



Florian Winkelbauer, BSc

Tackling Software Quality Problems in a Free and Open Source Software Project

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Software Development and Business Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Institute of Software Technology

Co-Supervisor: Dipl.-Ing. Annemarie Harzl, BSc

Graz, January 2016

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

The quality of a software product depends on several aspects found in a development environment. Writing software is influenced by factors such as the knowledge of developers, team collaboration, as well as different processes and tools.

This master thesis explores how code reviews, automated testing and version control workflows effect software quality in the development process of the Pocket Code mobile application. The major contributions to the Pocket Code project include:

- A checklist for performing code reviews
- An adaptation in the pull request acceptance process in which code reviews and static checks are a requirement
- An article series teaching aspects of how to write testable code
- An alternative git workflow using feature toggles instead of long-living feature branches

Contents

Abstract	v
1. Introduction	1
1.1. Problem Description	2
1.2. Thesis Overview	3
2. Catrobat	5
2.1. Extreme Programming	6
2.1.1. Code Refactoring	7
2.1.2. Automated Testing (Test Driven Development)	7
2.1.3. Pair Programming	8
2.2. Development Tools	9
2.2.1. Atlassian Software Tools	9
2.2.2. Git	11
2.2.3. Testing Tools	17
2.2.4. Gradle	20
2.2.5. Jenkins	20
3. State of Pocket Code's Development	23
3.1. Interviewing Team Members	23
3.1.1. Interview Preparation	23
3.1.2. Interviewee Selection Process	24
3.1.3. Interview Sessions	25
3.1.4. Interview Results	26
3.2. Personal Notes	27
3.3. Thesis Focus	28
4. Code Reviews	31
4.1. Current Situation	31

Contents

4.2.	Problem Description	32
4.3.	Related Work	33
4.3.1.	Code Reviews	33
4.3.2.	Checklist Based Reading	35
4.4.	Creating a Code Review Guideline	36
4.5.	Practical Application	38
4.5.1.	Creating a Checklist	38
4.5.2.	Search for Code Reviewers	39
4.5.3.	Checklist Usage Analysis	41
4.5.4.	Utilising GitHub's Branch Protection Feature	41
4.6.	Results	47
5.	Automated Testing	49
5.1.	Current Situation	49
5.1.1.	Jenkins Troubleshooting	49
5.1.2.	Broken User Interface Tests	50
5.2.	Problem Description	51
5.2.1.	Flaky Tests	51
5.2.2.	Testing Pyramid	51
5.2.3.	Test Statistics	53
5.3.	Related Work	55
5.4.	Creating Learning Material	57
5.5.	Practical Application	58
5.5.1.	Creating a Testing Article Series	58
5.5.2.	Roadmap: Refactoring Pocket Code's Test Suite	59
5.6.	Results	62
6.	Software Development Workflow	65
6.1.	Current Situation	65
6.1.1.	Working with Git	65
6.1.2.	Gitflow	66
6.1.3.	Pocket Code's Git Workflow	67
6.2.	Problem Description	71
6.2.1.	Different Pull Requests Sizes	71
6.2.2.	Feature Branch Lifetime	73
6.2.3.	Maintaining Two Central Branches	73
6.3.	Related Work	73

6.4. Proposed Workflow Changes	75
6.4.1. Using Feature Toggles	76
6.4.2. Handling Master and Develop Branch	77
6.5. Practical Application	79
6.5.1. Removing Feature Branches	81
6.5.2. Removing One Branch	84
6.6. Results	85
6.6.1. Feature Toggles	85
6.6.2. Using a Single Branch	86
7. Conclusion	89
8. Future Work	91
A. Questionnaire (German)	95
B. Consent Form (German)	101
C. Interview Results (German)	103
D. Checklist	111
E. Pull Request	123
F. Feature Toggles	125
G. Testing Articles	129
H. Roadmap to Improve Pocket Code’s Test Suite	163
I. How Git Handles Commits	165
J. Confluence Decision Log: Git Workflow	167
Bibliography	171

List of Figures

2.1. Pocket Code App	6
2.2. Kanban Board	11
2.3. Confluence	12
2.4. Git Branching	13
2.5. Git Merging	14
2.6. Git Tagging	16
2.7. GitHub Pull Request	18
2.8. GitHub Pull Request Details	19
2.9. GitHub Pull Request Summary	20
2.10. Jenkins	21
4.1. GitHub's Review Feature	42
4.2. GitHub's Pull Request Merge Information	43
4.3. Merge Problem Example Part 1	44
4.4. Merge Problem Example Part 2	45
4.5. Merge Problem Example Part 3	45
4.6. Merge Problem Example Part 4	46
5.1. General Testing Pyramid	52
5.2. Pocket Code's Testing Pyramid	53
6.1. Gitflow's Development Workflow	68
6.2. Gitflow's Release Workflow	69
6.3. Pocket Code's Development Workflow	70
6.4. Merging Sub-projects	72
6.5. Feature Toggle Example	76
6.6. Alternative Release Workflow	80
6.7. Release Information	81
6.8. Defining A Gradle Feature Toggle	82

List of Figures

6.9. Using A Gradle Feature Toggle	82
I.1. Git Commit Details	166

1. Introduction

Software users and developers have different sets of needs: A user mainly cares about what he or she can achieve with an application. In this context "good software" could mean that the application is reliable and that it fulfils its intended purpose. For a developer on the other hand, "good software" could mean that a code base is maintainable, understandable and easily extendible. Even though these needs seem different, we can see a connection between them: An understandable code base makes it easier for a developer to create more reliable software.

Software quality in the context of the development process depends on many different factors including:

- Available resources (e.g. team members and their work schedule)
- Individual experience of each member
- Team collaboration
- Testing and validation processes
- Tools which further assist the development process (e.g. testing tools, version control software, project planning software or knowledge distribution software)

The goal of this master thesis is to explore how a team of developers is effected by some of the above mentioned factors. To do so, I chose to work with the team which develops the Pocket Code mobile application. This team is not only the biggest and most active team in the Catrobat project, but its work also influences several other sub-projects in Catrobat. These reasons made Pocket Code a very appealing study subject as potential improvements could spread to other Catrobat teams.

1. Introduction

1.1. Problem Description

Aspects of the Pocket Code project include:

- Pocket Code is a student project - most of the team members are undergraduate students
- The project currently has about 15 members
- Students with the highest amount of project experience are very likely to leave the project soon as they are getting closer and closer to finishing their studies. As a result new team members may not always get the chance of collaborating with highly experienced colleagues
- Team members receive little to no formal training. Instead students are encouraged to explore recommended reading material or learn through working together with other students
- Even though the team tries to keep the code base understandable and bug free, meeting deadlines created by stakeholders leads to a conflict in priorities in which quality aspects may get neglected

Different sources indicate negative aspects concerning Pocket Code's software quality:

- Raphael Sommer, [2016](#) has studied motivation in the Catrobat project through a survey. His master thesis shows that one of the most present demotivators in the project is software quality. Students reported several aspects which drain their energy including:
 - The current code base is very complex
 - Having a broken testing environment
 - Fixing old functionality
 - Software development methods and practices are poorly implemented
- The occurrence of negative Pocket Code user reviews: 12 out of 115 Google Play Store ratings (about 10%) that were written between the 17th of April 2016 and the 19th of October 2016 mention dissatisfaction because of application crashes and data loss

Initially this master thesis focused on the following topics:

1.2. Thesis Overview

1. Onboarding: A process which helps to integrate new team members into a project or an organization. The goal is to minimize the time needed before they can start to add value (Fagerholm et al., 2014).
2. Coding dojo: Events at which a group works on a certain coding challenge (also known as a kata). The focus of these challenges is on learning through discussions and sharing experiences (Sato, Corbucci, and Bravo, 2008)
3. Code review: A process which aims to reduce software defects by letting other team members review source code (Bosu, Greiler, and Bird, 2015)

These topics were further adapted and refined through learning more about Pocket Code's development process as well as the day-to-day problems Pocket Code developers face. Interviewing team members appeared to be an appropriate first step to gain more insight into the project. After analysing the gathered data I decided to drop the onboarding and coding dojo topics in favour of two alternative subjects while still keeping the code review topic:

1. Automated testing: How developers can write code to check the expected behaviour of the software product in an automated fashion (Kent Beck, 2002)
2. Software development workflow: How a team can work together to develop a software product (Driessen, 2010)

A detailed description on why this shift occurred and how these fields are connected to the code review process is provided in Chapter 3.

1.2. Thesis Overview

This master thesis is organized as follows:

Chapter 2 introduces the Catrobat project and presents development practices and tools the team applies in its daily development work.

Details about the interview process of Pocket Code team members and how this information has influenced this thesis are presented in Chapter 3.

1. Introduction

Chapter 4 to 6 each present one of the three main topics of this thesis:

- The process of supporting code reviewers through learning material as well as the attempt to motivate team members to contribute in code reviews are presented in Chapter 4
- A detailed analysis of Pocket Code's testing suite as well as created learning material and a proposed roadmap to change the current testing suite is presented in Chapter 5
- Chapter 6 discusses Pocket Code's development workflow using the version control software git

Finally, an overall summary of this thesis and future work are outlined in Chapter 7 and 8.

2. Catrobat

Catrobat is a free open source visual programming language developed by the Catrobat project at Graz University of Technology (Austria). It promotes computational thinking skills by enabling children and teenagers to create and share their own applications. Catrobat is inspired by Scratch, another visual programming language developed by the Lifelong Kindergarten Group at MIT Media lab. While Scratch is designed for the use on desktop computers, Catrobat is designed for mobile devices with multi touch capabilities. Pocket Code is a mobile application which is used to create projects written in the Catrobat language. It is available on Android devices through the Google Play Store; the iOS version reached public beta status in autumn 2016, while the Windows Phone version is still in development (Catrobat, 2016), (Scratch, 2016).

Programs in both Catrobat and Scratch are developed by chaining blocks together, where each block represents different programming statements (e.g. loops, conditions, events or other statements). Blocks are split into categories through a color code. Figure 2.1 shows a small example of source code written using Pocket Code.

The development of Pocket Code is split into different sub-projects, each with its own team. Some examples of these sub-projects include:

- Core - covers the development of most of Pocket Code's features
- Chromecast - a project about using Pocket Code with the Google Chromecast HDMI dongle
- Drone - a project which adds the capability to navigate a flying drone through Catrobat bricks
- Musicdroid - a project about creating music using Pocket Code

2. Catrobat

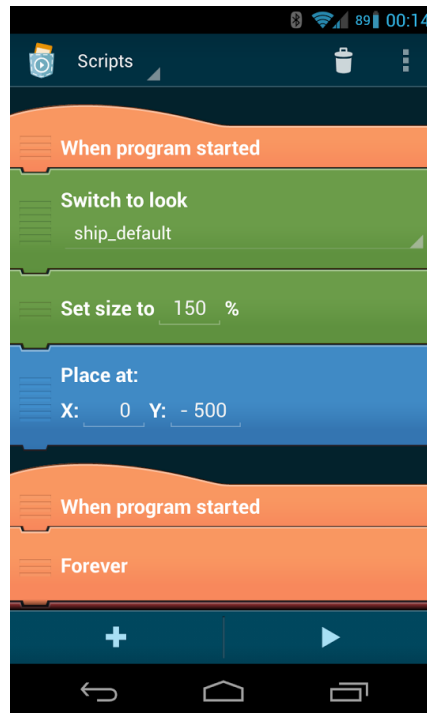


Figure 2.1.: Code example written in Pocket Code.

- APK Generator - a generator which is used to create standalone Pocket Code applications which are based on user-written projects. Such applications can be downloaded through Pocket Code's website.

In order to get an overview of the Catrobat development environment, the following sections cover aspects of how the developer team works and which tools they use.

2.1. Extreme Programming

Extreme Programming (XP) refers to a software development style which focuses on applying a set of programming techniques, clear communication and teamwork. Members of the Pocket Code team attempt to apply this development style in their daily work. The following sections describe a

2.1. Extreme Programming

subset of XP principles which are relevant for this thesis (K. Beck and Andres, 2004).

2.1.1. Code Refactoring

Refactoring is a technique which changes the internal structure of source code without changing its external behaviour. The goal of refactoring is to minimize the chance of introducing bugs by "cleaning up" existing source code. It improves the overall software design through simple steps such as moving a variable from one class to another, or splitting up a single big method into several smaller ones (Fowler, 1999).

2.1.2. Automated Testing (Test Driven Development)

The goal of software testing is to check that a piece of code behaves as expected. Manual testing has downsides as it becomes more tedious and error-prone the bigger the given code base. Automated testing refers to a practice in which code is used to create and execute a test suite consisting of several test cases which check the behavior of e.g. a single function, a class, or even the whole application. A single test case can either pass or fail. This approach has the following advantages over manual testing (Kent Beck, 2002):

- Better scaling
- Potential fast test runs which don't have to be supervised or observed
- Test cases can be used to identify bugs in existing code after changes were introduced (regression testing)
- Create reports for each test run which also include detailed error descriptions
- Tests can easily be executed over and over again, which creates a faster feedback loop

Test-Driven Development (TDD) emphasizes the importance of testing by writing test code prior to the "actual" code (often referred to as "production

2. Catrobat

code”). TDD relies on a short development cycle which consists of three phases (Kent Beck, 2002):

- Write a failing test case
- Write only enough production code to make the test pass
- Refactor

This cycle is repeated until all software requirements are satisfied.

2.1.3. Pair Programming

Pair programming refers to a practice in which two people share one computer (with one mouse and one keyboard) to work together on a programming task. Each pair has two roles (K. Beck and Andres, 2004):

- The partner in the “driving seat” (meaning that he or she currently uses the keyboard and the mouse) thinks about the best way to implement whatever task that is currently relevant
- The other partner considers the bigger picture, while asking several strategic questions:
 - Is there a better way to achieve what we are doing? What are the advantages or disadvantages?
 - What test cases do we have to consider?
 - Can this be simplified?

Roles and pairs are dynamic, meaning that a pair may switch roles from time to time or that people form new pairs. Pair programming can have several advantages even though it increases the man-hours that are required to finish a task. Advantages include (K. Beck and Andres, 2004):

- Collective code ownership
- Design quality (code contains fewer defects)
- Supports team communication
- Spreads knowledge (for the given code base and also in general)

2.2. Development Tools

The Catrobat team uses a broad set of development tools. The following listing is an incomplete short overview, as it is focused on topics that are relevant for this thesis.

2.2.1. Atlassian Software Tools

Atlassian is a company which creates products that assist software developers in their daily work. Two of their services are heavily used in the Pocket Code project:

- JIRA
- Confluence

JIRA

JIRA is a project management system which also provides issue tracking and bug tracking possibilities. For each issue (e.g. a new feature, a bug fix or some refactoring work) a JIRA ticket is created. These tickets contain all information concerning the issue, e.g.:

- A descriptive name
- A detailed description of what has to be done
- A priority indicator
- The current development status

While developing an issue the ticket status can change depending on the work progress (e.g. "in development" or "done"). These steps are mapped in a workflow which could for example look like this:

1. A ticket is created and put into the "issue pool" (a container for all new issue tickets)
2. If a project member is assigned to an issue, the ticket status changes to "in development"

2. Catrobat

3. After the development is done it could be labelled "in review"
4. Finally after the content of the ticket has been integrated into the software, the status could be set to "done"

To visualize such a workflow, a graphical representation is used which can display issues and their current status. This is commonly referred to as the board. Such a board can give an easy but detailed overview of the current development progress. Even though boards can be designed differently depending on development needs, they all display issues with their current status (Atlassian, 2016a). Figure 2.2 shows an example of Pocket Code's board. The listed issues consist of:

1. Issue tags (e.g. CAT-1134)
2. Issue titles
3. A current status (e.g. "Ready for Development")
4. Priorities through visual cues (e.g. a red arrow pointing up for important issues)
5. The current assigned project member (displayed through an avatar)
6. The age of an issue (displayed through grey dots at the bottom of an issue)

To further improve the visibility of the issue status, columns split issues by their status.

Confluence

Confluence is a Wiki software which is used for team collaboration, knowledge distribution and knowledge management. It can be used to store documentation including:

- Meeting notes
- Project pages
- Member profiles with additional information (e.g. a picture, an e-mail address or a phone number)
- News articles (blogs)
- Task lists
- General Wiki-style pages

2.2. Development Tools

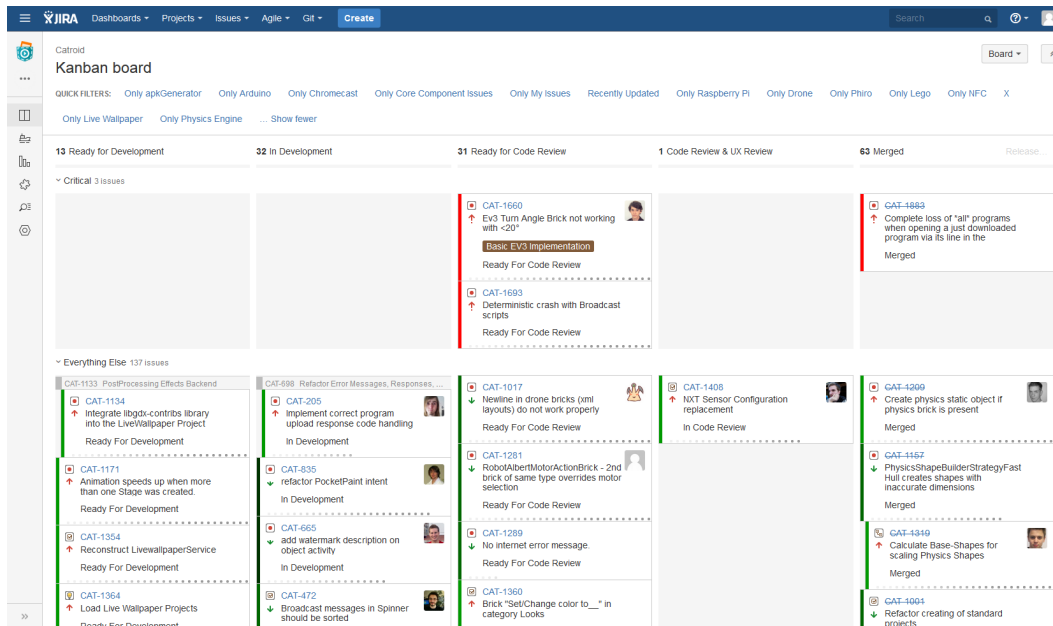


Figure 2.2.: Kanban board of Pocket Code.

- Calendar

Through the use of a markup language, an integrated calendar and plugins, Confluence is a versatile tool to store information. This markup language is also tightly connected to JIRA and other Atlassian products, which makes it for example possible to link JIRA issues in Confluence pages (Atlassian, 2016b).

2.2.2. Git

Git is a popular free and open source version control system which is mainly used to manage software development projects. It tracks file changes in a specific folder (also known as a repository). These changes are saved in commits, which represent a snapshots of the repository state. Git distinguishes itself from other version control software (e.g. Subversion) through two major features:

2. Catrobat

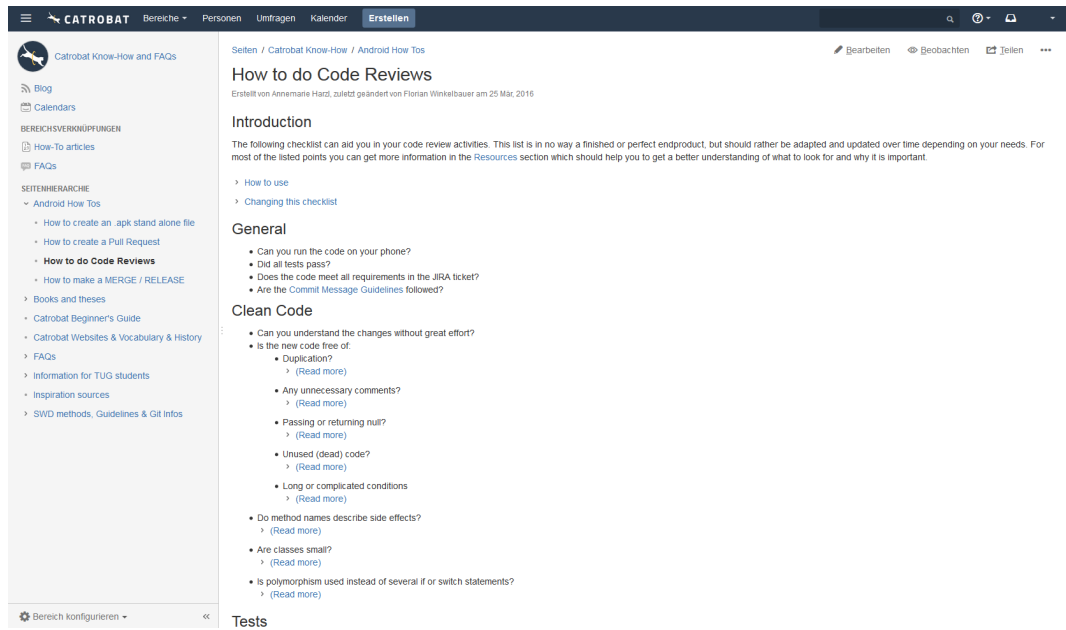


Figure 2.3.: A Wiki page about code reviews.

Decentralized Distributed System

Git is a completely distributed system. This means, that each user owns a complete copy of all files in the repository including its history. Instead of having a single entity which is responsible for sharing work done by team members (e.g. a central server), any repository can be used to up- or download collaborative work, though because of convenience such central hosting services are commonly used. An example of such a service is GitHub, which is described in more detail in Chapter 2.2.2.

Branches

Software projects can consist of several components which are developed by members simultaneously. Working with a single repository would lead to interference problems if for example members try to change the same set of files at the same time. Git solves this problem through branches, which can

2.2. Development Tools

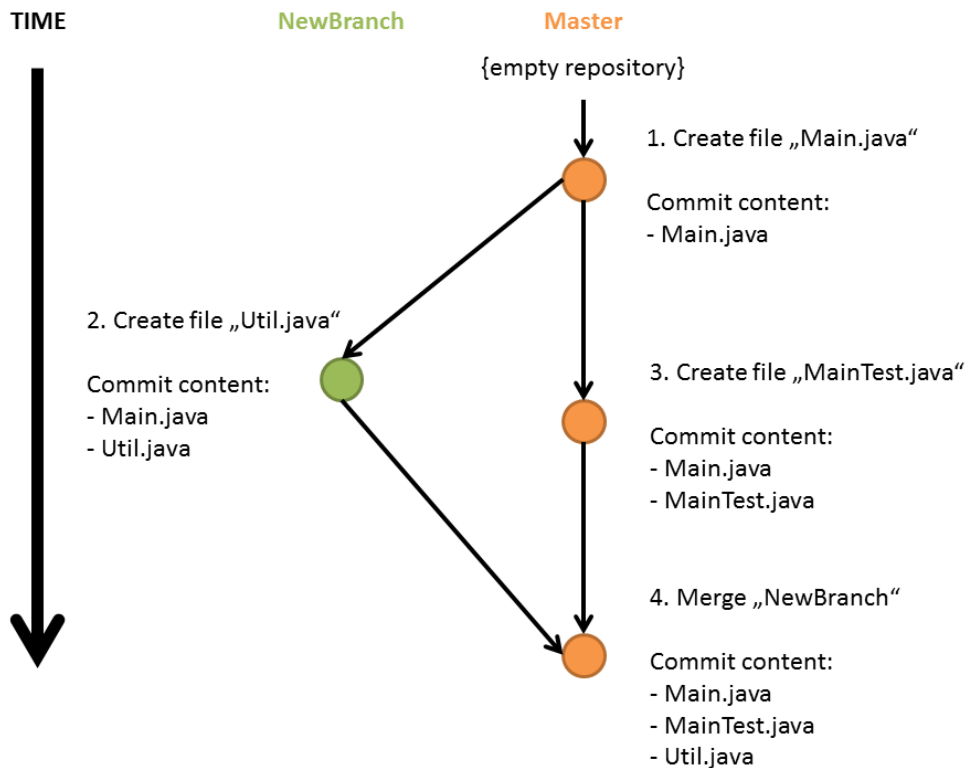


Figure 2.4.: Basic branching

be compared to having an organized set of repository copies, each with a unique name and its own set of commits. These changes can be pushed into other branches through a merge operation. Branches can be created, deleted or merged at any given point, which can become complex very quickly. Because of this complexity, there exist several branching strategies which define workflows, which can be used to effectively work with branches. Such workflows are further discussed in Chapter 6 (Git, 2016).

Figure 2.4 shows a very basic use case: Git is initialized in an empty repository. The first commit consists of a single file called "Main.java". After this commit a new branch is created which consists of another file "Util.java". Note that the class "Util.java" only becomes known to the "Master" branch, after the content of "NewBranch" has been merged.

2. Catrobat

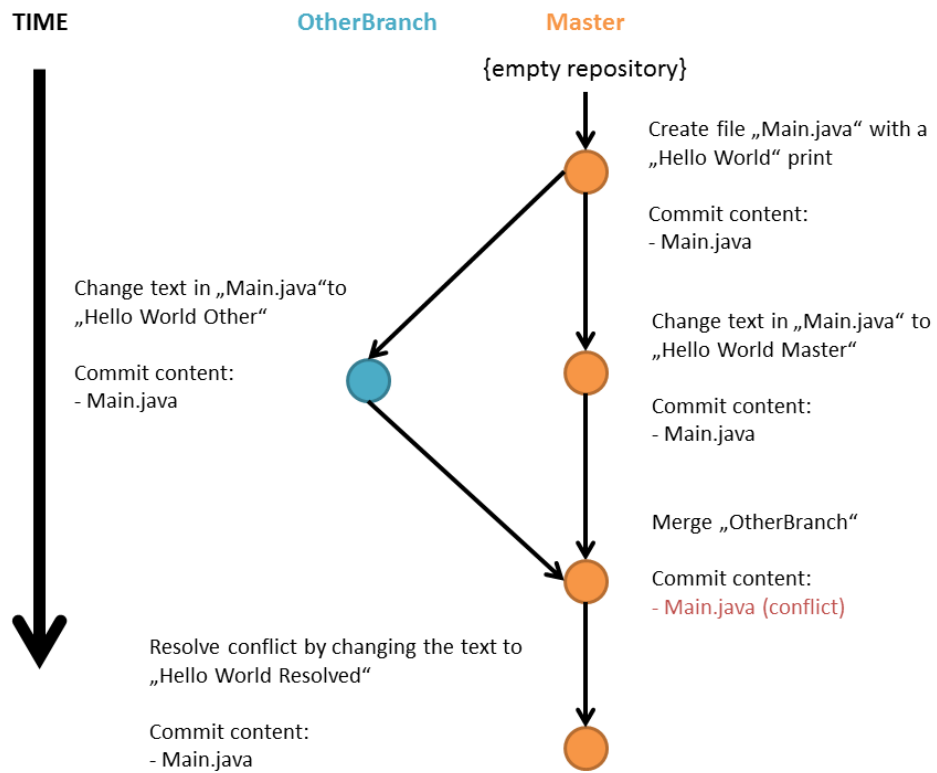


Figure 2.5.: A simple merge conflict

2.2. Development Tools

Merging branches can create conflicts if people work on the same set of files in different branches. Figure 2.5 visualizes such a scenario: Both branches "Master" and "OtherBranch" change the same file. After merging "OtherBranch" into "Master", git does not know which change it should apply. Git warns the user that a conflict has occurred and it is the user's duty to resolve this conflict in a following commit.

