum retries before marking the test as failed. However, to allow local test runs to deal with flaky tests another approach is required. Android does only provide annotations[4][5] and means for exclusion of flaky tests and not for rerunning of said tests. To provide this rerunning behavior for both local, and Jenkins CI test runs, the desired behavior was implemented as a JUnit rule.

The annotation @Flaky for a test method will rerun the test if it fails. By default the number of reruns is limited to three. This can be overridden in the annotation by adding a parameter. @Flaky(5) for example, will have the test rerun 5 times before marking it as failed and progressing to the next test. To have the test runner account for the annotations the Rule has to be added to the test class.

```
@Rule public FlakyTestRule flakyTestRule = new FlakyTestRule
    ();
```

5.3.2 Category annotations

Annotations for test categories were introduced and the org.junit.experimental.categories Categories.class is used to construct category based TestSuites.

[3]https://plugins.jenkins.io/flaky-test-handler
[4]https://developer.android.com/reference/android/test/FlakyTest.html
[5]https://developer.android.com/reference/android/support/test/filters/FlakyTest.html

```
  //AppUi for all  tests focusing on the pocket
  //code application, so the menus, fragments,
  //lists, and their functionality
  @Cat.AppUi

  //CatrobatLanguage for all  tests focusing on
  //catrobat language correctness (eg. tests
  //verify in stage correctness
  //of some catrobat program
  @Cat.CatrobatLanguage

  //Device for all tests that are required to run /
  //are only runnable on a physical android device
  @Cat.Device
subparagraph
  //Gadgets for all  tests focusing on peripheral
  //hardware and gadgets (eg. RasPi, Drone, LegoNXT)
  @Cat.Gadgets

  //SettingsAndPermissions for all tests probably
  //require some settings change or permission
  //confirmation.
  //(eg. NFC, Bluetooth, Camera, etc)
  @Cat.SettingsAndPermissions

  //Network for all tests that do require an internet
  //connection and or any network services, etc.
  @Cat.Network

  //SensorBox include all tests using the Sensor
  //testing box
  @Cat.SensorBox

  //Regression tests for Api 19(KitKat Android version 4.4 -
     4.4.4), stuff that just works
  //on higher Apis, but needs special treatment on 19
  @Cat.ApiLevel19Regression

  //Educational tests that are in place to demonstrate
  //how to test something
  @Cat.Educational

  //Tests that have side effects that can break other
  //tests, therefore should be excluded from bulk
  //test runs
  @Cat.Quarantine
```

Additional to categories, levels were introduced to distinguish basic functionalities from special use case tests. This annotation should also reflect the number of affected users in case a bug appears in one of the tested functionalities.

```
@Level.Detailed

@Level.Functional

@Level.Smoke
```

This is intended to enable the project to split test runs into smaller packages, once the test suites grow big. For example, to have nightly test runs on all levels, while a pull request triggers on GitHub only run the ones annotated with Smoke and Functional automatically, with the option to manually trigger detailed runs if necessary. All tests are supposed to have the following annotations:

- Either *CatrobatLanguage* or *AppUi*
- Optionally more categories from the above listing.
- *Level*

The following minimal usage example, shows a test that verifies basic behavior of the UI:

```
@Category({Cat.AppUi.class, Level.Smoke.class})
@Test
public void testChangeSizeByNBrick();
```

Test suites based on category annotations

JUnit4's Categories.class can be used to generate a test suite based on a number of listed tests or test suites, and filter those tests according to their category annotations.

```
@RunWith(Categories.class)
@Categories.ExcludeCategory({
    Cat.SettingsAndPermissions.class,
        Cat.Device.class,
        Cat.Educational.class,
        Cat.SensorBox.class,
        Cat.Quarantine.class})

@Suite.SuiteClasses(PhiroIfBrickTest.class,
        ThinkForBubbleBrickTest.class,
        SayBubbleBrickTest.class,
        WhenConditionBrickTest.class,
        ChangeTransparencyByNBrickTest.class,
        BroadcastBricksTest.class,
        ...)
public class FilteredTestSuite {
}
```

This approach requires the developers to manually list the test classes that should be included. However, it is possible to have @Suite.SuiteClasses based on other test suites instead of specifying test classes inline. In the first iteration a bash script was implemented that generates a test suite that generates a list of all available test classes, but later this solution was replaced by a custom JUnit test runner that would automatically include all tests classes in a given package.

5.3.3 PackageTestRunner

JUnit4 doesn't provide any functionality or methods to create test suites based on package paths. Instead, it provides the AllTests.class to emulate JUnit3.x like handling of dynamic test suite generation[6]. This however, requires all test classes to implement a "suite()" method and be added to a specific suite manually. Third party tools like Takari/Cpsuite[7] offer a ClassPathSuite to generate TestSuites based on a given class path.

[6]https://junit.org/junit4/javadoc/4.12/org/junit/runners/AllTests.html
[7]https://github.com/takari/takari-cpsuite

```
@RunWith(ClasspathSuite.class)
@ClassnameFilters(".\*UnitTest"})
public class MySuite {
}
```

These tools however, have compatibility issues with Android projects using Dex. To mitigate these problems a custom JUnit test runner that provides all tests in a given package path was implemented.

```
public class AndroidPackageRunner
    extends ParentRunner<Runner>
```

This custom runner can now be used to build a test suite that is composed of all test classes within the given package path.

```
@RunWith(AndroidPackageRunner.class)
@AndroidPackageRunner.PackagePath(
      "org.catrobat.catroid.uiespresso")
public class AllEspressoTestsSuite {
}
```

The resulting test suite can in turn be used as a basis for filtering with Categories.class to get the desired test suite.

```
@RunWith(Categories.class)
@Categories.ExcludeCategory({
    Cat.SettingsAndPermissions.class,
        Cat.Device.class,
        Cat.Educational.class,
        Cat.SensorBox.class,
        Cat.Quarantine.class})

@Suite.SuiteClasses(AllEspressoTestsSuite.class)
public class PullRequestTriggerSuite {
}
```

With the above solution the developer does not have to add new tests manually to be included in the filtered suites.

Currently set up test suits

As of mid-September 2018 the following test suites are set up and are used by the project's continuous integration system:

- SensorboxTestSuite
  Containing only tests that are run on a real device that is attached to the sensor test box.
- PullRequestTriggerSuite
  Contains all Espresso tests that are runnable on Android emulator instances, do not require the user to perform any actions in Android settings beforehand (eg. activating NFC), and filtered by problematic tests that increase flakiness in other tests.
- ApiLevel19RegressionTestsSuite
  Tests that have to run on API level 19(KitKat Android version 4.4 - 4.4.4).

