Thomas Schranz, BSc

Refactoring: Reintroducing Architecture into a TDD Project

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, July 2019

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: 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

During the lifetime of complex, large-scale, long-term software systems requirements change. Systems need to adapt to these changes. The scope of their functionality typically changes as well and many design decisions taken in the early stages do not age very well. A growing system needs continuous refinement and an aging system needs maintenance. Growth typically causes a system's integrity to deteriorate. Prioritizing growth over quality introduces what is known as technical debt. Continuous attention to code quality is necessary to prevent deterioration.

This thesis examines the problems that can be caused by a decline in code quality and describes the implications for developers and users. It details and evaluates a dedicated effort to increase code quality and to counteract a decline in system integrity. Catroid, a complex, large-scale Android application, serves as a real-world example.

The implicit understanding of code quality is quantified by metrics that evaluate the complexity, size and design of the system. An analysis on the refactored classes and an overall assessment of code quality is presented. The thesis shows that refactoring can be beneficial for large-scale systems and provides an outlook on alternative approaches and follow-ups. It indicates that in order to sustainably deliver working software it is necessary to continuously invest in code quality.

Kurzfassung

Die Anforderungen an komplexe, groß angelegte Softwaresysteme ändern sich typischerweise im Lauf der Zeit. Systeme müssen sich and diese Änderungen anpassen. Üblicherweise verändert sich auch der Funktionalitätsumfang und viele Designentscheidungen, die in den frühen Entwicklungsphasen getroffen werden, altern schlecht. Wachsende Systeme brauchen kontinuierliche Verbesserungen und alternde Systeme Instandhaltung. Ein wachsendes System verliert oft an struktureller Integrität. Wenn Erweiterung wichtiger ist als Softwarequalität, entstehen technische Schulden. Um Qualitätsverfall zu verhindern, ist es notwendig, sich andauernd um die Softwarequalität zu kümmern.

Diese Arbeit untersucht die Probleme, die vom Qualitätsverfall verursacht werden und beschreibt deren Auswirkungen auf Entwickler und Benutzer. Die Arbeit analysiert und evaluiert die Implikationen eines dezidierten Prozesses zur Verbesserung der Systemqualität. Catroid, eine komplexe, groß angelegte Androidapplikation dient hierbei als reales Anwendungsbeispiel. Des Weiteren werden alternative Lösungsansätze und mögliche weiterführende Themen besprochen.

Das implizite Verständnis von Softwarequalität wird mit Metriken quantifiziert, die die Komplexität, Größe und das Design des Systems evaluieren. Eine Analyse der refaktorisierten Klassen und ein Gesamtanalyse der Qualität werden präsentiert. Die Ergebnisse implizieren, dass Refaktorisierungen positive Verbesserungen bringen und dass es notwendig ist andauernd in die Erhaltung der Softwarequalität zu investieren, um nachhaltige Entwicklung sicherzustellen.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Booch (1991) proposed that the complexity of large-scale software systems exceeds the human intellectual capacity. A plethora of software design principles, developed over the last few decades, has provided help in managing complexity. Large-scale software is often long lived, and during its lifetime requirements change. With the rise of agile development, *continuous integration (CI)* (Booch, 1991) has become common practice. Continuous delivery and refinement have found wide acceptance in the software development community. Automated testing and *test driven development (TDD)* (Beck, 2003) have helped to systematically assess software integrity and introduce incremental development approaches.

Booch (1991) has mentioned the terms software evolution and software preservation. He used them to describe the effort necessary to respond to changing requirements and the effort to keep software from decaying. Evolving software needs refinement. Agile development practices are designed to promote sustainable development and encourage continuous refinement (Google, 2018a). Spinellis (2006) has highlighted the importance of software quality and proposed that quality is a crucial, non-functional property of a system. While functional defects in a software impede user experience, non-functional errors can be catastrophic. It stands to reason that in the scope of long-term, large-scale software development maintaining code quality is essential.

The cost of changing software is said to increase exponentially over time because design typically deteriorates. Researchers and developers have tried to reduce the cost of change by refining development practices, environments and tools. Refactoring (Fowler, 2018) is a means to counteract quality deterioration and to improve the design of existing code. It has found widespread

acceptance in the software development community. Extreme Programming (XP) (Beck and Gamma, 2000) suggests that refactoring should be incorporated into the day-to-day routine of all developers. However, in this thesis we want to examine how a dedicated refactoring effort can help to improve code quality and how code quality influences developers and users. All considerations and actions described here are based on the experiences gathered during an improvement process applied to the real world FOSS project Catrobat[1].

1.1 About Catrobat and Catroid

Catrobat is a Free Open Source Software (FOSS) project that develops a visual programming language for teenagers. It allows them to design their own applications, games and animations. The project encourages teenagers to be creative and fosters computational thinking skills. The Catrobat project is organized according to the principles of the Extreme Programming (XP) development framework. Contributors are encouraged to apply agile development methods such as test driven development (TDD) and continuous integration (CI). A Jenkins CI server environment supports TDD and CI by automating builds, tests and deployment (Luhana, Schindler and Slany, 2018).

Catrobat has a large base of local and international contributors including developers, usability and user experience engineers, designers and translators. The project is organized into teams, each of which develops, adapts and maintains a distinct part of the catrobat system. The *Catroid*[2] team builds and maintains an interpreter and integrated development environment (IDE) for the Catrobat language on Android. Providing a mobile platform allows teenagers to develop programs using their smartphones instead of traditional personal computers (Harzl et al., 2013). Catroid is available freely as the app *Pocket Code*. Ever since its first beta release in 2013 Pocket Code

[1] https://www.catrobat.org/
[2] https://github.com/Catrobat/Catroid

was installed over 500,000 times[3] through Google Play.

Aside from many international contributions, Catroid is mainly developed by university students at Catrobat's headquarters in Graz, who are given the opportunity to voluntarily collaborate within the scope of their curricula. Their experience with software development and XP practices varies Müller, Schindler and Slany, 2019 significantly and their skill levels are just as diverse.

Working on complex large-scale systems can require extensive training. However, some contributors, especially students, do not spend a lot of time with the project. Thus, providing an efficient onboarding process is vital. Software quality plays a role in this process. A systematic analysis of entrance barriers to FOSS projects by Steinmacher et al. (2015) has found that architecture complexity for example constitutes a common obstacle for newcomers. Hence, it stands to reason that reducing complexity can effectively help to flatten the steep learning curve faced by newcomers.

[3]https://play.google.com/store/apps/details?id=org.catrobat.catroid

2 Related Work and Background

2.1 Extreme Programming (XP)

Extreme Programming (XP) is a lightweight, agile software development framework. Agile processes promote sustainable, continuous delivery of working software to users (Google, 2018a). XP encourages continuous integration (CI) of contributions, close collaboration of developers and automated testing (Beck and Gamma, 2000). XP is aimed at small to medium sized, self-organizing development teams.

In agile development there is continuous attention to design quality. Good design promotes agility. It allows developers to respond to changing requirements and to keep delivering valuable software. Especially in long-lived software projects such as Catrobat it is necessary to respond to changing requirements. As an Android application Catroid relies heavily on the Android framework. Because of Google's continuous development, frameworks and policies change regularly. To keep delivering, the Catrobat organization has to continuously respond to these changes and adapt the Catroid codebase accordingly.

2.2 Technical Debt

The term technical debt was first mentioned in the context of software development by Ward Cunningham. He says that code quality can be traded

off for a short term gain, only as long as it is mitigated as soon a possible by a rewrite (Cunningham, 1992). Cunningham compares this trade to going into financial debt. While a little debt can actually help, too much can have severe ramifications.

In Catroid it is very common that, in light of time constraints, code quality is neglected. However, it is very unlikely that developers actually mitigate quality deterioration by a rewrite, often because they have to meet some other time constraint. This has lead to a significant buildup of technical debt over the years and in many cases these short term solutions have grown into the foundation for a lot of other important functionality.

2.3 Refactoring

In the scope of large-scale, long-term systems, software development is an evolutionary process (Lehman, 1980). Requirements change during the lifetime of a system, often because of the mere existence of the system itself. Maintaining and adapting software is a central objective for any developing organization. Booch (1991) proposed that software maintenance only describes error correction and that maintenance alone is not sufficient to keep a system running. He claims that evolution and preservation are more suitable terms and concepts to consider when describing this process.

Preserving software can be tedious, especially when code quality deteriorates. Counteracting deterioration supports maintenance and keeps a software system adaptable. Agile software development embraces changing requirements. XP promotes flexible planning and feature development (Beck and Gamma, 2000). Agility and XP practices are built on promoting technical excellence and good design. In XP design has a central role and XP practices suggest integrating design concerns in the day-to-day business of every contributor in the form of refactoring.

Refactoring can be understood as a means of improving the internal structure of a system without changing its external behavior (Fowler, 2018). It

is tightly connected to the concept of technical debt. If done correctly, it reduces complexity and (re-)introduces structure into existing code. It is an effective measure to reduce technical debt and a necessary process in system maintenance and evolution. Refactoring helps create a more expressive architectural representation of the target domain. It aims at minimizing the chance to introduce bugs while counteracting a decline in system integrity. To do so it relies on solid automated tests that capture system behavior.

2.4 Automated Tests

In a software system testing can take place on three abstraction levels. i) Unit testing; ii.) Integration testing and iii) System testing.

2.4.1 Test Levels

Unit tests verify correctness of a system on small abstraction levels and pinpoints faults very precisely. Fowler (2012d) says that in an object oriented software system, units are typically classes, but that cohesive class clusters can be tested as units as well. Google's testing blog mentions some confusion between the exact distinction between unit and integration tests (Stewart, 2010). Unit tests are small and fast. They can be run frequently during the development process and verify each incremental change. Unit testing is only possible if the class hierarchy is independent enough to test classes in isolation.

Integration tests verifies more complex interactions in a system and assures that the independent components of a software interact correctly. Depending on their scope integration tests can more difficult to understand and maintain. Fowler (2012a) proposes that there are different notions of integration tests: narrow integration tests and broad integration tests. This has lead to some confusion in the software development community, which is why broad integration tests are often called system tests instead. Narrow

integration tests and unit tests are sometimes close in scope and typically use the same test framework, while broad integration tests need a special framework. Using narrow style integration test can significantly speed up test runs and improve resiliency (Fowler, 2012a)

System tests can be used to verify end-to-end user interactions. Tests on the user interface (UI) require a special setup and framework. In the case of Catroid a Jenkins CI server is used to run tests built on the Espresso[1] framework. End-to-end tests are brittle and prone to non-deterministic failures (Fowler, 2012b).

2.4.2 Mocks and Stubs

During unit testing it can happen that a class under test depends on some class method or function that should not be part of the test. *Mocking* and *Stubbing* are a means of replacing production code with some implementation that is defined within the test (Fowler, 2012c). There is a plethora of mocking frameworks for Java and Android. In Catroid Mockito[2] is used. Mocks and stubs can be particularly useful when they replace functionality outside the system such file operations or access to device sensors.

2.4.3 Testing on Android

Google splits testing into three categories(Google, 2018d): small, medium and large. Android applications depend on Java classes from the Android framework. This necessitates a classification into tests that can be run on a developer's local Java Virtual Machine (JVM) and tests that have to run on a real or virtual (emulated) Android device.

Local unit or integration tests on the JVM are considered small.

[1]https://developer.android.com/training/testing/espresso
[2]https://site.mockito.org/

Instrumented integration tests that rely on the Android framework but do not test the user interface, constitute medium tests.

Large tests validate end-to-end interactions on the application's user interface and run on a (virtual) Android device. Automatically testing an Android UI is inherently unreliable (Coppola, Morisio and Torchiano, 2018) and resource-intensive. These tests are significantly slower and more complex than all of the others. Android developer guidelines suggest using as few of these large UI tests as possible.

Software architecture influences how a system can be tested. If classes depend on each other it is difficult to test them in isolation. Sometimes classes provide some common functionality so that testing them together is reasonable, however, if classes that provide very different functionality depend on each other even narrow integration testing is difficult. In Catroid developers who have little experience in writing automated tests sometimes found it easier to write large end-to-end system tests because of Catroid's dependent architecture.

2.5 Object Oriented Design (OOD)

Today there exists extensive research and literature on *object oriented design (OOD)* methods. The most fundamental principle is abstracting the domain underlying a software into objects. The concept of objects is tightly connected to the concept of classes. Classes represent an abstraction of a cohesive part of the software and an object is an actual instance of such a class. An object thus has a concrete lifecycle and concrete characteristics. Booch (1991) proposes that classes are a necessary but insufficient vehicle for decomposition. In the scope of software design it is sometimes inadequate to describe the system in terms of single classes. Instead it is often convenient to consider groups of collaborating classes, which together provide some specific functionality. Booch calls these class clusters class categories. They consist of cohesive, tightly coupled classes that cannot be separated easily. If categories are reused, they have to be reused as a whole. Classes in categories

are highly dependent, so if any class within the category changes, the others have to change too. Proper class design requires developers to be aware of dependencies within these categories and between categories. In order to build adaptable, robust and modular systems dependency management is essential.

2.6 Dependencies

Martin (1994) highlighted three requirements for object oriented design. Systems are supposed to be robust, maintainable and reusable. He proposed that a system is rigid, fragile and hard to reuse if dependencies between subsystems are not properly managed.

- In an interdependent system changes to one module are propagated through to dependent subsystems. Developers cannot properly estimate the extent and impact of a change. When the cost of change is hard to estimate, project managers become reluctant to authorize them and the design becomes rigid.
- In a fragile system, changing one part of the system is likely to cause side effects in other parts. Fragile software is unpredictable and fixing errors often breaks other functionality. Problems can spiral out of control and maintenance becomes tedious. This problem is exacerbated when automated tests are unreliable. However, interdependent and fragile design is difficult to test properly; Hence, it is very likely that there are there is no solid set of tests for a fragile system.
- Dependent designs are hard to reuse. It is often disproportionately more difficult to sufficiently uncouple a reusable subsystem from the existing design than to simply implement it again.

Dependency management is crucial in software evolution and preservation. It is important to notice that not all dependencies are inherently bad. Collaborating class clusters or class categories are typically highly cohesive and tightly coupled. Besides, systems remain robust, adaptable and reusable if dependency targets are stable. Independent classes, which do not depend

on others, are less likely to change because there is no external reason for them to do so. Similarly, class categories with a large number of dependents have a good reason not to change, i.e., that changing them would require changing all dependents. Independent, responsible classes are stable and unlikely to change. Consequently, they are very unlikely to cause changes in their dependents. Thus, dependencies upon stable categories do not cause adverse effects on system stability. However, because stable categories discourage modification they limit adaptability and extensibility. Hence, to keep a system adaptable, stable class categories have to be abstract because only then it is possible to extend them without having to change them.

Martin (1994) introduced a set of metrics that can be used to assess dependency management. The metrics can be used analyze subsystems in terms of their stability, responsibility and independence. *Afferent couplings (CA)* are the connections between classes within a category to dependent classes outside the category. They highlight dependencies on a category. The number of CA to stable categories can be high, while it is necessary to reduce CA to unstable ones. Besides, completely abstract categories necessarily have a high number of CA because there have to be classes outside the category that implement them.

Martin defines the connections between classes outside a category to dependent classes within the category as *efferent couplings (CE)*. A high number of CE signals that classes within this category are highly dependent on classes in other categories. Hence, these classes are susceptible to change. The ratio of CE to the total number of couplings measures *Instability*. A value of *0* denotes a maximally stable and *1* a maximally instable category.

Stable categories cannot change easily. However, it is necessary to keep a system adaptable. It is important that stable categories can be extended without modification. Hence, stable categories have to be abstract. *Abstractness* is defined as the ratio of abstract classes to the total number of classes in a category. An abstractness of *1* denotes a completely abstract category and *0* a completely concrete one.

In theory two combinations of characteristics are desirable in categories: maximally stable and completely abstract or maximally unstable and con-

Figure 2.1: The main sequence between abstractness and instability

crete. However, in practice it is not always possible to design categories with only either combination. Martin introduces the concept of a main sequence (see the graph in Figure 2.1), which denotes all balanced combinations between instability and abstractness. All categories along or close to the main sequence are considered to have the proper proportions between afferent (CA) and efferent couplings (CE), abstractness and instability. The *distance from the main sequence (D)* is a metric that can be used to assess the design quality of categories in terms of the other metrics and a system as a whole. It is calculated as the perpendicular distance from the linear function representing the main sequence, normalized into an interval between *0* and *1*.

In this thesis dependency considerations are based on the propositions put forward by Martin (1994). However, often single classes are considered instead of categories, which means that single classes are treated as categories. This is due to the fact that in Catroid's architecture it is sometimes difficult to identify which classes collaborate on a functional level and which ones are simply coupled because of missing encapsulation. Besides, classes in Catroid often have many responsibilities and hence behave like categories anyway.

2.7 Cyclomatic Complexity (CC)

System complexity typically depends on the number of entities and their relationships. In a complex system there is a large number of relationships and dependencies. It is difficult to change or test a complex system because it is impossible to properly understand all its relationships and interactions.