Extra Tools

In addition to the two mentioned features git also offers a broad set of extra tools (Git, 2016):

Tags can be used to add meta information to commits. Git also offers the feature to list all tagged commits. Tagging is often used to add version number information to the developed software (see Figure 2.6).

The rebase operation is an alternative to git's merge operation. While a merge simply creates new commits, a rebase can effect prior commits as well which can create changes in the existing commit graph. Both tools can be used to integrate changes from one branch into another.

GitHub

Github.com is a hosting service for remote git repositories, which adds social features to git. GitHub's community currently consists of more than 14 million people who work on over 35 million projects.

One of GitHub's most interesting features is its pull request system, which offers a pull based development style. Instead of directly merging a branch into another branch (e.g. merging "newFeature" into "master"), a pull request is created. Such a request not only lists a code diff of all new commits in a branch, but it also serves as a communication platform which is used to discuss the proposed changes (e.g. in a code review). This messaging system offers several additional features including:

- adding a reference to a GitHub user
- adding comments to a pull request or to selected code lines in a diff

2. Catrobat

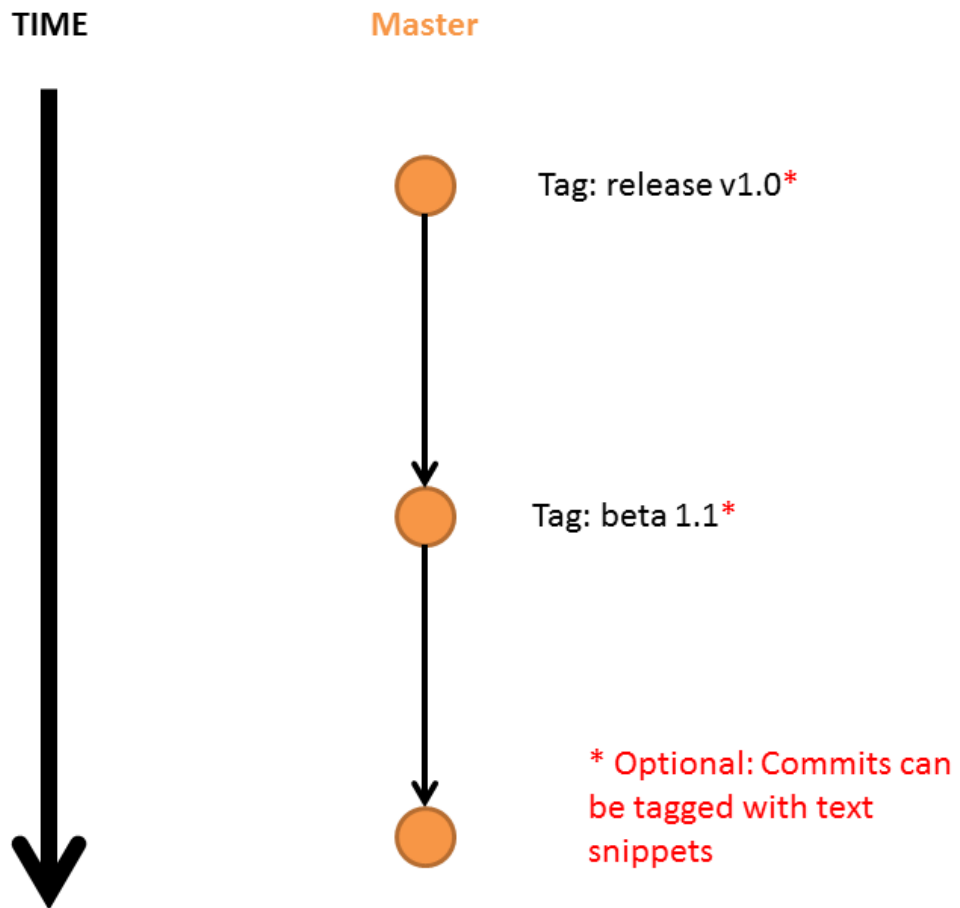


Figure 2.6.: Git allows commits to be tagged with text

2.2. Development Tools

- support to create syntax-highlighted code snippets

As long as a pull request is open, all additional changes which are committed to the particular branch, as well as all text messages that were added to the discussion are added to the pull request. Finally a pull request is either accepted or rejected. In this context accepting means that the branch is merged, while rejecting simply states that a branch will not be merged (GitHub, 2016b).

Figure 2.7 and Figure 2.8 show a typical example of a pull request. This pull request consists of an ongoing discussion about a proposed code change, a commit history (consisting of only a single commit) and a code diff which also includes comments. Figure 2.9 finally shows the overall summary of the pull request. The summary shows that two static checks (building the code and checking the code format) have passed and that no other conflicts exist, meaning that the pull request can be merged. Any discussion that happened in this pull request does not change the summary in any way.

2.2.3. Testing Tools

JUnit

JUnit is a framework which can be used to write unit tests in Java and Android. Pocket Code currently uses Android unit test features which are based on JUnit version 3 (JUnit, 2016).

Robotium

Robotium is an open source test framework for Android which allows developers to write automated tests which can access the graphical user interface of an app. Pocket Code currently uses Robotium version 5.2.1 (Robotium, 2016).

2. Catrobat

The screenshot shows a GitHub pull request interface for the repository 'Catrobat / Catroid'. The pull request is titled 'CAT-1863 Enumerating newly created backgrounds #1598' and is in an 'Open' state. It shows 1 commit from 'Achilles-96' merged into the 'Catrobat:deve1op' branch. The interface includes navigation tabs for Code, Pull requests (17), Wiki, Pulse, and Graphs. The main content area displays a conversation with three comments from 'robertpainsi' (a Catrobat member) and one from 'Achilles-96'. The comments discuss the implementation of background enumeration, mentioning 'getUniqueLookName' and 'SoundController'. A search link is provided: <https://github.com/Catrobat/Catroid/search?utf8=%E2%9C%93&q=getUniqueSoundName>. The right sidebar shows metadata for the pull request, including labels (None yet), milestone (No milestone), assignee (No one—assign yourself), and notifications (Unsubscribe). There are 4 participants in the conversation and a lock conversation option.

Figure 2.7.: Discussion of pull request on the issue CAT-1863.

2.2. Development Tools

The screenshot shows a GitHub pull request interface for the repository 'Catrobat / Catroid'. The pull request title is 'CAT-1863 Enumerating newly created backgrounds #1598'. It is created by 'Achilles-96' and targets the 'develop' branch. The pull request shows 2 files changed. A diff view is displayed for the file 'catroid/src/org/catrobat/catroid/ui/controller/LookController.java'. The diff shows a change on line 316, where a new line of code is added: 'imageName = LookFragment.getFinalLookName(imageName,lookDataList);'. A note is added by user 'robertpainsi' 3 days ago, mentioning a Jenkins build issue with Checkstyle warnings. The note includes a screenshot of a Checkstyle warning message: '[ant:checkstyle] .../catroid/src/org/catrobat/catroid/ui/controller/LookController.java:316:56: warning:'. Below the note, there is an 'Add a line note' button and a diff view for lines 316-318, showing the context of the change.

Figure 2.8.: Detailed overview of the proposed code in CAT-1863.

2. Catrobat

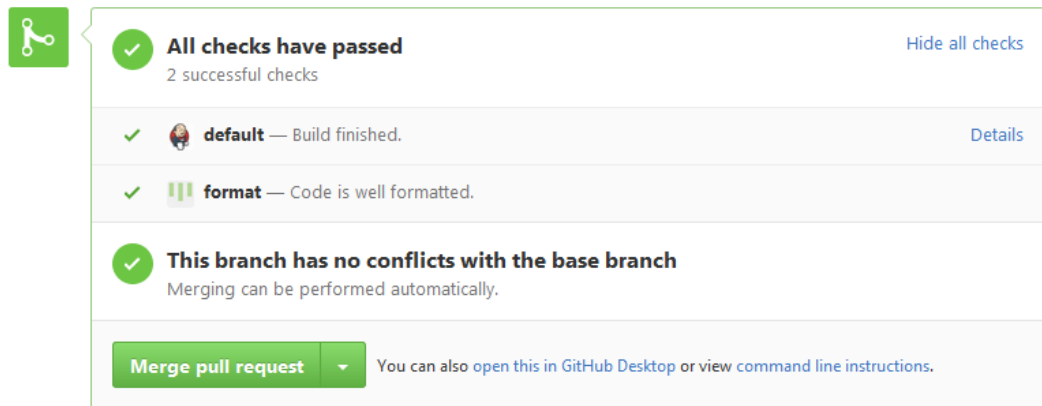


Figure 2.9.: Summary of the proposed code in CAT-1863.

Espresso

Espresso is another open source graphical user interface testing framework which is developed by Google. The Pocket Code team is trying to transition from Robotium to Espresso as Espresso is also used in all testing literature created by Google (Espresso, 2016). This framework transition is further discussed in Chapter 5.

2.2.4. Gradle

Gradle is a highly customizable open source software which is used to automate build processes. It uses a domain-specific language to define build steps and actions. Android projects use Gradle as their default build tool (Gradle, 2016).

2.2.5. Jenkins

Jenkins is an open source automation server which supports software project processes like building or deploying. This is achieved through jobs which

2.2. Development Tools

Welcome to Catrobat Jenkins 2.0 CI Server!

S	W	Name ↓	Letzter Erfolg	Letzter Fehlschlag	Letzte Dauer	Deployed On
		CatroidCheck	11 Tage - #187 develop	16 Tage - #181	1 Minute 47 Sekunden	N/A
		CatroidSingleClassEmulatorTest	11 Tage - #52 test.common.DefaultProjectHandlerTest	6 Tage 20 Stunden - #57 uitest.ui.fragment.SpritesListFragmentTest	2 Minuten 37 Sekunden	N/A
		CatroidSinglePackageEmulatorTest	5 Stunden 26 Minuten - #1024 uitest.ui.fragment.UserBrickDataEditorFragmentTest	4 Stunden 29 Minuten - #1026 uitest.drone	3 Minuten 10 Sekunden	N/A
		CatroidTriggerPullRequestTest	1 Tag 16 Stunden - #208 origin/pr/1719/merge	1 Tag 16 Stunden - #206 origin/pr/1731/merge	5 Minuten 22 Sekunden	N/A

Symbol: S M L

Legende RSS Alle Builds RSS Nur Fehlschläge RSS Nur jeweils letzter Build

Figure 2.10.: Jenkins overview for several jobs.

consist of several automated steps. Jenkins supports a broad range of tasks which can also be extended through plugins (Jenkins, 2016).

An example of a Jenkins job which is used for Pocket Code would look like this:

- Trigger a specific job if a new pull request is opened on GitHub for this project
- Checkout all source code from GitHub's pull request
- Start an Android emulator
- Create an executable file through building the source code using Gradle
- Start Pocket Code through uploading its executable on the emulator
- Perform test runs on the emulator
- Collect all test results, archive and visualize them
- Archive all information about this job (performed steps, date of execution and job result)
- Report the job result back to GitHub. The success of a job is visualized using a traffic light system:
 - Green = success (no errors)
 - Yellow = a build step was unstable
 - Red = a build step reports a critical error

Figure 2.10 shows a table of a few Pocket Code jobs. This view summarizes some data for each job:

2. Catrobat

- Job name
- Latest result (e.g. the latest execution of "CatroidCheck" was successful, while the latest execution of "CatroidSingleClassEmulatorTest" was not) and its execution time
- Statistics on when the last failure and success occurred

3. State of Pocket Code's Development

This chapter covers how I have improved my insight into the Pocket Code development process. This was done through interviewing team members as well as through creating notes based on personal observations while working in the project. I have used this information to further refine and adapt the focus of this master thesis.

3.1. Interviewing Team Members

3.1.1. Interview Preparation

Hove and Anda, [2005](#) wrote a great summary about different interview types:

- Structured interviews focus on a very specific purpose with very specific questions, which makes it possible to quantify its results.
- Unstructured interviews only suggest an overall theme with only a few specific prepared questions.
- Semi-structured interviews dive into both worlds, so they use very specific, as well as open-ended questions.

Semi-structured interviews seemed to be the perfect tool to dive into the development process of the Pocket Code core development team. As I have never worked in this team before it was essential to not only gain an overview, but also to gather qualitative data to not miss important details. To achieve this I have created a questionnaire which I could use throughout the

3. State of Pocket Code's Development

interview process. It consists of several specific and open-ended questions which are grouped in four categories:

- General information
- Onboarding
- Coding Dojos
- Code Reviews

The first section was used as a very light starting point for each interview. It consists of questions like "How did you join Catrobat?", "In which team are you currently active?" or "Have your programming skills changed since you have joined Catrobat?". These questions were not only used to let the interviewee loosen up, but also to create a first impression of his or her experience. Onboarding covered the topic of how the interviewee started out working in the project and how he or she received help while doing so. In the third section we talked about coding dojos and what one would expect of such events. Central questions were: What would make a coding session interesting? How often should it be done? The last section dove into the current code review process: How does it work? What is working great? What is not? What problems have occurred in the past concerning code reviews?

I have started to do test interviews with colleagues to not only gather experience in interviewing, but also to further refine the first draft of my questionnaire. Through these interviews I could create a second questionnaire which I used in all following sessions. You can find a copy of these questionnaires in [Appendix A](#).

Receiving honest and detailed answers was critical to further narrow down the focus of my thesis, which is why I decided to make all interview results anonymous. You can find a copy of the consent form which each interviewee signed in [Appendix B](#).

3.1.2. Interviewee Selection Process

Before I could start interviewing Pocket Code members I had to decide on which team members I should focus on:

3.1. Interviewing Team Members

- Juniors are new members which are currently in the process of getting started to work in the project
- Seniors are members with great knowledge about the code base and overall team processes

While juniors could potentially provide more information about their onboarding experience, seniors could provide a detailed look into code reviews and issues with past releases. Both groups could offer different but still insightful information, which is why I decided to interview both groups.

At the end of an interview I would often ask what other project members could be interesting candidates for an interview. This approach is also known as "Snowball sampling" (Atkinson and Flient, 2001).

After two interviews the questionnaire was further cut down to exclude onboarding and coding dojos. Even though they might be used effectively, their impact was estimated to be unfavourable: Onboarding affects new members only and leads to slow change, which seemed problematic for the scope of this thesis. Coding dojos would have been an extensive organisational investment with several open questions including:

- Who would organize these dojos?
- Who would prepare a dojo session?
- How can these events be made accessible? Student's schedules are very different and depend on different courses or exams.
- What would make such events interesting?

Code reviews on the other hand seemed promising as they could be done by several project members and could be done without major organisational effort, which is why they became the focus of all following interviews.

3.1.3. Interview Sessions

The interviews took place at the university in March 2016. We started each conversation with simple small talk and after the interviewee signed the consent form I would start a recording device. The recording made it possible to concentrate on the interview instead of having the extra task

3. State of Pocket Code's Development

of taking notes. Depending on how the interview went, I would leave out different parts of the questionnaire, while also weaving in other open-ended questions to steer the conversation into different directions. I could also cut the sections about onboarding and coding dojo from my questionnaire as they were no longer part of my thesis, which lead to shorter sessions. All interviews took between 30 to 90 minutes.

3.1.4. Interview Results

The last step was to summarize all findings:

- There is a strong reliance on Jenkins and other static analytic tools. The Jenkins system stopped working around the winter semester 2015/2016, which results in a disruption of the whole development process, as it is used as a reference for all tests.
- Actions have been taken in the past to motivate more people to do code reviews, though none of these attempts lead to changed behaviour. It is unclear why these changes did not stick. Potential hypotheses are that students are not confident enough, lack project and review experience or simply don't want to be bothered with additional effort. Other possible reasons are that code reviews seem like an ungrateful task, or that doing reviews is such a big habit change that it takes a significant amount of time investment to turn it into a regular and lasting habit.
- Reviewing pull requests which include several thousand lines of code changes is challenging and demanding. As a result bugs and design problems could potentially slip through more easily.
- There exists no written guideline on how to perform code reviews. A reviewer only relies on his or her own experience. The Confluence How-To section contains an article about creating a pull request, though this content seems to be outdated.
- Having only one code reviewer in the team, as well as having issues with Jenkins lead to a persisting bottleneck. Prioritizing pull requests can lead to situations in which some pull requests might be open for weeks or even months before they are processed. This delay further hinders processing older pull requests: In the meantime the common

code base may change significantly which increases the chance of creating non-trivial code conflicts in older pull requests.

You can find the results of all interviews in Appendix C.

3.2. Personal Notes

Catrobat members have access to a dedicated project room which they can use for all tasks surrounding the project. Through continuous presence in this room I was able to further improve my insight into the project. This was achieved by attending the weekly Pocket Code meetings, as well as observing live code reviews and online discussions on GitHub. I have also documented my experiences and thoughts as notes. The following list shows a summary of the major topics of these notes:

- Only a very small group of team members participate in code reviews
- Project members take issues concerning Jenkins with a sense of humour, even though this is (in my opinion) a very serious problem. Failing tests can be ignored more easily, leading to a delay in finding bugs
- People seem to lack knowledge about how to successfully write unit tests. This leads to code that is very hard to test. Existing tests are slow and may pass or fail randomly on successive test runs
- The Catrobat team lacks good material to teach testing concepts
- The most experienced members are those who will potentially leave the project very soon as they get closer to finishing their academic studies
- Catrobat's development workflow suffers from unnecessary maintenance and merge work which are a result of how sub-projects are incorporated in the workflow
- Pull requests consisting of huge code changes (several thousands lines of code) are practically impossible to review and can only be merged in good faith
- Pull requests are often merged in bursts of "review sessions". I believe that these time intervals are too short to perform proper reviews

3. State of Pocket Code's Development

- Adding new features happens through a "merge now, fix later" approach. There are always new issues which are more important and urgent, leading to neglected problems. This does not happen intentionally, but it is rather a result of several other reasons which are mentioned in this listing

3.3. Thesis Focus

Summarizing the interview results as well as my personal notes lead to this list of problematic areas:

- Finding and motivating team members to do code reviews
- Working with automated tests: writing tests and using them to gain quick feedback to find bugs
- Issues which are missed while reviewing large pull requests
- Lack of knowledge distribution
 - Performing code reviews
 - Writing testable code
- Unnecessary extra work which is caused by how Pocket Code handles sub-projects in its development workflow

On the surface these topics may seem diverse, but on closer examination we can see that there is a link between them:

Code reviewers rely on automated tests to find bugs or unexpected side effects in new code. If team members lack knowledge about how to write tests or how to write testable code, this crucial feedback is non-existent. And even if a proper set of tests is provided, a lack of maintenance work on these tests or an unstable test execution environment renders these tests unusable.

Large pull requests are a burden to code reviewers. The sheer size of a pull request seems to have an effect on the quality of the review itself. The size of a pull request is influenced by the design of Pocket Code's development workflow.

3.3. Thesis Focus

As a result of these findings I have decided to change the overall topic of this thesis to focus on three topics:

- Code reviews:
 - Motivating team members to participate in reviews
 - Providing learning material for potential new code reviewers
- Automated testing: Identifying and analysing issues in Pocket Code's testing environment. The findings should be used to educate team members about software testing
- Software development workflow: Analysing Pocket Code's development workflow to identify and address potential problems which have an effect on the release cycle

4. Code Reviews

The goal of code reviews is to improve software quality by getting feedback on an artefact (e.g. a piece of source code, or a technical document) from other engineers. Code reviews originate from software inspections which refer to a formal peer review process developed by Fagan, 1976. In its core, software inspections are meetings in which a group of participants review an artefact (Aurum, Petersson, and Wohlin, 2002).

Due to their formal nature, software inspections are very time consuming. Since its introduction about thirty years ago, less formal and more lightweight approaches have emerged. Researchers have termed them "modern code reviews" (Bosu, Greiler, and Bird, 2015).

This chapter discusses Pocket Code's modern code review process. The following sections cover these topics:

- Current problems in the review process
- Educational material which was created to help team members to get started in performing code reviews
- Attempts to motivate team members to participate in code reviews

4.1. Current Situation

The general workflow Pocket Code uses to work on JIRA tickets looks like this:

- A project member assigns himself to a ticket
- He or she works on said ticket

4. Code Reviews

- After the member deems his or her work finished, he or she will create a pull request on GitHub and change the JIRA ticket status to "Ready for Code Review"
- A project member reviews the pull request. This member will also change the ticket status to "In Review". Reviewers may start a discussion because of two major reasons:
 - There is a lack of understanding about specific details that have to be clarified before the review can be finished
 - Change requests are made by pointing out specific parts in the committed source code the reviewer would like to have improved. Depending on the severity of the findings, merging the pull request may get delayed until said changes have been implemented
- Lastly the pull request is merged and the ticket closed

4.2. Problem Description

The most problematic points regarding the current state of Pocket Code's code reviews have already been mentioned in Section 3.1.4. Here is a short summary:

- Learning material regarding code reviews is either non-existent or outdated
- Reviewers receive no kind of training
- Even though actions have been taken in the past to increase the number of code reviewers, most pull requests are reviewed and merged by a single person. Other project members seem to neglect the review task
- Understanding and reviewing pull requests which contain several thousand lines of code changes is problematic. This topic is further discussed in Chapter 6

4.3. Related Work

4.3.1. Code Reviews

Bernhart and Grechenig, [2013](#) did a study in an industrial context about the effects of continuous code reviews in regards to spreading knowledge and collective code ownership. They found that creating smaller code review tasks is the main reason for gaining positive effects in regard to knowledge distribution and code ownership. As this study was based on a 1:1 ratio between code sample tasks and reviewers, they proposed that more reviewers should be used so that this 1:1 bottleneck can be avoided.

Bosu, Greiler, and Bird, [2015](#) determined characteristics of useful code reviews through an empirical study at Microsoft. Implications of their results include that:

- The selection process for finding a reviewer has an important effect on the amount of useful comments in a code review. Even though this might suggest that one should only select experienced project members, they recommend that inexperienced reviewers should be added as well so that they can gather knowledge about the process.
- Review effectiveness decreases with an increase in the amount of files in a code review.
- Classification tools could be used to analyse code review comments to identify weak areas in the process.

Swamidurai, Dennis, and Kannan, [2014](#) compared pair programming and peer code reviews in the context of TDD. Their results suggest that peer code reviews seem to be at least as effective as pair programming while also offering two advantages:

- Peer code reviews are asynchronous
- Reviews take on average nearly 30% less time than pair programming

Bosu, Greiler, and Bird, [2015](#) performed a study at Microsoft in which they analysed review comments from five different projects to identify factors which have an impact on the usefulness of code review feedback. Their findings include:

4. Code Reviews

- The reviewer's experience with the code base improves feedback
- Review effectiveness decreases with an increase in the overall number of file changes included in a change set

Yu et al., [2015](#) studied factors which have an effect on pull request evaluation latency for open source projects hosted on GitHub. While pull requests make it easy for a potential contributor to propose code changes, an increasing amount of requests also increases the burden on integrators (project members responsible for evaluating and integrating proposed code changes). The following predictors in terms of pull request review latency were found:

- Pull request size: Shorter pull requests are reviewed faster
- Availability of an automated build and test service (e.g. Jenkins): Improving the service availability may reduce review latency
- The number of comments: A great number of comments in a pull request tends to signal controversy which can lead to longer evaluation times

Krusche, Berisha, and Bruegge, [2016](#) designed a university course which teaches how code reviews improve maintainability. In this course 300 students performed nearly 3000 code reviews for different projects with industry customers. A final interview session showed that students did no longer see code reviews as a bureaucratic burden and that they were convinced that code reviews improve software quality. Students did no longer see reviews as organisational extra work, instead reviews were perceived as an opportunity to learn from more experienced colleagues.

A study done by A. Bacchelli and Bird, [2013](#) explored modern code reviews in terms of expectations, outcomes and challenges. They analysed several hundred review comments in several teams at Microsoft. The study results show:

- Reviews do not identify defects as often as one would like. Deep, subtle or higher level issues may pass unnoticed
- Reviewers with prior knowledge of the context of the given code can give more valuable feedback
- Reviews in general bring other advantages besides finding defects. They can:

- spread knowledge
- increase code ownership
- help to improve code style
- be used to find alternative solutions
- increase learning

4.3.2. Checklist Based Reading

In a study done by Rong, Li, et al., [2012](#) the effect of code review checklists for beginners was investigated. Two groups of students conducted code reviews: one group used a checklist, while the other group did not. The study concluded that:

- Checklists help beginners to conduct code reviews by guiding them through the process
- Checklists do not help to improve review efficiency for inexperienced students. Rong, Li, et al., [2012](#) recommends that one should not rely solely on a checklist to perform good code reviews

A second study done by Rong, Zhang, and Shao, [2014](#) further investigated code reading techniques (including checklist based reading) through a case study in a small software company. The case study results conclude that:

- Checklists provide help when starting the process of code reviews
- Using a checklists slows down the code reading process, which can be beneficial in finding defects in code

Dunsmore, Roper, and Wood, [2002](#) discussed three reading techniques for inspecting object-oriented code: checklists, a use-case based strategy and a systematic technique. In their study checklists were not only the quickest technique that could be applied, but it also produced the overall best performance. In addition to that the researchers did not find a significant difference between best and worst reviewers in terms of defect detection. They also mention that checklists may show some weaknesses, including that:

4. Code Reviews

- Checklists rely on historical information being recorded and analysed so that the content can be improved
- Checklists lack effectiveness for certain defects like issues with missing lines of code

Hatton, 2008 found no evidence that checklists make a difference in defect detection using a formal statistical analysis of around 300 people. He states that the simplest explanation of these results might be that experienced engineers fall back on personalized internal methods instead of using checklists.