5.4 Increase coverage through JUnit tests

5.4.1 JUnit 4.0

By abandoning the PolideaTestRunner in favor of the AndroidJunitRunner the number of Instrumentation crashes significantly decreased and all JUnit tests were updated from JUnit3.8 to JUnit4. Most of the work for the update was automatable with a custom python script:

- Recursively traverse the unit test package
- For each test
  - Remove JUnit3.8 imports
  - Add JUnit4 imports
  - Remove inheritance from AndroidTestCase
  - Add @RunWith annotation to class
  - Remove @Override annotations
  - Annotate setUp method with @Before
  - Annotate tearDown method with @After
  - Annotate all test methods with @Test

Structure of an Android JUnit3.8 test:

```
import android.support.test.InstrumentationRegistry;
import android.test.AndroidTestCase;

public class ActionTest extends AndroidTestCase {
    @Override
    protected void setUp() {
        ...
    }
    @Override
    protected void tearDown() {
        ...
    }
    public void testSomething() {
        ...
    }
 }
```

Structure of an Android JUnit4 test:

```
import android.support.test.InstrumentationRegistry;
import android.support.test.runner.AndroidJUnit4;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

import static junit.framework.Assert.assertEquals;

@RunWith(AndroidJUnit4.class)
public class AskActionTest {
    @Before
    protected void setUp() {
        ...
    }
    @After
    protected void tearDown() {
        ...
    }
    @Test
    public void testSomething() {
        ...
    }
 }
```

5.4.2 Replace UI tests with Unit tests

A significant number of UI tests originate from a lack of knowledge about JUnit testing amongst contributors. Replacing those UI tests with JUnit tests without losing coverage was trivial. Mocking frameworks like PowerMock and Mockito enabled the reimplementation of further UI tests as either instrumented JUnit or local JVM tests. The same approach was applied to the existing instrumented JUnit tests, which allowed a substantial number of them (approximately 600 tests, i.e., 50% of the instrumented test suite at that time) to be relocated to the local JUnit package. Despite mocking frameworks, there are limits to the type and number of tests that can be modified to run as local JUnit tests. The main reason for this is that Catrobat relies heavily on third party libraries that in some parts are Android native, precompiled libraries, for example LibGdx.

An example usage of PowerMock to prevent loading of native precompiled library in a static block of a class:

```
public class PhysicsWorld {
    static {
        GdxNativesLoader.load();
    }
    ...
}

@RunWith(PowerMockRunner.class)
@PrepareForTest(GdxNativesLoader.class)
public class PhysicsTest {
    @Before
    public void setUp() {
        PowerMockito.mockStatic(GdxNativesLoader.class);
    }
    ...
}
```

However, this only suffices if loading the native library is only an unwanted side effect of the class under test. If any methods of the native library are to be called during the test run this test must run as an instrumented test.

An example usage of PowerMock to inject mocks into class under test on creation of mocked type:

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(JSONObject.class, ClassUnderTest.class)
public class Test {
    @Test
    public void test() {
        JSONObject response = PowerMockito.mock(JSONObject.
            class);
        PowerMockito.whenNew(JSONObject.class)
            .withArguments(anyString())
            .thenReturn(response);
    }
    ...
}
```

5.5 Quality and performance measuring

Reduction of complexity

This section(5.5) has been previously published in Hirsch et al., 2019.

All inheritance hierarchies in test classes were flattened to reduce complexity and to stay closer to testing best practice suggested by Alégroth, Steiner and Martini, 2016. Wherever necessary, code used in multiple test classes was moved into test objects (see Section 5.1.1) or when more related to the test lifecycle to custom JUnit rules[8].

An example of a test with a deep inheritance hierarchy:

```
public class CollisionBetweenTest extends
   PhysicsCollisionBaseTest {
    ...
}

public abstract class PhysicsCollisionBaseTest extends
   PhysicsBaseTest implements PhysicsCollisionTestReceiver {
    ...
}

public class PhysicsBaseTest extends InstrumentationTestCase
   {
    ...
}
```

Can be refactored into a simple “has-a” relationship to a rule:

```
public class CollisionBetweenTest {
       @Rule
        public PhysicsCollisionTestRule rule = new
           PhysicsCollisionTestRule();
    ...
}
```

Nearly all branching and loop statements were removed from tests during reimplementation, either using parameterized tests or by more thoughtful test design. A parameterized test is a test that is run multiple times with

[8]https://junit.org/junit4/javadoc/4.12/org/junit/Rule.html

differing, predefined inputs. When running the test, each test method and its evaluation are displayed together with an identifier for the specific parameter used. Therefore, in case of a failure, providing the developer the information on which input led to the failure.

Example test with a for loop:

```
@RunWith(JUnit4.class)
public class AngleTest {
    @Test
    public void testAngle() {
        for (int i = 0; i < 4; i++) {
            assertEquals(expected, Rotation.Angle(i*90.0f));
        }
    }
}
```

An example parameterized test, providing the same test coverage as the example above, while also providing more information about a failure to the developer:

```
@RunWith(Parameterized.class)
public class AngleTest {
    @Parameterized.Parameters(name = "{0}")
    public static Iterable<Object[]> data() {
        return Arrays.asList(new Object[][] {
                {0.0f},
                {90.0f},
                {180.0f},
                {270.0f}
        });
    }

    @Parameterized.Parameter
    public float angle;

    @Test
    public void testAngle() {
        assertEquals(expected, Rotation.Angle(angle));
    }
}
```

Unnecessary Try/catch constructs were removed to the test fail directly in case of an exception and provide a stack trace:

```
@Test
public void test() {
    try {
        formula.interpretFloat(sprite);
    } catch (InterpretationException e) {
        fail("Exception")
    }
    ...
}
```

Refactoring the test to throw the Exception, in case of a failure, the developer is provided a stacktrace instead of an assertion error message:

```
@Test
public void test() throws InterpretationException {
    formula.interpretFloat(sprite);
    ...
}
```

Testing expected Exceptions using try/catch inside the test method is hard to read and can be misleading especially when combined with above described try/catch construct.

```
@Test
public void test() {
    try {
        try {
            formula.interpretFloat(sprite);
            fail("Exception not thrown")
        } catch (InterpretationException e) {
        }
    } catch (IOException e) {
        fail("Exception")
    }
}
```

Above example refactored to ExpectedExceptionRule[9] use as suggested by Meszaros, 2006:

```
@Rule
public final ExpectedException exception = ExpectedException.
   none();

@Test
public void test() throws IOException {
    formula.interpretFloat(sprite);
    exception.expect(InterpretationException.class);
}
```

Requiring reflection to enable a test to perform its task is a telltale sign of bad testability of the code under test. Those tests are brittle and maintenance consuming. All reflections to access private members and methods for testing, as well as a big number of stubs were removed and replaced by mocks using the Mockito framework instead. This, of course, required some production code refactorings as well.

Example of using reflection to access a private field of an object:

```
String version = (String) Reflection.getPrivateField(
   projectXmlHeader, "applicationVersion");
```

Android test support library provides the @VisibleForTesting[10] annotations that allow calls to the method from a test context while defaulting the access modifier to private in production use. If another access modifier is desired it can be set as parameter of the annotation.

[9]https://junit.org/junit4/javadoc/4.12/org/junit/rules/ExpectedException.html
[10]https://developer.android.com/reference/kotlin/androidx/annotation/VisibleForTesting

The reflection example above refactored to use @VisibleForTesting annotation on a getter method:

```
public class XmlHeader {
    ...
    @VisibleForTesting
    public String getApplicationVersion() {
        return applicationVersion;
    }
}

public class XmlHeaderTest {
    ...
    @Test
    public test() {
        ...
        String version = xmlHeader.getApplicationVersion()
    }
}
```

Due to the neglect of mocking frameworks in the Catrobat project prior to this thesis, a variety of stubs were found in the test code base. Arguably inelegant, this works with small, simple, and stable classes as in the example below. However this approach does not scale for bigger, more complex classes that are susceptible to change over time.
An example of a File stub:

```
private class TestFile extends File {
    private boolean exists;
    TestFile(boolean exists) {
        super("");
        this.exists = exists;
    }
    @Override
    public boolean exists() {
        return exists;
    }
}

public class Test {
    @Test
    public test() {
        File nonExistingFile = new TestFile(false);
        ...
    }
}
```

Refactored using Mockito Mocks:

```
public class Test {
    @Test
    public test() {
        File nonExistingFile = Mockito.mock(File.class);
        when(existingFile.exists()).thenReturn(true);
        ...
    }
}
```

5.6 Catrobat tests

The conformance of a Catrobat interpreter to the Catrobat languages specification is inherently hard to test in its entirety. The main reason for this is that without elaborate Hamcrest Matchers and Espresso actions it is impossible to perform actions or assertions on the LibGdx rendered display. To further complicate this endeavor, setting up Catrobat projects in tests using Java is very cumbersome, verbose and brings some fragility due to the tight binding to objects within the language. A white box test approach as applied with Espresso is absolutely necessary for such kinds of tests to be implemented. Due to the reasons stated above the Catrobat language and its interpreter were almost exclusively tested with unit tests. However, there is a need to test this on a system level due to the pseudo multithreading event based approach of the language which cannot be adequately tested with unit tests only.

5.6.1 Behaviour Driven Development

Additional to the problematic testing situation described above, the wish for implementation independent tests was expressed by the product owners. This was intended to reduce development effort of Pocket Code implementations for other platforms such as iOS and make them comparable. The requirements for those language tests were as follows: In the spirit of BDD (Behaviour Driven Development) they should be written on a high abstraction level, easily understandable by people without knowledge of the underlying implementation, and provide a test suit that can be used to verify language and interpreter correctness on different platforms and implementations of the Catrobat language. Multiple attempts to introduce BDD tests based on the Cucumber[11] tool into the Catrobat project have failed. Hobl, 2018 basing Cucumber on top of the Espresso framework and struggling with high maintenance efforts and lower reusability than expected. Pokrievka, 2019 basing Cucumber on the already outdated Robotium framework and struggling with stability issues of his test setup. Lessons learned from those thesis (Hobl, 2018; Pokrievka, 2019) are that the introduction of additional test frameworks to provide means for BDD testing adds sig-

[11] https://cucumber.io/

nificant maintenance efforts on multiple platforms while at the same time reducing stability.

5.6.2 Testing Bricks

Example was taken from other programming languages, and means and methods were developed to enable self testing of the Catrobat language. This approach avoids the pifalls (Hobl, 2018; Pokrievka, 2019) of an additional test framework. The Catrobat language was extended with a number of test Bricks, that enable developers to write tests in the language itself. Those tests are directly executable on any other Catrobat implementation supporting Catrobat language version 0.99995 or higher. The test Bricks are only available in development builds of the application, and are deactivated for releases on Google Play.

The Assert equals Brick

The assert equals Brick shown in Fig. 5.7 contains two formula fields, one for the actual value and one for the expected value. Behaviour in case of inequality is similar to JUnit assertEquals. The test result is set to failure including information on the actual value and the position of the assertion Brick that failed, and terminating the test.

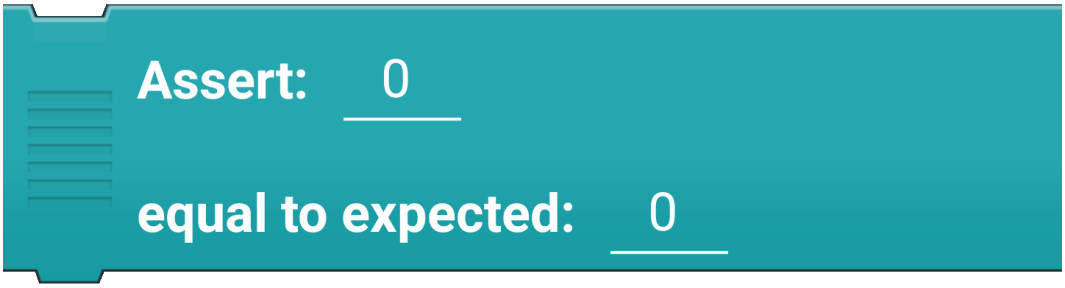

Figure 5.7: Assert equals Brick.

The Single tap Brick

To give the developer minimal means of interaction with the program under test, the single tap Brick was implemented that simply triggers a tap event at specified coordinates (see Fig. 5.8).

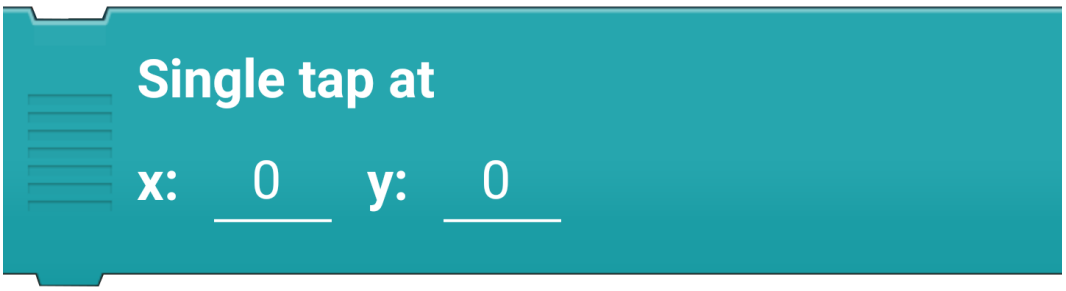

Figure 5.8: Single tap Brick.

The Wait until idle Brick

To get around extensive usage of wait statements in tests, the Wait until idle Brick was added to provide basic synchronization capabilities. This Brick will be blocking as long as this and other actors in LibGdx have scripts running, and will release only if all others have finished (see Fig. 5.9).

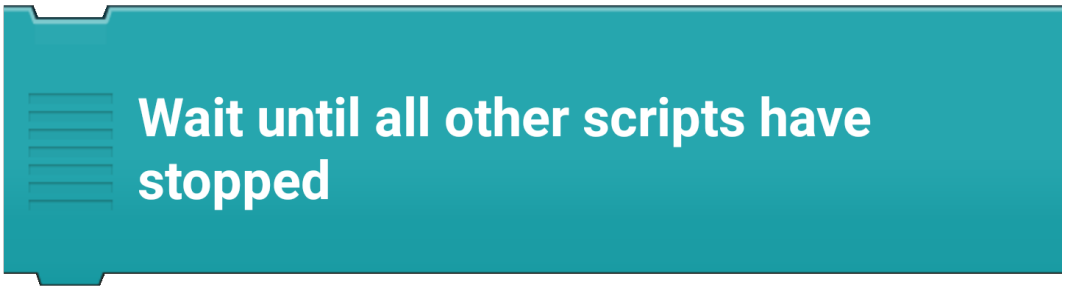

Figure 5.9: Wait until Brick.

The Finish Stage Brick

Catrobat projects do not have an end due to the event based nature of the language. Up to this point a system for the Stage to deliver a result back to the calling Activity was never required. However, for the testing use case it is necessary to specify when a test is done and have means of reporting test results. The Finish Stage Brick sets the test result as Android ActivityResult for the runner and closes the Activity under test. Every test must have such a Brick(see Fig. 5.10).