There are various approaches to quantifying software complexity. The *cyclomatic complexity metric (CC)* was proposed by McCabe (1976). It counts linearly independent paths through the control flow graph of a program. High complexity implies that it is difficult to understand all possible ways in which the system can behave. A larger number of independent paths also means that more tests are required in order to ensure proper coverage. This metric can be calculated on the system as a whole or on any abstraction level, such as modules, packages, classes or methods.

Complexity is influenced by class design. When new features were added to Catroid, existing classes and methods were regularly extended to assume additional responsibilities. This means that class behavior is often primarily dictated by control structures. It becomes difficult to predict what happens during runtime.

In proper object oriented design classes should have one responsibility. Cyclomatic complexity typically increases when class design becomes complex and classes assume too many responsibilities.

2.8 Chidamber and Kemerer Metrics (CKM)

In the mid 90s Chidamber and Kemerer responded to the demand for metrics that can be used to assess software development processes. With growing popularity of object oriented development (OOD) approaches they devised a metrics suite specifically tailored to it (Chidamber and Kemerer, 1994). The metrics were developed upon the conceptualization of a software

as a relational system and are thus based on mathematical theory. They are designed to measure class design instead of implementation. Hence, the metrics are technology-independent.

The measures are a formalization of the intuitive understanding developers have about complexity. Thus they quantify implicit, empirical characteristics of a system into relations and values, such as class complexity. Chidamber and Kemerer propose that the metrics can be used for process improvement and benchmarking. Today, with the advent of XP and agile development methods the CKM find new applications, such as quantifying the implications of a refactoring process.

2.8.1 Weighted Methods Per Class (WMC)

Methods are considered properties of a class and properties define class complexity. The *weighted methods per class metric (WMC)* is the cumulative cyclomatic complexity of all methods in a class (Chidamber and Kemerer, 1994). Methods inherited from a superclass do not count towards it. Conversely, overridden methods are considered different implementations. Hence, they contribute to the metric. If all methods have unit complexity the WMC is simply the number of methods in a class.

WMC formalizes some intuitive notions that developers have: Classes with a large number of methods are more difficult to understand, maintain and test. The same is true for classes that have very complex methods. The more methods a class has the more methods child classes inherit. Hence, method complexity in a class does not only influence the class itself but also affects child classes.

Application-specific classes often have a large number of methods, which limits their reuse. Similarly, methods are often complex because they have too many responsibilities or because they have to behave differently depending on the context they were invoked in. In both cases the complexity is likely caused by interdependent design and interdependent design is

difficult to understand and test. Consequently it is typically difficult to reuse, extend or adapt classes with a high WMC.

2.8.2 Coupling Between Objects (CBO)

Dependencies or couplings between classes or class categories significantly influence design quality. Thus, while valuable, it is often insufficient to consider the complexity of a system only in terms of class or method complexity. *Coupling between objects (CBO)* can be used to evaluate class interaction design.

Classes A and B are considered coupled if an object of class A uses methods or instance variables of an object of class B. Chidamber and Kemerer (1994) define CBO as the number classes to which one class is coupled. Multiple couplings between the same two classes are counted only once. Access to static constants, type-definitions or constructors as well as couplings because of inheritance are not counted toward this metric.

While not all couplings necessarily affect code quality negatively, they limit reuse. Dependent classes cannot easily be reused without the ones they depend on. Coupled classes are sensitive to changes, i.e., changes in one class likely require changes in dependent classes. Hence, couplings increase maintenance effort.

Couplings influence testability because it is difficult to test coupled classes in isolation. In tests it is consequently necessary to either provide dummy implementations (stubs or mocks) for dependent classes or test the complete coupled class cluster in an integration test. Both solutions are often less than ideal. Implementing mocks, especially within the scope of a framework such as the Android operating system can be difficult or even impossible, depending on the characteristics of the class. Classes that depend on functionality provided by the system for example, such as sensor values, cannot be mocked easily. Testing complete clusters requires integration testing. Integration tests are inherently more complex, hence it becomes harder to

find the source of failures. Since maintaining complex tests is tedious, test quality often deteriorates over time.

2.8.3 Response for a Class (RFC)

The response set consists of all methods that can potentially be invoked when an object of a class receives a message. *Response for a Class (RFC)* is defined as the cardinality of the response set (Chidamber and Kemerer, 1994). The metric is an upper bound on possible paths through program flow in a class. A large response set contributes to class complexity.

Response set cardinality assesses class hierarchy design. RFC can be high either because classes contain a large number of methods or because they call a large number of methods. RFC can be used to increase test coverage because it helps identify classes that require additional testing effort.

2.8.4 Lack of cohesion in methods (LOCM)

Modularity is closely related to dependency management. Cohesiveness between the methods in a class is desirable because it promotes encapsulation (Chidamber and Kemerer, 1994) and indicates proper class design. However, cohesiveness is an abstract notion that developers may perceive subjectively. Chidamber and Kemerer introduced a relational metric to quantify cohesion by measuring the inverse. *Lack of cohesion in methods (LOCM)* counts the number of method pairs that operate on disjoint sets of instance variables. The metric helps to identify where class design can be refined. Diverging method clusters in classes can be an indicator that they should be split. Because of issues with the calculation of the measure, LOCM was subject to a modification by Hitz and Montazeri (Hitz and Montazeri, 1996) which is based on graph theory and finds method clusters, instead of pairs, that access disjoint sets.

2.9 AntiPatterns and Code Smells

A common approach for a developer is to try to apply *design patterns* wherever possible. Design patterns are formalized, abstract solutions to recurrent problems Gamma et al., 1995. However, a design pattern might be a suitable solution for one problem and a rather poor one in another depending on the context. It is important for developers to understand which problem a design pattern is supposed to solve, how the pattern works and where it is applicable.

A design pattern that is used in the wrong context or any general recurring pattern at all that causes negative consequences constitutes an *AntiPattern* (Brown et al., 1998). *Code smells* are a similar concept to AntiPatterns introduced by Fowler (2018). Code smells are well defined, recurring implementations or patterns that are considered poor design choices. Like Antipatterns they highlight structures in code that are known to negatively influence system integrity.

Studies propose that code smells within classes indicate that they are more likely to require changes (Khomh, Di Penta and Gueheneuc, 2009) and that code smells affect maintainability (Sjøberg et al., 2012), (Fowler, 2018).

2.10 Android User Interface Basics

Google (2018b) uses activities as a metaphor for windows in the Android user interface (UI). Activities control user interaction and navigation through the application. Activities can host UI submodules called fragments. Developers can use fragments to build flexible and adaptive user interfaces. Fragments can be added and removed dynamically and reused in multiple activities. Activities provide a stack where transactions between fragments can be stored. These fragment transactions can occur when users navigate through the activity. The stack can be used as a history, for example a transaction can be pushed to the stack when users navigate to a certain

screen within the activity and popped back when they press the back button. The Android system maintains activities in a similar manner. When a new activity starts, the currently running activity is paused and the new one is placed on top of the system's activity stack. When the user navigates back the activity on top is removed and the one below resumed. Activities and fragments have what Google (2018g) refers to as lifecycles. When users navigate through the app they transition through states in these lifecycles. The system provides callback methods for lifecycle events and Android Developer Guides strongly recommend properly handling the transitions in order to build stable and performant apps. It is important to understand that the lifecycles of the components in the current window do not necessarily coincide. In general, it is bad practice to retain object references to components that have a lifecycle, because they are modified directly by the operating system. For example, the system typically destroys activities on configuration changes, such as switching orientation, or when the user puts the application into the background. Dereferencing pointers to destroyed activities or fragments results in `NullPointerExceptions`, and these runtime exceptions cause the entire application to crash.

Views are atomic user interface components (Google, 2018h) in Android. Each window in the UI is built from a view tree. Views can be added and removed dynamically during runtime. A fragment, for example typically consists of a view group. When a fragment is attached to its host activity the fragment's view group is added to the activity's view tree. To obtain an instance of a view, it has to be *inflated* from a layout specification using a system service.

In Android datasets can be visualized as a collection of vertically ordered views. Usually, these views are put into a scrollable container. Typically the container knows about the content of the data and delegates data binding to an adapter. From April 2015 onward Google (n.d.) recommend using the `RecyclerView` model and its supplementary components to visualize datasets because of superior flexibility, modularity and backwardcompatibility over other, older frameworks such as `ListView` or `ListFragment` (Google, 2018e).

Google (2018f) proposes handling user interaction with lists through contextual actions. A contextual action mode displays a contextual action bar

and allows users to select and deselect items from the list. The contextual action bar contains actions users can perform on selected items. An action mode has a distinct lifecycle. Action mode behavior can be defined by implementing the `ActionMode.Callback` interface. The methods in the interface are handles to events in the action mode's lifecycle. The system invokes these methods automatically when the action mode is created or displayed, when an action item is clicked or when the mode is destroyed. An action mode with a specific callback can be started from any activity and in response to any user interaction. In Catroid for example, users can start the action modes from the options menu in most activities. The action mode is usually finished when the back button is pressed or when the user selects an option item from the contextual action bar.

3 Problem Statement

Code quality is important because it affects developers. Complex code constitutes an entrance barrier (Steinmacher et al., 2015) for newcomers and in the case of Catroid some of them have been unsuccessful in overcoming it. Dependent software architecture obfuscates the scope of tasks because changes to one component typically require dependent components to be changed as well. Seemingly trivial tasks can turn out to be disproportionately complicated. If it is difficult to estimate the impact and extent of a change developers can become reluctant to implement it. It can be difficult for developers, especially for newcomers, to find tasks appropriate to their skill level and experience. Being able to identify and understand the relevant parts of a complex, interdependent system such as Catroid confronts newcomers with a steep learning curve.

Dependent architecture cannot be changed easily. Extending a dependent system is tedious because it is difficult to reuse existing functionality. Extracting a certain functionality from a dependent system can require more effort than simply implementing it again.

Dependent design is not extensible because components know about each other. If developers want to add new components to a dependent system they either have to add the new dependency to all existing components or completely redesign the system. Adding to dependent design exacerbates the issue and increases technical debt. On the other hand, rewriting certain subsystems is not always possible. Especially for less experienced developers it can be difficult to completely redesign an entire subsystem to integrate a change.

In other cases rewrites are impossible because of time constraints. Catrobat

wants to promote continuous integration. However, every integration must be buildable and pass all automated tests. Hence, it is typically impossible to subdivide more extensive rewrites into multiple tasks and integrate each change separately. This causes a bit of a predicament because on the one hand integration should be fast and continuous but on the other hand Catroid's architecture often requires changes to be considerable.

Fixing errors in an interdependent system is difficult because changes to one part of the system can have side effects in other, seemingly unrelated parts. Hence, patches can break other important functionality and developers often have to rewrite parts of the code that should have nothing to do with the original fault. This slows down the development process and limits how fast the Catrobat organization can respond to critical errors.

Symptoms of poor code quality often appear together. For example, dependent design might be hard to understand. In some cases contributors have left the project because of issues caused by poor code quality after a short period of time. For those who stayed with the project beyond the onboarding phase problems with code quality sometimes started a vicious cycle. Frustration with code quality lead them to add more low quality code, which frustrated others who in turn wrote low quality code and so on.

Features and patches have to be integrated into the codebase. In the case of Catroid this is accomplished via pull requests (PRs). Before they are merged however, PRs undergo a review process, which is usually lead by senior developers and supported by automated tests. Code quality in the PR as well as the quality of the codebase influence this process. If a PR is long it can be difficult for reviewers to thoroughly verify all changes and the chance of missing some crucial issue is higher.

The need for extensive changes, however, is often due to dependent design in the codebase. Conversely, if the PR is short but adds to the problem of deteriorating code quality it might also have to do with the codebase's rigidness. In either case the reviewer would reject the PR request the author to change it. The author would then try to address the issues with the PR and request another review once the necessary changes are implemented. However, this is not always straightforward. Sometimes PRs undergo

multiple iterations of reviews and changes, which can be demotivating for reviewers and developers alike.

It is important to consider the role the quality of the existing codebase plays here. As a developing organization it is difficult to directly influence the code quality of a PR. However, if the design of the codebase is modular, PRs can be shorter and the review process more thorough and faster. It is less likely that there will be multiple review iterations because the reviewer's feedback can be more concise. If dependencies are properly managed automated test can provide more coverage and are more reliable.

Code quality is important because it affects users. How fast a developing organization can respond to changing requirements is limited by the adaptability of the system. An example: Catroid allows users to incorporate device sensor readings, GPS coordinates, built-in face detection and text-to-speech capability into their projects. Catroid can be used to control drones and robots via bluetooth, write and read near field communication (NFC) and supports ChromeCast features.

Many of these features require the use of restricted endpoints. Even read and write access to the *external file storage* (Google, 2018c), where Catroid's home directory was located, were restricted from 2018 onward. Being able to read and write files is a fundamental functionality for Catroid, because this is how a user's programs are saved and loaded.

In order for an application to use restricted endpoints, users must grant the application the necessary permissions. Before 2018 all of the permissions an application requires were listed during installation of the app and once a user accepted them they were granted. Developers had to consider them only as far as listing them in a file. However, in 2018, due to changes in Google's policies, permission handling became more complex. All permissions had to be integrated and checked during application runtime. Besides, this runtime permission handling allows users to revoke any permission at any time. From November 2018 onward Google prohibited releasing any applications to Google Play that do not conform to these requirements. This includes updates for apps that are already available on Google Play.

Because of Catroid's rigid architecture adapting to these new requirements was difficult. The process of accommodating the necessary changes effectively took more than four months. During the adaptation process it was impossible to release any of the new features or patches that had been developed during that time. This was particularly frustrating for developers and users alike.

3.1 Test Driven Development and Continuous Integration

In August 2015 automated testing had effectively become unavailable for two reasons: First, test results were inconsistent and second, full test executions, if they finished at all, took up to 16 hours. Test driven development (TDD) (Beck, 2003) relies on a solid set of automated tests that can be executed quickly after each incremental change. Waiting for 16 hours after each small change is clearly infeasible and discourages developers from using a proper TDD approach. Hence, verification was typically postponed until developers were ready to open a pull request (PR).

Although it was a reasonable approach in light of the situation, it meant that was more difficult to trace back development steps if tests failed. Another strategy for mitigating the runtime issue was to use only subsets of the test suite as immediate verification and have the whole suite run nightly. In any case, development was slowed down significantly and systematic verification of the development steps impossible.

Empirical studies (Luo et al., 2014) show the consequences of *flaky* (non-deterministic) test behavior. Inconsistent test results severely limit productivity because it is difficult to decide if a certain test fails because of problems with the system or because of issues with the test framework or the test itself. At first the test framework and testing environment were blamed for the stability issues. However, after switching to a newer, more stable framework and extensive updates and configuration optimizations to the

continuous integration (CI) server it became apparent that the problems were actually caused by fundamental quality issues with the existing tests and the production codebase (Luhana, Schindler and Slany, 2018).

4 Solution Approaches

Developers of all skill and experience levels are influenced by software quality. Contributing to a software project that has evolved over more than six years can be challenging, especially if code quality concerns have often been neglected. The inherent complexity of such a large-scale software project can overwhelm developers. Functional quality suffers and users are affected directly. Catroid has suffered from the consequences of technical debt regularly. Hence, counteracting a decline in code quality and system integrity was a central objective in the improvement process described here.

Extreme Programming (XP) practices suggest that refactoring should be integrated into the day-to-day routine of all developers (Beck and Gamma, 2000) in order to mitigate quality deterioration and the buildup of technical debt. However, if refactoring has merely had little priority in a software project, it can be challenging to (re)introduce it. A bundled effort to mitigate the most urgent design issues in Catroid was outlined via four main objectives: i.) Improve stability, maintainability, and adaptiveness of the system; ii.) Raise awareness about the necessity of refactoring and highlight its benefits to developers; iii.) Streamline the workflow by reducing overall project and PR size, and iv.) Support efficient and stable automated testing.

Refactoring was an integral step towards improvement. It constituted a necessary foundation for other measures such as improving the code review process, onboarding and test automation.

Modularized architecture simplifies integration and encourages developers to reuse code. Clean code can be understood by newcomers more easily. If the code is self-explaining they do not have to rely on seniors to guide

their onboarding process. Functional separation allows newcomers to start developing as soon as they are familiar with the subsystem they want to work on, instead of having to learn the details of all dependent parts of the code first. Especially for contributors not seeking a long term commitment this is essential.

Lower complexity in class and method design helps developers to understand program flow and allows them to design more reliable automated tests. Looser coupling between classes reduces the necessary testing effort. In general, fewer dependencies allow for proper integration and unit testing. Moving the focus from system tests to integration and unit tests increases performance and coverage. Besides, in simpler tests it is easier to pinpoint the reasons for failures.

It was clearly infeasible to refactor the whole project immediately. Hence, it was a key objective to divide the process and implement the most beneficial refactorings first. It stood to reason to prioritize improvements based on two criteria: i.) How likely are they to mitigate faulty behavior such as application not responding errors (ANRs) and failures that cause persistent damage, e.g., malformed project files or data loss; ii.) How much other, potentially new, functionality can benefit from them.

4.1 Catrobat Language Specification

With Catroid users can develop and run programs written in the Catrobat language. The programs can consist of multiple *scenes*. Scenes can be thought of as chapters in a story or levels in a game. Scenes contain actors or objects. These objects are called *sprites* in the Catroid codebase.