Rong, Zhang, and Shao, 2014 compared checklist based and ad hoc based reading for industrial novice inspectors. Their results show that checklists may improve the performance of novice inspectors for finding defects. However, inspectors might be limited by such a checklist, meaning that they would miss defects which are not mentioned on the list.

4.4. Creating a Code Review Guideline

To address the above mentioned issues I decided to provide learning material for project members who would like to start to do code reviews, while also trying to convey that the Pocket Code team is in desperate need of additional code reviewers.

Code reviews can be based on different strategies (or reading techniques) which help to identify potential issues. These strategies range from having a detailed approach which has to be learned, to less specific approaches. Aurum, Petersson, and Wohlin, 2002 list several reading techniques:

- Ad-hoc reading is the least structured strategy which solely relies on the personal experience of a reviewer. It requires no additional training and is the most natural approach a person would use if given the task of performing a review
- Using stepwise abstraction a reviewer determines the function of subprograms. Through combination of these subprograms the reviewer can build their own specification of the program. This derived

4.4. Creating a Code Review Guideline

specification is then compared to the official specification to identify differences

- Scenario-based reading and perspective-based reading are techniques which develop scenarios to assist the review process. These scenarios are created from a particular viewpoint and contain specific questions to find particular classes of defects
- Checklist based reading is an approach which uses a document (a checklist) consisting of several specific questions to pinpoint common problems of the past (Dunsmore, Roper, and Wood, 2003)

Of all these reading techniques, checklist based reading seemed promising as it can:

- be applied by review beginners without any extensive preceding training (see Section 4.3)
- serve as a knowledge database which can be used by anyone who is involved in the development process
- be integrated into Pocket Code's project environment (Confluence and GitHub) easily
- be maintained and updated after it has been created to keep its content relevant

Reasons for why the other techniques seemed less effective for the Pocket Code project include:

- Each other technique (excluding ad-hoc reading) is more complicated than ticking of items from a checklist. Applying these techniques can only be done after some form of training
- Pocket Code does not have an official specification, which is essential to apply stepwise abstraction
- Scenario-based reading and perspective-based reading would require the most time investment which might be discouraging for code review beginners

4. Code Reviews

4.5. Practical Application

The following sections cover my process of creating learning material for potential new code reviewers as well as attempts to increase the number of reviewers in Pocket Code.

4.5.1. Creating a Checklist

The task of creating a checklist lead to several questions including:

- What is considered best practice for creating such lists?
- What are typical formats?
- How long and detailed should they be?

Brykczynski, [1999](#) gave a detailed overview of effective checklists and what they have in common:

- They are updated regularly to include frequently occurring defects
- Their items should be phrased as a question (e.g. "Are there any areas in which null values are either passed or returned which could be avoided all together?")
- A checklist should be short. Having to scan several pages while doing a review should be avoided at all cost
- They should not cover topics that could be handled by other means (e.g. static code analysis tools)

I have used several resources including books, blogs and videos to create a checklist as a How-To article in Confluence, of which you can find a copy with all references in [Appendix D](#). It was designed to be used in three very distinct ways: First and foremost it is a list of items which should be considered while performing a code review. Second it can be used as a learning resource as most items can be expanded to get more information on what the item is about with examples, reasoning or additional resources (ebooks and video material). Lastly it serves as a central source for future work: How could this list be improved? What topics could be considered?

4.5. Practical Application

The checklist consists of 19 major items and is structured in the following way:

- Introduction: How should this checklist be used? How can it be changed?
- Major topics:
 - General: major questions include “Did all unit tests pass?” or “Does the code meet all requirements in the JIRA ticket?”
 - Clean code: includes items that check how readable the source code is
 - Tests: checks the quality of new test cases
 - Testable Code: identifies areas in the source code which are hard to test
 - Design: includes items that improve maintainability
- Blockers: gives advice on when accepting a pull request should be delayed because of issues found while doing a review
- Resources: includes references to the material used to create the checklist
- Future Work: a list of potential new items which could be considered for future updates of the list

Confluence did already contain a how-to article about creating pull requests, though some of its information and links were outdated. I have rewritten this page to not only update its content, but to also create coherence between this article and the newly created checklist.

As external contributors do not have access to Confluence I have also added this information to Catrobat’s Wiki on GitHub. You can find a copy of the pull request how-to article in [Appendix E](#).

4.5.2. Search for Code Reviewers

The release of both how-to articles (“How to do code reviews” and “How to create a pull request”) was announced through a Confluence blog post in March 2016. This post was also displayed on the Confluence start page. About one month after these announcements, a Confluence addon was

4. Code Reviews

installed which made it possible to track the usage data of the code review how-to article. This delay was used to collect view count information without considering views due to simple curiosity from the announcing blog posts.

The next step was to create awareness that one person cannot be responsible for all pull requests on GitHub. For this a separate meeting was held on the 17th of March 2016 at which one senior of each sub-team of Pocket Code had to be present. This meeting was used to define a new set of guidelines which should not only spread responsibility, but also to further define the overall development workflow. The most important aspects were:

- Seniors of a sub-team are responsible to review pull requests of their team members
- To decrease the size of feature branch pull requests, these branches should be merged back into the develop branch after certain milestones which consist of a set of issues. These milestones should be defined by each sub-team and should result into several smaller pull requests which are easier to handle. I was able to identify problems surrounding this milestone approach in hindsight. This topic is discussed in more detail in Section [6.4.1](#)
- Regular code review meetings are going to get scheduled so that the team can keep track of its progress

Pocket Code's weekly meetings were used to ask for help in handling pull requests instead of the previously mentioned separate regular code review meetings.

From March 2016 to June 2016 more than 130 pull requests were merged on PocketCode's GitHub page. The reviewer distribution looks like this:

- About 100 pull requests were merged by a single person
- The remaining requests were merged by less than ten different members

These findings show that barely any change in terms of code review participation was achieved. A possible explanation could be that team members had no significant reason to change their behaviour: Other tasks such as

developing a new feature for Pocket Code could be perceived as more fulfilling than reviewing the code of a colleague. This topic is further discussed in Chapter 8.

4.5.3. Checklist Usage Analysis

Because of a bug which occurred due to a Confluence update in July 2016, most of the usage information gathered by the Confluence addon regarding the checklist was lost. In June 2016 about 20 individual project members had visited the page, with a total of about 35 views. A small set of visitors viewed the page twice, while an even smaller group came back even more often. These numbers indicate that the checklist was not incorporated in the general pull request review process.

4.5.4. Utilising GitHub's Branch Protection Feature

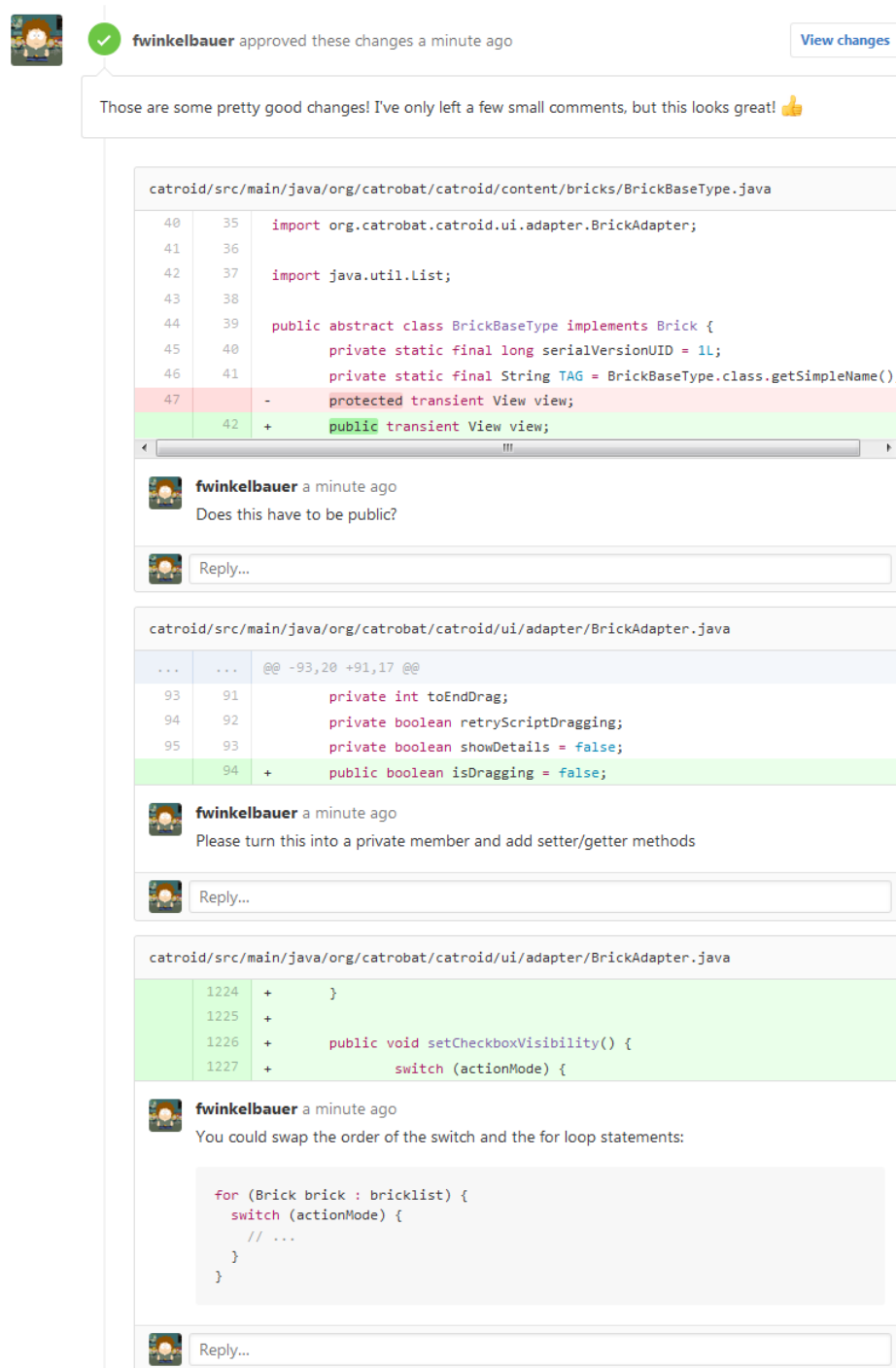
GitHub's New Review Feature


On the 14th of September 2016 GitHub announced additional features which were added to the site. One of these features included a set of new code review tools which were added to the existing pull request system (GitHub, 2016a). With these tools it is possible to:

- Group up comments into a single review session
- Leave a summary for the whole review session (including all made comments)
- Mark the review as "approved" or "request changes"

Another very interesting feature enables branch protection. This disables the merge action on pull requests until all static checks pass and a reviewer has approved of the pull request. Blocking reviews (reviews in which changes were requested) can be dismissed if a good reason is given, but there still has to be at least one positive review before a merge can happen. Dismissing a review could for example be reasonable if the reviewer is on a vacation and merging the pull request is urgent.


4. Code Reviews




 **fwinkelbauer** approved these changes a minute ago [View changes](#)

Those are some pretty good changes! I've only left a few small comments, but this looks great! 🍌


```
catroid/src/main/java/org/catrobat/catroid/content/bricks/BrickBaseType.java
40 35 import org.catrobat.catroid.ui.adapter.BrickAdapter;
41 36
42 37 import java.util.List;
43 38
44 39 public abstract class BrickBaseType implements Brick {
45 40     private static final long serialVersionUID = 1L;
46 41     private static final String TAG = BrickBaseType.class.getSimpleName();
47 -     protected transient View view;
48 +     public transient View view;
```

 **fwinkelbauer** a minute ago
Does this have to be public?

```
catroid/src/main/java/org/catrobat/catroid/ui/adapter/BrickAdapter.java
... .. @@ -93,20 +91,17 @@
93 91     private int toEndDrag;
94 92     private boolean retryScriptDragging;
95 93     private boolean showDetails = false;
96 +     public boolean isDragging = false;
```

 **fwinkelbauer** a minute ago
Please turn this into a private member and add setter/getter methods

```
catroid/src/main/java/org/catrobat/catroid/ui/adapter/BrickAdapter.java
1224 +     }
1225 +
1226 +     public void setCheckboxVisibility() {
1227 +         switch (actionMode) {
```

 **fwinkelbauer** a minute ago
You could swap the order of the switch and the for loop statements:

```
for (Brick brick : bricklist) {
    switch (actionMode) {
        // ...
    }
}
```

Figure 4.1.: GitHub's Review Feature

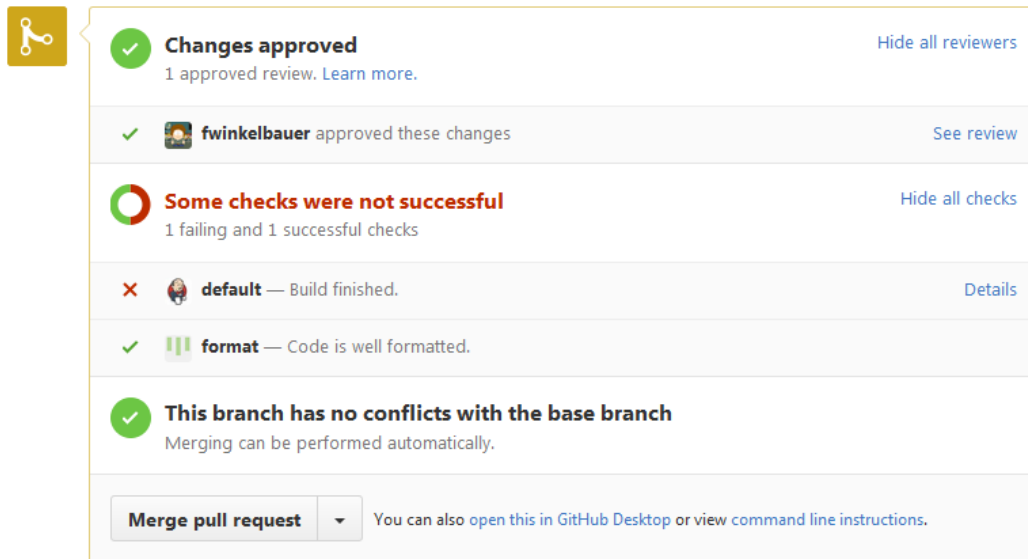


Figure 4.2.: GitHub's Pull Request Merge Information

Figure 4.1 displays an example of a review I did on a pull request created on the 23rd of September 2016. The topmost comment is the overall review summary, which contains all review comments and questions.

Figure 4.2 shows that I have approved the content of this pull request. Notice that a project member with merge rights could potentially merge this pull request even though a static check failed (indicated by the "Merge pull request" button in the picture). It would have also been possible to merge this pull request even if a reviewer had requested changes. This is possible because GitHub's branch protection is disabled which would block the merge operation of this particular pull request.

Importance of Static Checks

Chapter 2.2.2 previously mentioned that two static checks (building the code and code formatting) are executed for each Pocket Code pull request. These checks operate on the merge result instead of the code in the pull request itself. Cases exist in which a build passes for code in a particular

4. Code Reviews

```
1 // -----  
2 // Content Of Branch 'Master'  
3 // -----  
4 public class Person {  
5     private String firstName;  
6     private String lastName;  
7     public Person(String firstName, String lastName) {  
8         this.firstName = firstName;  
9         this.lastName = lastName;  
10    }  
11    public void sayFullName() {  
12        System.out.println("Hello, my name is " + firstName + " " + lastname + "!");  
13    }  
14 }
```

Figure 4.3.: Common Code Base: Content Of Branch 'Master'

pull request, while the resulting code (after a merge operation) cannot be built. The following example shows such a scenario (see Figures 4.3-4.6):

- The master branch contains a small code snippet as seen in Figure 4.3
- Figure 4.4 shows a code change which is part of the first pull request (PR #1). In it two class variables have been combined into a single one. PR #1 has not been merged yet
- Another code change is introduced in Figure 4.5 (PR #2). It adds a completely new method using a class variable. Notice that the change in PR #1 is not included in this code snippet
- The final result (Figure 4.6) is created by merging PR #1 and PR #2 in any order. We have now created a code snippet which contains an error, even though each pull request on its own was fine: the variable "firstName" used in method "sayFullName()" has not been declared, which results in a build error

This scenario highlights how important static checks are, as performing checks by hand may lead to a false sense of security.

4.5. Practical Application

```
1 // -----  
2 // Content Of Pull Request #1 To 'Master'  
3 // -----  
4 public class Person {  
5     // combining two variables into one  
6     private String fullName;  
7     public Person(String firstName, String lastName) {  
8         fullName = firstName + " " + lastName;  
9     }  
10    public void sayFullName() {  
11        System.out.println("Hello, my name is " + fullName + "!");  
12    }  
13 }
```

Figure 4.4.: First Pull Request To Branch 'Master'

```
1 // -----  
2 // Content Of Pull Request #2 To 'Master'  
3 // -----  
4 public class Person {  
5     // changes from Pull Request #1 are not applied  
6     private String firstName;  
7     private String lastName;  
8     public Person(String firstName, String lastName) {  
9         this.firstName = firstName;  
10        this.lastName = lastName;  
11    }  
12    public void sayFullName() {  
13        System.out.println("Hello, my name is " + firstName + " " + lastname + "!");  
14    }  
15  
16    // Introducing new method  
17    public void sayFirstName() {  
18        System.out.println("Hello, my name is " + firstName);  
19    }  
20 }
```

Figure 4.5.: Second Pull Request To Branch 'Master'

4. Code Reviews

```
1 // -----
2 // Result After Merging Both Pull Requests Into 'Master'
3 // -----
4 public class Person {
5     private String fullName;
6     public Person(String firstName, String lastName) {
7         fullName = firstName + " " + lastName;
8     }
9     public void sayFullName() {
10        System.out.println("Hello, my name is " + fullName + "!");
11    }
12
13    public void sayFirstName() {
14        // Result after merging Pull Request #2:
15        // variable 'firstName' does not exist - build error!
16        System.out.println("Hello, my name is " + firstName);
17    }
18 }
```

Figure 4.6.: Code After Merging Both Pull Requests

Evaluation

The following list displays advantages and disadvantages concerning introducing branch protection in Pocket Code:

- Advantages
 - No pull request could be merged without a review
 - Clear documentation of who has reviewed which code
 - Change requests done by a reviewer prevent premature merge operations. Not only does the pull request author have to take action by changing the submitted code, the reviewer has to approve these new changes as well
 - Static checks cannot be bypassed any longer. This ensures that a merge operation can be performed without side effects
 - The author of a pull request gains a better understanding about the current status of his or her pull request. He or she also gains more direct feedback on when additional action has to be taken (e.g. fix static checks or deal with code changes requested by a reviewer)

- Disadvantages
 - Project members could cheat the system by creating an empty review to approve of a pull request

Concerning the “cheat the system” argument: Even if such a scenario would happen, there would be a clear documentation of what happened, which means that the long term effect could be less severe as it would seem at first.

I believe that enabling GitHub’s branch protection for Pocket Code could potentially be an additional nudge for all members. Code reviews are a team effort and they should be taken seriously.

4.6. Results

Section 4.5.3 discussed that the checklist was not incorporated in the pull request review process. This situation could be linked to the overall lack of review participants mentioned in Section 4.5.2. Team members who have already reviewed several pull requests in the past are most likely depending on their own experience and are as a result not dependent on a checklist. This assumption coincides with the findings of Hatton, 2008 (see Section 4.3).

The branch protection feature on GitHub was activated on the 17th of November 2016. It is currently unclear if this step will change the ongoing lack of participants in reviewing pull requests discussed in Section 4.5.2.

Pocket Code’s stakeholders have interests and goals which could be a risk for the continuous use of the branch protection feature: Requiring code reviews for each pull request potentially delays the integration of new features, which increases the probability of having a conflict between product quality and deadline adherence.

Analysing the age of unprocessed pull requests also gave an interesting insight. On the 20th of November 2016 23 pull requests were still open:

- 11 out of these 23 pull requests were opened less than a week ago

4. Code Reviews

- three were about two weeks old
- The remaining 9 pull requests were older than one month

This data highlights once again the bottleneck created by the absence of additional code reviewers. Three pull requests contained either comments regarding requested changes by a reviewer or errors in the static check system. The combination of these findings could indicate that the authors of pull requests no longer feel responsible for any pull request related task after they have created their pull request.

5. Automated Testing

Performing automated software tests is a central part of Pocket Code's development cycle (see Chapter 2.1.2). This chapter gives an overview on Pocket Code's current test strategy, identifies issues in its test suite and presents a series of articles about how to successfully write testable code, which were provided to the Catrobat team.

5.1. Current Situation

5.1.1. Jenkins Troubleshooting

Even though automated tests can be executed on a local computer using a real phone or an Android emulator, these test results might not be meaningful. Different smart phones, emulator versions and settings or even different Android operating system versions might yield different test results. A central build server (e.g. using Jenkins) serves as a reference, which means that its build and test result are more relevant than any individual build or test result on another computer.

Pocket Code has a dedicated Jenkins team, which struggled with a legacy build server setup for more than half a year. Due to this, information about Pocket Code's test quality was missing. To address this problem I was partly involved in a separate group of Pocket Code members who experimented with a completely new Jenkins setup in the spring semester of 2016. Instead of following prior attempts to repair the current setup, a new system using a Debian Linux distribution, an Android emulator and a minimal Jenkins installation with a set of plugins was used. This new instance seemed promising as its configuration was much slimmer than all previous systems.

5. Automated Testing

Building the Pocket Code application through a job on the new Jenkins instance revealed that several tests were broken, which might have been a result of not having a build server over the course of more than half a year. Successive job runs also revealed that some tests lead to inconclusive results, which means that they would either pass or fail at random without any change in-between these builds. Such flakiness is troubling as these tests cannot be used to form reliable assumptions about the code base.

This led the Pocket Code team to two decisions:

- Create JIRA issues to address the failing tests
- Set up a build job which gets triggered by creating a GitHub pull request which only performs static code checks without any tests

Having a build server which does not execute any tests may create the false assumption of having a stable code base, which was a situation which had to be changed as fast as possible.

Over time each broken test was fixed, which made it possible to trigger build and test runs through GitHub pull requests on Jenkins. Remaining unreliable tests were further addressed by automatically re-executing them once in case of an error.

5.1.2. Broken User Interface Tests

On the 23rd of March 2016 a JIRA ticket with the name "Fixing UI Tests" was created. After merging several new pull requests approximately 80 User Interface (UI) tests started to fail. A quick investigation revealed that one of the leading problems concerning these failures was outdated test code: UI changes were introduced while existing UI test code was not adapted to these changes, resulting in the given situation of failing test cases. Possible explanations for how this scenario occurred might be:

- The Jenkins job (mentioned in the last section) which is triggered on each pull request does not execute any UI test, delaying the feedback on these tests
- Several other issues with the UI test suite on Jenkins were hiding failing tests. This issue could be linked to the infrequent UI test execution

Instead of fixing these tests the development team decided to perform a total rewrite of the UI tests. Because of different priorities, this task was delayed for several months in which the UI tests could not be used. This topic is further discussed at the end of this chapter.

5.2. Problem Description

The analysis of the current situation of Pocket Code's test suite lead to the following key questions:

- Why do some tests show flaky behaviour? This refers to the fact that tests either pass or fail without any change in-between test runs
- How many unit and UI tests does Pocket Code have? Is there a recommended or preferred composition? How does this effect the overall test suite?
- How much time is needed to execute each group of tests?

5.2.1. Flaky Tests

Analysing Jenkins' log files of prior builds revealed that the flaky tests mainly failed due to external dependencies. One of these examples is a dependency on a memory card which can produce an error stating that the card is busy even if an Android emulator is used. Further investigation revealed that this dependency cannot be avoided as it is hard wired in several components in the Pocket Code source code. Refactoring the code so that a test double could be introduced instead of the real memory card is a challenging and very time consuming task.

5.2.2. Testing Pyramid

The testing pyramid (see Figure 5.1) is a concept which recommends how a testing suite should be composed (Google, 2015). It distinguishes three types of tests:

5. Automated Testing

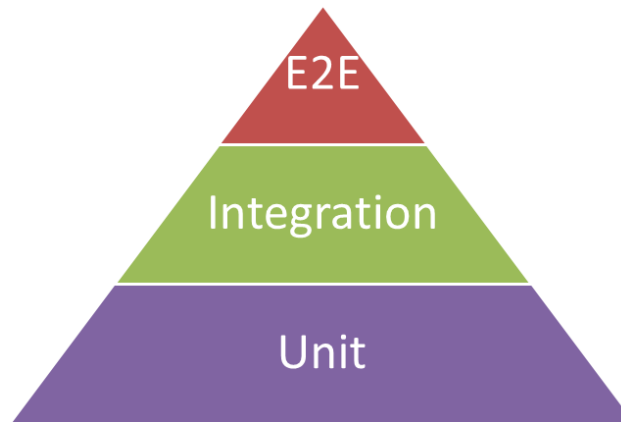


Figure 5.1.: General testing pyramid.

- Unit tests: they test a small piece of code in isolation
- Integration tests: they test that a small group of units work together as expected
- End-to-End Tests (E2E): they test a real user scenario (e.g. a user logs into a piece of software using a login form)

E2E tests have a few disadvantages, including:

- Long test runs
- Can have unreliable behavior, meaning that a failing test case could be a false positive error (false alarm)
- No failure isolation

Because of these issues the amount of end-to-end tests should be kept as small as possible. To rely less on these tests while also having meaningful test results for most of a code base, it is recommended that the major portion of a test suite should consist of unit and integration tests. This preference is based on the fact that these tests provide faster and more reliable feedback (Google, 2015).