Figure 5.10: Finish Stage Brick.

5.6.3 Portability

Catrobat projects can be exported and shared easily crossing platform and implementation boundaries. In the current implementation these exported tests are stored in test assets in the Catrobat projects repository. It is planned to move this test suite into a separate repository to allow integration into different implementations of Catrobat.

5.6.4 Test runner

A custom test runner for the language tests was implemented. This runner utilizes AndroidJUnit4 and ActivityTestRule to directly start the Stage Activity for each test project. It is implemented as a parameterized test that automatically collects all available Catrobat tests from the assets package, runs them and evaluates the results.

The initial implementation allowed only a single assertion at the end of the test and relied on Catrobat's variable handling for communication with

the underlying runner. This enabled the runner to provide proper exception handling and failure reporting in the same style as regular JUnit assertions, as well as trivial integration into existing test suites for CI integration and local execution. However, communication through user variables has a number of drawbacks. The latest version of the Catrobat testing Bricks and Catrobat test runner use Android ActivityResult for communication between runner and test. Besides addressing the aforementioned limitations, this reduced the complexity and code size of the test runner and assertion Brick to an absolute minimum, enabled the possibility to evaluate test results on a device, and made it possible to remove all Espresso dependencies of the test runner.

5.7 Ongoing and persistent measures

This Section (5.7) has been previously published in Hirsch et al., 2019.

A number of persistent and long term measures based on the aforementioned findings were introduced to the project. These measures are intended to prevent regression in test code quality, testability, and stability of the test suite.

5.7.1 Educate developers

Developers are being educated in proper use of mocking frameworks like Mockito and PowerMock, JUnit rules, and HamcrestMatchers that allow the creation of Matcher objects and increase readability. Further usage of parameterized tests is encouraged and developers are urged to favor unit tests over UI tests. Educational tests were implemented to serve as templates of proper usage of those tools and to serve as templates for new developers.

Code smells and anti patterns that were in widespread use in the test code base as well as best practices are discussed with developers. The most prevalent anti patterns and code smells in the project were as follows:

- *Obscure test - It is difficult to understand the test at a glance.* (Meszaros, 2006)
- *Conditional test logic - A test contains code that may or may not be executed.* (Meszaros, 2006)

- *Slow test - The tests take too long to run.*
  (Meszaros, 2006)
- *Test logic in production - The code that is put into production contains logic that should be exercised only during tests.*
  (Meszaros, 2006)
- *Assertion roulette - It is hard to tell which of several assertions within the same test method caused a test failure.*
  (Meszaros, 2006)
- *Frequent debugging - Manual debugging is required to determine the cause of most test failures.*
  (Meszaros, 2006)
- *Long macro event - A macro event contains too many actions.*
  (Chen and Wang, 2012)
- *Long parameter list - The parameter list of a keyword or macro event is too long; it is difficult to understand the meaning of each parameter.*
  (Chen and Wang, 2012)
- *Shotgun surgery - Multiple places need to be modified with a single change.*
  (Chen and Wang, 2012)

Due to the fact that the test code base was cleaned and the code quality increased in the immediate measures, tests are now closer to executable specifications and hence work as documentation to showcase how to properly test certain components. This is especially beneficial for developers who are new to the project. However, a re emergence of those smells can only be prevented by code reviews from senior developers that are aware of those issues.

5.7.2 Educate senior developers

An additional education focus was set on the project's senior developers who usually conduct code reviews. The goal is to trigger knowledge transfer within the team about test code quality and to make them aware of smells and anti patterns, which became widespread in the code base. The same level of quality assurance for production code and test code lead to a direct improvement of test code maintainability and readability which in turn flatten the learning curve for new developers. This process of seniors

educating other seniors has to be kept in motion constantly by the projects management.

5.7.3 Code reviews

New and modified test code is now closely examined in frequent code reviews. Developers are encouraged to refactor and clean up old tests if they have to modify them. Unnecessary UI tests are to be removed whenever the same test coverage can be achieved with unit or integration tests. These measures shall prevent the re-emergence of the test ice cream cone structure, and increase the test code quality of legacy tests. From early 2018 on a positive code review is mandatory to be able to merge a pull request. This is currently enforced via configuration on GitHub. Project management must keep this restriction in place, while constant grooming of the approval permission has to be performed.

5.7.4 Persistent monitoring of the test suites