Users can define sprite behavior in event-based *scripts*. Sprites have visual representations, so-called *looks*. Besides, users can include *sounds* into their programs. These looks and sounds are a reference to files on the device's storage and represent these files in Catroid. Looks and sounds can be included into the behavior definition in such a way that sprites may change

their appearance or play sounds during runtime.

All projects are executed on the *stage*. The stage is a window in Catroid where Catrobat programs run. There users can see and hear the sprites act according to the behavior defined in their scripts.

4.2 Conceptual Catroid Architecture

On the most abstract level Catroid consists of two main components: An interpreter for the Catrobat language and an integrated development environment (IDE). The interpreter contains the entire Catroid language specification and executes programs on the stage. The IDE allows users to create and modify Catrobat projects through a graphical user interface (UI). There are supplementary components such as web interface where users can access some web services such as a sharing platform or a tutorials pages.

Catrobat projects are developed by users, hence they may assume faulty behavior during runtime is caused by errors in their programs. Thus, as long as these faults are not too severe or occur determinately, users might not think about them too much. In the case of faulty behavior in the IDE however, blaming Catroid is more obvious. In order to improve user experience prioritizing improvements to the IDE seemed preferable.

4.3 IDE windows for Projects, Scenes, Sprites, Looks and Sounds

Projects, scenes, sprites, looks and sounds are visualized in lists. Because the lists are self contained UI components they were implemented as fragments. These fragments, seen in Figures 4.1 through 4.5 are very similar in appearance and in terms of the functionality they offer. In all of these lists, except for the project list, users can rearrange items via drag and drop. In the scene

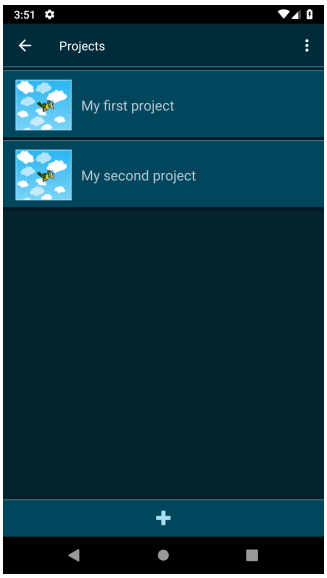

Figure 4.1: The project list in the IDE

list the order defines the sequence in project flow, a sprite's position within the sprite list determines its position on the stage's z axis, i.e., sprites on top will overlay the ones below them, if their x and y positions coincide, and the first look in the list is the one that represents the sprite on the stage.

Catroid provides a copy-paste functionality called backpack, where users can copy items into a persistent clipboard. The backpack can be used to copy scenes, sprites, looks, or sounds between projects or to copy sprites between scenes, or looks between sprites or any other combination. Users can also store items in the backpack and use them at a later time because the contents of the backpack are serialized into application storage. Scenes, looks and sounds hold references to physical files in storage. When these items are added to the backpack their files are copied into a backpack directory as well. Users can view and manage the backpack from special fragments, seen in Figures 4.6 and 4.7. These fragments are accessible from the IDE fragments through the options menu.

Backpacking, copying, deleting and renaming items is handled by action modes. Users can start the action modes from the respective entry in the

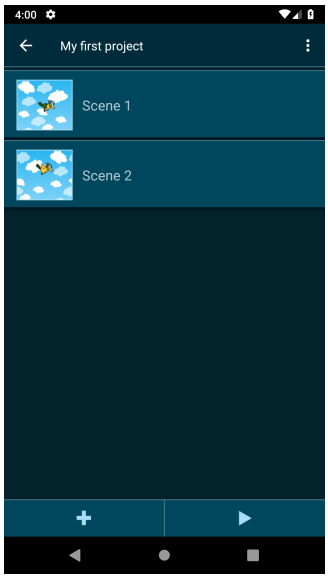

Figure 4.2: The scene list in the IDE

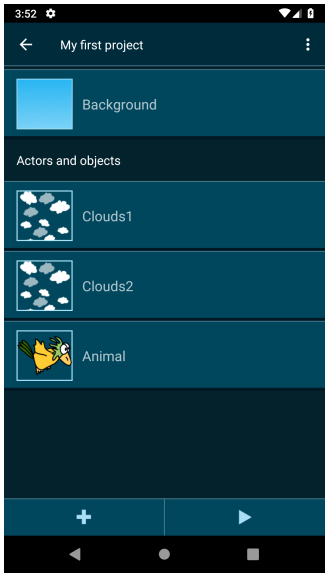

Figure 4.3: The sprite list in the IDE

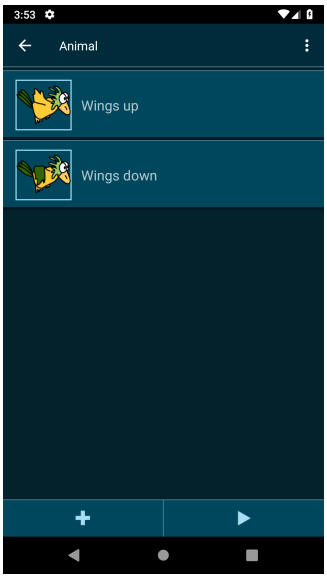

Figure 4.4: The look list in the IDE

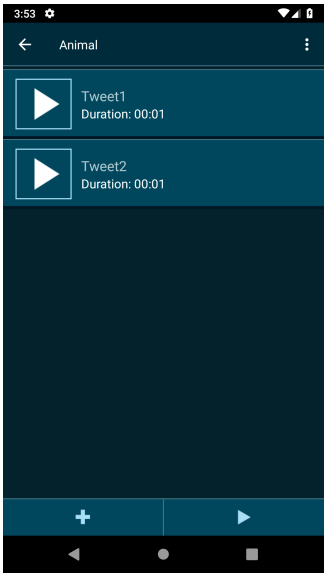

Figure 4.5: The sound list in the IDE

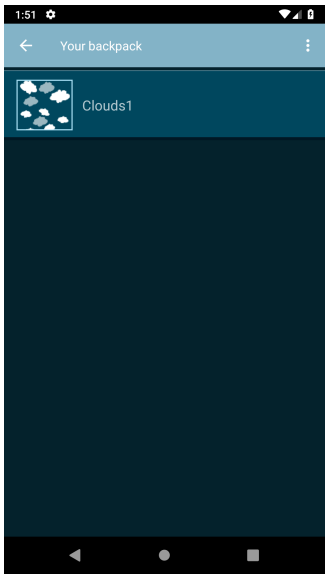

Figure 4.6: The sprite backpack in the IDE

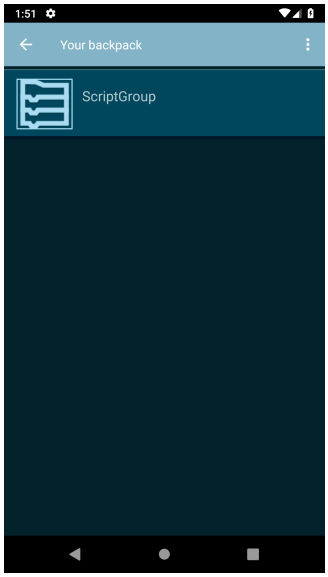

Figure 4.7: The script backpack in the IDE

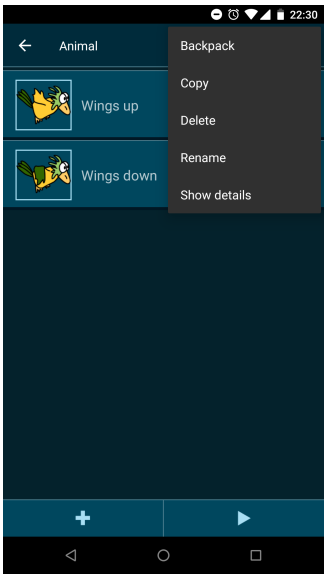

Figure 4.8: The options menu where users can start action modes to modify the look list.

options menu (see Figure 4.8). When the action mode is active, users can select items via checkboxes, as seen in Figure 4.9. The action mode is closed when a user selects the confirmatory button in the contextual action bar or the device's back button. Users can add items to the underlying dataset from the left button in the bottom bar. This button typically starts a workflow that involves selecting a source for the new item and naming it. The exact nature of the workflow depends on the item type, i.e., adding a new project is somewhat different from adding a new look. But in general the workflow is complex and involves starting other activities or applications, or performing storage operations.

4.3.1 Pre-refactoring Implementation and Issues

Each fragment was coupled to an adapter and a supplementary controller. The controller classes did not actually have a specific responsibility, but rather provided some utility methods to the fragments and adapters. It is reasonable to assume that developers put code into the controller classes

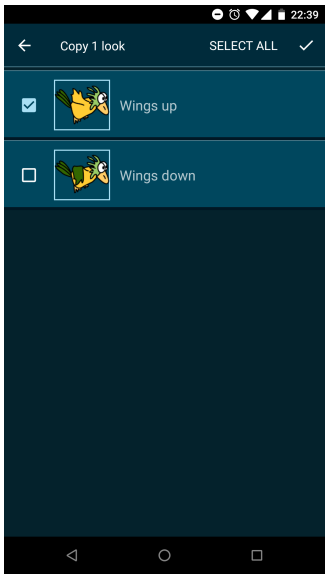

Figure 4.9: The checkboxes where users can select and deselect items they want to copy.

because they thought that the fragments were large enough already. This shows that developers have an intuition about appropriate class size and class design. However, it also shows that some developers apparently lack knowledge about object oriented design principles and that once design decisions are taken, class design is hardly ever re-evaluated.

Despite their similarity in appearance and functionality the implementations for the different fragments and adapters hardly shared any code. Improper dependency handling and insufficient modularization were clearly at fault. Providing view inflation and data binding, checkbox handling and selection management were not properly modularized and distributed between the fragments, the adapters and the controllers. User interaction was handled by event based message broadcasting. While this was a conceptually valid approach, the message receivers or listener classes often held object references to Android UI components. Because these components are handled by the Android system, object references regularly become invalid during app runtime, causing memory leaks. Receiver and listener implementations were not shared between fragments hence properly registering and unregistering listeners on Android lifecycle events had to be handled for each fragment

Table 4.1: Class Size Metrics Details

LOC	Non-comment lines of Java code
CA	Class size in terms of attributes, not counting inherited attributes or static fields
OA	Number of operations added, i.e., number of methods
Dpt	Number of dependent classes
Dcy	Number of classes that a class depends on

separately. Maintaining and adapting these broadcast receivers and listeners was error prone. This lead to frequent and inconsistent crashes. Mitigating the issues was challenging because the large number of methods in each class and their complex interactions. In addition the implementation was fragile, hence changing one method typically caused faults in other, seemingly unrelated parts of the code, which significantly limited maintainers in their efforts. Duplication caused inconsistencies, because behavior changes or patches had to be implemented for each fragment separately to ensure consistency. This process becomes increasingly difficult, the more components are involved and it becomes more likely that a developer misses some important detail. Especially in the scope of a long-lived system, such as Catroid, duplications constitute a recurring problem limiting maintenance and adaptation.

The statistics in tables 4.2 and 4.4 list size metrics for the classes that constitute the backpack. As suggested by Martin (2009) physical lines of code are often not sufficient to assess class size. Instead, it is preferable to describe size in terms of responsibilities. However, it is difficult to quantify the responsibilities of a class numerically. As an approximation a set of metrics was used to assess class design which includes the number of methods and the number of dependencies between the classes. These metrics are described in table 4.1. The numbers were gathered from release 0.9.27 (October 2016[1]).

Concerning the architecture there is very little complexity. There is virtually no inheritance or composition. There is no base class for the adapters,

[1]https://github.com/Catrobat/Catroid/tree/96a24298d557f6305c1158bbf4f8abc43e516008

thus there is no shared functionality between adapters holding different data types. However, the backpack, look, and sound adapters share a base class with their respective non-backpack adapters. The adapter classes are relatively small because they do not actually implement dataset visualization. Instead, view inflation and data binding was delegated to the relatively large fragment classes. For example, the look adapter and the sound adapter only provide one method. Conversely the sprite adapter has 14 methods. This obviously points to some irregularities in class design because all adapters actually provide the exact same functionality. In terms of physical class size in lines of code (LOC) it is evident that the fragment and controller classes are relatively large.

It stands to reason that class design can be improved by moving visualization to the adapters and by providing base classes for shared functionality. It can be seen that the fragment classes depend on a comparatively large number of other classes. Hence, because changes in any of these classes are likely going to require changes in the fragments as well, they are prone to change regularly. The fragment's efferent couplings (CE) or dependencies (Dcy) significantly outnumber their afferent couplings (CA) or dependent classes (Dpt), hence, as suggested by Martin (1994) they are highly unstable.

The Chidamber and Kemerer metrics in table 4.3 and 4.5 corroborate the interpretations proposed before. The fragment classes have a high method complexity (WMC) and response for class (RFC). Hence, besides their large physical size, the classes are complex in terms of method complexity and in terms of program flow. Because the large, complex classes are most likely to change maintenance is tedious and changes are likely to cause side effects or introduce faulty behavior. The coupling between objects (CBO) and the lack of cohesion between methods (LOCM) metrics hint that class design can be improved. Classes are tightly coupled to each other and partly incohesive. It is reasonable to assume that some of the larger classes, especially the controllers and fragments have too many responsibilities. Hence functionality should be split and properly encapsulated.

The IDE supports rearranging items via drag and drop. Because no built in solution existed within Android's `ListFragment` model Catroid used its own implementation that was extended from it. The `dynamiclistview` pack-

Table 4.2: Class Size Metrics on Pre-refactoring Backpack Classes

Class	LOC	CA	OA	Dpt	Dcy
BackPackListManager	318	4	64	35	18
.LoadBackpackAsynchronousTask	29	0	2	1	7
.SaveBackpackAsynchronousTask	7	0	1	1	2
BackPackSceneController	221	3	14	4	18
BackPackSpriteController	166	2	12	8	18
BackPackScriptController	186	2	11	5	32
LookController	565	13	35	19	24
SoundController	596	9	45	11	19

Table 4.3: CK Metrics on Pre-refactoring Backpack Classes

Class	CBO	LOCM	RFC	WMC
BackPackListManager	49	5	103	101
.LoadBackpackAsynchronousTask	7	1	13	8
.SaveBackpackAsynchronousTask	2	1	4	1
BackPackSceneController	21	2	76	43
.OnBackpackSceneCompleteListener			1	
BackPackSpriteController	25	3	64	39
.OnBackpackSpriteCompleteListener			1	
BackPackScriptController	36	3	89	45
LookController	41	4	174	107
.OnBackpackLookCompleteListener			1	
SoundController	29	3	188	122
.OnBackpackSoundCompleteListener			1	

Table 4.4: Class Size Metrics on Pre-refactoring Backpack UI Classes

Class	LOC	CA	OA	Dpt	Dcy
BackPackActivity	210	20	5	26	15
BackPackActivityFragment	20	2	3	11	0
BackPackSceneFragment	400	14	28	2	24
SceneAdapter	180	7	14	7	14
.ViewHolder	6	4	0	1	0
BackPackSpriteFragment	355	15	26	4	28
.SpriteDeletedReceiver	9	0	1	1	2
BackPackSpriteAdapter	185	7	14	2	13
.ViewHolder	13	11	0	1	0
BackPackScriptFragment	366	16	21	5	23
.ScriptGroupDeletedReceiver	9	0	1	1	2
BackPackGroupViewHolder	9	7	0	4	0
BackPackScriptAdapter	151	4	14	2	9
BackPackLookFragment	369	16	24	6	25
.LookDeletedReceiver	9	0	1	1	2
.LooksListInitReceiver	8	0	1	1	2
DeleteLookDialog	36	2	2	7	8
LookViewHolder	9	7	0	3	0
LookBaseAdapter	68	7	12	7	3
BackPackLookAdapter	29	8	1	2	8
BackPackSoundFragment	381	16	26	5	23
.SoundDeletedReceiver	9	0	1	1	2
DeleteSoundDialog	35	2	2	1	8
SoundViewHolder	10	8	0	3	0
SoundBaseAdapter	101	10	21	7	4
BackPackSoundAdapter	29	9	1	2	8
Average	115.62	7.38	8.42		
Total	3,006	192	219		

Table 4.5: CK Metrics on Pre-refactoring Backpack UI Classes

Class	CBO	LOCM	RFC	WMC
BackPackActivity	34	5	73	45
BackPackActivityFragment	11	9	10	3
BackPackSceneFragment	25	8	138	80
SceneAdapter	20	5	63	41
.OnSceneEditListener			2	
.ViewHolder	1	0	0	0
BackPackSpriteFragment	29	6	134	73
.SpriteDeletedReceiver	2	1	7	2
BackPackSpriteAdapter	14	1	59	34
.OnSpriteEditListener			2	
.ViewHolder	1	0	0	0
BackPackScriptFragment	24	5	127	68
.ScriptGroupDeletedReceiver	2	1	7	2
BackPackGroupViewHolder	4	0	0	0
BackPackScriptAdapter	10	1	51	33
BackPackLookFragment	26	6	128	72
.LookDeletedReceiver	2	1	7	2
.LooksListInitReceiver	2	1	4	2
DeleteLookDialog	15	1	26	5
LookViewHolder	3	0	0	0
LookBaseAdapter	10	5	21	15
.OnLookEditListener			2	
BackPackLookAdapter	9	1	15	7
BackPackSoundFragment	24	9	135	71
.SoundDeletedReceiver	2	1	7	2
DeleteSoundDialog	9	1	25	5
SoundViewHolder	3	0	0	0
SoundBaseAdapter	11	9	31	25
.OnSoundEditListener			3	
BackPackSoundAdapter	9	1	15	7
Average	11.62	3.00	36.40	22.85
Total				594

Table 4.6: Class Size Metrics on Pre-refactoring Drag And Drop Classes

Class	LOC	CA	OA	Dpt	Dcy
`DynamicExpandableListView`	55	1	9	3	3
`DynamicListView`	50	1	5	7	1
`UtilDynamicListView`	617	29	34	3	10
Average	240.67	10.34	16.00		
Total	722	31	48		

Table 4.7: CK Metrics on Pre-refactoring Drag And Drop Classes

Class	CBO	LOCM	RFC	WMC
`DynamicExpandableListView`	4	4	31	15
`DynamicListView`	7	5	27	13
`UtilDynamicListView`	10	1	115	127
Average	7.00	3.33	57.67	51.67
Total				155.0

age constituted the class category that provided the desired functionality. It consisted of two different derived list fragment, the `DynamicListView` and the `DynamicExpandableListView`, and a complementary utility class, the `UtilDynamicListView`. The two different fragment classes were necessary because of slightly different requirements in the sprite list and all other fragments. The underlying dataset operations were handled by the util class. While the implementation was relatively cohesive and encapsulated, it was large and complex. The numbers in table 4.6 show that the utility class has over 600 lines of code (LOC), 29 member fields and 34 methods. The size of the response set (RFC) and the weighted method complexity (WMC) with 115 and 127 are relatively high. Conversely lack of cohesion (LOCM) and couplings (CBO, Dpt and Dcy) are within the desired range (see table 4.7). The size and complexity of the util class limited maintenance and adaptability because it was very hard to understand for developers.