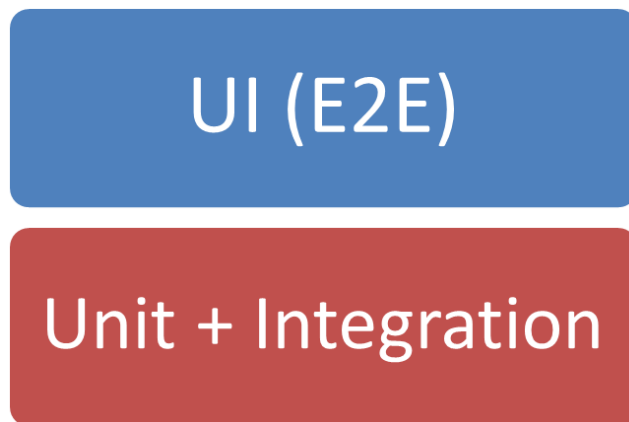


Figure 5.2.: Actual test composition of Pocket Code.

Table 5.1.: Test statistics

Category	Class Count	Test Count	Execution Time (Single emulator)
Unit Test	156 (51%)	835 (52%)	less than two minutes
UI Test	147 (49%)	780 (48%)	several hours
	303 (100%)	1615 (100%)	

5.2.3. Test Statistics

Using `grep`, a Linux command line tool which can be used to search for text in files, I was able to create statistics of Pocket Code's test base (Master branch, 27th September 2016). Table 5.1 shows an overview of my results.

The distribution between unit and UI tests (a form of E2E tests) in Pocket Code is about even while both groups contain more than 700 test cases (see Figure 5.2). Having such a deviation from the previously mentioned testing pyramid distribution has a significant impact on the overall execution time: While all unit tests can be executed in a few minutes, the UI tests (executed either on a smart phone or an Android emulator) take several hours.

A closer look on UI test case distribution also revealed an interesting insight: Table 5.2 shows that three out of all 15 packages contain more than 70 percent of all UI test cases (packages: `ui/fragment`, `ui/activity`, `content/brick`).

5. Automated Testing

Table 5.2.: UI Test Package Distribution

Package	Test Classes	Test Cases	Production Classes
ui/fragment	8	254 (33%)	23
ui/activity	11	161 (21%)	26
content/brick	77	139 (18%)	116
formulaeditor	6	63 (8%)	28
content/interaction	7	39 (5%)	0
ui/dialog	11	38 (5%)	43
web	2	31 (4%)	3
stage	11	26 (3%)	11
drone	4	11 (1%)	6
devices/mindstorms	3	7 (1%)	28
facetedetection	3	6 (1%)	4
bluetooth	1	2 (0%)	11
devices/arduino	1	1 (0%)	3
devices/phiro	1	1 (0%)	3
ui/menu	1	1 (0%)	1
	147	780 (100%)	306

If we compare the test class and the production class count for each package we can see that the production class count outweighs the test class count in 12 out of 15 packages.

The production classes count for the package "content/interaction" is zero in Table 5.2 because it does not exist in the production environment.

These findings led me to the following assumptions:

- Some UI code seems to be more important or at least easier to test than other code parts
- The high test count in the previously mentioned top three packages could be an indicator that these classes have a lot of content and responsibilities

5.3. Related Work

Kochhar et al., [2015](#) studied the automated test culture among open source Android projects. Analysing over 600 projects showed that most of the studied applications are poorly tested: 14% of the studied applications contained test cases, while only 9% contained enough test cases to test about 40% of the production code. Even though the Android environment has access to many different automated testing tools (e.g. JUnit or Robotium), developers tend to rely more on manual testing. In an additional survey developers mentioned the challenges they face with automated testing. The top four challenges include:

- Time: Developers would rather spend time developing instead of testing an app in order to push their product on the market before a competitor offers an alternative solution
- Compatibility: Some automated testing tools are not generic enough or rely on specific technology which makes developers fall back to manual testing
- Awareness: Some developers were simply not aware of automated testing tools or believed that this practice is not common
- Usability: Testing tools can be unintuitive or complex to use

In their paper Vasilescu et al., [2015](#) analysed software projects using historical data to find how automation services such as Jenkins influence the quality and productivity of handling pull requests on GitHub. Their findings mention that:

- Relying on services such as Jenkins and having a large set of automated tests allows teams to process (merge or reject) pull requests quicker than without an automated environment. This result indicates that the accelerated feedback in terms of building and testing has a positive effect on the decision making process concerning accepting or declining pull requests
- An automated environment as described above enables core members to discover more bugs in pull requests

Yang, Tempero, and Melton, [2008](#) discussed the positive effects on quality of Dependency Injection (DI) mentioned in the literature. DI is a concept in

5. Automated Testing

which a class receives all collaborating dependencies it needs as an input (see Appendix G for an example). This concept helps to increase:

- extensibility as a developer can decide on the implementation of the injected collaborator each time it uses a class
- testability by injecting specific implementations of collaborators which can be used to control and observe a class under test. (This process is known as mocking a collaborator)
- reusability by improving flexibility and breaking dependencies

Further Yang, Tempero, and Melton, 2008 examined the usage of DI in over 30 open source Java applications by searching for the following types of DI:

- Constructor No Default: Passing a collaborator via the constructor
- Constructor With Default: Same as above but with a provided default dependency
- Method No Default: Passing a collaborator as a method parameter
- Method With Default: Same as above but with a provided default dependency

The study suggests no widespread usage of DI among the tested applications. Yang, Tempero, and Melton, 2008 conclude that the most likely explanation for this finding is that DI is simply not taught in software design courses and as a result DI is not known as a common design principle.

Wick, Stevenson, and Wagner, 2005 added JUnit as a topic to their curriculum at the University of Wisconsin-Eau Claire which helped students to appreciate the value of automated testing in software development. The programming courses introduced several guidelines to how students should apply testing:

- Create a test class for each production class
- Introduce mocks for each collaborator in the test class
- Derive a class from the test class for each method that should be tested. This rule should help to reduce complexity but it also increases the amount of test classes significantly

5.4. Creating Learning Material

- Add test cases for each possible scenario of how a method is used in the derived test class
- Create a test suite for each test class and its derived test classes. The test suite acts as a container for all test cases which belong to a single production class

Feathers, 2010 gave an insightful presentation about testing at the Norwegian Developers Conference (NDC). He states that the problems software developers might face with writing test code arise from underlying software design problems in their code base. Feathers' main statements include:

- Good design makes it easy to write tests, but
- Making code testable might not lead to good design
- In this context good design is defined through several well known design principles such as:
 - Open/Closed Principle: A class should be open for extension, but closed for change
 - Single Responsibility Principle: A class should care about a single functionality which further means that it should only have a single reason to be changed

5.4. Creating Learning Material

The results from Section 5.2 and the findings of Feathers, 2010 suggest that the root cause of Pocket Code's test situation is linked to underlying design problems in the code base. Redesigning or refactoring the code base including the test cases is a task which cannot be achieved by one person for several reasons:

- There are over 1.600 tests
- Most of these tests can only be changed if the code under test is changed as well
- A single large refactoring would not qualify as a long term solution, as it does not change the behavior of other team members

5. Automated Testing

Based on these findings I have decided to educate the Pocket Code team about writing testable code by providing online learning material. This material should emphasize two key concepts:

- Improving test execution time by relying less on E2E tests
- Creating a more robust test suite by favoring unit tests over integration and E2E tests through good design

5.5. Practical Application

5.5.1. Creating a Testing Article Series

I have created a series of articles about writing testable code which were published as blog posts on Confluence. The first article was uploaded in the beginning of July 2016, with a following article published about every week. Each of the nine articles covered an aspect of testing principles or strategies in around 1000 words. I have chosen this format instead of a book-like format to slowly build up knowledge, while also leaving enough time in between articles so that members could perform their own research. For this purpose these articles contained resource sections with additional reading material. The two most influential resources which were used to create this article series were Hevery, [2008](#) and Rainsberger, [2013](#), who both talked extensively about common design problems in testability.

The following list gives a few examples of topics that were covered by the article series:

- Improve software design
 - How to use DI. This makes it possible to test classes in isolation using test doubles instead of using concrete class collaborators
 - All problems which arise by using some form of global state
 - How to separate external dependencies from one's own code. This topic builds on the principles of DI
- Maintaining test code
 - Structure of a unit test

- How to use creational design patterns to handle objects in test cases

You can find the complete article collection in [Appendix G](#).

5.5.2. Roadmap: Refactoring Pocket Code's Test Suite

The testing articles did not only consist of explanations and code examples, they also contained "Getting Practical" sections, which were used as a step by step description on how the presented information could be practically applied. This was done by analysing the current code base while focusing on Pocket Code's test suite problems discussed in [Section 5.2](#).

The following sections give a complete overview of my personal recommendations that could be applied to improve Pocket Code's test suite. All following concepts and terms are explained in more detail in [Appendix G](#). A summary of the following content was also provided to the Pocket Code team through Confluence (see [Appendix H](#)).

Test Structure

Applying these changes would make it easier for a developer to read and maintain test cases:

- Upgrade the JUnit testing framework from version 3 to version 4 to apply its improved Exception testing features
- Restructure test cases to comply with a Create, Act, Assert structure style. Using this principle test cases would feature a consistent structure:
 1. Create the necessary components which are needed in the test (Create)
 2. Perform all steps which should be validated in the test (Act)
 3. Finally, check that the outcome matches the expected result (Assert)

5. Automated Testing

Use Dependency Injection / Reduce Global State

Decreasing the amount of integration tests is only possible after improving Pocket Code's decoupling situation. A big blocker for this step is the heavy use of global state which should be avoided most of the time in the future. This would allow the Pocket Code team to introduce test doubles to make tests smaller, faster and more reliant (less flaky). To do so the Pocket Code team has to:

- Refactor classes that rely on global state
- Introduce DI
- Hide external dependencies (e.g. code that accesses the memory of a smart phone) behind wrapper classes

DI could further be used to address the high amount of UI tests Pocket Code has. This would not only drastically improve the execution time of tests (from hours down to minutes or even seconds), but it would also reduce the maintenance effort caused by failing test cases which can occur while changing UI components. Having a faster test suite also increases the chances that these tests are executed at a regular basis, speeding up the feedback process.

Focus on Relevant Details

Leaving out irrelevant details makes tests shorter and easier to read, which in turn makes them more maintainable. To achieve this the Pocket Code team could:

- Use a mocking framework to create test doubles
- Refactor test code to use test doubles through DI
- Apply creational design patterns to simplify object creation in tests:
 - Object Mother: A class which can be used to build objects while only focusing on details which are relevant for a particular test case
 - Test Data Factory: A class which can be used to create predefined objects

Where to Start

Of all the three mentioned topics, using DI (this includes reducing global state) is the most important, but also most time consuming task. Applying any other change to the test suite prior to this task could lead to extra work. Changing the test structure and introducing a new JUnit version is the least significant task as it improves readability just slightly. Because of this I believe that the order of implementing the roadmap topics should be:

1. Use Dependency Injection / Reduce Global State
2. Focus On Relevant Details
3. Test Structure

Effort Estimation

I estimate that implementing the above changes could take several weeks or even months. The reasoning for this estimation is as follows:

- The recommended task "Use Dependency Injection / Reduce Global State" is rather complex and time consuming. Reducing the dependency on global state is difficult as it can potentially be used anywhere in the code. Changing something on one side may have a none-obvious effect on some other part in the code
- Several parts of Pocket Code are not tested, which makes it more difficult to verify that introduced changes do not break existing functionality
- Team members do not have a regular work schedule:
 - Most team members are students, which means that they have additional responsibilities such as attending classes, learning for exams or working on assignments
 - Some members have a part-time or even a full-time job
 - Most members are not very active in the holiday time
- Section 1.1 mentioned that the most experienced students are very likely to leave the project soon. Less experienced students would need more time to familiarize themselves with the code base before they can start to contribute

5. Automated Testing

- All major changes need to be communicated so that the team can learn from past mistakes
- Refactoring and redesigning large parts of the code base has an effect on the usual development cycle (implementing features and fixing bugs). The team has to prioritize activities in order to effectively use its time

5.6. Results

I have received positive feedback concerning the article series about how to write testable code, although I suspect that most project members have not taken the time to read the material. This assumption is based on my constant effort to inspect the code of new pull requests on GitHub in which I could not find any major change in how code is written. On the 18th of November the article series was declared as a mandatory read for all Catrobat members. This decision could help to spread knowledge about how to write testable code.

In the Pocket Code team meeting on the 9th of November 2016 a complete rewrite of the UI tests was announced. Up to this announcement several UI tests were still not functional, though there has been great effort to refactor the UI code to remove duplicated code. The UI test rewrite is going to focus on a few key points:

- Replacing the UI testing framework Robotium with Espresso as parts of Robotium have been declared as deprecated. This change would also lead to an upgrade from JUnit version 3 to version 4
- Removing redundant UI test cases or replacing them with non-UI test cases if possible
- Integrating Espresso with Gradle to use it on Jenkins

My suggestions concerning how to refactor and redesign the test suite were considered by the Pocket Code team, but they have decided to only focus on rewriting the test code instead. Focusing on rewriting and changing the UI test code will most likely improve the overall test suite, but it might not have a significant impact on the core problems of Pocket Code's test suite:

5.6. Results

- Writing test cases without operations such as accessing a memory card or using the UI cannot be done with the current state of the code base, suggesting that the amount of integration and E2E tests will grow further
- An increase of the integration and E2E test count could further shift the test suite composition to resemble a reverse pyramid shape (meaning that there might be far more integration and E2E tests than unit tests). This could have a negative impact on the reliability and the runtime of the test suite

Feathers, 2010 (see Section 5.3) mentions that hard-to-test code is an indication for underlying design problems. In addition to that Vasilescu et al., 2015 (see Section 5.3) emphasized that an automated testing environment is a valuable tool to integrate code changes. These statements suggest that without any major change, Pocket Code's test suite could slow down the overall development process of Pocket Code as it becomes harder and harder to integrate and verify new changes.

6. Software Development Workflow

On the surface creating software only revolves around writing source code, though on closer examination we can identify a workflow which consists of three distinguishable steps:

- Feature development: creating new functionality for upcoming software releases. Several developers can work on different features simultaneously
- Creating software releases: delivering a set of new features to a user
- Hotfixing releases: correcting wrong behaviour in a prior software release without introducing any additional features

This chapter covers the following topics:

- Pocket Code's current workflow using git
- Design decisions in the current workflow which have led to additional maintenance work in the past
- Alternatives to address the identified disadvantages
- The process of how these alternatives were implemented

6.1. Current Situation

6.1.1. Working with Git

Git is a powerful source control management system which offers versatile possibilities on how it can be used. Its branching concept can be used to

6. Software Development Workflow

separate tasks, so that developers can work without interfering with each other. Creating a specific branching strategy builds the groundwork to successfully bundle features into a software release.

6.1.2. Gitflow

The current Pocket Code workflow is based on Gitflow, a branching strategy which was first published in a blog post by Driessen, [2010](#). It describes a model to develop software using git which distinguishes two types of branches:

- Main branches
 - One master branch
 - One develop branch
- Support branches
 - Release branches
 - Hotfix branches
 - Feature branches

Main branches are used to maintain stable software: Past versions which have already been released are saved on the master branch. Newly developed software features (which are going to be included in an upcoming release) are put on the develop branch.

Support branches fulfill specific tasks:

- Release branches are used to prepare new software versions (releases), which are later pushed to the master branch
- Hotfix branches are used to fix bugs in a software release, which leads to a new release
- Feature branches are used for ongoing development. They originate from the develop branch and are used to implement new functionality (new features). These branches are later on merged back into the develop branch.

Feature Development

Figure 6.1 shows a basic overview of how feature development is done using Gitflow. Each feature (#1 and #2) originates from develop and is later on merged back into develop itself. Note that feature #2 can only be integrated after all changes from feature #1 which have already been integrated into develop have been pulled into feature #2 to resolve possible merge conflicts.

Software Releases

Gitflow uses two support branches to prepare a software release. Figure 6.2 gives an example of the release preparation workflow:

A release branch is derived from a particular state of the develop branch. In this example such a branch is used to prepare the first software version which is tagged (1.0) and pushed to the master branch afterwards. Git uses tags to label commits with a custom name so that they stand out from other commits.

If a bug is found in an already released software it can be dealt with by using a hotfix branch. After a bug has been fixed in such a branch, a new commit is created, tagged (1.1) and pushed on the master branch. In addition to that the content of a hotfix is also merged into the develop branch.

Note how all ongoing development in the develop branch does not interfere with past releases in any way. The commit "New feature for 2.0" is not included in any release (1.0 and 1.1).

6.1.3. Pocket Code's Git Workflow

To include Pocket Code's sub-projects (e.g. physics, drone or Musicdroid), a variation of Gitflow was created, which also incorporates a slightly different naming convention. Pocket Code views sub-projects as large features which consist of several functions. To allow an independent development of these sub-projects, each of them lives in its own feature branch until its first

6. Software Development Workflow

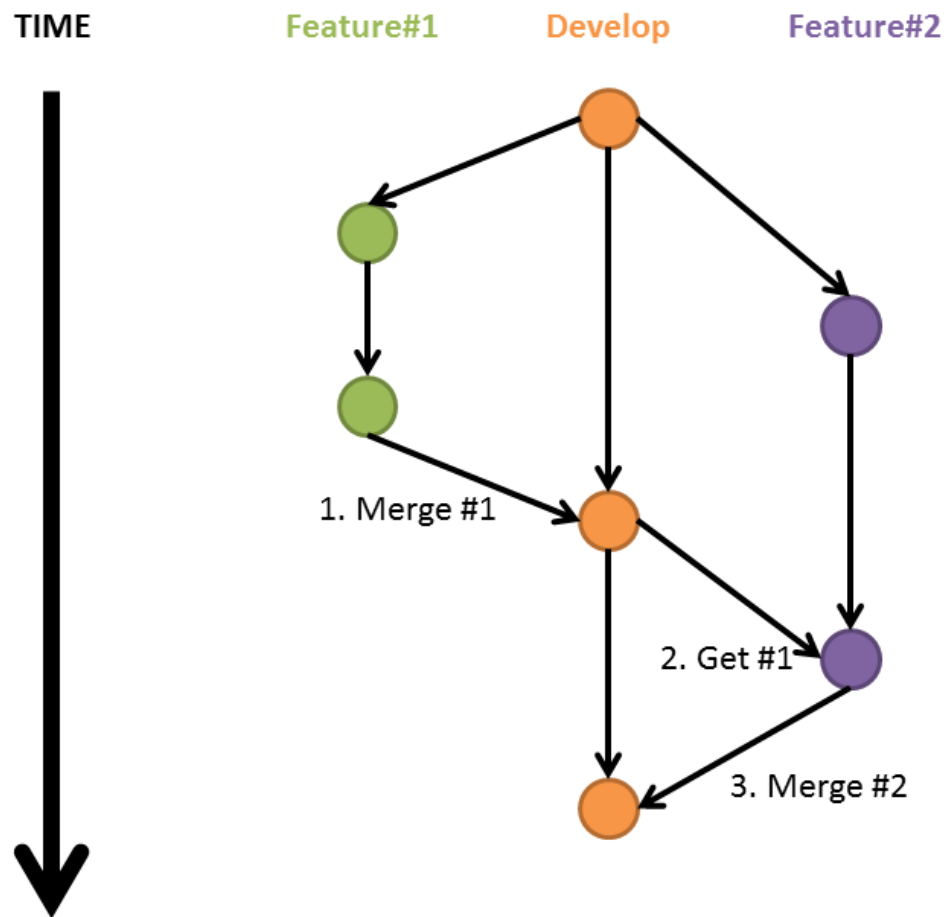


Figure 6.1.: Gitflow's development workflow

6.1. Current Situation

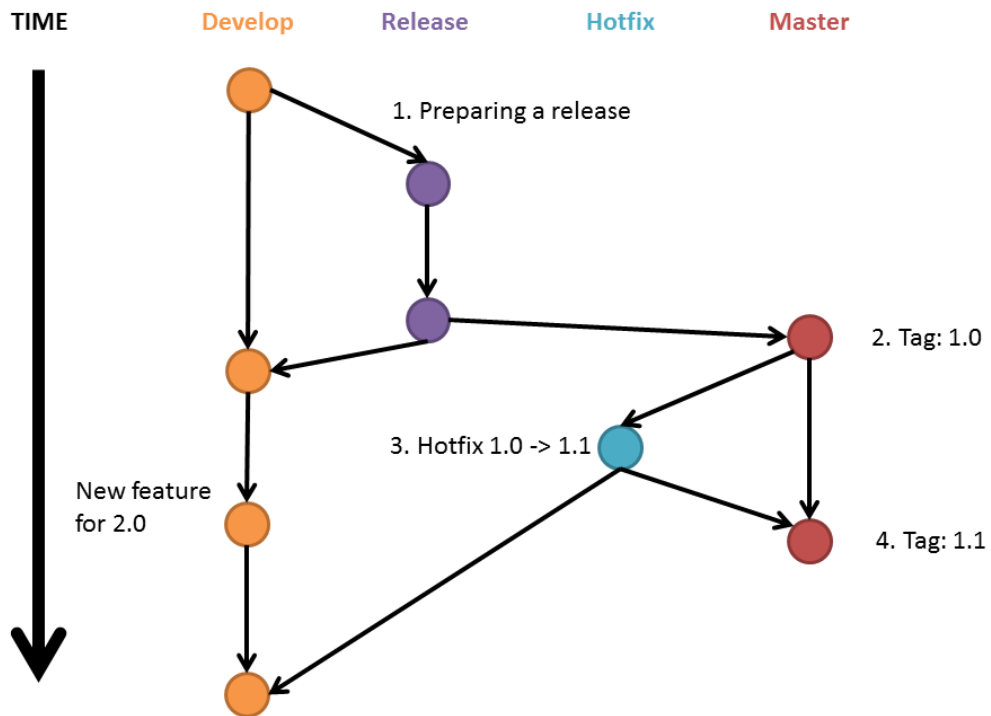


Figure 6.2.: Gitflow uses two helper branches to prepare releases

6. Software Development Workflow

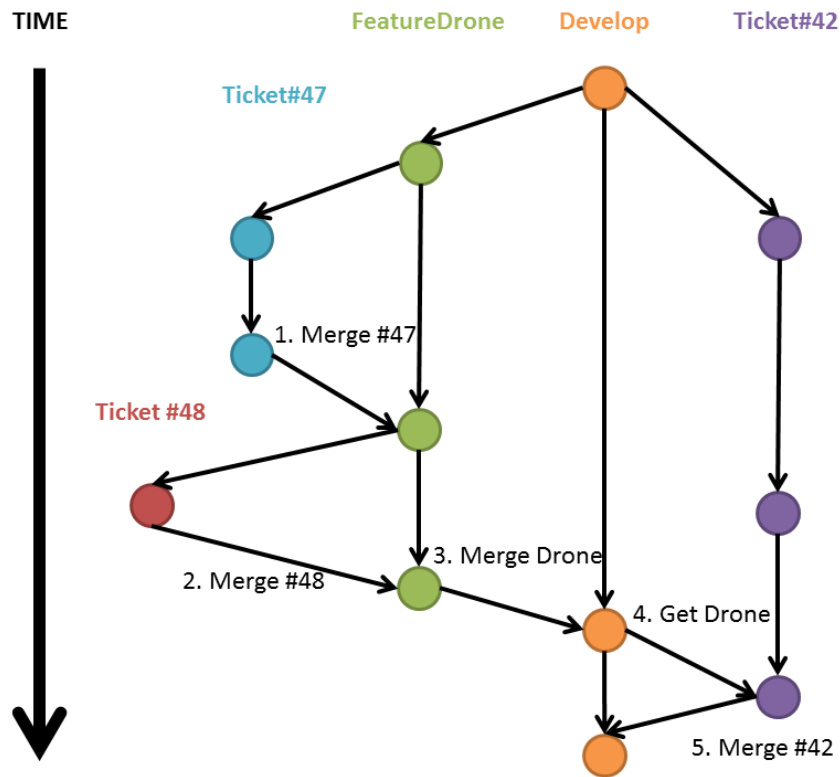


Figure 6.3.: Pocket Code's development workflow

version is finished. This is a rather confusing circumstance, as Gitflow uses the term "feature branch" to address branches which are used to develop a single smaller functionality. Pocket Code members on the other hand do not use a specific term to talk about branches which contain small functionality, which is why I am going to refer to them as ticket branches. I have chosen this name as each of these branches is described by a JIRA ticket.

Figure 6.3 illustrates an example of Pocket Code's workflow. The Drone feature, as well as the JIRA ticket #42 originate from develop. After implementing #47 and #48 the Drone feature is considered to be done and is merged back into develop. To merge #42 into develop the Drone feature first has to be pulled into #42 to resolve possible merge conflicts.

If we compare both workflows we can in summary identify two major

differences:

- Pocket Code's branch hierarchy uses an additional branch layer when working with sub-projects (e.g. Drone) and a different naming convention (sub-projects are called features)
- Pocket Code's feature branches could be compared to temporary develop branches for the first development of a sub-project

6.2. Problem Description

Pocket Code's git workflow has three major drawbacks which increase the overall maintenance effort:

6.2.1. Different Pull Requests Sizes

The combination of using feature and ticket branches means that two pull requests are necessary to pull sub-project code in the develop branch. The size of these two pull requests types differs significantly, as feature branches consist of a collection of several JIRA tickets. Figure 6.4 shows an example using four different pull requests: one for each JIRA ticket (#137, #138, #139) and one containing these tickets.

Feature and ticket branch pull requests pass through the same code review cycle, which means that some code is reviewed twice. On first sight this might suggest an increase in error detection, though on closer examination a common pattern emerges: Reviewing big pull requests becomes exhausting very quickly, which can decrease the applied level of detail. Due to their size, subtle code changes may pass unnoticed more easily, which could introduce bugs (e.g. having wrong assumptions about the code base while trying to resolve a merge conflict can lead to unexpected behaviour, see Figure 6.4). Several studies suggest that the size of a pull request has a negative impact on the code review process (see Related Work, Section 6.3).