The test suites have to be permanently monitored for flaky tests, their runtime, and their resource requirements. Both, running test suites locally on a developer's machine, and running the test suites on the CI system have to be considered. If any problems arise, measures have to be taken immediately to prevent the test system from deteriorating. Tests are to be partitioned into groups, and depending on their importance and impact, either run on every pull request, or only on nightly runs. This holds especially true for resource or runtime expensive tests. Developers and CI engineers have to collaborate to keep the test suite healthy.

5.7.5 Future of the test suite

All measures described above depend heavily on a project management focus and prioritization on code quality. Due to the high rate of personnel turnover in the project a loss in knowledge can occur very fast leading to a deterioration of the code quality soon after. Currently management pays a lip service to code quality while its actual focus lies on very specific toy projects. Leaving the task of quality assurance and pushing aforementioned

measures to a small group of dedicated senior developers, that feel their efforts being belittled by management. Ideas and valid concerns of these senior members are often dismissed in a rather unfriendly fashion. However, any negative results from the omission of the suggestions, is again blamed on the senior developers. As a cautious reminder, fixing the test suite was a huge effort over the past four years, breaking it can be done within a small number of months. A TDD project without a working test suite is neither TDD nor can it be considered tested. There is little to no value to tests that are not executable. Impact on performance, stability, and maintainability of the product would be severe, as the past has shown.

5.8 Evaluation - Comparison before and after

This section (5.8) has been previously published in Hirsch et al., 2019.

5.8.1 Size of the test suite

Reimplementing a portion of the UI tests as unit tests and encouraging developers to write unit tests rather than UI tests when the same coverage can be achieved has shown a significant change in size of the respective test suites and a slow tendency away from the ice cream cone structure. This is depicted in Fig. 5.11 and Tab. 5.1.

Table 5.1: Number of test methods per group

Code Base	**Test Methods**		
	JVM unit tests	***Instrumented Unit/Integration tests***	***Ui tests***
Catrobat-2015	0	550	597
Catrobat-2019	682	507	397

Again visualizing the status of the test code base through the Android test pyramid comparing it before (see Fig. 5.12) and after (see Fig. 5.13) the refactoring shows significant shift towards a healthier test distribution on the pyramids horizontal and vertical layers. Although system portions in

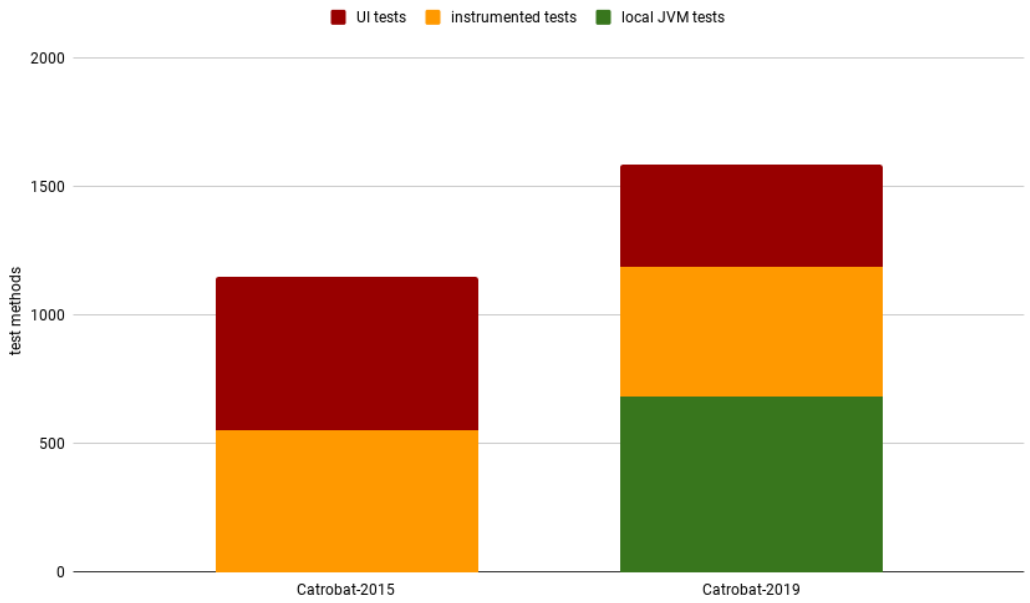

Figure 5.11: Number of test methods grouped by underlying technology and platform.

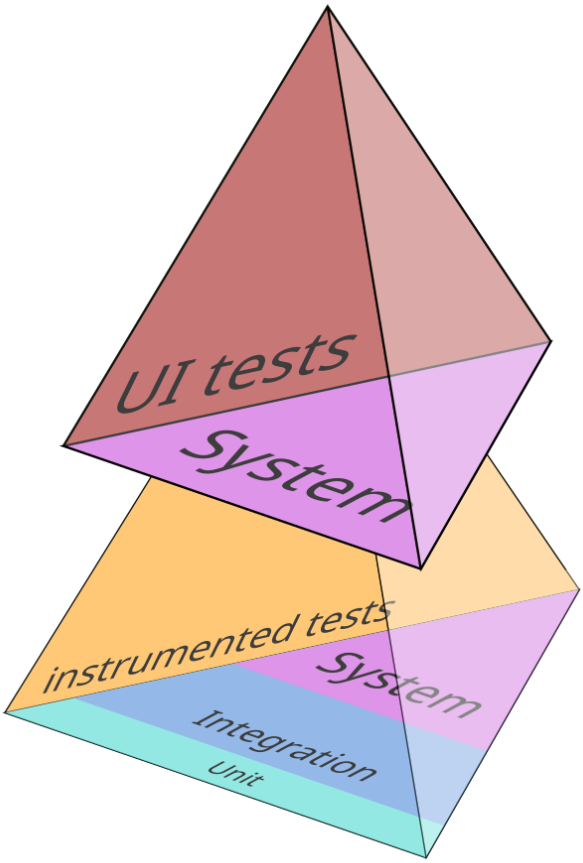

Figure 5.12: Catrobat test code base as of 2015 visualized.

the instrumented layer are still on the heavy side, while unit tests are rather dominant in the local JVM layer.

5.8.2 Test code quality

Test code quality is difficult to pin down in a measurable way. However, LoC per test method, including all utility classes in the total LoC, and average cyclomatic complexity of the test code were chosen as indicators. Both metrics show a significant increase of code quality (i.e., a decrease in those values). The average size (in LoC) of instrumented unit and integration test

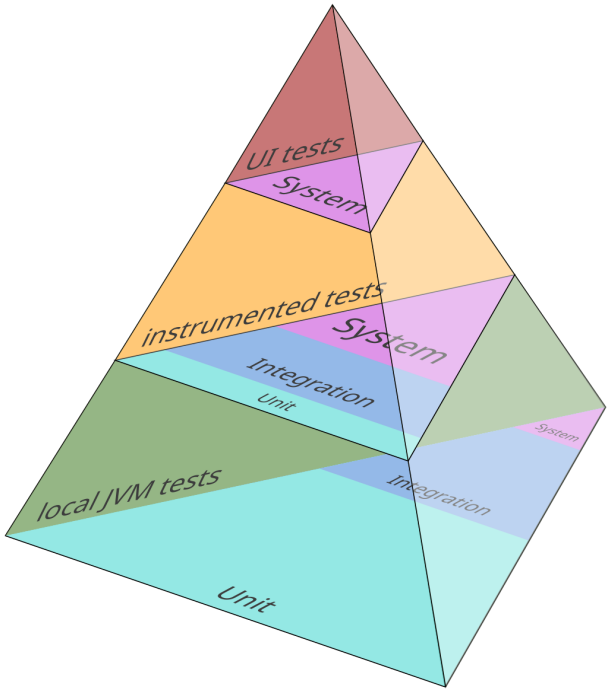

Figure 5.13: Catrobat test code base as of 2019 visualized.

methods shows a reduction of 48.8%, the average size of UI test methods a reduction of 32.8%. The now local unit tests, compared to their former location in instrumented tests show a reduction of 49.5% of LoC, as depicted in Fig. 5.14.

Table 5.2: Loc per test suite.

Code Base	**LoC**		
	JVM unit tests	***Instrumented Unit/Integration tests***	***Ui tests***
Catrobat-2015	232	12,681	23,659
Catrobat-2019	11,946	14,375	21,332

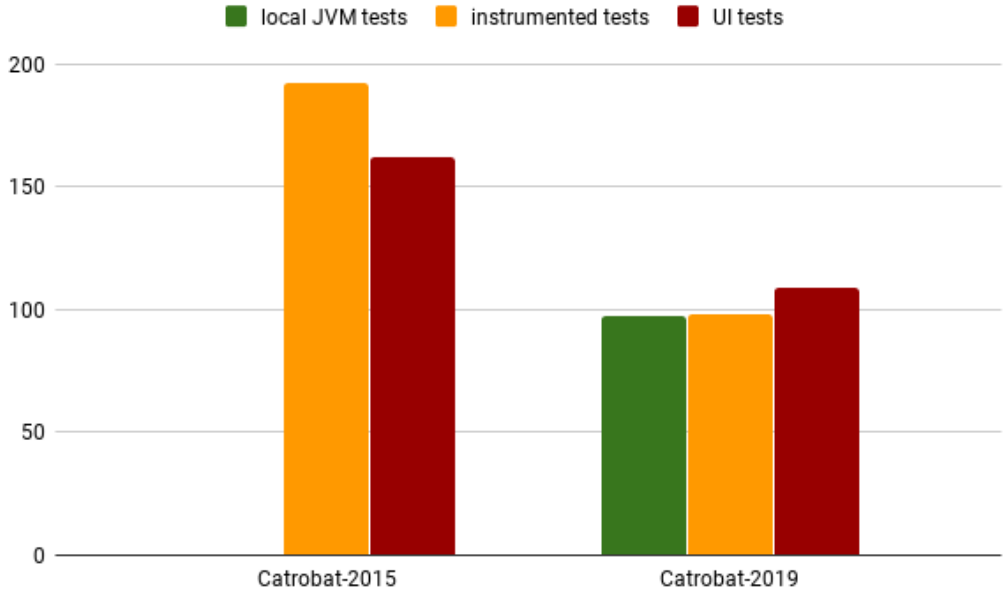

Figure 5.14: Average LoC per test method, grouped by underlying technology and platform.

Average cyclomatic complexity of the unit and integration test package shows a reduction of 18.1%, UI test package shows a reduction of 19.8%, and the now local unit tests compared to their former location in instrumented

unit and integration tests a reduction of 22.6%, as described in detail in Table 5.3.

Table 5.3: Average cyclomatic complexity.

Code Base	**Average cyclomatic complexity**		
	JVM unit tests	***Instrumented Unit/Integration tests***	***UI tests***
Catrobat-2015		1.46	1.49
Catrobat-2019	1.16	1.17	1.22

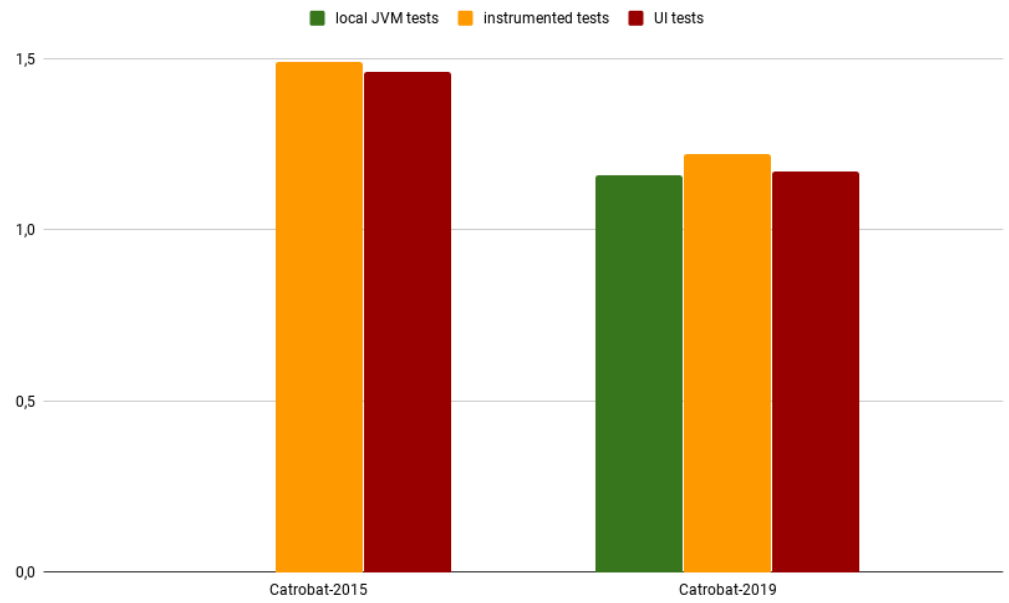

Figure 5.15: Average cyclomatic complexity, grouped by underlying technology and platform.

5.8.3 Test suite runtime

On the project's Jenkins CI instance the execution time for a full test run including all UI and unit tests is now down to an average of 14 min 30 sec, described in detail in Table 5.4. This was achieved by:

- Lowering the number of UI tests by 42%.
- Espresso synchronization capabilities used over *sleep* statements in code.
- Isolation of flaky tests.
- Parallelization efforts on the project's Jenkins CI instance.

Table 5.4: Average runtime for full test suite.

Code Base	**Suite size and runtime**	
	Number of tests	***Average runtime***
Catrobat-2015	1,147	12-16 hours
Catrobat-2019	1,586	14 min

5.9 Proposed followup research

5.9.1 Ratios of the proposed Android test pyramid