4.3.2 Refactoring Fragments Introduction

Fowler (2018) says solid automated tests are a prerequisite for refactoring. Tests ensure that the system behaves exactly the same before and after the refactoring process. However, because of dependent class design and insufficient separation between backend logic and UI reliable tests were unavailable. Dependencies impeded unit testing, thus any existing test were integration or UI tests. Because maintaining complex integration or UI tests is tedious, test quality and coverage was poor. Refactoring on the basis of flaky, incomplete tests was difficult and implementing new, more reliable test for the existing code was challenging. In any case the improvement process had to include building more testable implementations and ideally designing classes so that they can be either unit tested or at least tested without having to run a UI on the test system. Strictly speaking, replicating the exact same system behavior was not the objective anyway, because the current behavior was inconsistent and failure prone. Hence, the objective was to reimplement the fragments to meet the same requirements as the ones defined for the previous implementation while providing a more consistent and stable user experience.

The initial objective was to simply rewrite the fragments. However, because of the overwhelming complexity in the existing implementation the first refactoring did not replace all fragments and became a two stage process. The first part consisted of re-writing the fragments and adapters in a way that program flow becomes more apparent and to reduce unused or unnecessarily complicated code. Here the objective was to build upon existing technologies, such as the `ListFragment` model and to mitigate problems with the drag and drop implementation and checkbox handling. It was an attempt to generify the implementation and build a reusable solution. However, for reasons described in more detail in the subsequent sections, not all fragments were successfully reimplemented.

Because of changing requirements and lessons learned from the first refactoring iteration the second part was based on reimplementing the fragments and supplementary components, such as adapters and controllers on the basis of the `RecyclerView` model.

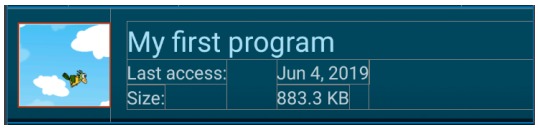

Figure 4.10: An inflated version of the viewholder for projects with layout bounds and subtexts.

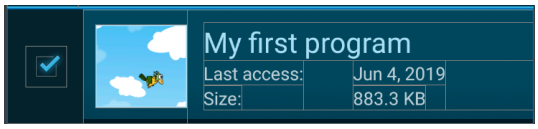

Figure 4.11: An inflated version of the viewholder for projects with layout bounds and subtexts and a visible checkbox.

4.3.3 First Iteration: Reimplementing as ListFragments

Because of recurring issues with the visualization and user interaction it seemed reasonable to redesign checkbox handling. In a first refactoring PR[2] an adapter was introduced that provides consistent view inflation and data binding. The adapter uses a special viewholder class that wraps the layout into a class. This class provides handles to a checkbox, an imageview where any bitmap can be loaded, a title and four subtitles or details textviews. An inflated version of the viewholder's layout from the `ProjectListFragment` can be seen in Figure 4.10 and a version with a visible checkbox in Figure 4.11. The figures show the view tree's layout bounds. The four subtitles or details views are wrapped into a container, so that they can be hidden as a group. The viewholder class was not a new idea, however, in the reimplementation the viewholders that existed for some of the data items as separate classes were replaced by one generic implementation.

To obtain visualizations for the data objects the Android system calls the adapter's `getView(final int position, View convertView, ViewGroup parent )` method. In this method it provides the adapter with the view object that exists at a specific position in the list. This view object is passed in through the `convertView` parameter. If there is no current view object, such as when

[2]https://github.com/Catrobat/Catroid/pull/1999

the list is initially built or the user scrolls to a part of the list for which no view objects exist, the adapter sets up a new one. To do so it creates a new viewholder object, inflates the layout and binds the data, i.e. the title, the image and any number out of the four details texts. Providing a viewholder class allowed to streamline list efficiency. In Android it is possible to reference objects in view tags. These tags are member variables of a view object and hence have the same lifecycle. A view object's lifecycle is managed by the Android system, thus it is more efficient to have the system handle the viewholder objects as well, instead of maintaining a list in the adapter. Whenever a data binding operation is invoked on the adapter and the `convertView` is created, the adapter puts the viewholder object into the view object's tag. This means that whenever, the adapter has to update the view, for example when the checkbox visibility changes, it can simply obtain the viewholder object from the tag and update the view state accordingly. Here the adapter also handles details visibility and onclick behavior.

The adapter was built as a generic abstract class so it can be extended and typed according to its intended usage. In the first iteration this abstract adapter was extended for the project and scene list as well as for all backpack fragments. While viewholder inflation and subview setup is handled in the base class, actual data binding is delegated to the child classes. Because no common base class for the items in the dataset existed, this seemed to be the most reasonable approach. The adapter provides view objects to the fragment and sets up listeners for user interactions. The adapter supports three listener types: a `ListItemClickHandler`, a `ListItemLongClickHandler` and a `ListItemCheckHandler`. These interfaces allow other classes to register themselves as listeners on the respective user interactions. The adapter invokes the appropriate callback method on each user action. The interfaces are generic as well, thus they can be typed according to the items in the dataset. The classes that implement the interface can thus be passed the object that was clicked as the correct data type. In the project list for example the overridden on click method is passed an object of type `ProjectData` (see Listing 4.1), while the base version of the adapter does not even have to know about the existence of this class.

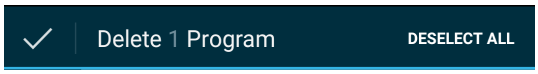

Figure 4.12: The contextual action bar with select/deselect all button and a title that changes with the number of currently selected items.

```
@Override
public void handleOnItemClick(int position, View view,
    ProjectData listItem) { ... }
```

Listing 4.1: The typed version of the onClick interface method.

Complementary to the generic adapter an abstract base class for the fragment was implemented. It integrates with the adapter and provides some functionality that is shared through all IDE fragments such as displaying a contextual action bar with an adaptive title depending on the active action mode. The class implements the adapter's `ListItemCheckHandler` interface and updates the title with the number of currently selected items. Besides it provides public methods through which activities that host the fragment can set details visibility on the dataset visualizations. This is used for example when users toggle details visibility from the activity's options menu. Additionally the fragment provides a select/deselect all button in the contextual action bar and handles communication with the adapter when it is clicked by the user. The contextual action bar and the button can be seen in Figure 4.12.

The abstract base class `CheckBoxListFragment` is extended by two other abstract classes: one for the fragments in the backpack and one that should have been the base for all non-backpack fragments. The `BackPackActivityFragment` provides two action modes that are shared among all fragments in the backpack: the unpack and the delete mode. The callbacks for the action modes implemented in this class handle action mode setup, which includes calling the adapter to display the dataset's checkboxes, setting up the contextual action bar title and clearing the selection on action mode destruction. The operations on the dataset that occur when the action mode finishes are delegated to the child classes via abstract methods. The `ListActivityFragment` was intended to be the base class for all non-backpack fragments. It provides

four action mode callback implementations for deleting, copying, renaming and backpacking items. The callback capabilities are very similar to the ones provided by the action mode callbacks in the `BackPackActivityFragment`. Actual dataset manipulation is also delegated to the child classes. Upon exiting the delete action mode users are asked if they really wish to proceed through a dialog. Both base classes provide a method that generates and displays this dialog. The `ListActivityFragment` however, was used as a base class for the project and scene list only because refactoring the classes for sprites, looks and sounds foundered on the complexity of these classes.

The drag and drop functionality was redesigned on the basis of the previous implementation. This means that many of the methods that handle the actual movement of the views on the screen and the responses to user gestures were reused in the `DragAndDropListView`. However, the dataset manipulation, i.e., reordering the items in the list was delegated to the adapter via the `DragAndDropAdapterInterface`. This is advantageous because it is extensible without modification, meaning that each adapter can override the `int swapItems(int position1, int position2)`, which is called by the drag and drop view whenever items are moved over each other. The adapter then can prevent dragging certain items over others for example, or update the dataset in some special way. In the old version when the data manipulation had to be different for any of the implementations, the utility class would have had to know about it and handle it via some control structure. Thus this improvement helps decouple the view and the dataset through inverting the low level (adapter implementation) dependency on the high level (the fragment) to a design where both the adapter and the fragment only depend on the abstract interface.

4.3.4 First Iteration: Results and Insights

In the first iteration of rewrites it was possible to reimplement the entire backpack UI. The tables 4.8 and 4.9 show the class sizes in terms of LOC, attributes, operations and dependencies and the Chidamber Kemerer metrics respectively. The tables contain both the cumulative and average metrics.

Because the reimplementation consists of fewer classes the total numbers give a better indication about how the two implementations compare to one another in terms of overall complexity. Conversely, the average numbers assess class design and describe the internal structure of the implementations. There are metrics however, where total or average numbers are not meaningful. For example lack of cohesion in methods (LOCM) is a metric that assesses class design. A value of one indicates a cohesive class and larger numbers suggest that design can be improved. Summing over these numbers for example has no significance whatsoever. Hence, the respective cells in the tables were left blank intentionally.

The overall LOC was reduced by 50% from 3,006 distributed within 26 classes to 1,450 in 15 classes, which decreased the average class size from approximately 116 to 97 lines of code. However, the improvement can be seen even more clearly in the number of operations, or methods (OA) in the classes. The total number of methods in all backpack UI classes was reduced to approximately one third, from 219 methods to 77 methods. This decreased the average number of non-inherited methods per class from approximately 8 to 5. Fewer methods can have either mean that complexity was removed from the implementation altogether or that many simple methods were replaced by fewer, more complex ones. Hence, to properly assess the changes it is necessary to evaluate method complexity. While the LOC give a rough upper bound on the size of the methods, cyclomatic complexity properly describes method complexity. The cumulative cyclomatic complexity (WMC) over all methods in the implementation was decreased by more than 50% from 594 to 273 and the average WMC was reduced from 22 to 18. This corroborates that class complexity was in fact reduced and not hidden within more complex methods.

It can be seen that common functionality has moved from the concrete implementations to the base classes because the number of methods (OA) in the child classes was reduced by approximately 50%. The OA in the adapter child classes were even reduced to zero. This means that the adapters themselves only provide functionality by implementing methods from the abstract base class. The adapter base class is stable because it mostly depends on the abstract fragment base class. These two, with the view holder base class, form a class category by providing one specific functionality. Hence

dependencies between those classes are desirable. The abstract base classes are very unlikely to change, thus having the child classes implement only methods inherited from their parents constitutes a robust, yet adaptable class hierarchy.

While the average number of attributes has increased, this is mainly due to the fact that some very simple classes, such as duplicate listeners have been removed. Looking at the cumulative number of attributes confirms this proposition. The total number of attributes was reduced from 192 to 173. More interestingly however is how these changes affect the quality of class design. Cohesive classes are desirable. The lack of cohesion in classes (LOCM) metric is based on the number of methods that operate on disjunct sets of instance variables or attributes. Here the tables show significant improvement in certain classes such as the fragment base class, the `BackPackActivityFragment`. Overall LOCM was reduced in the reimplementation.

The changes significantly improve readability and testability because program flow is less complex and class design is more expressive. Couplings on adapter and fragment classes have improved slightly, mainly because dependencies have moved to the base classes and because fragment and adapters now form a more cohesive class category. This can be seen in the Dpt, Dcy and CBO metrics.

The metrics on the drag and drop classes can be seen in tables 4.10 and 4.11. It can be seen that class size and complexity was significantly reduced while class design remained cohesive.

While the metrics show improvements there were some unresolved issues after the first iteration. It had been clear that the first step would leave out refactoring the script and brick IDE because of its complex requirements and dissimilarity to the other IDE fragments. However, during the improvement process it became apparent that it was not possible to refactor all of the other fragments within the first iteration. This corroborates the proposition that the issues described in sections 3 and 4.3.1 limit software evolution.

It was difficult to estimate the scope of the rewrite beforehand because of dependencies and code obfuscation. Missing separation between backend

Table 4.8: Class Size Metrics on Refactored Backpack UI Classes

Class	LOC	CA	OA	Dpt	Dcy
`BackPackActivity`	176	18	5	17	14
`BackPackActivityFragment`	125	9	6	6	10
`CheckBoxListFragment`	115	8	12	16	9
`CheckBoxListAdapter`	112	9	11	17	7
`.ListItemViewHolder`	11	9	0	7	0
`BackPackSceneListFragment`	176	15	10	2	19
`SceneListAdapter`	26	10	0	3	8
`BackPackSpriteListFragment`	179	15	9	2	23
`SpriteListAdapter`	30	9	0	2	7
`BackPackScriptListFragment`	136	15	7	3	17
`BackPackScriptListAdapter`	23	9	0	2	6
`BackPackLookListFragment`	142	14	8	3	17
`LookListAdapter`	24	9	0	2	6
`BackPackSoundListFragment`	140	15	8	3	16
`SoundListAdapter`	35	9	1	2	7
Average	96.67	11.53	5.13		
Total	1,450	173	77.0		

Table 4.9: CK Metrics on Refactored Backpack UI Classes

Class	CBO	LOCM	RFC	WMC
BackPackActivity	29	5	64	38
CheckBoxListFragment	25	3	51	23
CheckBoxListAdapter	24	2	33	22
.ListItemCheckHandler			1	
.ListItemClickHandler			1	
.ListItemLongClickHandler			1	
.ListItemViewHolder	7	0	0	0
BackPackActivityFragment	15	5	36	26
BackPackSceneListFragment	21	3	80	33
SceneListAdapter	11	1	15	4
BackPackSpriteListFragment	25	3	90	35
SpriteListAdapter	9	1	24	4
BackPackScriptListFragment	20	3	70	25
BackPackScriptListAdapter	8	1	19	3
BackPackLookListFragment	20	3	72	27
LookListAdapter	8	1	17	3
BackPackSoundListFragment	19	3	76	25
SoundListAdapter	9	1	22	5
Average	16.67	2.33	37.33	18.20
Total				273

Table 4.10: Class Size Metrics on Refactored Drag And Drop Classes

Class	LOC	CA	OA	Dpt	Dcy
DragAndDropListView	122	11	9	1	2
Average	122	11	9	1	2
Total	122	11	9	1	2

Table 4.11: CK Metrics on Refactored Drag And Drop Classes

Class	CBO	LOCM	RFC	WMC
`DragAndDropAdapterInterface`			1	
`DragAndDropListener`			4	
`DragAndDropListView`	3	1	47	25
Average	3.00	1.00	17.33	25.00
Total				25

logic and UI proved to be especially problematic. In the pre-refactoring version data manipulation was distributed between fragments, adapters and activities. Automated testing was difficult because it was impossible to test single classes. In the first improvement iteration data manipulation, including storage operations was moved to the fragment child classes. However, this still impeded automated testing because all tests for data manipulation required a UI and all UI tests had to be provided with a file system for the storage operations. It was clear that in the second iteration data manipulation should be removed from the fragment classes.