6. Software Development Workflow

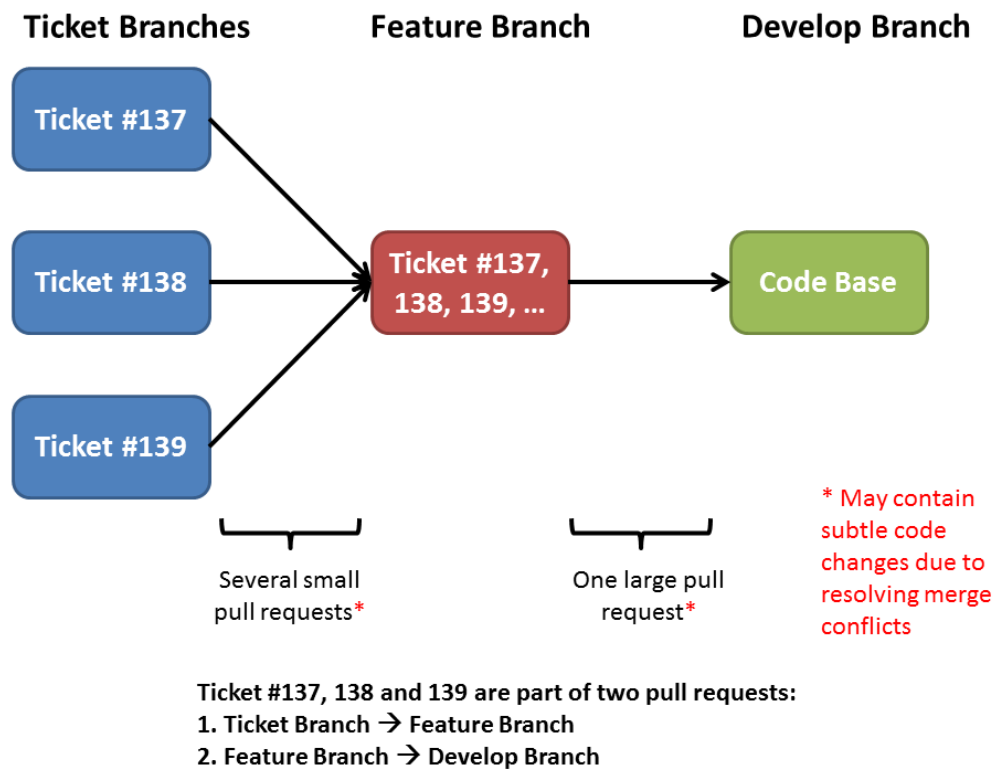


Figure 6.4.: Merging feature branches (sub-projects)

6.2.2. Feature Branch Lifetime

Pocket Code's feature branches may have a lifetime of several months or even years. Merging such branches becomes more and more complex if several extensive changes have been performed on the develop branch. As a result, keeping up with changes in this branch becomes tedious, as each change in develop has to be merged into every feature branch to avoid even more complicated merge operations later on.

6.2.3. Maintaining Two Central Branches

Using a set consisting of a develop and a master branch offers no additional information or serious benefit than having a single-branch approach. In fact it can even lead to issues if the Gitflow instructions for merging different branches into develop and master are not followed strictly. Pocket Code had for example cases in which integrating a hotfix in an incorrect way lead to having inconsistencies between both branches. Due to this no pull request could be merged prior to resolving this inconsistency.

6.3. Related Work

In their paper G. Gousios et al., [2015](#) focus on the integrator's (a person who processes pull requests) view in a pull based development environment. A survey of over 700 integrators has shown amongst other things:

- General work practices
 - 80% of the interviewed integrators use the pull based development model to perform code reviews and to resolve issues
 - About 50% use pull requests to discuss new features
- Accepting or declining contributions

6. Software Development Workflow

- Code quality is the most important factor concerning accepting or declining contributions. Topics include: understandable and elegant code, good documentation and adding value with minimal impact
- Integrators also consider how the submitted code fits into the overall goals and targets of the project
- Trust in the submitted code is enforced through automated and manual testing. A contribution that includes testing code (which offers good coverage of the proposed changes) is considered as a positive signal by integrators. Automated testing is performed by services such as Travis CI or CloudBees (which are similar to Jenkins)
- Challenges
 - Maintaining quality while including external contributions is considered a serious challenge. This is also why reviewing and testing processes are put in place
 - Rejecting a contribution is tough because it is hard to motivate contributors to keep on working even though their submissions are declined
 - Integrators struggle to keep up with the increasing volume of incoming contributions in large projects. In a follow-up study Veen, G. Gousios, and Zaidman, [2015](#) presented a tool to automatically prioritize pull requests as a first step to improve this situation

Georgios Gousios, Storey, and Alberto Bacchelli, [2016](#) performed another study in which they focus on the contributor's perspective. Using a survey with over 600 top contributors to active open source projects they gave insight into the work practice and challenges of contributors. The following list shows a few of the mentioned challenges:

- The time until a contributor gets feedback by an integrator was pointed out as a hard issue. Contributors report that it is extremely frustrating to create a pull request which is not getting processed by an integrator. Contributors also mention that they would rather like to receive a rejection on their pull request instead of not getting any feedback at all

6.4. Proposed Workflow Changes

- Even though contributors would prefer negative feedback over no feedback, the fear of rejection is still present
- It is hard to understand the code base and its structure
- Another challenge contributors face is estimating the impact of a contribution: how do new changes effect existing code? This issue goes hand in hand with the previous point about understanding the code base
- Survey participants also reported issues regarding using git and resolving branch conflicts (merging)

Phillips, Sillito, and Walker, [2011](#) studied branching and merging practices across different version control systems. An online survey which included 140 version control users gave insight into the development of several software projects. The following listing shows parts of the study results:

- Merge problems are common in source control systems. Branching and merging strategies are used to reduce the complexity and frequency of conflicts
- There exist two common branching strategies
 - Flat branch layouts (layouts in which every branch is directly connected to the main branch) allow easier merge operations, although this layout comes with a higher risk of destabilizing the project when errors are introduced
 - Branch layouts with a hierarchy of branches offer a separation between responsibilities at the cost of having more effort to propagate changes between branches

6.4. Proposed Workflow Changes

The previously mentioned issues with Pocket Code's workflow are problematic because they:

- Make reviewing and merging feature branch pull requests exhausting and error prone
- Introduce constant effort to maintain long living branches

6. Software Development Workflow

```
1  boolean useNewImplementation = false;
2  // useNewImplementation = true; // uncomment this if you develop the new algorithm!
3
4  if (useNewImplementation) {
5      newImplementation(arguments);
6  } else {
7      oldImplementation(arguments);
8  }
```

Figure 6.5.: A very basic feature toggle in Java

- Increase the workflow's complexity for releasing software

To reduce merge and maintenance work created by these drawbacks I propose two workflow changes:

- Replace long living feature branches with feature toggles
- Change the current two branch approach (master and develop branch) to a single branch approach to reduce merge problems between them

6.4.1. Using Feature Toggles

Fowler, 2010 gave an extensive insight into the concept of feature toggles which use boolean conditions to activate or deactivate certain parts of a software.

Figure 6.5 illustrates a very basic example of a toggle. Instead of having separate branches to develop new implementations of a single feature, a boolean value is used to switch between the old and the new implementation. Feature toggles should be deleted after a feature is fully developed or after a certain testing period has passed so that the amount of boolean flags does not become overwhelming. Comparing feature toggles to Pocket Code's previous feature branch workflow leads to the following list of advantages and disadvantages:

- Advantages

6.4. Proposed Workflow Changes

- Fewer merge issues as a result of not having to deal with feature branches
- Being able to activate or deactivate features through flipping boolean switches
- Only a single pull request is needed to merge a finished JIRA ticket into the develop branch
- Disadvantages
 - Developers have to gain knowledge about how feature toggles should be implemented and used
 - Having several feature toggles may become increasingly hard to maintain

Section 4.5.2 described an approach to address large pull requests which result from closing feature branches. This approach is based on milestones which should be used to split the integration process into several smaller sized pull requests.

On first sight a milestone approach may sound like an alternative to feature toggles, but on closer examination we can see, that such an approach cannot be implemented without toggles: Integrating parts of a feature through a milestone may expose unfinished functionality to the user, creating the necessity to turn these functions off completely. This means that implementing a workflow which relies on feature branches and milestones would also include feature toggles, leading to a mixed approach. Because of these observations I am leaning towards switching the current sub-project workflow to an alternative one which relies only on toggles.

6.4.2. Handling Master and Develop Branch

There exist two alternatives to solve the previously mentioned problems concerning the set of a master and a develop branch:

- Introduce an extension tool to streamline Git operations in the Gitflow workflow
- Fall back on a single branch approach

6. Software Development Workflow

Gitflow Extension

The author of Gitflow offers an extension tool for git which provides additional command line features including the ability to handle hotfixes with a single command (Driessen, 2012). Using this tool comes with these advantages and disadvantages:

- Advantages
 - It offers a very simple solution which does not change the overall workflow, reducing some organisational burden
 - It simplifies the Gitflow workflow through additional command line functions
- Disadvantages
 - The Gitflow extension tool is only usable through a command line window, which means that there is no additional support through the GitHub web page which is the primary way in which all of Pocket Code branches are managed
 - The tool has to be installed on every computer on which it should be used

In my opinion using the Gitflow extension tool would overall only provide a quick fix without changing the underlying problems that occur because of having a master and a develop branch.

Using A Single Branch

We can simplify Gitflow's software release approach by changing the workflow to only use a single branch instead of a combination of master and develop. Figure 6.6 illustrates how the prior Gitflow release example (Figure 6.2) could be solved by using a single branch workflow: The overall information (tag 1.0 and 1.1) is unchanged even though the commit graph looks different. Appendix I goes into more technical detail on why this can be done.

Using a single branch instead of two includes these advantages and disadvantages:

6.5. Practical Application

- Advantages
 - It offers the same information as a two branch solution
 - Removing branch redundancy makes it easier for a developer to prepare a release or integrate a hotfix as he or she only has to maintain one instead of two branches
 - It flattens the learning curve for applying the current git workflow. Learning about git can already be overwhelming for new developers, which is why the Pocket Code team should aim to keep its git workflow as simple as possible so that new members can start to work with it as quickly as possible
- Disadvantages
 - Even though this change leads to simplifications, team members have to be educated about the change so that they do not get confused by the absence of a familiar branch
 - Currently every commit to the master branch could be identified as a new release, without the need to do any tag lookup. In a single branch approach this tagging information is needed to find releases in the commit graph. Due to git's toolset this can still be done easily, though it is a little less convenient than a two branch solution. Figure 6.7 compares both approaches side by side

Overall I believe that a single branch approach offers the most benefits in the long term. The initial cost of changing the workflow is outweighed by the simplification that occurs. Integration problems between the master and the develop branch simply cannot occur in this approach as only one branch is utilized.

6.5. Practical Application

This section covers how the proposed workflow changes were applied in Pocket Code.

6. Software Development Workflow

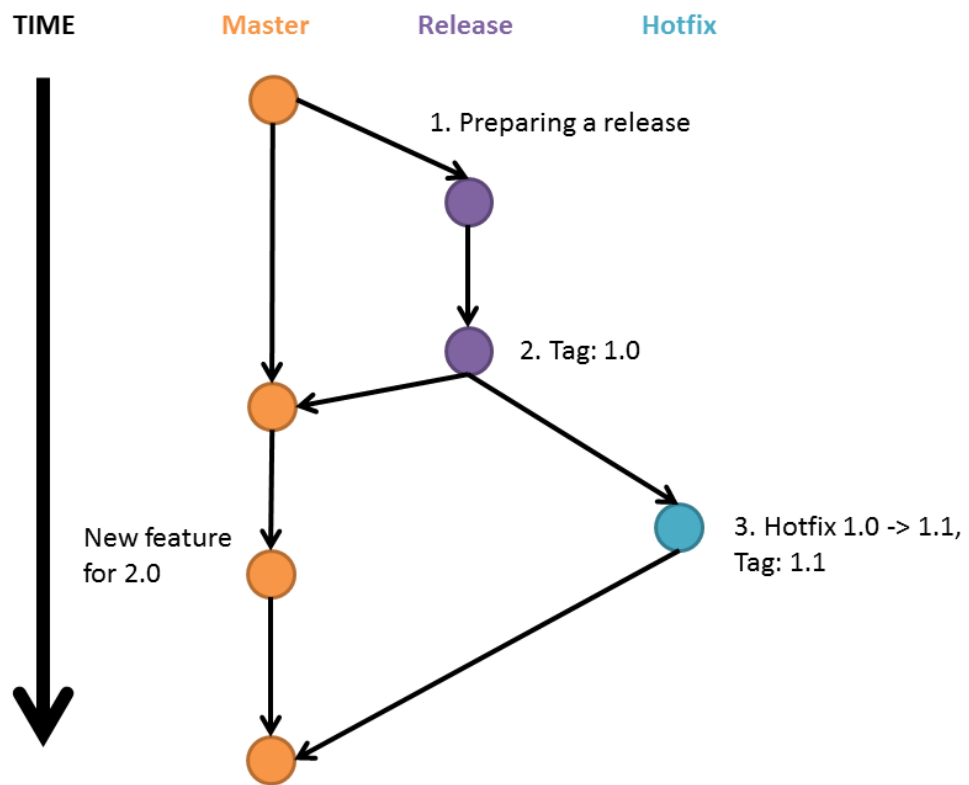


Figure 6.6.: Alternative concept which provides the same information as Gitflow's release workflow

6.5. Practical Application

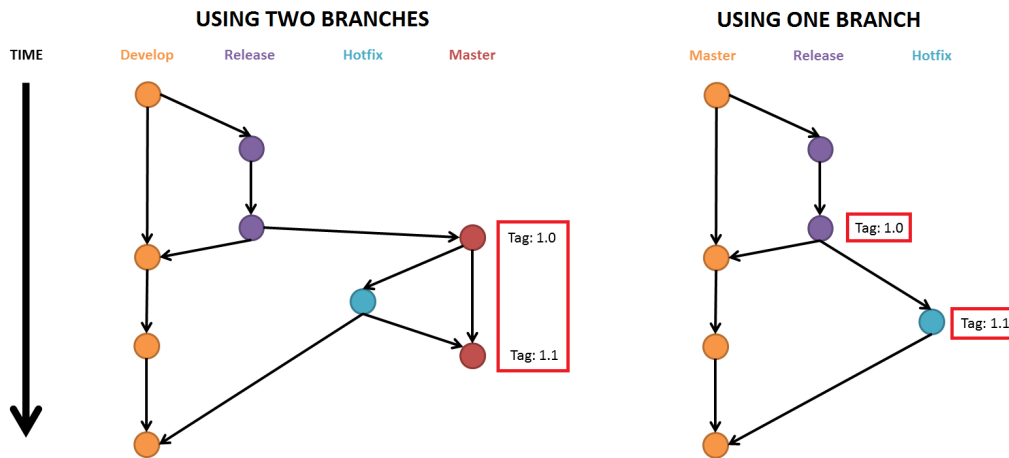


Figure 6.7.: Comparing release information between branch approaches

6.5.1. Removing Feature Branches

Musicdroid

Musicdroid is a sub-project of Pocket Code which allows users to create custom music. It is the latest project which got integrated into Pocket Code, which means that its feature branch is the youngest one. As all other sub-projects are getting merged one after another, only Musicdroid remains as a constant feature branch. This made it a perfect candidate to implement feature toggles. This was done in three steps:

1. Educate the team about how toggles are used
2. Add a Musicdroid feature toggle
3. Merge Musicdroid's feature branch into Pocket Code's develop branch

Educating all members of Musicdroid was done during one of their regular meetings in which the workflow changes as well as its implementations were discussed in detail. I have also added a how-to article on Confluence about feature toggles (see Appendix F).

Creating a feature toggle was done by adding a constant to Pocket Code's Gradle configuration file (build.gradle). Through a Gradle plugin these

6. Software Development Workflow

```
1  android {
2      // ...
3
4      buildTypes {
5          debug {
6              // ...
7              buildConfigField "boolean", "FEATURE_<DESCRIPTION>_ENABLED", "true"
8          }
9          release {
10             // ...
11             buildConfigField "boolean", "FEATURE_<DESCRIPTION>_ENABLED", "false"
12         }
13     }
14 }
```

Figure 6.8.: Adding a feature toggle to build.gradle

```
1  import org.catrobat.catroid.BuildConfig;
2
3  // ...
4
5  if (BuildConfig.FEATURE_<DESCRIPTION>_ENABLED)
6  {
7      // ...
8  }
```

Figure 6.9.: Using the previously defined toggle in Java

constant properties can then be accessed through a Java class in the code itself. This approach was chosen because of two reasons:

- Gradle and the plugin were already in use
- Defining feature toggles as static properties in Gradle gives the advantage of having all toggles in a single place, which improves maintainability

Figures 6.8 and 6.9 show an example of how a feature toggle can be added and used in code. Additionally defining these toggles as properties through Gradle allows us to change its state through a run configuration (here: debug and release). In this example the new feature is only activated in a development environment.

6.5. Practical Application

As a final step Musicdroid's feature branch was merged into the develop branch. All upcoming finished tickets involving this sub-project are going to get merged directly into the develop branch.

Chromecast

This sub-project focuses on adding support for Google Chromecast to Pocket Code. Through this feature it is possible to play a game (which was created in Pocket Code) on a TV using the Google Chromecast HDMI dongle. In this scenario the mobile phone can be used as a game controller device.

We added a feature toggle to this project in a similar way as the previously mentioned Musicdroid project. The only difference is, that this feature is enabled by default, which means that it is included in the released Pocket Code application. This means that in case of a major problem this feature can be deactivated completely in a release version through changing a single boolean value in the build.gradle file.

APK Generator

The standalone application download feature of the APK generator is implemented through a custom Jenkins job which uses the APK generator feature branch to build a custom version of Pocket Code which only runs a single Catrobat project without any additional functionality. Because of this Jenkins job there is no intent to merge the APK Generator feature branch into the master branch. As a result constant maintenance has to be done to keep this feature branch up-to-date with all the changes in the master branch. Integrating the APK generator feature branch removes the need for this maintenance work. Feature toggles are in this scenario unnecessary as most of the generator code is completely separated from Pocket Code. Using a separate repository is currently not possible, as this would require major refactoring to resolve dependency issues.

Removing the feature branch was done in two steps:

6. Software Development Workflow

1. Merge the feature branch into the develop branch through a pull request and delete the feature branch afterwards
2. Alter the Jenkins APK generator job to use the latest release version of Pocket Code instead of the latest snapshot found in the deleted feature branch

As a result of the above Jenkins change, any new feature or bug fix concerning the APK generator has to be added to a Pocket Code release in order for it to be active. This could be seen as a drawback to the overall approach, as generator changes are mostly not directly linked to the main application itself. On closer examination we can identify two major advantages of this approach:

- Maintaining the feature branch by keeping it up-to-date with the develop branch is no longer necessary
- The prior Jenkins job took the latest status of the feature branch, which means that no explicit versioning was applied. In some cases it could also include code that was not included in Pocket Code's current release. Including generator code changes into Pocket Code's release cycle solves this problem by giving the developers more control on what they want to include in a release

6.5.2. Removing One Branch

Even after several extensive discussions with the Pocket Code team we were unable to fully understand past merge problems between the develop and the master branch. We assume the following:

Section 6.1.2 explained that the content of a hotfix has to be integrated into the develop and the master branch. Integrating hotfixes through a rebase instead of a merge operation might have caused unexpected changes in the commit graph. As a result, changes in the develop branch could not be integrated into the master branch which in turn made it impossible to create a new release of Pocket Code. If these assumptions are true, such scenarios could be avoided if hotfixes are applied without using rebase operations.

Because of several project related deadlines and priorities, a decision about what should be done about the current two branch approach was postponed for several weeks. Past problems between the master and the develop branch were rare, which made this issue a low priority topic. The final outcome concerning this topic is described in Section 6.6.2.

6.6. Results

6.6.1. Feature Toggles

Pocket Code currently uses a total of nine feature toggles to handle eight sub-projects and one new UI feature (GitHub, 2016c) (18th of November 2016). While these toggles remove the necessity of using feature branches and feature branch pull requests, large pull requests with several thousand lines of changes can still occur: JIRA tickets which describe new functionalities which are not split into sub-tickets can still lead to massive pull requests. One example of such a ticket is CAT-1717 which was merged in August 2016. Its pull request contained over 7000 lines of code changes (more than 5500 insertions and more than 1500 deletions) (GitHub, 2016d). These changes also introduced a feature toggle, but the intended purpose of the toggle (to allow Pocket Code to use a set of smaller pull requests) got lost. Even though CAT-1717 was reviewed by several team members, a total of eight pull requests were created to further fix bugs in CAT-1717 after it has been merged. The following list shows all eight pull requests with their unique number (created by GitHub) and their title:

- #1981 Fix scene escaping
- #1978 CAT-2161 Fix copying of spriteVariables in scenes
- #1963 Quickfix
- #1961 Fix default scene name string in all languages and first scene string
- #1957 CAT-FIX Grouping/Backpack/Scenes
- #1956 Fix Crash when pressing back in Scenes Action Modes, Fix bug when pressing back in MainMenu, Fix bug when dismissing delete message in Scene Action Mode

6. Software Development Workflow

- #1916 CAT-FIX Several Bugs and UX Issues in Scenes, PenBricks and Bubbles
- #1892 Fix build error

This bug fixing situation could be the result of not splitting CAT-1717 into several smaller JIRA tickets. It is more likely for reviewers to miss important details when reviewing such a large code change at once (Bosu, Greiler, and Bird, 2015). Splitting CAT-1717 into several smaller tickets while using a feature toggle to merge these tickets back into the main code base could potentially have reduced the amount of subsequent bug fixes.

An example which highlights the successful and intended use of feature toggles is the Musicdroid sub-project. Since their feature branch has been removed, a total of six pull requests were created. Each of these pull requests contained less than 200 code changes. The complete list of pull requests is:

- #1975 MUS-194 Change button sizes
- #1948 MUS-197 Structure to GUI
- #1942 MUS-196 Set or unset note image on button toggle
- #1924 MUS-195 different color for black and white key rows
- #1914 MUS-201 Create a Dummy Song
- #1910 MUS-189 Replace piano place holder with image

6.6.2. Using a Single Branch

The Pocket Code team decided to leave the current two branch structure for these reasons:

- The two branch approach offers the same information as the one branch approach
- We suspect that all past problems could be avoided if hotfixes are integrated with a merge instead of a rebase operation
- Introducing a single branch approach would have led to changes on Jenkins. The team did not feel confident to adapt the current configuration because of past problems (see Section 5.1)

6.6. Results

I have summarized all information concerning this particular workflow change in Confluence (see [Appendix J](#)). This information could be used if for example new problems between the develop and master branch occur in the future.

7. Conclusion

Developing software is a complex task which depends on a vast set of factors. This thesis focused on code reviews, automated testing and development workflows using git in the Pocket Code project.

All attempts to motivate team members to participate in code reviews were unsuccessful and as a result, the created checklist was not utilized by the Pocket Code team. The enabled branch protection feature on GitHub now blocks pull requests until they are reviewed by at least one team member. This feature might positively influence the review participation of the team in the future as one of the main contributors to processing and accepting pull requests has recently left the team.

The introduction of feature toggles has removed the dependency on long-living feature branches for sub-projects such as Musicdroid, Chromecast or the APK generator. As a result, reviewing changes in these sub-projects, as well as keeping them up-to-date with Pocket Code's core features has been simplified tremendously. A future challenge for the Pocket Code team will be maintaining the list of all existing feature toggles. The team has to formulate concrete strategies on when and how to utilize feature toggles, so that the positive effect of these toggles is not outweighed by the effort needed to maintain them.

The Pocket Code team decided to rewrite its UI test suite in the future, although additional work in the code base is needed so that the team can effectively utilize automated tests. A refactoring and redesigning roadmap which outlines potential changes to the code base to improve its testability was provided to the team. Without any major action towards addressing Pocket Code's current test suite problems, the overall distribution of unit, integration and E2E tests will remain unchanged, suggesting that the test suite will remain slow and potentially unreliable.

7. Conclusion

Recent work done by a small group of team members show the tremendous impact refactoring and redesigning activities can have on the current code base. In a series of three pull requests over 11.000 lines of duplicate UI code was removed. The following list shows these pull requests:

- #1934 Refactored BrickAdapter and Brick handling
- #1940 Further Improvements for Brick handling
- #1999 CAT-2148 Refactor ListViews, ListAdapters and Fragments

This work suggests that team members deeply care about the project and are willing to invest their time to improve the maintainability of the project. Even though this refactoring and redesigning work will most likely have a positive impact on the future development of Pocket Code, it should only be seen as the beginning. A continuation of such effort is what the Pocket Code project desperately needs to be a more reliant and more enjoyable mobile application.

8. Future Work

Potential future work could include:

A recommender system could be used to automatically find team members who can review a particular pull request. Similar work was done by Veen, G. Gousios, and Zaidman, [2015](#).

The use of feature toggles as a way to activate or deactivate functionality could be extended by developing a web-based configuration system. Such a system could be used to dynamically configure feature toggles in Pocket Code. Possible applications could include:

- Deactivate faulty functions in Pocket Code without depending on the user to download an application update
- Presenting different functionality to different users with a single application. Combined with a monitoring system this could for example be used to improve the UI experience by gathering real user data

The presented learning material about how to write testable code could be used as a basis to develop additional learning material (e.g. a video series), training courses or even a lecture.

Pocket Code's development workflow could further be improved by building a continuous delivery pipeline (Humble and Farley, [2010](#)). Such a pipeline streamlines the release process by automating several steps. A potential basic pipeline could look like this:

- Compiling the source code directly from version control (e.g. git)
- Testing the code using a suite of unit, integration and E2E tests
- Building a releasable Android application (an .apk file)
- Distributing the .apk file to a group of manual testers

8. Future Work

- If the manual testers approve of the application: Uploading the new .apk file to the Google Play Store

Further studies could investigate how the onboarding experience could be improved to help new team members get started in the Pocket Code project. A survey done by Raphael Sommer, [2016](#) indicates that the two biggest issues new members face when joining the project include missing documentation and understanding the existing large code base of Pocket Code.

Section [4.6](#) concluded that Pocket Code's team members may not feel responsible for pull request related tasks. Future work could study a concept in which a team member can only pick new tasks after his or her pull requests were processed. This concept might create an environment in which team members gain more responsibility for handling pull requests as they directly benefit from a fast integration process.