The applicability of the 70% - 20% - 10% split suggested for the original test pyramid (Cohn, 2009; Google, 2019a; Wacker, 2015) in the context of the horizontal and vertical planes of the proposed Android test pyramid is still unanswered. This would require performing a study on a large number of Android applications and their test code bases.

5.9.2 Code quality and and product stability

Intuitively good code quality will cut down bug rates and time to fix. Catrobat could serve as a good research project to empirically measure the

effects from introducing static analysis tools as SonarQube[12]. Development resources would have to be dedicated to refactoring hotspots highlighted by those tools. Furthermore, it would be interesting to identify threshold values for code quality metrics that ensure product quality and foster its ratings on the app store.

5.9.3 Catrobat tests as reference

Besides writing more and more Catrobat tests to enable the project to verify correct and consistent behavior of different implementations of the language, care must be taken to not overly inflate the system portion of the projects test pyramid since these tests are to be considered system tests. A study on the effectiveness of a comprehensive test suite of Catrobat tests serving as reference tests for multiple implementations could provide valuable insight regarding the value of those tests. Metrics for quality and code style of those tests could be established. This would require a Catrobat to be implemented on a new platform, in contrast to the iOS implementation, this time with the support of such a test suite.

5.9.4 Study on test suite evolution

Catrobat was suffering from a build-up of technical debt in the form of outdated frameworks and generally being a late adopter of new testing frameworks and tools. A study on the adoption behaviour of other open source Android applications could give valuable insights. Those insights could be compiled into recommendations to support in resource planning for the management of the Application in future.

5.9.5 Production code refactoring

The biggest problem of the project is the production code base which is still far away from being properly testable. Refactoring efforts are still ongoing, given the size of the project (approx. 70.000 LoC in approx. 800 Java classes, excluding all Android XML resources) this is still a major effort in the context of a mobile application. The most challenging parts of this refactoring effort

[12]www.sonarqube.org

will be the reduction of global states in form of singletons which are heavily overused in the codebase and severely impact testability. The insufficient separation of concerns, especially the tight binding to Android or third party APIs is another big problem.

5.9.6 Follow the proposed test pyramid and move tests down the levels

As the aforementioned production code refactoring is ongoing, more tests can be refactored from instrumented unit tests to local unit tests. For new tests an emphasis is put on writing local unit tests instead of instrumented tests. Robolectric is already in use in the projects, and its use should be pushed further to move more tests from emulators or real devices to the local JVM. However this has its limits since the LibGdx[13] engine used for the execution and rendering is tightly bound to Android APIs and partially consists of native libraries. Since LibGdx and Unity are widely used in open source Android applications a study on the testing approach of other such applications could uncover tools, libraries, and patterns that could support Catrobat's developers.

[13]https://libgdx.badlogicgames.com/

6 Conclusion

This section has been previously published in Hirsch et al., 2019.

6.1 Key findings

Updating deprecated test frameworks and tools is a key factor for fast and reliable test suites. Intelligent partitioning of tests into their respective categories along the vertical dimension, established in Android test pyramid shown in Fig. 4.5, further increases stability and decreases runtime of the test suite, while at the same time highlighting architectural shortcomings in the production code. During the partitioning it also became clear that the tests contained in the local JVM and instrumented test category again contain the full spectrum of test classes that the original testing pyramid displays. Reducing the size and complexity of tests, while keeping the coverage level by adding more tests, as well as establishing an object oriented approach to UI tests has shown to be beneficial for maintenance and flattened the learning curve for new developers in the project. Permanent measures such as educating developers and keeping the same quality measures for test and production code are mandatory to keep moving towards a more stable and fast test suite.

6.2 Lessons learned for practice

6.2.1 Technical debt in tests

Architectural flaws

Improper separation of concerns, and other architectural flaws in the production code base will lead to more tests in the instrumented category, requiring them to run on an emulator or real device. Having the majority of

tests in the instrumented category as found in Catrobat is a telltale sign of bad architecture and can lead to maintenance problems downstream. Tools like PowerMockito, that provide functionality as mocking static methods and injecting mocks on new instance creation, or Robolectric, that provides Android APIs on a local JVM, can help moving more tests to the local unit test category. However, the underlying architectural flaws stay uncorrected. Establishing an Android test pyramid incorporating a best practice split for integration and local tests as shown in Section 4.1.3 and conscious usage of frameworks as PowerMock or Robolectric can help uncover architectural flaws in a TDD project at an early stage.