Before the first improvement iteration user interactions were modelled by using singleton patterns and static classes. Instead of proper object oriented design, certain functionality was implemented as a series of method calls to various static classes or singleton objects. Instead of passing arguments, parameters were often set at globally accessible locations, such as a public field in a singleton. Sometimes class design and user interaction between activities was handled via setting boolean flags in some global singleton. Data manipulation for example, especially storage operations were handled mainly through singletons. Throughout the system these singletons have taken over an increasing number of responsibilities over the years. Especially the `ProjectManager` and the `StorageHandler` classes have become what Fowler (2018) identifies as god objects. A god object constitutes a class level code smell. It describes a class that has too many responsibilities. Common symptoms are a large number of LOC, large numbers of methods and attributes. The numbers can be seen in table 4.12. Throughout Catroid there are 347 classes that depend on the `ProjectManager` and 93 depending on the `StorageHandler`. Both classes are extremely complex with around 50 methods

Table 4.12: Class Size Metrics on God Object Classes

Class	LOC	CA	OA	Dpt	Dcy
`StorageHandler`	953	11	49	93	188
`ProjectManager`	593	15	59	347	40

Table 4.13: CK Metrics on God Object Classes

Class	CBO	LOCM	RFC	WMC
`StorageHandler`	276	5	195	137
`ProjectManager`	376	8	173	144

(OA), a response set size (RFC, i.e., all methods that can be potentially invoked) of close to 200 and a weighted method complexity (WMC, i.e., cumulated cyclomatic complexity of all methods in a class) of approximately 150. Because both classes have many responsibilities they depend on a large number of classes. The `StorageHandler` for example knows about and depends upon all Catrobat language objects, including all script and brick classes. Consequently the coupling between objects CBO metrics are extremely high with 267 for the `StorageHandler` and 367 for the `ProjectManger`.

These god objects are difficult to test and maintain and create a tightly coupled architecture. Because a large number of classes depend on these objects it is hard to change them without causing side effects. Hence they promote rigidity by fragility. Consequently methods are hardly adapted but rather copy-pasted. Because the non-backpack activities and fragments relied heavily on these singletons it stood to reason that refactoring and eventually splitting the god objects had to become part of the improvement process.

Quality deterioration was more severe in the non-backpack fragments, especially in the look, sprite and sound fragments. Scenes had only been introduced into the Catrobat language shortly before the improvement process described here. Because the scene fragment was relatively new, it still had some structural integrity. Besides, user interactions on scenes and projects are not as complicated as for the other language elements. For example adding new projects or scenes is significantly less complex than

adding sprites or looks. Copying or deleting projects or scenes is simpler too, because it just involves storage operations on the respective directory in storage, while sprites, looks or sounds can have additional dependencies. Hence it was possible to rewrite the scene and project fragment within the first iteration.

Setting up tightly connected yet encapsulated components proved to be a feasible approach. Separating dataset manipulation, visualization and the fragment helped reintroduce modular class design and promote adaptability and reusability. However, because there are similar user interactions and lifecycle events for all fragments it should be possible to move more of the common functionality, such as starting action modes and displaying various dialogs to the base classes. Conversely, using composition instead of inheritance for the Catrobat language element specific functionality should help build more adaptable classes.

4.3.5 Second Iteration: Separating Frontend and Backend

One objective was to remove operations on data from the fragment classes, thus separating UI functionality and backend logic. This was realized through applying composition rather than inheritance and setting up the fragments to hold a controller object and delegating data operations to it. Controller classes wrap operations necessary for the copying, moving or deleting Catrobat language elements, such as storage operations. These controller then take over one distinct functionality from the fragments. The `LookController` for example calls the file operations necessary to copy, pack, unpack or delete looks.

Advantages are: i.) Controllers can be tested in isolation without having to run the UI on the test system; ii.) It is possible to test the fragments without having to call storage operations by mocking the controllers. Thus it is possible to test how the fragments react if storage operations fail, and iii.) Controllers can be adapted or exchanged if requirements change, and changes to the storage operations do not require changes in the fragments.

Table 4.14: Class Size Metrics on Classes Built from StorageHandler

Class	LOC	CA	OA	Dpt	Dcy
`BackpackSerializer`	57	3	3	2	6
`StorageOperations`	234	2	15	41	1
`XstreamSerializer`	434	5	12	58	191
Total	725	10	30		

Table 4.15: CK Metrics on Classes Built from StorageHandler

Class	CBO	LOCM	RFC	WMC
`BackpackSerializer`	8	2	25	6
`StorageOperations`	42	4	64	56
`XstreamSerializer`	249	2	92	45
Total				107

4.3.6 Second Iteration: Splitting God Objects

The `StorageHandler` class with approximately 1,000 lines of code (LOC), 93 dependent classes and dependencies on 188 classes was obviously one source of Catroid's tightly coupled design. It stood to reason that splitting the class distributes dependencies and improves class design. Clean code principles (Martin, 2009) suggest that classes should have only one responsibility. Hence, it seemed reasonable to identify all responsibilities the `StorageHandler` has, and construct new classes for each of these responsibilities.

The `StorageHandler` provided three main functionalities: i.) Serializing and deserializing Catrobat programs; ii.) Performing storage operations, such as copying or deleting files, and iii.) Serializing and deserializing the backpack. Consequently, the class was split into the `XstreamSerializer`, the `StorageOperations` and the `BackpackSerializer` classes. The process was relatively straightforward because there was no change in functionality or rewrites.

The metrics for the new classes are listed in tables 4.14 and 4.15. The total number of lines of code (LOC) in all three classes is smaller than the number

of LOC in the `StorageHandler` class by approximately 200 lines. Moreover, the dependency metrics have improved significantly. The backpack-serializing functionality for example, is only used by two other classes (Dpt value of two) and depends on six other classes (Dcy value of six). The storage operations class is used by 41 other classes (Dpt value of 41). However, it only depends on one other class (Dcy of one). The xstream serializer is used by 58 classes and depends on 191 classes. This is interesting because it shows how class design influences dependency management. The storage-operation functionality for example, is highly independent and reusable. It is very unlikely to change and consequently a stable target for other classes to depend on. Conversely, the xstream serializer part is highly dependent on the Catrobat language specification, i.e., on all classes that constitute language elements. This part was responsible for the `StorageHandler`'s large number of dependencies. Consequently, it was not possible to reduce the number of dependencies here. However, the backpack serializer has significantly fewer dependencies than the xstream because the backpack only contains language elements above brick level, i.e., scenes, sprites, scripts, looks and sounds. In any case, splitting the class decoupled the three distinct functionalities in the `StorageHandler`. This implies that any classes that use either of the functionalities, no longer simultaneously depend on the other two.

The Chidamber and Kemerer metrics in table 4.15 show that splitting the `StorageHandler` class has helped reduce complexity. The potential number of method invocations (RFC) was decreased significantly, from 195 in the `StorageHandler` to 25, 64 and 92 for the backpack serializer, the storage operations and the xstream serializer respectively. A lower number of potential method invocations improves testability because there are fewer scenarios to consider. Furthermore, lower RFC implies that is easier for developers to understand the control flow through the class. The same change can be seen in the weighted method complexity (WMC). The new classes have significantly lower WMC per class, and even the cumulative method complexity of all three classes is lower by 30 paths through control flow (reduced from 137 to 107). The decision to split the `StorageHandler` is corroborated by reduced lack of cohesion in methods (LOCM) measures. However, the values of two, four and two for the backpack serializer, the storage operations and the x stream serializer respectively, imply that the still have multiple responsibilities and that they should be split into even more, smaller classes.

The results imply that it is beneficial to split god objects even without rewriting the methods in the class. Consequently, it is reasonable to assume the splitting the `ProjectManager` can help to improve class design just as significantly.

4.3.7 Second Iteration: Reimplementing as RecyclerView

Based on the developer guidelines provided by Google (2018e) it seemed reasonable to reimplement the fragments based on the `RecyclerView` model. Conceptually component design in the `RecyclerView` model is similar to the architecture set up in the first refactoring iteration. The `RecyclerView` is used as a container for data that is visualized in viewholders. Data binding is handled by an adapter. The `RecyclerView` model is flexible because it is a view instead of a fragment. Hence, it can be used within any view tree and offers more customization options. `RecyclerView` handles view binding and reuses viewholders to optimize app performance and reduce memory requirements. It provides a consistent look and feel on devices running different Android versions, because of its superior backward compatibility and supports drag and drop out of the box.

Based on the experiences from the first iteration it stood to reason that the class hierarchy for the `RecyclerView` model should be similar to the one used for the `ListFragment` implementation. The rewrite introduced a generic adapter class that is responsible for data visualization. It manages the checkboxes that are used for item selection during the action modes and keeps track of the items a user selects. In the first iteration selected items were tracked in the adapter class directly. This functionality was moved to a separate selection manager class in this iteration. This way multi-selection can be tested in isolation, without the adapter. Conversely, the adapter does not depend on the specifics of the multi-selection. It simply depends on the public methods to set, remove and toggle item selection and a method that provides the currently selected positions. This design is more adaptable because if selection has to work differently for one child class of the adapter the multi-selection manager can simply be overridden there and the base

class does not have to change.

The adapter uses a viewholders to visualize the dataset. The `ViewHolder` class is extended from the `RecyclerView` model's viewholder class. It provides a checkbox, an image view, a title and four details views. The adapter binds the respective values from the dataset to the view elements. Changes to the checkbox state are detected by the adapter. It uses the multi-selection manager to keep track of the checked items in the dataset and allows other classes to register themselves as listeners, so that they are notified when the selection changes. Additionally, the adapter listens for click and long click events on the items in the dataset. Other classes can register as listeners for click events. The listener interfaces can be seen in Listing 4.2.

Two abstract base classes for the UI fragments were set up. These fragments provide functionality to handle user interactions in the backpack and non-backpack fragments. Both base classes support action modes. In the backpack there are two action modes, one for deleting and one for unpacking items. In the non-backpack fragments there are four action modes that pack, copy, delete or rename items. The base classes start an action mode when users click on the respective item in the options menu. They notify their adapter to display the checkboxes and handle finishing the action modes. Actual dataset manipulation, such as deleting or unpacking items is different for each data type, meaning that deleting a look is different from deleting a sprite. Hence, this functionality is delegated to the child classes through abstract methods. This way the fragments are extensible. The base class does not have to know about all different data types and depends only on the abstract method and not on implementation details. In order to pass the correct data type for manipulation, the base class is generic and can be typed through a parameter. The class headers and the interfaces the base fragments implement can be seen in Listings 4.3 and 4.4.

The fragments respond to Android lifecycle events. For example, they finish any active action mode when they are put into the background. Dataset setup depends on the data item type. For example, obtaining all sprites from a scene works differently than obtaining scenes from a project. Hence, initializing the adapter with the dataset is delegated to the child classes. This keeps the UI independent from the specifics of the Catrobat language

elements.

When the adapter initialization is finished the fragments register as listeners for selection changes and click events on the dataset. How the fragments respond to these events can be defined in the child classes by overriding the respective methods. Both base classes handle long click events. The backpack fragment displays a contextual action dialog where users can unpack or delete items from the dataset (see Figure 4.13) and the non-backpack fragment initiates drag and drop.

```
public interface SelectionListener {

  void onSelectionChanged(int selectedItemCnt);
}

public interface OnItemClickListener<T> {

  void onItemClick(T item);

  void onItemLongClick(T item, ViewHolder holder);
}
```

Listing 4.2: Recyclerview listener interfaces for user interactions with the dataset.

```
public abstract class RecyclerViewFragment<T> extends
   Fragment implements
    RecyclerViewAdapter.SelectionListener,
    RecyclerViewAdapter.OnItemClickListener<T>,
    NewItemInterface<T>,
    RenameItemDialog.RenameItemInterface { ... }
```

Listing 4.3: The generic recyclerview class and the interfaces it implements.

```
public abstract class BackpackRecyclerViewFragment<T> extends
    Fragment implements
    ActionMode.Callback,
    RVAdapter.SelectionListener,
    RVAdapter.OnItemClickListener<T> { ... }
```

Listing 4.4: The generic recyclerview class for backpack fragments and the interfaces it implements.

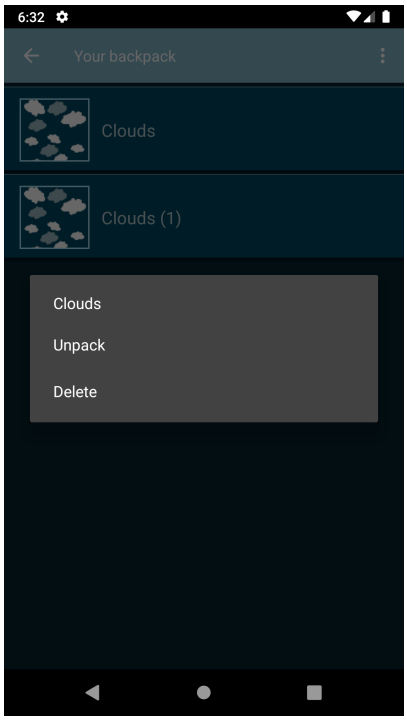

Figure 4.13: The contextual action dialog initiated by a long click on a data item.

4.3.8 Second Iteration: Results and Insights

In the second iteration of the IDE refactoring it was possible to reduce the size of the codebase significantly. The pull request (PR) that introduced the `RecyclerView` implementation[3] removed 44,352 lines of code and replaced them with 9,539. This is a reduction of 34,813 lines of code (including comments and non-production code) or 78.5%. The scope of the PR with changes to 573 files was far beyond what is usually acceptable in the project. The review process took 7 days which clearly did not adhere to the principles of continuous integration (CI) at all.

However, because of the tightly coupled architecture of the previous implementation it was difficult to split the changes and integrate them through

[3]https://github.com/Catrobat/Catroid/pull/2682

Table 4.16: Class Size Metrics Before and After Reimplemenation

	Before RV	After RV
LOC (total)	89,132	77,773
LOC (average)	83.90	76.84
CA (total)	8,398	8,016
CA (average)	8.98	9.06
OA (total)	5,814	5,148
OA (average)	6.22	5.82
CBO (average)	17.18	16.57
LOCM (average)	1.92	1.85
RFC (average)	24.37	22.61
WMC (average)	16.48	14.91

multiple PRs. The fact that even after an effort to reduce the changes to the absolute minimum the PR still changed 573 files highlights the impact of dependent design on the development process.

Considering the scope of the changes it is clearly unreasonable to evaluate the metrics on class level. Instead, the metrics in table 4.16 show the numbers calculated on the entire project. The columns compare the metrics calculated directly before and immediately after the PR that introduces the `RecyclerView` model. The size metrics are the cumulative number of lines of code, attributes and methods over all classes and the averages per class. For the Chidamber and Kemerer metrics (CKM), which are calculated per class, the table shows the averages over all classes.

The numbers show a significant reduction in the total number of lines of production code from 89,132 to 77,773. The average class size was decreased by around 10% from 84 to 77 lines of code. The class size in terms of attributes and methods remained approximately the same. Couplings (CBO), response set size (RFC) and method complexity (WMC) was reduced and cohesion between methods (LOCM) improved.

The implications of the rewrite are not confined to changes in code metrics. It influenced the development process and how developers collaborate

remotely. In 2019 remote contributors implemented a new contextual action for the one of the non-backpack fragments: A *merge* functionality for projects, where users can combine two projects into one. This comprised two extensions to existing functionality: a new action mode definition with a corresponding callback and a new selection mode that allows users to select exactly two items in a list. The developers managed to implement the changes to the user interaction handling in a PR[4] with less than 300 lines of code (including comments, copyright headers, UI message texts and automated tests). It is worth noting that these remote contributors neither had any part in the refactorings described here nor were they briefed on the details of the implementation. Hence, it is reasonable to assume that clean code is in fact reused and that the fragment and adapter implementations are sufficiently adaptable and independent to be changed easily.

Despite the improvements it is worth discussing some considerations and improvement opportunities regarding the reimplementations:

- Class have too many responsibilities. While it was possible to separate functionalities and add some modularization, many classes still do too much. Especially the IDE fragments handle a majority of the user interactions. They provide a very specific set of functionalities and hence it might be difficult to reuse them in other contexts.
- Backpacking is still a mess. The dependencies between Catrobat language elements makes it difficult to properly duplicate items and correctly set all references. The controller classes are highly dependent on the implementation details of the language specification. Here it could be beneficial to encapsulate certain functionality into the language elements themselves.
- Class design for Catrobat language elements could be improved. It is an objective that Catrobat programs are backward compatible, meaning that programs created with an old language version have to be interpretable by newer versions of the interpreter and the IDE. Because of this requirement changes to the language element implementation meant that xstream serialization and deserialization had to be adapted as well.

[4]https://github.com/Catrobat/Catroid/pull/3173/files

- Class design for the UI relies on inheritance. However, in certain cases using composition would have been preferable. Building UI classes from composite elements could be used to remove more backend logic from the UI components. Which implies that more of the functionality would become testable in unit and medium-sized integration tests that do not have to be run on the UI. This could help reduce test flakiness and build more reliable tests. Additionally, composition would help build more reusable UI components. Action modes and callbacks, for example could be components, instead of having the fragments implement action mode behavior. With inheritance any new fragments that want to reuse action mode implementations always inherit all action modes form the base class, regardless if they use all of them or not. Conversely, adding a new action mode for only some of the child fragment classes is difficult. If the action mode is added to the base class, all other children inherit it as well, even if they do not need it and if it is not added to a common base class sharing it between to children becomes complicated, unless another common base class for the fragments sharing this action mode is added. However, this complicates the class hierarchy and promotes rigid design.