Finally, future work could build upon work from former Pocket Code team members:

- In his master thesis (which has not been published yet), Roman Mauhart is evaluating SonarQube, a system which can be used to perform static source code analysis (SonarQube, [2016](#)). Such a system could further be used to track Pocket Code's progress in terms of software quality
- Maximilian Fellner, [2013](#), Christian Hofer, [2014](#) and Manuel Wallner, [2014](#) studied Behaviour Driven Development (BDD) frameworks such as Cucumber (Cucumber, [2016](#)). BDD is a testing technique in which stakeholder's requirements can be documented as automated tests. These tests can further be used as a complement to Pocket Code's existing test suite

Appendix

Appendix A.

Questionnaire (German)

Interview PocketCode (Version 1)

(an Koordinatoren, Seniors und Juniors)

Allgemein

- Was studierst du? Welches Semester?
- Wie bist du zu Catrobat gekommen?
- In welchem Team bist du?
- Wie lange bist du schon bei deinem jetzigen Team?
- Wie würdest du deine Programmierkenntnisse zu Beginn des Projektes und jetzt einschätzen? (jeweils eine Skala von 1-10, wobei 10 die höchste Bewertung ist)
- Wie definiert sich in deinem Team ein Senior? (Wann ist jemand ein Senior?)
- Wie arbeitest du am Projekt? (allein, Pair Programming)
- Arbeitest du nebenbei als Softwareentwickler/Programmierer? Beeinflusst das deine Art wie du im Team arbeitest?
 - Setzt ihr agile Methoden in der Firma ein? Wenn ja: Wie läuft das ab?

Onboarding

- Wie wurde dir beim Einstieg ins Projekt geholfen?
- Was hättest du dir als Einstiegshilfe gewünscht?
- Hattest du einen Mentor, bzw. eine Ansprechperson?
 - Wenn ja – inwiefern hat er dir geholfen?

Coding Dojo

- Was fängst du mit dem Begriff Coding Dojo (Prepared Kata, Randori Kata) an?
- Würdest du so etwas gerne in unserem Projekt haben? Wenn ja/nein warum?
- Würdest du so eine Veranstaltung als Teilnehmer besuchen wollen?
- [Senior] Würdest du bei so einer Veranstaltung helfen wollen?
- Was würde so ein Event für dich interessant machen? Welche Themen würdest du dir dort vorstellen? Was wäre in deinen Augen wichtig?
- Wie häufig sollte so etwas stattfinden? Welchen Zeitrahmen (Dauer) fändest du gut?
- Sollte es soziale Events zum Ausklang geben? Was stellst du dir vor? (z.B. etwas trinken gehen)

Code Review

- Wie viele Leute bearbeiten einen einzigen Pull Request? Ist das unterschiedlich?
- Gibt es so etwas wie einen Review Leitfaden? Wenn ja, in welcher Form? Wenn nein, warum nicht?
- Wie wird bestimmt ob jemand bereit ist Code Reviews durchzuführen?

- [Reviewer] Wie führst du ein Code Review durch?
 - Wird das Projekt auch händisch getestet oder werden nur Diffs betrachtet?
 - Auf was achtest du allgemein? (Formatierungsfehler, Logikfehler, Designverbesserungen, Namensgebung, fehlende Tests oder nicht aussagekräftige Tests)
 - Welche Tools benutzt du neben Android Studio und Git?
 - Was läuft gut in eurem Review Prozess?
 - Was sollte man an eurem Review Prozess verbessern?
 - Gibt es Dinge, die dich bei eurem Review Prozess stören?
 - Wie geht ihr damit um, dass die Programmierqualitäten der Member sehr unterschiedlich sind? Wird von Juniors schlechterer Code akzeptiert als von Seniors?
 - Gibt es Fehler die öfters übersehen werden?
- Welche Mittel werden zur Pull Request Diskussion genutzt? (GitHub Kommentare, persönliche Gespräche, IRC, ...)
- [Author] Wie gehst du mit Verbesserungsvorschlägen bei einem Code Review um?
 - Welche Formen der Rückmeldung findest du besonders hilfreich? (Kommunikationsmittel und Kommunikationsart)
 - Welche Formen der Rückmeldung findest du nicht hilfreich?
 - Gibt es Member die besonders hilfreiches Feedback geben? Wenn ja, wer?
 - Ergibt sich deiner Meinung nach ein Lerneffekt?
 - Wie könnte man Code Review ändern um einen (besseren) Lerneffekt zu erzielen?

Interview PocketCode (Version 2)

Allgemein

- Was studierst du? Welches Semester?
- Wie bist du zu Catrobat gekommen?
- In welchem Team bist du?
- Wie lange bist du schon bei deinem jetzigen Team?
- Hat sich die Art wie du programmierst verändert seit du bei PocketCode bist?
- Wie definiert sich in deinem Team ein Senior?
- Wie arbeitest du am Projekt? Arbeitest du hauptsächlich allein? Betreibt ihr Pair Programming?
- Arbeitest du nebenbei als Softwareentwickler/Programmierer?
 - Beeinflusst das deine Art wie du im Team arbeitest?
 - Setzt ihr agile Methoden in der Firma ein? Wenn ja: Wie läuft das ab?
 - Beeinflusst das Projekt die Art wie du in der Firma arbeitest?

Onboarding

- Wie wurde dir beim Einstieg ins Projekt geholfen?
- Was hättest du dir als Einstiegshilfe gewünscht?
- Hattest du einen Mentor, bzw. eine Ansprechperson?
 - Wenn ja – inwiefern hat er dir geholfen?

Coding Dojo

- Was fängst du mit dem Begriff Coding Dojo (Prepared Kata, Randori Kata) an?
- Würdest du so etwas in unserem Projekt sinnvoll finden? Warum?
- Was müsste gegeben sein damit du zu so einem Dojo einmal hingehst? Was würde so ein Event für dich interessant machen? Welche Themen würdest du dir dort vorstellen?
- [Senior] Was müsste gegeben sein damit du bei so einer Veranstaltung helfen würdest?
- Wie häufig sollte so etwas stattfinden? Welchen Zeitrahmen (Dauer) fändest du gut?

Code Review

- Gab es bei euch Probleme durch neuen Code als er in den Develop Branch gemerged wurde? Was ist passiert?
- Wie viele Leute bearbeiten einen einzigen Pull Request? Ist das unterschiedlich?
- Gibt es so etwas wie einen Review Leitfaden? Wenn ja, in welcher Form? Wenn nein, warum nicht?
- Wie wird bestimmt ob jemand bereit ist Code Reviews durchzuführen?
- [Reviewer] Wie führst du ein Code Review durch?
 - Wird das Projekt auch händisch getestet oder werden nur Diffs betrachtet?
 - Auf was achtest du allgemein? (Formatierungsfehler, Logikfehler, Designverbesserungen, Namensgebung, fehlende Tests oder nicht aussagekräftige Tests)
 - Welche Tools benutzt du neben Android Studio und Git?
 - Was läuft gut in eurem Review Prozess?
 - Was sollte man an eurem Review Prozess verbessern?
 - Gibt es Dinge, die dich bei eurem Review Prozess stören?
 - Wie geht ihr damit um, dass die Programmierqualitäten der Member sehr unterschiedlich sind? Wird von Juniors schlechterer Code akzeptiert als von Seniors?
 - Gibt es Fehler die öfters übersehen werden?
- Welche Mittel werden zur Pull Request Diskussion genutzt? (GitHub Kommentare, persönliche Gespräche, IRC, ...)
- [Author] Wie gehst du mit Verbesserungsvorschlägen bei einem Code Review um?
 - Welche Formen der Rückmeldung findest du besonders hilfreich? (Kommunikationsmittel und Kommunikationsart)
 - Welche Formen der Rückmeldung findest du nicht hilfreich?
 - Gibt es Member die besonders hilfreiches Feedback geben? Wenn ja, wer?
 - Ergibt sich deiner Meinung nach ein Lerneffekt?
 - Wie könnte man Code Review ändern um einen (besseren) Lerneffekt zu erzielen?
- Gab es bereits Probleme durch Code aus Pull Requests? Was ist passiert? (z.B. Bug) Wie wurde das behandelt? (z.B. Revert)

Feedback

- Gab es Fragen die du als weniger bis nicht relevant, oder als zu sehr in der Tiefe einstufen würdest? Sollte ich etwas streichen?
- Wir haben über Onboarding, Coding Dojos und Code Reviews gesprochen. Gibt es noch etwas über das du reden möchtest? Habe ich einen wichtigen oder interessanten Punkt vergessen den ich für meine Interviews berücksichtigen sollte?
- Wer wäre sonst noch interessant um ein Interview durchzuführen?

Appendix B.

Consent Form (German)

Institut für Softwaretechnologie
Technische Universität Graz
Inffeldgasse 16b
A-8010 Graz
Austria / Europe

Florian Winkelbauer

Tel: +43 - 316 - 873 - 5720 (direkt)

Fax: +43 - 316 - 873 - 5706

DVR: 008 1833

UID: ATU 574 77 929

Einverständniserklärung:

Ich bin damit einverstanden, dass meine Daten, die im Zuge der wissenschaftlichen Arbeit von Frau Annemarie Harzl und Herrn Florian Winkelbauer erhoben werden, anonymisiert für wissenschaftliche Zwecke ausgewertet und verwendet werden dürfen. Ich stimme auch der Veröffentlichung dieser Daten bzw. Ausschnitte dieser Daten in **anonymisierter** Form zu.

Diese Daten können folgendes umfassen:

- Fragebögen
- Interviews und Ausschnitte aus deren transkribierten Audiomitschnitten
- Code und dessen Analyse
- Beobachtungen von Meetings
- Issues, Kanbanboard
- Zeitaufzeichnung

Name: _____ Matrikelnummer: _____

Ort, Datum: _____

Unterschrift: _____

Appendix C.

Interview Results (German)

Allgemein

Was studierst du? Welches Semester?

- **P1:** Informatik (Bachelor), etwa zwanzigstes Semester.
- **P2:** Informatik (Bachelor), achtes Semester.
- **P3:** Informatik (Master), studiere schon länger.
- **P4:** Softwareentwicklung und Wirtschaft (Master). Zwölftes Semester.
- **P5:** Softwareentwicklung und Wirtschaft (Master). Bin dabei meine Masterarbeit zu schreiben.

Wie bist du zu Catrobat gekommen?

- **P1:** Durch meine Bachelorarbeit.
- **P2:** Ich bin durch die Suche nach einem Thema für meine zukünftige Masterarbeit auf das Projekt gestoßen.
- **P3:** Durch meine Masterarbeit.
- **P4:** Ich bin durch meine Bachelorarbeit zum Projekt gekommen.
- **P5:** Durch Freunde die bereits beim Projekt waren.

Wie lange bist du schon beim Projekt?

- **P2:** Seit 2013
- **P3:** Seit November 2015
- **P5:** Ich bin seit drei Jahren dabei.

Hat sich die Art wie du programmierst verändert seit du bei PocketCode bist?

- **P1:** Ja. Ich habe es aber zum Beispiel noch nicht geschafft, Test-driven Development völlig umzusetzen. Gerade bei der Implementierung von neuen Features schreibe ich die Unit Tests nicht vorher. Ich habe jetzt ein großes Interesse für alle Themen die mit Code Qualität zu tun haben. Zum Beispiel Git Commit Messages, Refactoring, oder statische Code Analyse Tools wie Checkstyle und PMD.
- **P2:** Ja. Statt gleich auf eine Lösung hinzuarbeiten, überlege ich jetzt zuerst wie ich Komponenten abstrakter gestalten kann um den Wiederverwendungswert zu erhöhen.
- **P3:** Ich programmiere relativ wenig.
- **P4:** Ja, durch Code Reviews (als Autor und als Reviewer) habe ich viel dazu gelernt.
- **P5:** Ja. Ich habe als kompletter Neuling angefangen. Damals gab es einige Projektmitglieder die mir eine sehr gute Einführung gegeben haben.

Wie definiert sich in deinem Team ein Senior?

- **P1:** Es hat dazu nie genaue Regeln gegeben. Aber Leute, die durch ihre Fähigkeiten und durch ihr Interesse aufgefallen sind, sind Seniors geworden.
- **P2:** Es gibt keine genaue Definition, aber wenn man merkt, dass sich jemand gut auskennt, wird er bald als Senior gesehen.
- **P4:** Das waren für mich immer Leute, die schon länger dabei waren und sich gut ausgekannt haben.

Wie arbeitest du am Projekt? Arbeitest du hauptsächlich allein? Betreibt ihr Pair Programming?

- **P1:** Ich arbeite nicht mehr aktiv am Projekt. Ich kümmere mich nur mehr um kleinere Dinge und mache Code Reviews.
- **P2:** Meine meiste Programmierarbeit erledige ich über Pair Programming mit verschiedenen anderen Teamkollegen.
- **P3:** Anfangs hauptsächlich allein, jetzt öfters mit Kollegen im Projektraum.
- **P4:** Ich arbeite nicht mehr sehr aktiv am Projekt. Ich bin meist einmal die Woche im Projektraum, beziehungsweise arbeite sonst von Zuhause.

Arbeitest du nebenbei als Softwareentwickler/Programmierer?

- **P1:** Ja.
- **P2:** Nein.
- **P3:** Nein.
- **P4:** Ja.
- **P5:** Ja.

Beeinflusst das deine Art wie du im Team arbeitest?

- **P1:** Eher weniger, weil ich dort mit anderen Sprachen arbeite.
- **P4:** Teilweise. Das Thema Unit Tests kommt in der Firma leider noch immer zu kurz.
- **P5:** Nein, weil ich in einer ganz anderen Sparte arbeite.

Code Review

Wie viele Leute bearbeiten einen einzigen Pull Request? Ist das unterschiedlich?

- **P1:** Derweil werden Pull Requests hauptsächlich von Thomas Schranz bearbeitet.
- **P3:** Das macht bei uns hauptsächlich Thomas Schranz.

Wieso gibt es deiner Meinung nach so wenige die Code Reviews durchführen?

- **P2:** Ich glaube, dass einige Seniors einfach keine Lust haben ein Review durchzuführen. Andere fühlen sich auch einfach nicht verantwortlich. Es könnte auch genug Leute geben, die denken, dass sie sich nicht genug auskennen würden um Reviews machen zu können, obwohl das überhaupt nicht stimmt.
- **P5:** Wir haben dieses Problem schon öfters diskutiert. Eigentlich könnte jeder Code Reviews machen, aber bis jetzt wurde das Thema noch nicht wirklich angenommen. Ich weiß auch nicht wieso Code Reviews keine breite Akzeptanz finden. Ich glaube, dass Angst vor Verantwortung eine Rolle spielt.

War es schon immer so, dass es nur wenige Mitglieder gibt die sich um Reviews kümmern?

- **P5:** Früher gab es mehr Seniors (also Leute die sich gut ausgekannt haben) die sich um solche Dinge gekümmert haben.

Sind Pull Requests deiner Meinung nach ein Flaschenhals?

- **P3:** Besonders große Pull Requests mit mehreren tausend Zeilen Code finde ich problematisch. Man kann sich allein diese riesige Änderung nicht anschauen und meiner Meinung nach läuft es auf eine einfache Frage hinaus: Wie sehr vertraue ich dem Author, bzw. für wie kompetent halte ich ihn? Wenn man als Reviewer mit dem Thema des Tickets nicht gut vertraut ist, ist diese Frage sogar noch wichtiger.

Gibt es so etwas wie einen Review Leitfaden?

- **P1:** Im Confluence gibt es einen Eintrag für "How to create a PULL REQUEST".
- **P4:** Einen richtigen Leitfaden gab es nie, es gab aber implizit einige Anhaltspunkte wie zum Beispiel "laufen die Tests?" oder eben Clean Code.

Wie führst du ein Code Review durch? / Was war früher deine Herangehensweise als du dich um Code Reviews gekümmert hast?

- **P1:** Ich achte primär nur mehr darauf, dass die Coding Guidelines eingehalten werden, oder welche Refactoring Aspekte man einbringen könnte. Ich nehme keine Pull Requests alleine ab und verzichte daher auf einige wichtige Details, wie zum Beispiel das Testen des Codes.
- **P2:** An was ich bei einem Code Review denke:
 - Kann ich verstehen was der Code tut?
 - Programm ausführen am Handy
 - Gibt es Duplikate? Oder könnte man etwas wiederverwerten?
 - Zusätzlich verlasse ich mich auf analytische Tools (Checkstyle, PMD)
 - Sind die Tests vollständig und sinnvoll?
- **P4:** Gerade wenn es größere Pull Requests war habe ich mit dem Autor zusammengearbeitet. Auf was ich sonst geachtet habe:
 - Laufen die Tests? Machen die Tests Sinn? Sind sie vollständig?
 - Wie funktioniert das in der App?
 - Was war der Kommentar des UX Teams?
 - Clean Code? Kann man etwas lesbarer gestalten?
- **P5:** Wenn es größere Pull Requests waren, dann hab ich mir die Autoren geholt weil ich durch fehlendes Fachwissen anders nicht arbeiten hätte können. Ansonsten habe ich einfach versucht meine Erfahrungen und Meinungen einzubringen. Ich habe mich auch viel auf den Jenkins verlassen um zu überprüfen, ob die Änderungen möglicherweise Defekte bringen könnten. Wir haben auch nach unseren normalen Meetings Pull Request mit Autoren gemeinsam abgenommen. Das hatte den Vorteil, dass man Kleinigkeiten gleich vor Ort ausbessern konnte.

Welche Mittel werden zur Pull Request Diskussion genutzt? (GitHub Kommentare, persönliche Gespräche, IRC, ...)

- **P1:** Vor oder nach dem Montagsmeeting der Koordinatoren gab es immer wieder die Gelegenheit, mit Thomas gemeinsam Pull Requests anzuschauen. GitHub wird auch genutzt, weil man die konkreten Zeilen im Code kommentieren kann.
- **P3:** Hauptsächlich GitHub und persönliche Gespräche im Projektraum.
- **P5:** E-Mails wurden gerade zu Releasezeiten, bzw. für dringende Momente genutzt

Was läuft gut in eurem Review Prozess? Was sollte man verbessern?

- **P1:**
 - Negativ: nur Thomas Schranz kümmert sich offiziell darum
 - Negativ: je größer der Pull Request, desto weniger genau wird er überprüft
 - Negativ: keine Continuous Integration
- **P3:** Ich fand es immer schwer Issues abzunehmen, von denen man selbst wenig Ahnung hat. Außerdem war die Dauer bis zur Abnahme des Pull Requests (durch Zeitmangel von Thomas, oder durch Probleme mit dem Jenkins) manchmal zu lang.
- **P4:**
 - Negativ: Ich fand es immer schwer Issues abzunehmen, von denen man selbst keine oder wenig Ahnung hat. Außerdem war die Dauer bis zur Abnahme des Pull Requests (durch Zeitmangel von Thomas, oder durch Probleme mit dem Jenkins) manchmal zu lang.
 - Positiv: Gerade die zusätzliche Meinung fand ich hilfreich, weil man dadurch auch auf bessere Lösungen kommen kann.

Wie geht ihr damit um, dass die Programmierfähigkeiten der Member sehr unterschiedlich sind? Wird von Juniors schlechterer Code akzeptiert als von Seniors?

- **P1:** Ich denke, dass das von Reviewer zu Reviewer unterschiedlich ist. Es gibt kein offizielles "Wie gehe ich damit um?".
- **P2:** Im schlimmsten Fall merge ich einen Pull Request einfach nicht. Meistens versuche ich bei Problemen ein Treffen mit dem betroffenen Entwickler auszumachen um gemeinsam daran arbeiten zu können.
- **P4:** Wenn wir Probleme entdeckt haben, haben wir sie einfach per Refactoring versucht zu beheben bevor wir einen Merge durchgeführt haben.

Welche Tools werden neben Android Studio und Git für Code Reviews genutzt?

- **P2:** Checkstyle, PMD, Jenkins
- **P4:** Jenkins, Lint, PMD, Checkstyle, (FindBugs war in Diskussion)

Wie kam es zu der Umstellung zum Forking Workflow?

- **P2:** Durch ein großes Mergeproblem in der Vergangenheit fanden wir es nötig diese Umstellung durchzuführen. Dadurch ist die Chance viel geringer, dass Commits Fehler ins Projekt bringen, ohne dabei auf irgendeine Funktionalität zu verzichten.

Gab es Probleme durch neuen Code der gemerged wurde?

- **P3:** Ja. Es gab vor kurzem ein Ticket, welches gemerged wurde, obwohl es bekannte Probleme gab. Es wurde gemerged, weil es im Großen und Ganzen funktioniert hat. Die restlichen Fehler sollten im Nachhinein behoben werden.
- **P4:** Ja, ich kann mich an ein großes Issue erinnern, das nach dem Merge wieder rausgenommen werden musste, weil es nicht richtig funktioniert hat. Zu diesem Zeitpunkt hat der Jenkins nicht funktioniert, was dabei sicher eine Rolle gespielt hat.

Aus der Sicht des Erstellers eines Pull Requests: Gab es Rückmeldungen die dir besonders geholfen haben?

- **P1:** Das beste Feedback für mich war immer, als ich neben einem Reviewer gesessen bin. Durch das ständige Hinterfragen meiner Lösung haben wir schlussendlich einen Fehler gefunden.

Appendix D.

Checklist

This is an exported version of the Confluence checklist. Therefore it lacks all contained integrated content (links, videos and books).

How to do Code Reviews

Introduction

The following checklist can aid you in your code review activities. This list is in no way a finished or perfect endproduct, but should rather be adapted and updated over time depending on your needs. For most of the listed points you can get more information in the [Resources](#) section which should help you to get a better understanding of what to look for and why it is important.

▼ How to use

- Ask yourself the questions listed in the different sections. If the answer to a question is "**no**" you should either examine the problem in more detail or start a discussion by leaving a comment on GitHub.
- This list should only help you to identify common problems. Feel free to also review the code in your own way. Different opinions and discussions are welcome!

▼ Changing this checklist

- **Keep it as short as possible!** Only include the most important aspects. Aim for a length of no more than a single paper page. This does not include examples or further details from expandable text views.
- Is there anything in this list which is of no use for the majority of reviews? **Change or remove it!**
- Do you keep finding the same bugs over and over again? **Add them to the list or the [Future Work](#) section!**
- Did you find a good resource which is useful for this checklist? **Add it the relevant points and to the [Resource](#) section!**
- Are there any items which could be solved by automated tool or check? **Favor automation!**

General

- Can you run the code on your phone?
- Did all tests pass?
- Does the code meet all requirements in the JIRA ticket?

Clean Code

- Can you understand the changes without great effort?
- Is the new code free of:
 - Duplication?

▼ (Read more)

No

```
private void foo() {
    // ...
    Dialog dialog = DialogFactory.createDialog("Some title");
    dialog.show();
    // ...
}

private void bar() {
    // ...
    Dialog dialog = DialogFactory.createDialog("Some title");
    dialog.show();
    // ...
}
```

Follow the **DRY** principle: Don't Repeat Yourself. Duplicated code is very hard to maintain as a change in one part needs to be done in every copied location.

Yes

```
private void foo() {
    // ...
    showDialog();
    // ...
}

private void bar() {
    // ...
    showDialog();
    // ...
}

private void showdialog() {
    Dialog dialog = DialogFactory.createDialog("Some title");
    dialog.Show();
}
```

- o Any unnecessary comments?

▼ (Read more)

TODO comments

```
// TODO fix this bug
```

Such comments are fine while a task is still under development. In any other instance such a TODO should be added as a JIRA ticket.

Code in comments

```
// Foo foo = context.getFoo();
```

```
// foo.validate();
```

Don't keep code in comments you think you might need in the future. That's what version control is for. If no developer dares to delete such comments (somebody might still need this?!) they might rot forever.

Obvious comments

```
// returns an instance of Foo
public Foo getFoo() {
    return new Foo();
}
```

Looking at the method signature gives you the same information as this comment.

Using comments instead of making code more readable

```
// Checks if a person is bob. Says hi if it's him.
if (person != null && person.getFirstName().Equals("Bob") &&
    person.getLastName().Equals("Foo") && person.getAge() == 42) {
    sayHi();
}
```

Creating comments for complicated or hard-to-read code is an excuse for not making code more readable. You can do better. Such comments should be your last resort. See [Long or complicated conditions](#) for more information on this basic example.

- - Passing or returning null?

▼ (Read more)

You can greatly decrease the amount of null checks you need in your code if you stop returning or passing null. Instead return an empty collection or a [Null Object](#). Of course there are some cases (mostly through public methods in an API or through some user input) in which you still need those checks as a form of argument validation. See **Don't return null** and **Don't pass null** in [Clean Code](#) to get more information.

- Unused (dead) code?

▼ (Read more)

Unused code makes a class more bloated than it should be. Get rid of it and If you ever need it again you will find it in version control.

- Long or complicated conditions

▼ (Read more)

No

```
if (person != null && person.getFirstName().Equals("Bob") &&
    person.getLastName().Equals("Foo") && person.getAge() == 42) {
    sayHi();
}
```

Even though this condition might not be very complicated, its length makes the `if` statement harder to read.

Yes #1

```
boolean isBob = (person != null) && (person.getFirstName().Equals("Bob"))
&& (person.getLastName().Equals("Foo")) && (person.getAge() == 42);

if (isBob) {
    sayHi();
}
```

Now we know without much effort that the long condition just checks if a person is Bob.

Yes #2

```
if (isBob(person)) {
    sayHi();
}

// ...

private boolean isBob(Person person) {
    return (person != null) && (person.getFirstName().Equals("Bob")) &&
(person.getLastName().Equals("Foo")) && (person.getAge() == 42);
}
```

Use this alternative if you need the condition in several places. This follows the **DRY** principle.

- Do method names describe side effects?

▼ (Read more)

No

```
public ObjectOutputStream getOutputStream() throws IOException {
    if (outputStream == null) {
        outputStream = new ObjectOutputStream(socket.getOutputStream());
    }

    return outputStream;
}
```

This example is taken from [Clean Code](#), **N7: Names should describe Side-Effects**. The method name does not mention the side effect of creating a new stream. Such unexpected behaviour can lead to puzzling bugs.

Yes

```
public ObjectOutputStream createOrReturnOutputStream() throws IOException {
    if (outputStream == null) {
        outputStream = new ObjectOutputStream(socket.getOutputStream());
    }

    return outputStream;
}
```

- Are classes small?

▼ (Read more)

Warning signs for big classes:

- Summing up what a class does includes the word **"and"**
- A class contains several fields which get only used by a few methods
- It's hard to get an understanding of the purpose of a class
- You hear the phrase "You can look at anything except for this class"

See **Chapter 10** in [Clean Code](#), or the **Google Talk** of **Misko Hevery** about [Object Oriented Design for Testability](#) to get more information.