Outdated frameworks and technologies

The Android platform is a fast paced, ever changing platform to build applications upon. Care must be taken to keep frameworks and libraries up to date and embrace new technologies. While this is a well established rule for production code, it can be overlooked when it comes to automated tests. This leads to the accumulation of technical debt as the test code base stays behind on outdated frameworks and technologies, becoming increasingly complex and hard to maintain until either expensive big bang refactorings are undertaken or full test suites have to be dropped. This will reduce the visibility and the coverage of the production code base and therefore bring a reduction in quality and stability to the production code.

Test code quality

The same measures for code quality should be employed for test code as for production code, and enforced in code reviews as suggested by multiple researchers (e.g., Persson and Yilmazturk, 2004; Alégroth, Feldt and Kolström, 2016; Martin, 2008). Metrics that can help visualizing test code quality are cyclomatic complexity, average LoC per test method, runtime, flakiness, and stability. Developers should be educated on testing best practices, frameworks, and design patterns. Developers writing more and better tests lead to more testable production code, which in turn will show better architecture and proper separation of concerns (Martin, 2008).

6.3 General implications from the Catrobat case

The case of Catrobat should serve as a cautionary tale. After multiple thousands of hours of refactoring and reimplementing test and production code it still does not show a healthy test pyramid as suggested above, but has reached again a stable testing environment. This huge endeavor to get back on track was only possible because Catrobat is non profit project and there is practically no time constraints / release pressure and as an open source projects has a reasonable big workforce of contributors, mostly consisting of students, who put in thousands of hours to further develop this project.

6.4 Reflection

This thesis, and the test suite of Catrobat, benefited significantly from the long time span it covered. This enabled the writer to analyze problems, including the writer's own code and solutions in depth that otherwise would have been "quick fixed". This also provided the writer a chance to build up his development skills. The combination of aforementioned points results in little to no code in the test base that was written once and stayed untouched by the writer, constantly increasing code quality. The biggest drain of development hours due to multiple reimplementations arose from the order in which the work was done. At the beginning of this thesis the UI test suite and CI system was broken, therefore priority was given reimplementation of those tests to stabilize the CI system. As the work progressed to update Junit tests and introduction of testing frameworks as Robolectric and PowerMock a significant number of UI tests were again reimplemented, this time as unit tests. Although counter intuitive, the fastest way would have been to focus on updating the working portion of the test suite, the unit tests, before dealing with the broken UI tests. Research has to be done on the proposed Android test pyramid, its validity on a greater scale and its ratios (See 5.9). The effectiveness of Catrobat tests serving as reference tests for multiple implementations (See 5.9) has to be investigated.

6.5 Summary

Automated testing on Android forces developers to separate their test code base into two distinct categories, instrumented and local JVM tests, based on the target they are run on. It is not trivial to incorporate this split in the well established test pyramid. The simplistic approach suggested in the Android developer documentation places this split right on the border between unit and integration tests. Investigation into some well established open source projects in Section 4.1.2 show that this is not applicable in practice and led us to propose the extended Android test pyramid that adds a further dimension. This serves as a tool to properly visualize and categorize the test suites of Android applications, raising awareness that the test pyramid does play a role on on multiple levels, the test target on Android, local JVM tests, or instrumented tests. The question if the 70% - 20% - 10% split suggested for the original test pyramid is still applicable in this context leaves room for future research. The presented case of Catrobat illustrates the effects of these problems. This underpins that the emergence of mobile and distributed technologies is challenging theoretical models in practice and require further work to be applied in big projects.

Appendix

Bibliography

Alégroth, Emil, Robert Feldt and Pirjo Kolström (2016). 'Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing'. In: *Information and Software Technology* 73, pp. 66–80 (cit. on pp. 7, 10, 84).

Alégroth, Emil, Marcello Steiner and Antonio Martini (2016). 'Exploring the presence of technical debt in industrial gui-based testware: A case study'. In: *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, pp. 257–262 (cit. on pp. 7, 10, 29, 40, 60).

Beck, K. (2003). *Test-driven Development: By Example*. Kent Beck signature book. Addison-Wesley. ISBN: 9780321146533 (cit. on p. 7).

Chen, Woei-Kae and Jung-Chi Wang (2012). 'Bad smells and refactoring methods for gui test scripts'. In: *2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. IEEE, pp. 289–294 (cit. on pp. 7, 29, 40, 71).

Cohn, Mike (2009). *Succeeding with agile: software development using Scrum*. Pearson Education (cit. on pp. 7, 13, 14, 79).