4.4 Bricks and Scripts

Bricks are atomic elements within the Catrobat language. Currently Catrobat supports more than 120 different types of bricks. Each brick represents a statement and each statement corresponds to an *action*. Actions are atomic operations that can be performed during program runtime. They are the implementation of Catrobat statements based on the game-engine framework *libgdx*[5]. The statements in a Catrobat program are converted into actions before it is started on the stage. The Catrobat language is event-based and each statement is part of a thread. These threads are called scripts.

In Catroid's integrated development environment (IDE) there is a fragment

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

that displays all scripts and bricks that define the behavior of a sprite. Users can add, copy, rearrange, delete and deactivate scripts and bricks. Scripts have a header, subsequently called ScriptBrick, that visually marks the start of a script. While rearranging bricks within a script plays a central role in defining sprite behavior, rearranging whole scripts is purely cosmetic and should support users in maintaining structure within their code. By deactivating single bricks or whole scripts users can mark parts of their code as non-executable, similar to commenting out lines of code in a text-based programming language.

4.4.1 Brick Categories

Bricks are organized into color-coded categories. Besides bricks that can be used to define the motion, sound and appearance of a sprite, there are event, control and data bricks. Additionally there is the pen category that allows users to draw paths onto the stage. The pen supports customizations such as setting stroke width and color and the paths follow a sprites motion.

4.4.2 Event Bricks

The event category contains script bricks that act as script headers. While all scripts invocations depend on events, some scripts allow users to specify additional conditions. For example, there is a script type that allows users to react to broadcast messages. Here users can specify to which message the script should react. The event category also contains the bricks that send these broadcast messages. Other scripts types are executed on program start, when the user touches the screen, when some sensor value assumes some value or when the sprite collides with some other sprite on the stage.

4.4.3 Control Bricks

Users can control the program flow through the event threads with if-then-constructs, if-then-else-constructs, conditional and unconditional loops. Hence, the control flow graph of a script can contain branches. A simple example can be seen in Figure 4.14. It shows the program flow of a thread that is invoked when a scene starts.

In the IDE bricks are arranged below each other, similar to the statements in a text-based programming language. To visualize the grouping of statements in control structures there are else-, end-if- and loop-end-bricks. These bricks serve a similar function to braces around statement blocks in languages such as Java or C. The representation of an if-else construct can be seen in Figure 4.15.

In 2018 Catrobat started developing a 2D visualization and manipulation method for Catrobat programs based on Google's Blockly language specification[6]. Development decisions in Catroid influenced the development process for the Blockly extension. The Blockly-based extension interprets Catrobat programs built with Catroid and Catroid's modelling of the Catrobat language had some issues that will be discussed in the subsequent sections.

4.4.4 Script Interpretation

On the stage, the actions in a thread are executed sequentially. Branches in program flow are modelled as conditional actions. A conditional action, such as an if-then-else action, consists of a condition and two subsequences, an if-sequence and an else-sequence. To execute a conditional action, the condition is evaluated, and depending on the result, program execution continues on either the if- or else-sequence. The subsequences are lists of actions and that contain additional conditional actions. Repetitive actions or loops are

[6]https://developers.google.com/blockly/guides/overview

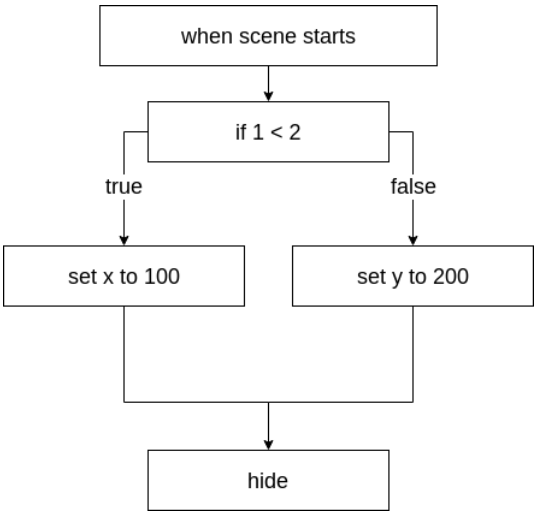

Figure 4.14: Conceptual representation of a if-then-else sequence

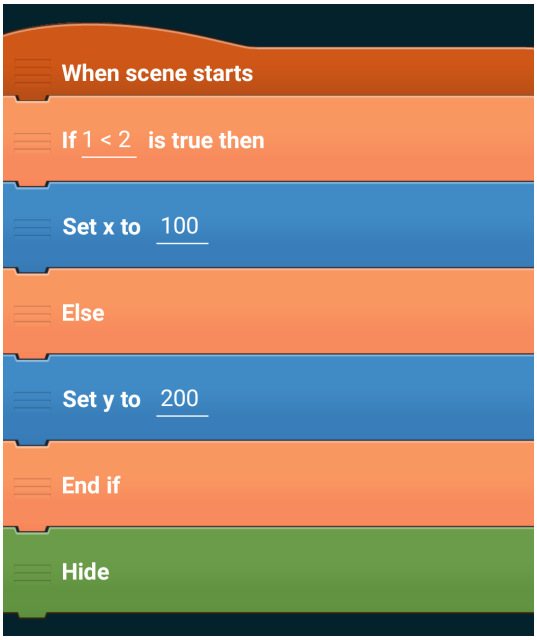

Figure 4.15: IDE representation of a if-then-else sequence

modelled similarly. They consist of a condition and a single subsequence of actions that are executed as long as the condition is fulfilled.

4.4.5 Pre-refactoring Data Modelling and Issues

The Catrobat language implementation in Catroid modelled the program flow of a script as a flat collection of bricks. When the statements in the thread were interpreted the else-, end-if- and loop-end-bricks were used to identify which bricks belong to which branch of the control flow and to construct the corresponding actions for the stage. Hence the scripts had to be parsed. The bricks in the collection had to be treated as tokens and the sublists for conditional or repetitive actions were built from subsections of this collection before the stage started.

For a script to be interpretable, there were requirements. The number and sequence of syntactic bricks had to be correct. However, because the syntactic bricks were part of the script's brick list as separate elements it was actually possible to create a sequence of bricks that is invalid. A missing syntactic brick for example, causes Catrobat programs to behave incorrectly. The same is true for the order of syntactic bricks: an if-end-brick before the else-brick result in an invalid program.

It is common that programming languages require the users to ensure that their programs to be correct in order to be executable. However, in Catroid the IDE was supposed to prevent users from modifying control structures in such a way that they would become invalid. If, for example a user selected an else-brick for modification the entire if-then-else construct was selected. This ensured that the user was only able to operate on these constructs as a whole. Figure 4.16 shows that selecting an if-then-else structure disables all checkboxes within the structure, so that parts of it cannot be modified separately.

The IDE, however did not parse the script in order to know the scope of a control structure. Instead all parts of a control structure, i.e., the control

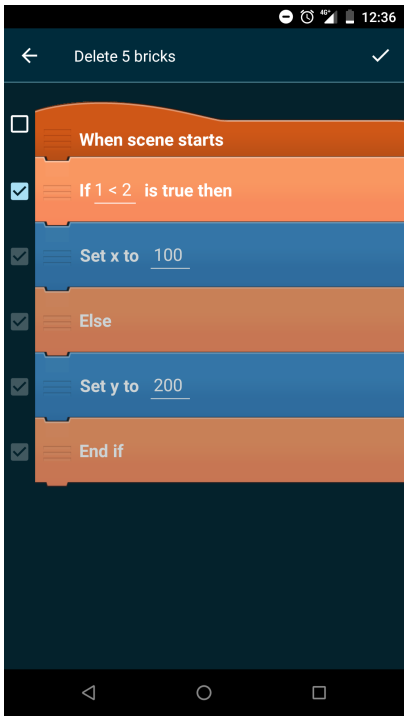

Figure 4.16: The IDE only allows control structures to be modified as a whole in order to prevent invalid program structures.

structure's header and its syntactic bricks were linked via object references.

It was a reasonable requirement for the IDE to ensure that users would only be allowed to build valid programs. However, there were some design decisions that had considerable drawbacks:

- Links between the bricks in the control structure were not part of the serialized version of the program. Hence the object references were not persistent. Every time a program was read from its file, each script in each sprite in each scene had to be parsed and the references between the bricks of a control structure set.
- Copying a selection of bricks within a script that contained control structures was complicated and did not work properly most of the

time. The main issue was correctly setting the references between the control structure headers and the corresponding syntactical bricks. Because control structures can be nested into each other and because there are many different possibilities for the sequence of bricks in a script, there were cases where references were set up incorrectly. This often resulted in `null` references, references to copied bricks that were not added to the script's brick list or references between copies and the original bricks. These problems caused Catroid to crash. There were patches for this behavior, however none of them actually fixed the issue for all use cases. This was mainly due to the fact, that in order to correctly address the problems large parts of the IDE would have had to be reimplemented.

- Because the syntactic bricks were treated as separate bricks, it was possible to serialize programs that contained incomplete or incorrectly ordered control structures. If for example during the copy process for a if-then-else structure the copied else- or if-end brick were not correctly added to the script it was still possible to serialize this invalid script because there was no other verification that ensured only correct programs were serialized. This had some serious ramifications. It was practically impossible for users to recover broken programs because the IDE did not allow them to modify incomplete control structures. Thus it was particularly frustrating for users if the programs were corrupted because of errors in the IDE.

Most of these issues were difficult to mitigate via patches because script and brick modification through the IDE because of the complex implementation. All classes concerned with this part of the system were highly interdependent and responsibilities were distributed among them. Hence, it was difficult to identify all classes that had to change in order to correct the behavior and to ensure that all other functionality remained unaffected.

Errors and crashes caused by problems with control structures in scripts were critical because they often resulted in persistent damage to a user's programs. Besides, it was difficult to convert the flat implementation to the Blockly language specification. Hence, it seemed reasonable to remodel script structure to provide a more resilient implementation.

4.4.6 Changing Data Modelling

It stood to reason to change the implementation in such a way that the data modelling does not allow invalid scripts. This means that the IDE does not have validate script correctness, which is preferable because the IDE as the user interface (UI) should not have to know about the specifics of the Catrobat language.

In other visual programming languages such as Blockly or MIT's Sratch[7] event-based threads are modelled as trees. Control structure statements have dedicated sublists with all statements that constitute their subtree. In such a hierarchical implementation, syntactic bricks do not exist as separate elements in the script's token list. Hence, the correctness of the script does not depend on the correct number and sequence of syntactic bricks.

Replicating this hierarchical data modelling for the bricks in Catroid seemed reasonable. Remodelling the data structure accordingly was relatively straightforward. However, the rewrite provided the opportunity to change class and dependency design.

The brick package has a large number of afferent couplings (CA). This implies that there are many classes outside the package that invoke public methods or directly operate on member variables of classes in the package. The rewrite provided the opportunity to change this. For the sake of robustness and to prevent changes to one of the 120+ concrete brick implementations from propagating to dependent classes all methods publicly available on the brick classes were added to the `Brick` interface. So that all classes outside the brick package depend on the interface rather than on the concrete implementations. The `Brick` interface is extremely stable (instability measure is 0.07), thus it is very unlikely to change and consequently it is highly unlikely that any classes that depend on it have to change because of it. Because the brick interface is abstract, it is possible to extend it without modification. Thus, classes and dependencies are designed according to the open-closed principle (Martin, 1996) which encourages extensibility (open

[7]https://scratch.mit.edu/

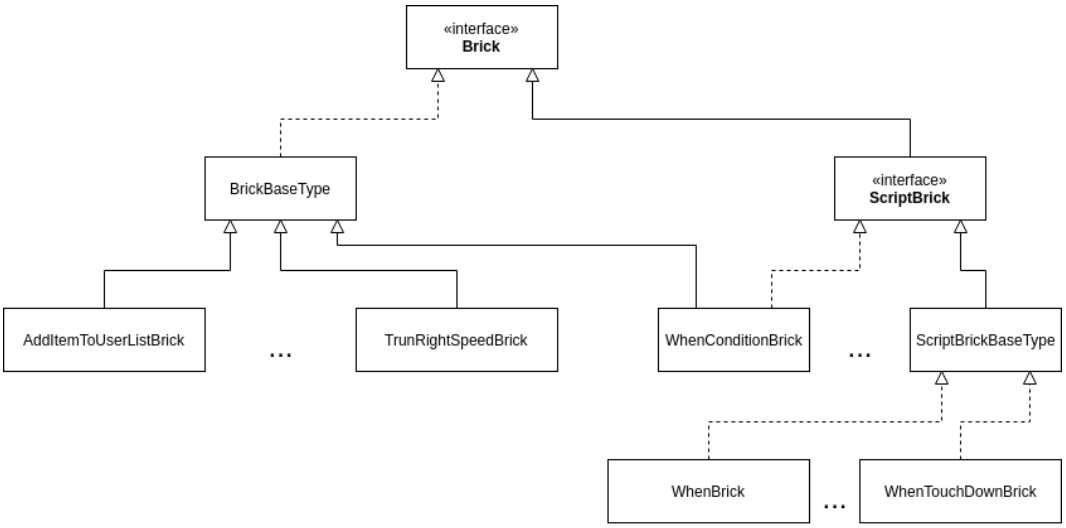

Figure 4.17: While there is additional hierarchy and various subclasses, all bricks implement the brick interface.

to extension) while discouraging modification (closed to modification). Figure 4.17 shows the brick class hierarchy. It can be seen that while there are base implementations and various child classes, all bricks implement the `Brick` interface.

By changing the data modelling to a tree, dataset operations had to be moved to the tree's children, i.e., the bricks. There are three operations that the data structure needs to support: removing children from anywhere in the tree, flattening the tree to a list and adding children to any level of the tree. In order to adhere to the open-closed principle, all methods necessary for these operations were added to the `Brick` interface. A representation of the tree data model and how it should be flattened can be seen in Figure 4.18. The tree's root is the script itself and any non-empty script without control structures has a height of one. The script brick is not part of the tree because it is the visual representation of the script and cannot be removed or added without adding or removing the script. Just like the syntactic bricks in a control structure it serves no functional purpose in the program. All children that have a depth of one, i.e., all subchildren of the root are in the scripts object's list of children. Any children with a depth of two have to be

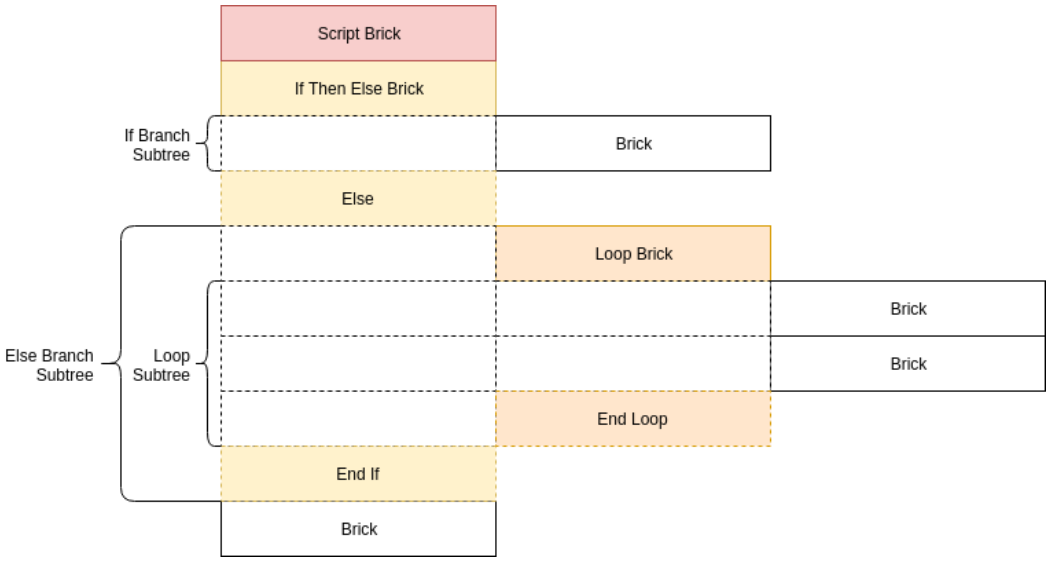

Figure 4.18: Representation of a script and its subtrees. The bricks in the leftmost column are on the first child level of the hierarchy. Syntactic bricks, i.e., the else-, end-if- and end-loop-brick are marked in dashed rectangles. They are only part of the flattened representation, but not of the actual tree.

subchildren of a child in the script's list. Any children with a depth larger than two have to be subchildren of subchildren. In order to find a brick in this structure the tree has to be traversed.