- Is polymorphism used instead of several if or switch statements?

▼ (Read more)

Writing the same `if` or `switch` statements in a class over and over again is a good indicator for the need of inheritance.

See **Item 20** in [Effective Java](#), or the **Google Talk** of **Misko Hevery** about [Object Oriented Design for Testability](#) to get more information.

Tests

- Are new features or fixes tested?
- Can you understand what the tests are doing without great effort?
- Are tests independent of each other?
 - Can tests be repeated in a different order without failing?

Testable Code

- Is it easy to instantiate classes in a testing environment?

▼ (Read more)

A constructor should be very simple. In the best case it only contains field initialization. Keep a constructor free of:

- static calls (you cannot change its behaviour in a testing environment)
- new keyword in constructors for the creation of anything else than value objects or collections (use **Dependency Injection** instead)

See the **Google Talk** of **Misko Hevery** about [Object Oriented Design for Testability](#) to get more information.

- Does code avoid the use of global state?

▼ (Read more)

Avoid the use of global space or the Singleton Pattern, as it makes instantiating and testing an object in testing environment **very hard**. You cannot change the behaviour of global space and it can lead to unexpected side effects. Tests may pass or fail depending on the order you execute tests.

See the **Google Talk** of **Misko Hevery** about [Object Oriented Design for Testability](#) to get more information.

- There are no workarounds or extra methods that are only used for testing

▼ (Read more)

An example of this would be a `setInstance()` method used by a Singleton Pattern so that you can change the behaviour of the global state in your testing environment. You will only ever find such calls in test code, never in production code.

Design

- Can new GUI strings be displayed in different languages?

▼ (Read more)

Text that gets shown to the user has to be defined in String Resources (R.String.<string_name>) so that different languages can be supported through [our translators](#).

- Does each caught Exception get handled properly so that there are no empty catch blocks?

▼ (Read more)

No

```
try {
    // ...
} catch (SomeException e) {
}
```

A bug could hide in such an empty catch statement. If you are certain that no error handling needs to be done or the exception gets never thrown add logging or assertion statements to it.

Yes #1

```
try {
    // ...
} catch (SomeException e) {
    // This Exception should never get thrown because ...
    log.Error(e);
}
```

Yes #2

```
try {
    // ...
} catch (SomeException e) {
    // This Exception should never get thrown because ...
    throw new AssertionError();
}
```

See [Item 65](#) in [Effective Java](#) to get more information.

- Are exceptions used instead of return codes?

▼ (Read more)

No

```
public int validate(Foo foo) {
    if (foo == null) {
        return -1;
    }
}
```

```
// ...  
  
return 0;  
}
```

Yes

```
public void validate(Foo foo) {  
    if (foo == null) {  
        throw new ValidationException("Foo cannot be null");  
    }  
  
    // ...  
}
```

See [Clean Code, Prefer Exceptions to Returning Error Codes](#) to get more information.

- Are `equals()`, `hashCode()` and `toString()` overridden in new data classes?

▼ (Read more)

Implementing `equals()` without `hashCode()` could create hard to detect errors when objects of the class are used in HashMaps. See **Item 8** and **9** in [Effective Java](#) to get more information about how to correctly implement `equals()` and `hashCode()`.

Override `toString()` to make print statements and debug information more readable.

- Are compositions favored over inheritance?

▼ (Read more)

Don't use inheritance if the superclass was either not designed for such a use or is in a different package. Instead use a **Wrapper Pattern** by implementing a common interface. See **Item 16** in [Effective Java](#) to get more information.

Blockers

This text is taken from: [Resources](#) (Other links - Dimagi Code Review)

A “blocker” is a change that the reviewer requests be made before the code is merged. A blocker should typically be in one of the following categories:

- An obvious defect
- An obvious improvement that is low effort
- An improvement that is high-effort, but important enough to still be worth doing

In general, and especially in the last scenario, it is important to be very careful and communicate clearly why something is being declared a blocker. Asking your teammates to put in substantial additional work should be viewed as a big deal, and the corresponding benefits of those changes should be easy to articulate and justify the value of the effort being requested.

Many times, items in the second category can be merged and followed up on in future pull requests. This is especially pertinent when the reviewer and reviewee are in different timezones. If the code does not obviously have a defect, the bar to merge should be lower for when reviewing those in a significant timezone difference. It is important though that the feedback does get responded to in a followup PR. Some practical examples

- Asking the author to write a few simple tests for a change is a reasonable blocker.
- Asking the author to write tests that would require bootstrapping an entire test framework should not be a blocker unless the code is absolutely mission-critical (and if it is mission-critical code it likely already has a test framework).
- Asking the author to rename a poorly-named function is a reasonable blocker, though can typically be addressed in a follow up PR
- Cleanup like converting a tuple to a namedtuple should likely not be a blocker

Resources

- ▼ YouTube: Jeremy Clarks on 'Clean Code'

Even though this presentation is about C#, the mentioned principles are all taken from the [Clean Code](#) book which also covers Java code. Watch this if you want to get a basic understanding without reading the book (which I would still recommend).

- ▼ Youtube: Google Talk by Misky Hevery on 'Object Oriented Design for Testability'

You can find the complete Google Talks playlist of clean code talks from 2007 to 2009 [here](#).

There also exists a small guide (web and pdf) about the content of this talk here: [Misko Hevery - Code Reviewers Guide](#)

- ▼ Book: Clean Code
- ▼ Book: Effective Java (Second Edition)
- ▼ Other links

This is a list of other links that were used to create this checklist:

- [blog.fogcreek Checklist](#)
- [DZone Checklist](#)
- [SmartBear Guide To A Better Code Review Prozess](#)
- [SmartBear 11 Proven Pratices For Code Review](#)

- [Java-Success Checklist](#)
- [Dimagi Code Review](#)

Last access: 18.02.2016

Future Work

This section lists ideas and areas which could improve this checklist.

- Adding extra sections for different teams - are there bugs which happen often in a few teams that we can identify and add here?
- Adding a section for XML issues:
 - Preventing a `NullPointerException` by a call to `inflate()` as a result of not defining a root element
 - Style checks (e.g. defining where paddings can be added)
- Checking correct handling of resources (e.g. camera or flashlight) for scenarios like pause or resume
- Are there any Android specific bugs that we should add here?
- Checks for valid and/or sanitized input/output
 - e.g. for serializing objects
- Keep the accessibility of classes minimal. Don't expose methods if you don't have to
- Adding a [Commit Message](#) checklist item
- Adding a checklist item for new warnings (Android Studio, Lint, ...)
- Adding a checklist item for Checkstyle violations instead of manually checking code format
- Adding a checklist item for uninitialized variables (no assignment in constructor)
- Adding a checklist item for reviewing UI unit tests - could they be replaced by normal unit tests?

Related articles

- Seite:

[How to do Code Reviews](#)

Appendix E.

Pull Request

This is an exported version of the Confluence pull request how-to. Therefore it lacks all contained links.

How to create a Pull Request

FAQ

- **Who reviews pull requests?**
 - A senior of your subteam and/or a senior of the core team. A list of seniors can be found on each [team page](#)
 - Review your own code using our [checklist](#)
- **Who approves (merges) pull requests?**
 - Ask a senior or join the weekly Catroid meeting (every Wednesday 17:30) for more information

Before you create a pull request

- Run the Android Studio Formatter **on your changes only**. More information can be found [here](#)
- Use the Gradle task **check** to fix issues concerning your changes
- Use the [Android Studio code inspection tool](#) to fix issues concerning your changes
- Squash your branch to **one commit** for small features or bug fixes
- Read about [consistent commit messages](#) and [prior pull requests](#)
- Merge the develop branch into your own branch (**feature branch with many commits only**)
- Review your own code using our [checklist](#)

After creating a pull request

- **Fix failing checks** on your pull request (Formatter, Jenkins)
 - Not fixing these checks will reduce the chance that somebody will have a look on your request
- See "**Who reviews pull requests**" to find a reviewer
- Move your JIRA ticket to **Ready for Code Review**
- Join the discussion on [GitHub](#)

Related articles

- [How to create a Pull Request](#)
- [How to do Code Reviews](#)
- [How to create an .apk stand alone file](#)
- [How to make a MERGE / RELEASE](#)

Appendix F.

Feature Toggles

This is an exported version of the Confluence feature toggles how-to. Therefore it lacks all contained links.

How to use Feature Toggles

Introduction

This article describes what feature toggles are, as well as how and why we use them.

History

We used to create feature branches for each sub-project (e.g. Drone, Physics, Arduino, Musicdroid) of PocketCode before it was merged into the develop branch. Feature branches were the central base of a sub-project from which several other branches (for each new JIRA ticket) were created. This ensured that no unfinished code interfered while preparing the next release. After the first implementation of a sub-project was finished and merged back into develop, no new feature branch was created. Instead all ongoing development of a sub-project was done in branches which originated from develop, which is identical to how PocketCode's core functionality is developed.

As each branch can only be merged through GitHub's pull request system, new functionality that was part of a sub-project could be found in two pull requests:

- From initial branch (created through a JIRA ticket) into a feature branch - this is a rather small pull request
- From a feature branch into develop - this is a massive pull request as it contains several JIRA tickets

Feature Toggles

Feature toggles (also called feature flags, or switches) are an alternative to feature branches. A toggle is a simple condition which guards the entry point of functionality which has not been finished yet. A very basic example which switches between implementations of some algorithm may look like this:

```
boolean useNewImplementation = false;
// useNewImplementation = true; // uncomment this if you develop the new algorithm!

if (useNewImplementation) {
    newImplementation(arguments);
} else {
    oldImplementation(arguments);
}
```

After the new implementation has been finished and tested, the condition logic can be removed, cleaning up the code base. Similar to feature branches, the toggle approach hides unfinished code from the user.

Advantages

- Fewer merge problems as a result of not having to deal with feature branches (which tend to have the biggest pull requests we have to deal with)
- Being able to easily activate or deactivate features through flipping boolean switches
- Only a single pull request is needed to merge finished JIRA tickets concerning sub-projects into the develop branch

Disadvantages

- Developers have to gain knowledge about how feature toggles are implemented and used
- Having many feature toggles may become increasingly hard to manage

Having to deal with two pull requests for features of a sub-project is something we want to avoid, which is why we are preferring feature toggles.

Adding Feature Toggles

Adding toggles for a sub-project is rather easy:

- Open build.gradle
- Add your new toggle `FEATURE_<DESCRIPTION>_ENABLED` to the buildTypes section (replace `<DESCRIPTION>` with a proper name, e.g. `PHYSICS`, or `MUSICDROID`)

```

android {
    // ...

    buildTypes {
        debug {
            // ...
            buildConfigField "boolean", "FEATURE_<DESCRIPTION>_ENABLED",
"true"
        }
        release {
            // ...
            buildConfigField "boolean", "FEATURE_<DESCRIPTION>_ENABLED",
"false"
        }
    }
}

```

- Synchronize Gradle in AndroidStudio so that the generated class BuildConfig gets updated
- Add conditions to all entry points of the new feature

```

import org.catrobat.catroid.BuildConfig;

// ...

if (BuildConfig.FEATURE_<DESCRIPTION>_ENABLED)
{
    // ...
}

```

- **Creating a pull request:** Make sure that your toggles are turned off in the build.gradle file!

Cleaning up

Consider when it's the best time to remove your toggles. This can for example be done after a sub-project is ready to be included in a release, or even after a release including an additional testing time has passed. To do that you have to:

- Remove the condition code for your toggle
- Remove the toggle from build.gradle
- Synchronize Gradle in AdnroidStudio to update the generated class BuildConfig

Resources

- [Martin Fowler](#)

Appendix F. Feature Toggles

This is an exported version of the Confluence pull request how-to. Therefore it lacks all contained links.

Appendix G.

Testing Articles

This is an exported version of the Confluence testing article series. Therefore it lacks all contained links.

[Testing] #0 How To Write Unit Tests

We have all seen examples of "how to write unit tests" using a basic application like a calculator or some other functional style program. They were rather straightforward and we thought we now would know how to write tests, but somehow, trying to write them in our everyday development work seems to not go too well. Testing feels weird and we have no idea where or how to start. It turns out that the reason for this is that we have not yet learned how to write production code that is testable.

As of June 2016 Pocket Code has over 1.400 tests. The distribution between those tests is even, meaning that we have +700 unit and user interface tests. On first sight this may sound like a solid test base, but on second thought we can identify two issues:

- Tests are flaky, which means that they either pass or fail without any applied change. This reduces the trust in our own test suite
- While our unit tests take around two minutes to run, the UI tests need well over **six hours**. This gets even worse with flaky UI tests

"Ok smart@!#" you might say, "*what should I do differently?*". Fear not, this upcoming article series should help you to get started in the right direction. In the upcoming first article you will learn:

- How to decrease coupling between classes
- Why we should split object creation and object usage
- How we can decrease the usage of external dependencies in our tests

In the end these articles should hopefully help you to write code that is easier to test!

[Testing] #1 Dependency Injection

Dependency injection is a concept which helps us to decrease coupling. It simply means that any dependency a class needs should be passed to it as a method or constructor parameter.

A very basic example without dependency injection:

```
public class Car {  
  
    private Engine engine;  
  
    public Car() {  
        engine = new Engine();  
    }  
  
    public void start() {  
        engine.start();  
    }  
}
```

Compare this to an example with dependency injection:

```
public class Car {  
  
    private Engine engine;  
  
    public Car(Engine engine) {  
        this.engine = engine;  
    }  
  
    public void start() {  
        engine.start();  
    }  
}
```

The difference may seem like no big deal, but from a testing point of view, this changes a lot. In the second example `Engine` could be an interface, which would not affect `Car` in any way. `Car` does not care which class it gets passed, as long as it can be used like an `Engine`. This also means that in a testing environment we might not even need to instantiate a "real" `Engine`:

```

public class CarTest extends TestCase {

    public void testEngineStarted() {
        EngineStub engine = new EngineStub();
        Car car = new Car(engine);

        car.start();

        assertTrue(engine.isStarted());
    }

    public void testEngineNotStarted() {
        EngineStub engine = new EngineStub();
        Car car = new Car(engine);

        assertFalse(engine.isStarted());
    }
}

public class EngineStub implements Engine {

    private boolean started;

    public void start() {
        started = true;
    }

    public boolean isStarted() {
        return started;
    }
}

public interface Engine {

    public void start();
}

```

Changing the implementation of `Engine` was impossible in the first example due to the `new Engine()` statement being in the `Car` class. Using classes like `EngineStub` has several advantages:

- We can change the implementation of `Engine` in our tests
- Using test doubles (as in stunt doubles) can be significant faster than the "real" object. This becomes relevant if we think about databases, network activities, file systems or in our case the Android hardware
- We can test the state or the behaviour of `Car` through our test doubles without any additional production code. Note that the "real" implementation of `Engine` in our second example may not need a `isStarted()` method at all

Creating a class like `EngineStub` may seem like a lot of extra work, but luckily frameworks have been created that can help us out by creating mock objects. The behaviour of these objects can be set at runtime, which means that we have direct control over how they act. In addition to that mocks can also verify that they are used in a predefined way. You can find more information about mocking and the difference between mocks and stubs in the resource section at the end.

Note: You don't always have to rely on test doubles like mocks or stubs. It is perfectly fine to use "real" objects if they are easy to use and do not depend on external components.

"But the new `Engine()` statement doesn't just disappear because of dependency injection" you might say. And you are right. Somewhere we will find `new Car(new Engine())` in some way. The important point is that the `new` call is in a **different** class. Dependency injection splits the responsibilities of object creation and object usage. Our first example did both: it created an instance of `Engine` and it performed actions on it. The second example did only the latter. Here is the missing part:


```
public class CarFactory {  
  
    public Car createCar() {  
        return new Car(new Engine());  
    }  
}
```

"Ugh a factory... but... I do have to test that method too. So what's the big deal?!" Bare with me. We need classes that care about object creation with as little other business logic as possible. Factories are one possible solution which fulfill this requirement. Testing such classes is different than writing isolated unit tests. We will discuss this in a later article. For now let's just remember that through dependency injection we gain more control in our `CarTest` class as we are not relying on the production code of `Engine`.

Note: Avoiding `new` in classes that act on objects does not necessarily apply to literal types such as `String`, `int`, `boolean` etc.

Conclusion

Alright, so far we have:

- Decreased coupling between `Car` and `Engine`, which allowed us to write independant unit tests in `CarTest`
- Split object creation (`CarFactory`) and object usage (`Car`) by deciding where and where not to put `new` statements
- Introduced a test double which interacts with our `Car` object, giving us more control in our test cases without creating any additional production code

The next article will build upon dependency injection and explore global space and its impact on testing!

Resources

- [Android Mocking](#)
- [Martin Fowler on Mocks and Stubs](#)
- [Misko Hevery: Guide about Writing Testable Code \(4 Flaws\)](#)
- [Misko Hevery: Flaw #1 - Constructor Does Real Work](#)

[Testing] #2 Global State

Video recommendation: [Misko Hevery on OO Design for Testability](#) (it only takes one hour and it will be worth your time)

Let's start talking about global state with a code snippet:

```
int a = new Worker().doSomething();
int b = new Worker().doSomething();

assertEquals(a, b);
```

The only case in which `a` and `b` contain different values is when `Worker` depends on some global state which changes how the class behaves. This becomes problematic in the context of testing, as the amount of executed tests, their execution order or some other factor (e.g. tests that touch `new Date()` or `Math.random()`) can change test results in unexpected ways. The above video talks a lot more about global space (among other topics), which is why I am focusing on a very specific issue:

Singletons

The [Singleton pattern](#) is a form of global state which we seem to **really** like in Pocket Code. I can't blame anyone, I used to believe that it is a pretty good idea too. Having exactly one instance of a specific class can be very beneficial, there is no doubt about that in my opinion. The evil thing about Singletons though is, that they are accessible by calling `MyFancySingleton.getInstance()` wherever we would like. Depending on the implementation of this global class, side-effects can occur which are far from obvious.

Note: This does not only apply to Singletons, but to any global accessible component!

But there is one more problem: Being able to access a component globally also means that any class which uses this component is tightly coupled to it, making it impossible to inject a test double in a testing environment. There is no such thing as dependency injection for global state. Some people fall back on crazy hacks like using `MyFancySingleton.setInstance(newImpl)` which they (hopefully) call in their tests. This is error prone and creates more problems than it tries to solve.

[ProjectManager](#) is one example of a Singleton we have in Pocket Code. Among other things it is used to load and save a [Project](#), which involves I/O operations. So any class which uses [ProjectManager](#) or some other class which relies on it is tightly coupled to the underlying I/O operations. Our test cases become fragile and slower as we cannot swap out any global state. This is also why tests may fail due to a "SD card is busy" `IOException` even though they should pass without any problem.

So what should we do? The long term solution should be to replace Singletons, so that we can make use of dependency injection instead. This of course leaves the possibility that more than one object of a previous Singleton class is created by accident, which means that we have to take more care while using them.

Note: There are two cases in which I believe that Singletons are fine:

- If they only consume data instead of returning it - e.g. loggers: Applications don't depend on loggers for data and their file writing behaviour can be deactivated in a testing environment, which speeds up the test run
- If they are used in a public API. There are some general things we should consider when using external dependencies (like an API) - we will talk about this in a future article!

Getting Practical

So let's imagine for a moment that we would want to get rid of the [ProjectManager](#) Singleton. This isn't easy for a few reasons:

- It has a dependency on [StorageHandler](#). Both bring nearly **1.800 lines of code** to the table
- Their size is an indicator that they have several responsibilities which are used all over the place. This means that we would have to toss these huge objects around everywhere (which makes you think that dependency injection is pretty useless)
- They use each others methods, which makes things more interesting

But still, how would we do it?

- The first step would be to take a close look at these classes to find out how many things they are trying to do
- The next step could be to split them up into more classes while still applying the Singleton pattern ("*Even after all your Singleton smack talk?*" - yep). Think about this as a transition phase
- The third and final step would focus on removing those newly created classes from the global state

Now that's just an overview. This alone will take a lot of time and effort, which raises the question if it is even remotely worth it. And that's a good question. I can't think of any other solution besides the previously mentioned `setInstance(...)` hack to get rid of all the trouble these global objects give us. They are both huge and this won't improve at all without any major change. Each class was changed in more than 30 commits this year. So I don't think that it's about *if* we should do it, but rather *when* we should start.

Conclusion

So let's summarize global state:

- Test cases involving global state can sometimes fail without any obvious reason
- Dependency injection cannot be applied
- Singletons are just another form of global state
- Having one object of a certain class is a good idea, but we need other solutions than relying on the Singleton pattern to create a reliable (as in non-flaky) test suite

The next article will cover these topics:

- The different types of tests
- How we can separate our code from the "outside world"
- The benefits we gain due to such a segregation
- If and how we should test external dependencies

Resources

- [Misko Hevery on OO Design for Testability](#)
- [Misko Hevery: Flaw #3 - Brittle Global State & Singletons](#)

[Testing] #3 Types Of Tests

It's time for some definitions. There exist a lot of different opinions on the types of automated tests and you may or may not agree with them. Instead of preaching a complete wall of definitions, I'll give you a few key points that I am using. This will be quick, I promise:

- Unit Tests
 - Test a single behaviour (e.g. a class or a method, but this is not set in stone. Read more about units of behaviour [here](#))
 - They do not cross boundaries, meaning that they are not using the file system, a database, a network connection or the Android hardware. Some people even go as far as to say that the class under test should be the only concrete class - any other collaborator has to be either a stub or a mock
- Integration Tests
 - Tests that cross boundaries
 - Use real collaborators instead of test doubles
- End-to-end Tests (e.g. UI Tests)
 - Tests which involve complete use cases (e.g. logging into an user account using a UI form)

Check out this [blog post by Google](#) to learn more about the testing pyramid and why no one should rely too heavily on end-to-end tests.

Let's face it, a lot of Pocket Code's tests are either integration or end-to-end tests. Today we are going to talk about how we could change this.

Building A Wall

We can split source code into three camps (I have stolen this from [J. B. Rainsberger](#)):

- **The happy zone:** Code that our team has written
- **The bad outside world:** Code that was written by other (non-team) people
- **The demilitarized zone:** Code that was written by us which taps into the bad outside world

So what does this mean? The "happy zone" contains only our code, it's that sunny place where everything is easy to test. The "bad outside world" refers to any code that is out of our control. That's the code we expect to work without any test (there are exceptions, but we are going to keep this simple). Finally there is this grey area in which both worlds collide. That's the camp we struggle with the most while writing tests.

Now the basic idea is to separate the "happy zone" from the grey area, while keeping the latter one as small as possible. If we think of the "happy zone" as a circle, our goal is to push any outside dependency at the border of the circle. This way the demilitarized zone should build a thin wall around the circle, shielding it from any outside crazyness. Calling the "bad outside world" from within the circle is something that has to be avoided at all cost.

Applying this three layered concept means that we communicate with the outside world through adapter classes, decoupling our code from external dependencies. Alright, how could this be applied?

Example

Let's look at a short example of a class which writes some text to a file:

```
public class GreetingsWriter {  
  
    public void writeGreeting(String yourName, String filePath) throws IOException  
    {  
        // The try-with-resources statement is a Java 7 feature. This way we can  
        ditch the writer.close() call  
        try (FileWriter fileWriter = new FileWriter(filePath)) {  
            fileWriter.write("Hello " + yourName + "!\n");  
        }  
    }  
}
```

Using the above principle we could split this code into one interface and two classes:

```

public interface WriterAdapter {

    public void writeLine(String line, String path) throws IOException;
}

// Demilitarized zone
public class FileWriterAdapter implements WriterAdapter {

    @Override
    public void writeLine(String line, String path) throws IOException {
        // Bad outside world
        try (FileWriter fileWriter = new FileWriter(path)) {
            fileWriter.write(line + "\n");
        }
    }
}

// Happy zone
public class GreetingsWriterNew {

    private WriterAdapter writer;

    public GreetingsWriterNew(WriterAdapter writer) {
        this.writer = writer;
    }

    public void writeGreeting(String yourName, String filePath) throws IOException
    {
        writer.writeLine("Hello " + yourName + "!", filePath);
    }
}

```

Your first reaction might be *"So what, all we did was adding a bunch of extra code for something really simple"*. That's true considering this short example. In a decent sized application though the adapter code should be significantly smaller than anything else in "the happy zone", which leaves us in a better place. To further understand this, let's look at some test cases we could write for our new code. We could test that:

- **Unit test #1:** GreetingsWriterNew passes a greeting note containing a name and a file path to WriteAdapter without using any file operation
- **Unit test #2:** GreetingsWriterNew re-throws any IOException that the writer.writeLine(...) could throw
- **Integration test #1:** FileWriterAdapter writes a new line to the file system

In our initial example such a clean separation of tests would not be possible, as two out of three tests would involve real file operations. Writing a unit test for another feature like writeGoodbye(String yourName, String filePath) becomes easy - we don't have to deal with I/O as we know that it works due to our integration test.

Getting Practical

How can we apply all this in Pocket Code? This is yet another multi-step process involving a lot of refactoring:

- Find a boundary that we cross (e.g. loading something from memory or using a smart phone camera)
- Create an interface and a concrete class which deals with the outside world
- Refactor classes to now depend on an adapter instead of the concrete implementation using dependency injection

So far we haven't talked about testing user interface features. That's because testing UI is yet another hard topic. UI code doesn't live in the "happy zone", which is why patterns like MVC or MVVM come in handy for writing unit tests. I'll leave you with this thought: Do we really want to test if typing a username and a password in a text box works, or do we rather care more about having a working login implementation?

Conclusion

This article talked about how we can split code in three different camps, allowing us to focus more on testing simple units instead of writing

bulky test cases involving several components.

In the next article we will talk about the structure of a single unit test!

Resources

- [Test Driven Development](#)
- [Google Testing Blog: Testing Pyramid](#)
- [J. B. Rainsberger On Unit Testability](#)
- [MVC](#)
- [MVVM](#)

[Testing] #4 Anatomy Of A Unit Test

The previous articles talked about several big picture topics. Today we are focusing on the format of an individual test.