Cohn, Mike (2019). *The Forgotten Layer of the Test Automation Pyramid*. accessed 17.4.2019. URL: `https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid` (cit. on pp. 8, 13).

Coppola, Riccardo, Maurizio Morisio and Marco Torchiano (2017). 'Scripted GUI Testing of Android Apps: A Study on Diffusion, Evolution and Fragility'. In: *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, pp. 22–32 (cit. on pp. 7, 9).

Coppola, Riccardo, Maurizio Morisio and Marco Torchiano (2018). 'Maintenance of Android Widget-based GUI Testing: A Taxonomy of test case modification causes'. In: *2018 IEEE International Conference on Software*

Testing, Verification and Validation Workshops (ICSTW). IEEE, pp. 151–158 (cit. on p. 9).

Dijkstra, Bas (2014). *The test automation pyramid*. accessed 17.4.2019. URL: https://www.ontestautomation.com/the-test-automation-pyramid/ (cit. on pp. 8, 13, 14).

Fowler, Martin (2011). *SubcutaneousTest*. accessed 17.4.2019. URL: https://martinfowler.com/bliki/SubcutaneousTest.html (cit. on p. 13).

Fowler, Martin (2012). *TestPyramid*. accessed 17.4.2019. URL: https://martinfowler.com/bliki/TestPyramid.html (cit. on pp. 8, 9, 13, 14).

Fowler, Martin (2013). *PageObject*. accessed 17.4.2019. URL: https://martinfowler.com/bliki/PageObject.html (cit. on p. 40).

Fowler, Martin (2014). *UnitTest*. accessed 17.4.2019. URL: https://martinfowler.com/bliki/UnitTest.html (cit. on p. 14).

Google (2018a). *Espresso target audience*. accessed 17.4.2019. URL: https://developer.android.com/training/testing/espresso/ (cit. on pp. 7, 23, 26).

Google (2018b). *ListView*. accessed 17.4.2019. URL: https://developer.android.com/reference/android/widget/ListView (cit. on p. 43).

Google (2018c). *Test your app*. accessed 17.4.2019. URL: https://developer.android.com/studio/test/ (cit. on pp. 7, 18, 22).

Google (2019a). *Fundamentals of Testing*. accessed 17.4.2019. URL: https://developer.android.com/training/testing/fundamentals (cit. on pp. 7, 13–15, 18, 26, 27, 79).

Google (2019b). *Guide to app architecture*. accessed 17.4.2019. URL: https://developer.android.com/jetpack/docs/guide (cit. on p. 5).

Hirsch, Thomas et al. (2019). 'An Approach to Test Classification in Big Android Applications'. In: *Proceedings - Companion of the 19th IEEE International Conference on Software Quality, Reliability and Security QRS-C 2019* (cit. on pp. v, viii, 7, 9, 13, 29, 60, 70, 73, 83).

Hobl, Markus (2018). 'Behaviour-driven development of a 3D programming environment'. MA thesis. Graz University of Technology (cit. on pp. 66, 67).

Kainulainen, Petri (2014). *Three Reasons Why We Should Not Use Inheritance In Our Tests*. accessed 17.4.2019. URL: https://www.petrikainulainen.net/programming/unit-testing/3-reasons-why-we-should-not-use-inheritance-in-our-tests/ (cit. on p. 35).

King, Tim (2018). *Test::Class Hierarchy Is an Antipattern*. accessed 17.4.2019. URL: http://blogs.perl.org/users/tim_king/2018/05/testclass-hierarchy-is-an-antipattern.html (cit. on p. 35).

Kochhar, Pavneet Singh et al. (2015). 'Understanding the test automation culture of app developers'. In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, pp. 1–10 (cit. on pp. 7, 22).

Koskela, Lasse (2013). *Effective Unit Testing*. ISBN: 9781935182573 (cit. on p. 35).

Lämsä, Tomi (2017). 'Comparison of GUI testing tools for Android applications'. MA thesis. University of Oulu. URL: http://jultika.oulu.fi/files/nbnfioulu-201706142676.pdf (cit. on pp. 22, 26).

Leotta, Maurizio et al. (2013). 'Improving test suites maintainability with the page object pattern: An industrial case study'. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, pp. 108–113 (cit. on pp. 29, 40).

Linares-Vásquez, Mario et al. (2017). 'How do developers test android applications?' In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, pp. 613–622 (cit. on p. 7).

Mar, Kane and Michael James (2006). *Technical Debt and Design Death* (cit. on p. 7).

Martin, Robert C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR. ISBN: 0132350882, 9780132350884 (cit. on pp. 10, 84).

Meszaros, Gerard (2006). *XUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR. ISBN: 0131495054 (cit. on pp. 7, 50, 63, 70, 71).

Micco, John and Atif Memon (2017). *How Flaky Tests in Continuous Integration: Current Practice at Google and Future Directions*. accessed 17.4.2019. URL: https://developers.google.com/google-test-automation-conference/2016/presentations (cit. on p. 9).

Persson, Christer and Nur Yilmazturk (2004). 'Establishment of automated regression testing at ABB: industrial experience report on'avoiding the pitfalls''. In: *Proceedings. 19th International Conference on Automated Software Engineering, 2004.* IEEE, pp. 112–121 (cit. on pp. 7, 10, 84).

Pokrievka, Milš (2019). 'Platform Independent Hardware Tests with Behaviour Driven Development'. MA thesis. Graz University of Technology (cit. on pp. 66, 67).

Scott, Alister (2019). *Testing Pyramids & Ice-Cream Cones*. accessed 17.4.2019. URL: https://watirmelon.blog/testing-pyramids/ (cit. on pp. 8, 9, 13).

Silva, Davi Bernardo et al. (2016). 'An analysis of automated tests for mobile Android applications'. In: *2016 XLII Latin American Computing Conference (CLEI)*. IEEE, pp. 1–9 (cit. on p. 7).

Stewart, Simon (2010). *Test Sizes*. accessed 17.4.2019. URL: https://testing.googleblog.com/2010/12/test-sizes.html (cit. on p. 14).

Wacker, Mike (2015). *Just Say No to More End-to-End Tests*. accessed 17.4.2019. URL: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html (cit. on pp. 13, 14, 79).

Zakharov, Valera (2014). *Espresso devs comment*. accessed 17.4.2019. URL: https://stackoverflow.com/questions/20046021/google-espresso-or-robotium/20487527 (cit. on p. 23).