Removing Bricks from a Script

The method `public boolean removeBrick(Brick brick)` in the script class traverses the tree implicitly. Hence, none of the classes that use this method to remove bricks from a script have to know about the tree structure. If the brick that should be removed is on the first child level it is removed from the list. If the brick is not found the script invokes `public boolean removeChild (Brick brick)` on each brick in the list, delegating removal to the bricks. All bricks provide the method `public boolean removeChild(Brick brick)` through the `Brick` interface. Bricks that do not have sublists simply do nothing and return false. Bricks that hold sublists, remove the brick if it is held by them

directly or otherwise delegate the call to the bricks in their sublists. The method returns true once the respective brick is found, hence, worst-case, i.e., if the brick to remove is not in the tree, the operation traverses the whole tree once. An example implementation for the if-then-else control structure can be seen in Listing 4.5.

This implementation has the advantage that it only relies on the `public boolean removeChild(Brick brick)` from the `Brick` interface so that neither the script nor a brick with sublists have to know the classes of the bricks in their lists. Hence, if any new brick classes are added, that handle removing children differently, the script class and all other brick classes do not have to be changed.

```
@Override
public boolean removeChild(Brick brick) {
  if (ifBranchBricks.remove(brick)) {
    return true;
  }
  if (elseBranchBricks.remove(brick)) {
    return true;
  }
  for (Brick childBrick : ifBranchBricks) {
    if (childBrick.removeChild(brick)) {
      return true;
    }
  }
  for (Brick childBrick : elseBranchBricks) {
    if (childBrick.removeChild(brick)) {
      return true;
    }
  }
  return false;
}
```

Listing 4.5: The method that removes a brick from an if-then-else control structure. It first checks if the brick is in one of the sublists and otherwise delegates the remove call to the child bricks recursively.

Flattening a Script

In the IDE scripts and bricks are visualized in a flat list. How a flat representation is obtained from the hierarchical data structure depends on the brick classes in the scripts. There are syntactic elements that are added to this flat list which are not part of any subtree. Because of the same dependency considerations that apply to removing elements from the tree, flattening is delegated to the brick classes. For this purpose the `Brick` interface provides the `public void addToFlatList(List<Brick> bricks)` method. The parameter is a, potentially empty, list where the bricks add themselves, all syntactic bricks that constitute their visual representation, and all bricks from their sublists. This is preferable over having the method return a list with all the bricks because multiple calls can simply reuse the same list and it is not necessary to create a new list for each call.

In order to flatten a script into a list first the script brick is added and then `public void addToFlatList(List<Brick> bricks)` is invoked on all bricks in the script's brick list, always passing the same list. Bricks with sublists add all their sublists recursively so that sublists of bricks that are in sublists are added as well. How a if-then-else control structure is flattened can be seen in Listing 4.6.

The flattening operation is used in the IDE's script fragment. Dataset visualization is handled by the `BrickAdapter` class. This adapter constructs and visualizes a list that was built from the flat representations of all scripts in a sprite. The adapter flattens the scripts as described above. Because the `public void addToFlatList(List<Brick> bricks)` method is part of the `Brick` interface the adapter only depends on the interface. This means that i.) because of the interface's stability it is unlikely that the adapter has to change because of changes to the bricks and ii.) if new brick classes are added, the adapter does not have to change either.

```
@Override
public void addToFlatList(List<Brick> bricks) {
  super.addToFlatList(bricks);
  for (Brick brick : ifBranchBricks) {
    brick.addToFlatList(bricks);
  }
  bricks.add(elseBrick);
  for (Brick brick : elseBranchBricks) {
    brick.addToFlatList(bricks);
  }
  bricks.add(endBrick);
}
```

Listing 4.6: The flattening method implementation for an if-then-else control structure. It adds the if brick via the super call, then all bricks in the if-branch then the syntactic else-brick, all bricks from the else-branch and a syntactic end-if-brick.

Selecting Bricks and Scripts in the IDE

Bricks and scripts can be backpacked, copied, deleted, and commented out through action modes in the IDE. However, there are some restrictions on how bricks and scripts can be selected. In general it is true that whenever a brick is modified, its children have to be modified as well. So if the users select a control structure they always operate on the entire control structure. Nevertheless, it is possible to select a child brick in a control structure and modify it separately. Entire scripts can be selected through their script bricks. Once the script is selected, all bricks in the script are visually highlighted and their checkboxes disabled. What selecting a script looks like can be seen in Figure 4.19.

Selecting one brick means selecting the entire subtree of the dataset that starts at this brick. Hence, to obtain all bricks that are influenced by the selection of one brick the method `public void addToFlatList(List<Brick> bricks)` can be used. The method recursively adds all children from the brick's subtree. If the UI detects that a user selects a brick in the list it can use this method to construct a list of bricks that should be selected in response.

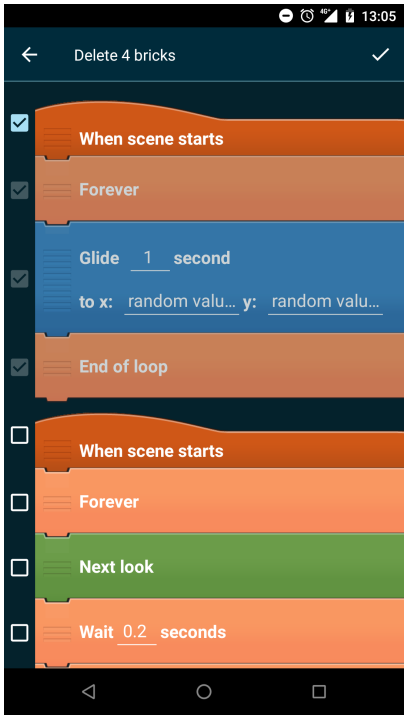

Figure 4.19: When users select a script for modification, all children are selected as well and their checkboxes disabled.

Encapsulating the logic in the bricks prevents dependencies between the UI and the dataset. As with the other operations any classes that use them only depend on the `Brick` interface. So any changes to how the selection works do not affect the UI.

Adding Bricks to Scripts

While adding a brick to any sublist is trivial, finding out to which sublist a brick has to be added in order to appear at a certain position in the flattened list is difficult. However, being able to do this is a prerequisite for reordering bricks in scripts. Rearranging bricks via drag and drop is subject to the same restrictions as the selection for modification, meaning that all child bricks have to move with their parents. Obtaining all bricks that have to be moved is trivial and simply works through the `public void addToFlatList (List<Brick> bricks)` method in the `Brick` interface. Moving subtrees in the data structure however, is more difficult. The main issue is to find the hierarchical parent from the flat representation and to interpolate the target position in the parent list. It stood to reason that the `Brick` interface should provide methods to obtain the target list and target position, because then the structure of the subtrees is encapsulated in the bricks and classes outside do not have to change if the subtree structure changes.

The method `public List<Brick> getDragAndDropTargetList()` can be used to get the list where a brick has to be added if it is dropped at the position of the brick on which the method was invoked. So if the user moves a brick from position one to position three, the UI calls the method on the brick that is currently at position three and adds the moving subtree to this list.

The data structure was implemented with links up and down the hierarchy. Parents have direct access to their children and children hold a reference to their parents. Calling `public List<Brick> getDragAndDropTargetList()` on bricks without sublists delegates the call to their parents. Script bricks return the script's brick list. Control structure bricks return their first sublist, and syntactic bricks that mark the end of the control structure return the control structure's parent list. In the case of an if-then-else construct the else brick

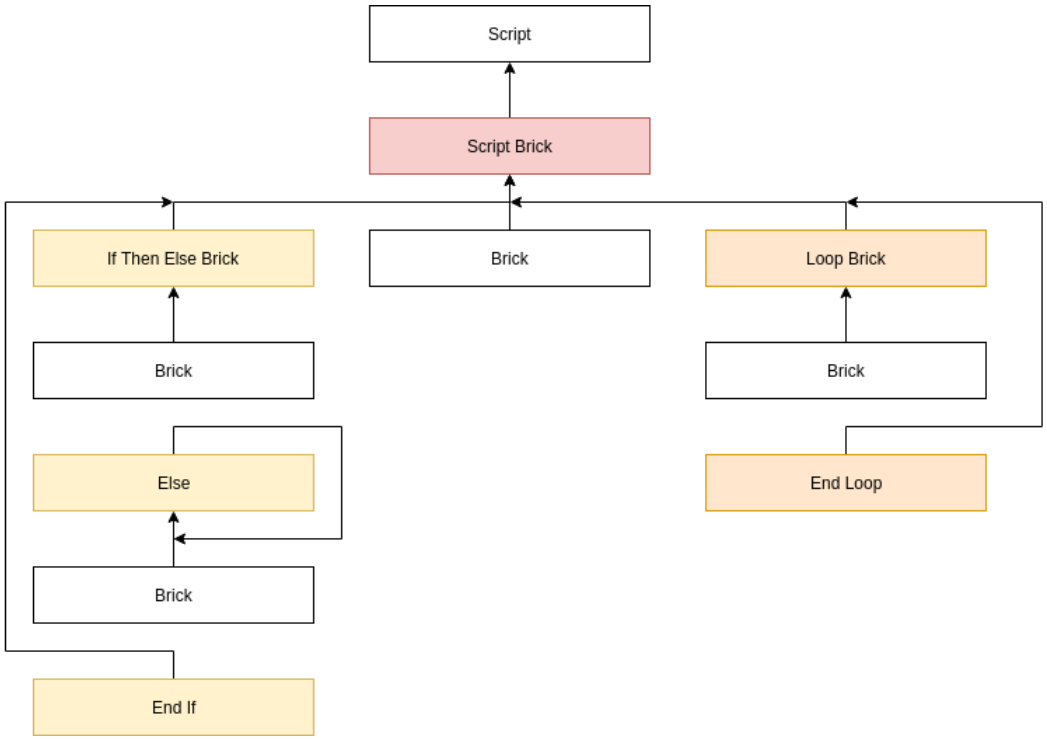

Figure 4.20: The Figure shows how the drop targets are obtained. The arrows indicate which sublist is returned when the method `getDragAndDropTargetList()` is invoked on each brick.

returns the control structure's else-branch list. Figure 4.20 visualizes the behavior in a diagram.

5 Results

5.1 Evaluation

The implications of the improvement process are clearly not sufficiently described by only analyzing code metrics for the refactored parts of the code. Instead, it might be interesting to see how the quality of the entire codebase has changed over a longer period of time. It is clearly true that changes in code quality can only be caused by code that was introduced during the development of new features, patches or rewrites. Besides, it is reasonable to assume that the knowledge, motivation and skills of the contributors are not significantly different on average. Additionally, literature and empirical data suggest that code quality typically decreases if there are no dedicated measures counteracting the decline. Hence, it stands to reason that code quality improvements are most likely caused by the developers who were involved in the refactoring process or by developers who were influenced positively by the process.

To capture changes in code quality it is interesting to look at metrics that describe the codebase's size, complexity and design. In addition to the metrics used in the results presented above, Table 5.3 provides an analysis of the code's average cyclomatic complexity (CC) per class and per method, as well as the cumulative CC of all methods in the project. Additionally, Table 5.1 lists the number of code smells. Code smells are well-known, recurrent patterns that are considered bad practice. Hence, occurrences of such patterns in the codebase indicate problems with the design.

The numbers in Tables 5.2 through 5.4 were calculated on each version

of Catroid that was released to Google Play. Hotfix releases (i.e., releases that were specifically tagged as hotfixes) were not considered because their scope was usually small and because they were typically released shortly after full releases. The first major refactorings described here were introduced in November 2016; between release v0.9.27 and v0.9.28. There were some refactorings between v0.9.24 and v0.9.27. However, these were not as extensive and dedicated as the rewrites that happened afterwards, which is why they were not described in this thesis.

Table 5.1 shows that code quality, described by the number of code smells, has improved over the last three years. The increase in the total number of code smells from 2016 to 2017 (between versions v0.9.24 and v0.9.32) is due to the large number of new features that were added to the codebase. Here, it becomes apparent why the measures of issues per line of code (LOC) and issues per file were chosen for this Table. When looking at the code size development in Table 5.2 it can be seen that the development that happened in parallel to the refactorings added approximately 250 files and 20,000 lines of code (LOC) to the codebase between August 2016 and November 2017 (between versions v0.9.24 and v0.9.32). The averages of code smells per file and per line of code (LOC) show that despite and increase in the total number of code smells, the relative number of code smells was reduced. While the relative number gives a more accurate description of the quality it is also interesting to look at the total numbers. Comparing the version v0.9.24 from 2016 with version v0.9.58 from 2019, it can be seen that despite both versions being roughly the same size in terms of LOC the number of code smells is lower by around 500, which is approximately 25%.

It is important to notice that the same number of LOC does not mean that both versions provide the same scope of functionality. In reality, the difference is significant, and the 2019 version provides considerably more bricks, scripts and additional features in the IDE and in auxiliary services. Examples include, among many others, a converter for Scratch programs, multiple new sensors values and functions that can be used in bricks and import/export features for Catrobat projects. An approximate indication to the scope of these features can be seen in the fact that v0.9.58 consists of 829 files, while v0.9.24 consists of 529 files.

Table 5.1: Code Smells

Version	Release Date	Files	Code Smells	Affected Files	Issues per File	Issues per LOC
v0.9.58	22.04.2019	829	1,474	451	1.78	0.0219
v0.9.46	06.03.2019	833	1,439	450	1.73	0.0212
v0.9.44	01.11.2018	816	1,504	451	1.84	0.0220
v0.9.42	24.10.2018	816	1,504	451	1.84	0.0220
v0.9.34	12.08.2018	834	1,773	469	2.13	0.0240
v0.9.32	27.11.2017	826	2,202	489	2.67	0.0249
v0.9.31	02.11.2017	825	2,202	489	2.67	0.0249
v0.9.29	23.09.2017	795	2,093	464	2.63	0.0245
v0.9.28	01.03.2017	764	2,230	461	2.92	0.0268
v0.9.27	26.10.2016	725	2,246	442	3.10	0.0274
v0.9.24	15.08.2016	590	1,906	380	3.23	0.0274

Changes in codebase size are captured in Table 5.2. Besides an indication as to how the scope of the codebase has changed, these numbers provide information about system architecture. They show how average class and method size has developed over time. Class size is affected by system design. Keeping classes small is a primary objective in the development of clean code (Martin, 2009). In clean code the number of physical lines of code is not considered sufficient to describe class size, instead the number of responsibilities is counted. However, in order for a class to be small in terms of responsibilities, small physical size is still a necessary condition. Hence, the number of lines of code (LOC) per class is an interesting measure to consider.

The numbers show that despite a large increase in total LOC between 2016 and 2017 the average LOC per class have steadily decreased from 91 to 65 lines of code, which constitutes a reduction by approximately 30%. There were four major jumps in average LOC reduction, one between v0.9.24 and v0.9.27, one between v0.9.27 and v0.9.28, one between v0.9.32 and v0.9.34 and one between v0.9.44 and v0.9.46. Each of the reductions cut about 6% of average class size and coincides with one concrete measure that

Table 5.2: Codebase Size

Version	Release Date	LOC (total)	Classes	LOC (per Class)	LOC (per method)
v0.9.58	22.04.2019	67,383	1,320	64.76	8.32
v0.9.46	06.03.2019	67,736	1,326	64.94	8.30
v0.9.44	01.11.2018	68,273	1,293	67.01	8.44
v0.9.42	24.10.2018	68,262	1,293	67.00	8.44
v0.9.34	12.08.2018	73.999	1,315	71.24	8.43
v0.9.32	27.11.2017	88,559	1,590	83.40	8.90
v0.9.31	02.11.2017	88,405	1,584	83.35	8.88
v0.9.29	23.09.2017	85,523	1,512	83.66	8.87
v0.9.28	01.03.2017	83,129	1,464	84.53	8.88
v0.9.27	26.10.2016	82,029	1,473	87.52	8.91
v0.9.24	15.08.2016	69,663	1,254	91.00	9.29

was implemented. For example, the reduction between v0.9.27 and v0.9.28 occured in the same iteration as the list fragment refactorings described in sections 4.3.3 and 4.3.4. The reduction between v0.9.32 and v0.9.34 coincides with the introduction of the `RecyclerView` model described in sections 4.3.5 through 4.3.8.

Just like classes, functions should be short (Martin, 2009). The numbers in Table 5.2 show the average physical size of the methods in the system. The development is similar to the development in class size. There has been a steady decline with some more pronounced jumps. These jumps happened simultaneously to the jumps in average class size and hence coincided with the refactoring efforts. Overall average method size was decreased from 9.29 lines of code (LOC) to 8.32 LOC, which amounts to a reduction of about 10%.

Clean code principles suggest that methods should be too small to hold nested structures. Indentation levels beyond one or two are considered hard to read (Martin, 2009). Table 5.3 provides the average cyclomatic complexity (CC) measures for methods and classes. CC captures the degree of nesting by counting all paths through the control flow. The numbers show that average

method complexity has decreased steadily by approximately 5%, from 2.16 to 2.06. Hence, reducing average method size has also reduced average method complexity. Here, the most significant jump can be seen between v0.9.32 and v0.94, which is when the `RecyclerView` model was introduced.

A more drastic change can be seen in cumulative method complexity per class. Average complexity per class has been reduced by almost 30%, from 17.74 in 2016 to 12.72 in 2019. This was caused by a reduction in method complexity and a reduction in class size. A larger reduction in class complexity than in method complexity implies that on average, classes have fewer methods. This conclusion is corroborated by the response for a class (RFC) measures in Table 5.4 which count the number of potential method invocations that can happen on an object of a class.

The Chidamber and Kemerer metrics (CKM) in Table 5.4 assess code quality in regard to object oriented programming principles. The weighted method complexity (WMC) is the cumulative cyclomatic complexity of all methods in a class and hence the same as the CC per class in Table 5.3. Lower WMC and smaller RFC values imply simpler class design. Less complex classes are easier to understand and test. These two metrics have improved by approximately 30% and 28% respectively, which is significant.

Coupling between objects (CBO) and lack of cohesion between methods (LOCM) show an overall improvement in class design. CBO has significantly decreased between v0.9.32 and v0.9.34 which is when the refactoring described in sections 4.3.5 through 4.3.8 happened. Coupling has decreased only very slightly, remained stable or even increased between all other releases. While the increases are small it is still interesting to see that, even if other code metrics improve, coupling tends to increase if no specific measures are taken. CBO is an important measure to describe class hierarchies and dependencies. While not all couplings are inherently bad, in a system that is as tightly coupled as Catroid it is still desirable to reduce overall coupling. Only when architecture improves, and the dependency measures are lower, it becomes more important to distinguish between desirable and undesirable couplings. In any case, this highlights the importance of dependency management considerations in software evolution.

Table 5.3: Codebase Complexity

Version	Release Date	CC (total)	Methods	CC (per method)	CC (per class)
v0.9.58	22.04.2019	12,204	6,261	2.06	12.72
v0.9.46	06.03.2019	12,285	6,323	2.05	12.79
v0.9.44	01.11.2018	12,442	6,321	2.08	13.13
v0.9.42	24.10.2018	12,438	6,321	2.08	13.13
v0.9.34	12.08.2018	13,337	6,899	2.03	13.84
v0.9.32	27.11.2017	16,285	8,040	2.14	16.40
v0.9.31	02.11.2017	16,272	8,037	2.14	16.40
v0.9.29	23.09.2017	15,838	7,800	2.15	16.52
v0.9.28	01.03.2017	15,585	7,593	2.15	16.81
v0.9.27	26.10.2016	15,255	7,498	2.15	17.43
v0.9.24	15.08.2016	12,646	6,142	2.16	17.74

Lack of cohesion between methods (LOCM) has not improved significantly, but continuously. This shows that splitting and refining classes improves overall design. However, with between 1,600 and 1,200 classes, it stands to reason that changes to a small number of classes, compared to the entire system, cannot be too significant. It is rather interesting that the refactoring process has affected these metrics at all.

5.2 Meta-reflection

In the process described here design decisions were often taken intuitively; not necessarily because of a lack of knowledge about software development methodologies, but because of time constraints or because following proper methods would have required even more extensive rewrites. So from a certain perspective, some refactoring decisions introduced new technical debt. However, as pointed out by Cunningham (1992) a deliberate decision to accept technical debt can be beneficial as long as it is mitigated by a rewrite. This rewrite, however, does not necessarily have to happen immediately.

Table 5.4: Chidamber and Kemerer Metrics

Version	Release Date	CBO	RFC	LOCM
v0.9.58	22.04.2019	16.29	19.02	1.94
v0.9.46	06.03.2019	16.29	19.16	1.94
v0.9.44	01.11.2018	16.11	19.56	1.92
v0.9.42	24.10.2018	16.10	19.56	1.92
v0.9.34	12.08.2018	16.12	21.15	1.92
v0.9.32	27.11.2017	17.07	24.17	1.92
v0.9.31	02.11.2017	17.02	24.15	1.92
v0.9.29	23.09.2017	17.98	24.32	1.91
v0.9.28	01.03.2017	17.64	24.36	1.98
v0.9.27	26.10.2016	17.66	25.02	2.03
v0.9.24	15.08.2016	17.55	26.12	2.05

More reasonably, it has to happen before the code that introduces the debt becomes a target for dependencies. This was a rationale in the refactoring process: Accept new technical debt as long as overall quality improves and remove newly introduced debt later.

This approach requires refactoring to be an ongoing process. Besides, it shows that the impact of a refactoring is best described by how it influences future development. In this regard, one challenging aspect was to know just how elaborate a refactoring should be. What this means is that on the one hand it is desirable to provide encapsulated components that can be reused by other developers. On the other hand, refactored code does not have to be a framework for other developers, meaning that it refactored code does not have to be held to higher standards than other code.

Instead, it stands to reason that rewrites are in general subject to the same considerations as other software development. Consequently, many approaches to software development could also be used to direct a refactoring. In hindsight it is obvious that using this rationale could have helped to optimize the rewrites. AntiPatterns for example, consist of a problem formalization and a systematic solution. Hence, AntiPatterns provide an effective

alternative approach to code quality improvement.

It would be interesting to compare the refactoring approaches suggested by literature to the ones that were implemented in the process described here. The decisions that the rewrites were based on tried to take into account conflicting objectives such as increasing code quality while quickly delivering new features without breaking existing functionality. Agile development approaches encourage combining these objectives. However, experience has shown that this is only possible to a certain degree. If a system has drowned in technical dept for example, it is virtually impossible to mitigate quality issues while simultaneously adding new features.

In Catroid the most difficult situations arose when the refactorings turned out to require significantly more extensive rewrites than anticipated. This was a problem because Catroid is actively used daily by people all around the world who expect to receive regular improvements. It was often challenging to combine refactorings with the day-to-day business of maintaining and extending the application. This however, presents an opportunity for follow-up research: Finding an effective way to combine the objectives, not only in theory but in the specific case of Catroid.

Refactoring itself has its pitfalls. Preventing regressions can be challenging, especially if the changes to the codebase are extensive. Automated tests are supposed to help to prevent regressions,. However, if the production code is difficult to test, tests cannot properly cover all potential faults. Writing new tests for untestable code is impossible. On the other hand, rewriting the code to be more testable without breaking functionality is impossible without proper tests.

In Catroid functional components were often intertwined with user interface components. For example, fragments and activities typically also handled file operations or dataset manipulation. Properly testing classes that have many different responsibilities is impossible and rewriting them without thorough verification via automated tests is just as undoable. This is a dilemma and one of the reasons why refactoring Catroid was difficult. It is also one of the reasons why many rewrites were not approached as systematically as described by Fowler (2018). This does by no means imply that

refactoring is not valuable, it just means that refactoring approaches have their limits and it is yet another argument for why effective refactoring has to happen continuously. This is nothing new, since Extreme Programming (XP) principles for example, suggest as much. However, this refactoring process has shown just how difficult it is to mitigate code quality issues if this advice is disregarded. Nonetheless, in hindsight it seems that putting even more emphasis on testing and trying to add more tests before rewriting the production code could have been helpful. Or put into a different perspective: testability is a prerequisite for successful refactorings. Hence, future refactorings should shift the focus to testability.

Regarding testability it is important to notice that the quality of tests plays a crucial role in the development process. Test quality is not confined to the code quality in tests, it also includes test composition. This means that the ratio between unit, integration and system tests is relevant. Most of the tests that were available before the refactoring were system tests. System tests on Android are inherently flaky (Coppola, Morisio and Torchiano, 2018). It is difficult to find the reasons for failures if tests are complex and test coverage was low. Hence, ensuring that the refactorings did not breaking anything was challenging. This is one more argument why reliable unit, integration and system tests are necessary and why it is a primary objective to write testable code.

Code quality metrics were used to provide a qualitative assessment of Catroid's codebase. However, they might as well be used as directives during development. For example, future refactorings could be planned in such a manner that they systematically reduce the number of code smells or the number of couplings or average method complexity. Considering code smells there is yet another possible application: integrating a quality gate into the automated testing process that prevent developers from adding new code smells. The question of how to use the metrics presented in this thesis during the development process provides an opportunity for further research.

The metrics used in this thesis can be grouped into two categories: The system-level analysis provides an abstract description of the overall system quality. Conversely, metrics calculated on the class level analyze the system's

atomic components (cumulative averages of class metrics constitute system-level abstraction). Atomic meaning that in terms of abstraction a class should be the smallest cohesive unit of a system. This thesis provides no explicit analysis on any levels in between. The reason can be found in Catroid's architecture. While it is typically convenient to describe a software in terms of collaborating class clusters, it was impossible to properly identify such groupings in Catroid. Because of the tight coupling between most of the classes it was not possible to decide which couplings were due to common functionality and which ones were just a result of poor dependency management. To properly apply software package metrics (Martin, 2002) it is first necessary to decouple classes throughout the entire system, until class categories become visible. While the measures described in this thesis are a step toward that development, there is still considerable potential.

Problems with modular design can be seen in the way classes are distributed into packages. Classes in packages often do not belong into the same package from a design perspective. The consequence is that visibility modifiers cannot be used properly and that classes are often more tightly coupled to classes in other packages than they are to classes within the same package.

Cleaning up package classification could significantly improve dependency management. While package metrics can be calculated regardless of the design quality, using them is only beneficial if there is a basic understanding of packaging and if packaging is used for its intended purpose. This provides an opportunity for future improvements and research: Identify cohesive parts of the class hierarchy, group them into packages, compare how this structure is different from the current packaging structure in Catroid and use the results to improve dependency design.

6 Summary

Catroid is a complex, large-scale, long-lived software system. During more than six years of lifetime, many requirements and the scope of the system's functionality have changed significantly. Some of the design decisions that were taken when the system was small have not aged very well. Besides, a number of development decisions that introduced a considerable amount of technical debt have effectively deteriorated system integrity. Dependencies between Catroid's subsystems have developed. Consequently, the scope and impact of changes has become difficult to estimate. Adaptability and extensibility has been limited severely.

This development has affected users and developers alike. A rigid system prevents fast, continuous delivery. Catrobat was often unable to properly respond to new requirements, such as changes to the Android framework by Google. Because of this, rollout of new features or patches was often impossible.

Developers have had problems with understanding the code and changes often required extensive rewrites. Proper automated testing has become unavailable because complex, interdependent code is hard to test. Without proper automated testing it is difficult to prevent regressions and changes to one part of the codebase regularly caused undetected side effects.

The work described in this thesis is a dedicated effort to counteract these problems. Most of the measures were based on the practice of refactoring. The overall objective was to reduce complexity and dependencies in order to promote a proper, modularized architecture. The expected benefits were to increase readability, allow for stable automated testing that provides proper coverage and to promote reusability and adaptiveness.

In order to quantify the results, metrics were introduced that capture code quality. Metrics such as cyclomatic complexity and code size provide a paradigm-independent evaluation. Metrics such as couplings between objects or weighted methods per class are part of a suite that was specifically tailored toward object oriented designs.

A thorough analysis on the refactored classes and an overall quality assessment of the codebase show that the measures have improved code quality. The metrics calculated on various versions of Catroid over the last three years have proven that refactoring can be used to counteract a deterioration of system integrity. Besides, the rewrites have promoted reusability by introducing code that is both understandable and adaptable.

Experience in this project has shown that clean, readable code that is properly encapsulated is in fact reused by developers. The results indicate that it is valuable to introduce refactoring as a common practice into the day-to-day routine of developers. Additionally, the findings present ideas for alternative approaches and present opportunities for future improvements.

Bibliography

Beck, Kent (2003). *Test-driven development: by example* (cit. on pp. 1, 24).

Beck, Kent and Erich Gamma (2000). *Extreme programming explained: embrace change* (cit. on pp. 2, 5, 6, 27).

Booch, Grady (1991). *object oriented design; with applications*. Tech. rep. (cit. on pp. 1, 6, 9).

Brown, William H et al. (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc. (cit. on p. 17).

Chidamber, S.R. and C.F. Kemerer (1994). 'A Metrics Suite for Object Oriented Design'. In: *IEEE Transactions on Software Engineering* 20.6, pp. 476–493 (cit. on pp. 13–16).

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 pp. 9, 87).

Cunningham, Ward (1992). 'Experience report-the wycash portfolio management system'. In: *Addendum to the proceedings on Objectoriented programming systems, languages, and applications (Addendum), OOPSLA* 92 (cit. on pp. 6, 84).

Fowler, Martin (2012a). *IntegrationTest*. visited 22.7.2019. URL: `https://martinfowler.com/bliki/IntegrationTest.html` (cit. on pp. 7, 8).

Fowler, Martin (2012b). *TestPyramid*. visited 23.7.2019. URL: `https://martinfowler.com/bliki/TestPyramid.html` (cit. on p. 8).

Fowler, Martin (2012c). *The Practical Test Pyramid*. visited 25.7.2019. URL: `https://martinfowler.com/articles/practical-test-pyramid.html` (cit. on p. 8).

Fowler, Martin (2012d). *UnitTest*. visited 23.7.2019. URL: `https://martinfowler.com/bliki/UnitTest.html` (cit. on p. 7).

Fowler, Martin (2018). *Refactoring: improving the design of existing code* (cit. on pp. 1, 6, 17, 42, 51, 86).

Gamma, Erich et al. (1995). 'Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley'. In: *Reading, MA*, p. 1995 (cit. on p. 17).

Google (2018a). *Activities*. visited 21.5.2019. URL: https://developer.android.com/reference/android/app/Activity (cit. on pp. 1, 5).

Google (2018b). *Activities*. visited 21.5.2019. URL: https://developer.android.com/reference/android/app/Activity (cit. on p. 17).

Google (2018c). *Data and file storage overview*. visited 16.5.2019. URL: https://developer.android.com/guide/topics/data/data-storage (cit. on p. 23).

Google (2018d). *Fundamentals of Testing*. visited 17.6.2019. URL: https://developer.android.com/training/testing/fundamentals (cit. on p. 8).

Google (2018e). *ListView*. visited 18.5.2019. URL: https://developer.android.com/reference/android/widget/ListView (cit. on pp. 18, 56).

Google (2018f). *Menus*. visited 18.5.2019. URL: https://developer.android.com/guide/topics/ui/menus.html#CAB (cit. on p. 18).

Google (2018g). *Understanding the Activity Lifecycle*. visited 18.5.2019. URL: https://developer.android.com/guide/components/activities/activity-lifecycle (cit. on p. 18).

Google (2018h). *View*. visited 21.5.2019. URL: https://developer.android.com/reference/android/view/View (cit. on p. 18).

Google (n.d.). *Create a List with RecyclerView* (cit. on p. 18).

Harzl, Annemarie et al. (2013). 'Comparing purely visual with hybrid visual/textual manipulation of complex formula on smartphones'. In: *VLC 2013-International Workshop on Visual Languages and Computing*. . (cit. on p. 2).

Hitz, Martin and Behzad Montazeri (1996). 'Chidamber and Kemerer's metrics suite: a measurement theory perspective'. In: *IEEE Transactions on software Engineering* 22.4, pp. 267–271 (cit. on p. 16).

Khomh, Foutse, Massimiliano Di Penta and Yann-Gael Gueheneuc (2009). 'An exploratory study of the impact of code smells on software change-proneness'. In: *2009 16th Working Conference on Reverse Engineering*. IEEE, pp. 75–84 (cit. on p. 17).

Lehman, Meir M (1980). 'Programs, life cycles, and laws of software evolution'. In: *Proceedings of the IEEE* 68.9, pp. 1060–1076 (cit. on p. 6).

Luhana, Kirshan Kumar, Christian Schindler and Wolfgang Slany (2018). 'Streamlining mobile app deployment with Jenkins and Fastlane in the case of Catrobat's pocket code'. In: *2018 IEEE International Conference on Innovative Research and Development (ICIRD)*, pp. 1–6 (cit. on pp. 2, 25).

Luo, Qingzhou et al. (2014). 'An empirical analysis of flaky tests'. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, pp. 643–653 (cit. on p. 24).

Martin, Robert C (1994). 'OO design quality metrics'. In: *An analysis of dependencies* 12, pp. 151–170 (cit. on pp. 10–12, 37).

Martin, Robert C (1996). 'The open-closed principle'. In: *More C++ gems* 19.96, p. 9 (cit. on p. 70).

Martin, Robert C (2002). *Agile software development: principles, patterns, and practices*. Prentice Hall (cit. on p. 88).

Martin, Robert C (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education (cit. on pp. 36, 54, 81, 82).

McCabe, Thomas J (1976). 'A complexity measure'. In: *IEEE Transactions on software Engineering* 4, pp. 308–320 (cit. on p. 13).

Müller, Matthias, Christian Schindler and Wolfgang Slany (2019). 'Engaging Students in Open Source: Establishing FOSS Development at a University'. In: *Proceedings of the 52nd Hawaii International Conference on System Sciences* (cit. on p. 3).

Sjøberg, Dag IK et al. (2012). 'Quantifying the effect of code smells on maintenance effort'. In: *IEEE transactions on Software Engineering* 39.8, pp. 1144–1156 (cit. on p. 17).

Spinellis, Diomidis (2006). *Code quality: the open source perspective*. Adobe Press (cit. on p. 1).

Steinmacher, Igor et al. (2015). 'A systematic literature review on the barriers faced by newcomers to open source software projects'. In: *Information and Software Technology* 59, pp. 67–85 (cit. on pp. 3, 21).

Stewart, Simon (2010). *Test Sizes*. visited 23.7.2019. URL: `https://testing.googleblog.com/2010/12/test-sizes.html` (cit. on p. 7).