A unit test should consist of three blocks:

- **Assemble:** Initializes anything the current tests needs
- **Act:** This is where all the action happens
- **Assert:** Finally we check if the actual behaviour matches our expected behaviour

Let's have a look at a set of tests taken from Musicdroid's `TrackTest` class. All we need to know about these tests is this: A `Track` is just a collection which holds objects of `NoteEvent` and some other information. Here are two slightly modified tests to highlight a few things:

```
public class TrackTest extends TestCase {

    private Track track;

    public void testAddNoteEvent() {
        track = new Track(MusicalKey.VIOLIN, MusicalInstrument.ACOUSTIC_GRAND_PIANO);
        assertNotNull(track);
        assertTrue(track.empty());

        track.addNoteEvent(0, new NoteEvent(NoteName.C4, true));
        assertEquals("Failed to add note event", 1, track.size());
    }

    public void testToString() {
        String expectedString = "[Track] instrument=" + track.getInstrument()
            + " key=" + track.getKey()
            + " size=" + track.size();

        assertEquals("Failed to create String from track", track.toString(),
            expectedString);
    }
}
```

What's the problem with these tests?

- They use the private member `track` instead of creating local members. Did you notice the bug? We forgot to initialize `track` in `testToString()`. This test would fail with a `NullPointerException` if it is executed without previously running `testAddNoteEvent()`.
- One of them contains three assert statement:
 - The first one is completely useless. How could this ever be null? And even if that would happen, the first call of a `track` method would give us the exact same information.
 - The `assertTrue(track.empty())` line should be a complete separate test if we really care about this check.
 - Finally we check what this test is all about: did we successfully add a `NoteEvent`?

Fixing tests that contain several assertions can become frustrating. We fix the first issue and re-run the test only to find that it fails yet again somewhere else down the line. This becomes even worse when we have to deal with assert statements in loops, or if the test takes a long time to run.

There are two additional issues:

- The assertion in `testToString()` swapped the expected and the actual value: `assertEquals(String expected, String actual)`. In case of a test failure JUnit would report us something similar to this: *"Expected '[Track] instrument=ACOUSTIC_GRAND_PIANO key=VIOLIN size=-123890', but was '[Track] instrument=ACOUSTIC_GRAND_PIANO key=VIOLIN size=0"*. In this case we can easily see that these values are swapped, but this might not be obvious in other cases.
- **Who the heck needs assertion messages?!** We really care about not creating useless comments (Clean Code, remember?), but yet somehow we think that we gain anything from these notes. The above tests are short and simple enough that we can figure them out without an assertion comment. We might think that such messages come in handy when dealing with larger tests, but we would much rather have a complicated test split up into smaller ones than having a note which reads "This failed".

Let's rewrite these tests:

```

public class TrackTest extends TestCase {

    public void testAddNoteEvent() {
        // Assemble
        Track track = new Track(MusicalKey.VIOLIN,
MusicalInstrument.ACOUSTIC_GRAND_PIANO);
        NoteEvent noteEvent = new NoteEvent(NoteName.C4, true));

        // Act
        track.addNoteEvent(0, noteEvent);

        // Assert
        assertEquals(1, track.size());
    }

    public void testToString() {
        // Assemble
        Track track = new Track(MusicalKey.VIOLIN,
MusicalInstrument.ACOUSTIC_GRAND_PIANO);
        String expectedString = "[Track] instrument=" + track.getInstrument()
            + " key=" + track.getKey()
            + " size=" + track.size();

        // Act (nothing to do)

        // Assert
        assertEquals(expectedString, track.toString());
    }
}

```

This looks pretty good, but there is still something left to do: These tests contain irrelevant information - we don't really care about the parameters needed to create `track` and `noteEvent`. We will get rid of them in the next article!

Note: The above structure comments were only inserted to highlight the structure. We don't need them in real tests.

Exceptions

Testing exceptions is yet another tricky topic. We are currently using JUnit 3:

```

// JUnit 3
public void testAddNoteEventException() {
    // Assemble
    Track track = new Track(MusicalKey.VIOLIN,
MusicalInstrument.ACOUSTIC_GRAND_PIANO);

    try {
        // Act
        track.addNoteEvent(-1, new NoteEvent(NoteName.C4, true));
        fail();
    } catch (IndexOutOfBoundsException e) {
    }

    // Assert (nothing to do)
}

```

- **Bad:** Forgetting the `fail()` statement would mean that our test would check nothing at all.
- **Bad:** No assertion at the end. In fact we are using an implicit assertion by using `fail` and an empty `catch` block.

Let's have a look at an example using JUnit 4:


```

// JUnit 4
@Test(expected=IndexOutOfBoundsException.class)
public void addNoteEventWithException() {
    // Assemble
    Track track = new Track(MusicalKey.VIOLIN,
MusicalInstrument.ACOUSTIC_GRAND_PIANO);

    // Act
    track.addNoteEvent(-1, new NoteEvent(NoteName.C4, true));

    // Assert (nothing to do)
}

```

- **Good:** The test fails if we forget to add the Exception annotation.
- **Bad:** We aren't using any assert statement at the end the test case. This is still not ideal, but due to the Annotation it's better than what we are using right now.

Note: Other approaches exist which also check the exception message as another form of verification. Read more [here!](#)

And finally here's a snippet of one test case taken from [ProjectManagerTest](#):

```

public void testSavingAProjectDuringDelete() {
    // some code here

    try {
        projectManager.loadProject(TestUtils.DEFAULT_TEST_PROJECT_NAME,
getContext());
    } catch (CompatibilityProjectException compatibilityException) {
        fail("Incompatible project");
    } catch (ProjectException projectException) {
        fail("Failed to identify project");
    }

    // more code here
}

```

What is the point of using a trycatch block with fail() like this? Instead we could add throws Exception to the method signature, which removes the need to use the trycatch block. Also the additional error messages in fail() don't offer any more information than the exceptions themselves would do.

Getting Practical

We should consider upgrading to JUnit 4 to further improve our exception testing. Changing our tests to be run by JUnit 4 comes with some refactoring work as it is different than JUnit 3: it uses different imports and it relies on annotations like @Test or @Rule.

[This link](#) gives a quick overview of how JUnit 4 and Mockito (a mocking framework) are used in Android.

Conclusion

Now that we understand the structure of unit tests, we can start writing tests which are easier to read and understand!

Next up: a closer look on creating objects in tests!

Resources

- [Different Ways Of Testing Exceptions](#)
- [Building Local Unit Tests](#)

[Testing] #5 Using 'new' in Test Cases

Let's start with the test cases we've used before:

```
public class TrackTest extends TestCase {

    public void testAddNoteEvent() {
        Track track = new Track(MusicalKey.VIOLIN,
MusicalInstrument.ACOUSTIC_GRAND_PIANO);
        NoteEvent noteEvent = new NoteEvent(NoteName.C4, true);

        track.addNoteEvent(0, noteEvent);

        assertEquals(1, track.size());
    }

    public void testToString() {
        Track track = new Track(MusicalKey.VIOLIN,
MusicalInstrument.ACOUSTIC_GRAND_PIANO);
        String expectedString = "[Track] instrument=" + track.getInstrument()
            + " key=" + track.getKey()
            + " size=" + track.size();

        assertEquals(expectedString, track.toString());
    }
}
```

Today's goal should be to simplify these tests even more by removing irrelevant information which add no real value to help us understand them:

- We don't really care about the parameters used to create a `track` in our tests
- We also don't need any information about a `noteEvent`

Note: The following strategies also help against cascading update problems in test cases. These problems occur if e.g. the constructor of a class is changed so that we end up with compile errors in our tests. Adding an argument to the `Track` constructor means, that we have to update both our test cases. This becomes more tedious the more we use `new`.

Test Data Factory (also known as Object Mother)

One approach would be to use factories to create objects with default values. You can find these in Musicdroid - take a look at [TrackTestDataFactory](#) and [NoteEventTestDataFactory](#). These classes contain methods to create any basic flavor we could possibly need. [TrackTestDataFactory](#) also contains methods to create examples of a `track` filled with more data. Using factories in our code looks like this:

```
public class TrackTest extends TestCase {

    public void testAddNoteEvent() {
        Track track = TrackTestDataFactory.createTrack();
        NoteEvent noteEvent = NoteEventTestDataFactory.createNoteEvent();

        track.addNoteEvent(0, noteEvent);

        assertEquals(1, track.size());
    }

    public void testToString() {
        Track track = TrackTestDataFactory.createTrack();
        String expectedString = "[Track] instrument=" + track.getInstrument()
            + " key=" + track.getKey()
            + " size=" + track.size();

        assertEquals(expectedString, track.toString());
    }
}
```

Now we have successfully hid any information about object creation that we don't need in our tests! We could argue that the 0 parameter in the `addNoteEvent()` call is irrelevant as well, but changing this would most likely be more confusing than helpful.

Using factories has one issue:

Take a look at the `createSimpleTrack()` method in [TrackTestDataFactory](#) (the details of this method are not important):

```

public static Track createSimpleTrack() {
    Track track = createTrack();

    long tick = 0;

    track.addNoteEvent(tick, new NoteEvent(NoteName.C4, true));

    tick += NoteLength.QUARTER.toTicks(Project.DEFAULT_BEATS_PER_MINUTE);

    track.addNoteEvent(tick, new NoteEvent(NoteName.C4, false));
    track.addNoteEvent(tick, new NoteEvent(NoteName.E4, true));

    tick += NoteLength.QUARTER.toTicks(Project.DEFAULT_BEATS_PER_MINUTE);

    track.addNoteEvent(tick, new NoteEvent(NoteName.E4, false));
    track.addNoteEvent(tick, new NoteEvent(NoteName.F4, true));

    tick += NoteLength.QUARTER.toTicks(Project.DEFAULT_BEATS_PER_MINUTE);

    track.addNoteEvent(tick, new NoteEvent(NoteName.F4, false));
    track.addNoteEvent(tick, new NoteEvent(NoteName.C4, true));

    tick += NoteLength.QUARTER.toTicks(Project.DEFAULT_BEATS_PER_MINUTE);

    track.addNoteEvent(tick, new NoteEvent(NoteName.C4, false));
    track.addNoteEvent(tick, new NoteEvent(NoteName.C5, true));

    tick += NoteLength.QUARTER.toTicks(Project.DEFAULT_BEATS_PER_MINUTE);

    track.addNoteEvent(tick, new NoteEvent(NoteName.C5, false));

    return track;
}

```

Let's assume that we have a new test in which we could use `createSimpleTrack()` if the created object was just *slightly* different (e.g. it needs more `addNoteEvent()` calls). What do we do?

- Do we change `createSimpleTrack()`?
- Add another factory method?
- Or do we call `addNoteEvent(...)` in our test?

The first approach is the most dangerous one, as it could break other tests. The other approaches are... alright.

Test Data Builder

This approach is taken from the book "Working Effectively With Unit Tests (Jay Fields)". Our data builders:

```

public class a {

    public static NoteEventBuilder noteEvent = new NoteEventBuilder();
    public static TrackBuilder track = new TrackBuilder();

    public static class NoteEventBuilder {

        private NoteName noteName;
        private boolean noteOn;

        NoteEventBuilder() {
            this(NoteName.C4, true);
        }

        NoteEventBuilder(NoteName noteName, boolean noteOn) {
            this.noteName = noteName;
            this.noteOn = noteOn;
        }

        public NoteEventBuilder w(NoteName noteName) {
            return new NoteEventBuilder(noteName, noteOn);
        }

        public NoteEventBuilder w(boolean noteOn) {
            return new NoteEventBuilder(noteName, noteOn);
        }

        public NoteEvent build() {
            return new NoteEvent(noteName, noteOn);
        }
    }

    public static class TrackBuilder {

        private MusicalKey key;
        private MusicalInstrument instrument;

        TrackBuilder() {
            this(MusicalKey.VIOLIN, MusicalInstrument.ACOUSTIC_GRAND_PIANO);
        }

        TrackBuilder(MusicalKey key, MusicalInstrument instrument) {
            this.key = key;
            this.instrument = instrument;
        }

        public TrackBuilder w(MusicalInstrument instrument) {
            return new TrackBuilder(key, instrument);
        }

        public TrackBuilder w(MusicalKey key) {
            return new TrackBuilder(key, instrument);
        }

        public Track build() {
            return new Track(key, instrument);
        }
    }
}

```

And our refactored tests:

```

public class TrackTest extends TestCase {

    public void testAddNoteEvent() {
        Track track = a.track.build();
        NoteEvent noteEvent = a.noteEvent.build();

        track.addNoteEvent(0, noteEvent);

        assertEquals(1, track.size());
    }

    public void testToString() {
        Track track = a.track.build();
        String expectedString = "[Track] instrument=" + track.getInstrument()
            + " key=" + track.getKey()
            + " size=" + track.size();

        assertEquals(expectedString, track.toString());
    }
}

```

Note: Using these builders may look a bit unusual. Other people like to write these builders in a different way. Check out [this link](#) for another example.

Aside from the uncommon syntax, these tests look very similar to the previous version using factories. Builders help us if we need to configure our objects in different ways in test cases. We could do some pretty convenient things, for example:

```

public void testNotEqual() {
    Track track1 = a.track.w(MusicalKey.VIOLIN).build();
    Track track2 = a.track.w(MusicalKey.BASS).w(MusicalInstrument.BANJO).build();

    assertFalse(track1.equals(track2));
}

```

Note: Builders really start to shine if we can nest them together, though it might not be beneficial in our current example. So just to showcase how it could look like without adding the actual implementation:

```

// this is similar to calling addNoteEvent(...) twice
Track track = a.track
    .w(0, a.noteEvent.w(NoteName.C4, true))
    .w(100, a.noteEvent.w(NoteName.C4, false))
    .build();

```

Conclusion

So which approach should we choose? Well, both have their up- and downsides. It depends on the actual use case, so let's summarize both strategies:

- Both offer an easy way to create objects in tests
- Both hide irrelevant information through default values
- Both can be customized easily
- **Test Data Factories**
 - Need separate methods to create special cases
- **Test Data Builders**
 - Need more upfront code for a basic implementation (without nesting)
 - Become more powerful if we can nest builders

The next article will talk about testing private components!

Resources

- [Book: Working Effectively With Unit Tests \(Jay Fields\)](#)
- [Martin Fowler: Object Mother](#)
- [Test Data Builders and Object Mother](#)

[Testing] #6 Testing Private Components

Should we test private methods or access private fields? The short answer is nope, we should not. Here's why:

Reflection

Testing private components can be done using our own test utils class [Reflection](#). This is for example done in the [LookTest](#) class by using `Reflection.invokeMethod(...)` and `Reflection.getPrivateField(...)`.

This approach is more error prone than a direct method call - let's look at a snippet taken from the `testBreakDownCatroidAngle()` test to see this in action:

```
public void testBreakDownCatroidAngle() {
    // some code here
    ParameterList params = new ParameterList(posigiveInputAngles[index]);
    float catroidAngle = (Float) Reflection.invokeMethod(look,
"breakDownCatroidAngle", params);
    // more code here
}
```

This snippet calls a method named `breakDownCatroidAngle(...)` which takes some parameters and returns a `float`. This code compiles without errors, even if we change the signature of the `breakDownCatroidAngle(...)` method (e.g. by changing the parameter list). We would only find out that something has changed if we execute the test. This unnecessary long feedback delay is really inconvenient.

But Why?

Why would we want to test private methods in the first place? We could think of two reasons: If a private method is either complicated and long, or if it performs some special operations for a set of parameters which are not provided by other (public) methods that call it. This could happen if for example this private method performs an algorithm with some special cases which we are currently not using.

Let's look at two common approaches to avoid private method testing:

Test Via The Public Interface

Here's a trivial example:

```
public class Main {

    public Money addMoney(Money first, Money second) {
        return new Money(Money.Type.EURO, add(first.getValue(), second.getValue()));
    }

    private int add(int first, int second) {
        return first + second;
    }
}
```

This example is pretty much pointless, but it should be *really* obvious that we can test `add(...)` through `addMoney(...)`. Every possible scenario that we would like to test considering `add(...)` can be done through our public method. We should not have to care about `add(...)` at all. The only thing that matters is that `addMoney(...)` is working as expected. That's all we need to do. If we can achieve that, we are done.

But sometimes testing the public interface doesn't cut it. Maybe the private method is big, or we are dealing with a series of methods that work together. Testing all this just feels wrong. These are the times where we have to take a deep breath and ask ourselves:

Do We Need Another Class?

Complicated private methods start to gather if we are mixing several responsibilities in one class. Writing tests for each different case that we have to deal with becomes tedious.

These are some examples that could indicate that we need a new class:

- The class contains methods that only operate on its parameters
- Groups of methods access only a subset of class members
- The class "feels large"

Note: Read more about classes that do too much [here](#) (Flaw #4)!

Here is a simple example of a class which tries to do too much:

```
public class Trainer {

    private String name;
    private Map<Item, Integer> inventory;

    public Trainer(String name) {
        this.name = name;
        this.inventory = new HashMap<>();
    }

    public String sayHello() {
        return "Hi, my name is " + name + "!";
    }

    public void buyItem(Item item, int count) {
        for (int i = 0; i < count; i++) {
            addItem(item);
        }
    }

    private void addItem(Item item) {
        int count = 0;

        if (inventory.containsKey(item)) {
            count = inventory.get(item);
        }

        count++;

        inventory.put(item, count);
    }

    public void useItem(Item item) {
        if (getItemCount(item) <= 0) {
            return;
        }

        item.use();
        removeItem(item);
    }

    private void removeItem(Item item) {
        if (getItemCount(item) <= 0) {
            return;
        }

        int count = inventory.get(item);
        count--;

        inventory.put(item, count);
    }

    private int getItemCount(Item item) {
```

```
        if (inventory.containsKey(item)) {
            return inventory.get(item);
        }

        return 0;
    }
}

public enum Item {
    POKEBALL, HEALTH_POTION, CANDY;

    public void use() {
```

```
        // more code here
    }
}
```

We are dealing with two clusters of methods:

- One method that accesses a Trainer's name
- Three methods that work with inventory

Here's what we could do:

```
public class Trainer {

    private String name;
    private Inventory inventory;

    public Trainer(String name, Inventory inventory) {
        this.name = name;
        this.inventory = inventory;
    }

    public String sayHello() {
        return "Hi, my name is " + name + "!";
    }

    public void buyItem(Item item, int count) {
        for (int i = 0; i < count; i++) {
            inventory.addItem(item);
        }
    }

    public void useItem(Item item) {
        if (inventory.getItemCount(item) <= 0) {
            return;
        }

        item.use();
        inventory.removeItem(item);
    }
}

public class Inventory {

    private Map<Item, Integer> inventory;

    public Inventory() {
        this.inventory = new HashMap<>();
    }

    public void addItem(Item item) {
        int count = 0;

        if (inventory.containsKey(item)) {
            count = inventory.get(item);
        }

        count++;

        inventory.put(item, count);
    }

    public void removeItem(Item item) {
```

```
    if (getItemCount(item) <= 0) {
        return;
    }

    int count = inventory.get(item);
    count--;

    inventory.put(item, count);
}

public int getItemCount(Item item) {
    if (inventory.containsKey(item)) {
        return inventory.get(item);
    }
}
```

```
        return 0;
    }
}
```

By splitting `Trainer` into `Trainer` and `Inventory` we could change the visibility of previous private methods. In addition to that, dependency injection gives us an easier time to test `useItem(...)` and `buyItem(...)` through `Inventory`. For example:

```
@Test
public void useItem() {
    Inventory inventory = new Inventory();
    Item item = Item.POKEBALL;
    Trainer trainer = new Trainer(null, inventory);

    inventory.addItem(item);
    trainer.useItem(item);

    assertEquals(0, inventory.getItemCount(item));
}
```

Note: It is totally fine to use a "real" `Inventory` object instead of a test double here. What we didn't test though was if `item.use()` was called through `trainer.useItem(...)`. This is an example where we would have to use a stub or a mock, because `Item` has no method that would give us information about this.

But What If I Really Have To Do It?

If you really, *really* need to test private components (e.g. when dealing with legacy code) you could change the visibility of the method to either `protected` or `default`.

Conclusion

Alright, in this article we've talked about strategies to avoid testing private components while improving our overall testing suite:

- Test private methods through their public interface, or
- Create additional classes to split responsibility

Next up: Working with other classes (collaborators)!

Resources

- [Blog: Unit Testing Private Methods](#)
 - The author later on changed his mind due to some reader comments. Checkout out **EDIT 2** and the comments at the end!
- [Misko Hevery: Flaw #4 - Class Does Too Much](#)

[Testing] #7 Working With Collaborators

Today we are going to talk about working with other classes. Let's start with a snippet taken from `StorageHandler`:

```
public File copyImageFromResourceToCatroid(Activity activity, int imageId, String
defaultImageName) throws IOException {
    Bitmap newImage =
    BitmapFactory.decodeResource(activity.getApplicationContext().getResources(),
imageId);
    String projectName =
    PackageManager.getInstance().getCurrentProject().getName();
    return createImageFromBitmap(projectName, newImage, defaultImageName);
}
```

Don't Dig

The above method requires a complicated test as it depends on several concepts including an `Activity` and file operations. We have already talked about how we can write code that can be tested without crossing boundaries, so instead let's focus on these two lines in particular:

```
Bitmap newImage =
BitmapFactory.decodeResource(activity.getApplicationContext().getResources(),
imageId);
String projectName = PackageManager.getInstance().getCurrentProject().getName();
```

These lines are examples where we dig into collaborators to access their data. Notice that we don't need either an `activity` nor a `project` for anything else in this method. We only care about `projectName` and `resources`. So what we could do instead is this:

```
public File copyImageFromResourceToCatroid(Resources resources, int imageId, String
defaultImageName, String projectName) throws IOException {
    Bitmap newImage = BitmapFactory.decodeResource(resources, imageId);

    return createImageFromBitmap(projectName, newImage, defaultImageName);
}
```

Note: If you have read the book [Clean Code \(Robert C. Martin\)](#) (if not, I recommend that you do) you might think that this method just became worse, as four parameters is a lot. In reality though these parameters did already exist in an implicit form in the method. All we did was to make it explicit.

We don't really get much from our refactoring work as we cannot simply instantiate an instance of Android's `Resources` class without an `activity`. But let's imagine for a second that we could and that this would work out totally fine with all following file operations that `copyImageFromResourceToCatroid(...)` includes. Putting this method in a test environment became easier by a mile for two reasons:

- We don't need to create an `Activity` (at least in our pretend game)
- We don't have to care about creating a `project` or about setting this `project` as the `currentProject` in our `Singleton`

In other words we would get rid of writing this boilerplate test code:

```
Activity activity = ...;
Project project = ...;
PackageManager.getInstance().setProject(project);
```

Note: Currently `StorageHandlerTest` doesn't include a single test method about images/bitmaps. Reasons for this could be that:

- Creating a test fixture (meaning all objects and operations that we need in order to start the test) involves a lot of moving parts
- Verifying test results is tedious as we have to load and check a written image from memory
- We have to perform manual clean up after these tests so that they won't become flaky

Lying API

Let's have a look at the `ServiceProvider` class. This is essentially a `Map` that can be used to access classes which implement `CatroidService`. Service providers are also commonly known as service locators or service containers. This particular class is static, which means that it comes with a series of global state problems, but for the sake of today's topic, let's imagine that we could simply instantiate a service provider:

```
ServiceProvider provider = new ServiceProvider();
```

This means that it could be used like this:

```
public class SomeService implements CatroidService { // some code }

public class SomeServiceWorker {

    private SomeService service;

    public SomeServiceWorker(ServiceProvider serviceProvider) {
        this.service = serviceProvider.getService(SomeService.class);
    }

    public void doStuff() {
        // some code which uses the service member
    }
}
```

We are using dependency injection, which means that testing this is straightforward, right? If we only look at the constructor signature of `SomeServiceWorker` we would think that we could write this:

```
public void testSomething() {
    ServiceProvider serviceProvider = new ServiceProvider();
    SomeServiceWorker worker = new SomeServiceWorker(serviceProvider);

    worker.doStuff();

    // assert something
}
```

Running this test will blow up with a `NullPointerException` because we didn't add an instance of `SomeService` to our service provider. How should we have known? The constructor signature didn't provide this information. Here's what we should do instead:

```
public class SomeServiceWorker {

    private SomeService service;

    public SomeServiceWorker(SomeService service) {
        this.service = service;
    }

    public void doStuff() {
        // some code which uses service
    }
}
```

This way our API becomes more transparent. Instead of using trial and error test runs or looking up the constructor body for more details, we can simply inspect the signature of `SomeServiceWorker` to gain all the information we need to instantiate an instance in our test:

```
public void testSomething() {
    SomeService service = new SomeService();
    SomeServiceWorker worker = new SomeServiceWorker(service);

    worker.doStuff();

    // assert something
}
```

Conclusion

Setting up a test case becomes easier if we only ask for collaborators that we really need in our production code. You can find more examples of how to work with collaborators [here](#).

Next week we will talk about details on using test doubles (stubs and mocks)!

Resources

- [Clean Code \(Robert C. Martin\)](#)
- [Misko Hevery: Flaw #3 - Digging into Collaborators](#)

[Testing] #8 More About Mocks And Stubs

Alright, today we are going to talk more about using test doubles. Here's a small code example that we have used in the past:

Note: Skip the following two code examples if you already know the basics about stubs and mocks. We won't need them for anything else!

```
public class Car {

    private Engine engine;

    public Car(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        engine.start();
    }
}

public class EngineStub implements Engine {

    private boolean started;

    public void start() {
        started = true;
    }

    public boolean isStarted() {
        return started;
    }
}

public interface Engine {

    public void start();
}

public class CarTest extends TestCase {

    public void testEngineStarted() {
        EngineStub engine = new EngineStub();
        Car car = new Car(engine);

        car.start();

        assertTrue(engine.isStarted());
    }

    public void testEngineNotStarted() {
        EngineStub engine = new EngineStub();
        Car car = new Car(engine);

        assertFalse(engine.isStarted());
    }
}
```

These tests would look like this if we would use a mocking framework like [Mockito](#) instead:

```

public class CarTest extends TestCase {

    public void testEngineStarted() {
        Engine engine = Mockito.mock(Engine.class);
        Car car = new Car(engine);

        car.start();

        verify(engine).start();
    }

    public void testEngineNotStarted() {
        Engine engine = Mockito.mock(Engine.class);
        Car car = new Car(engine);

        verify(engine, never()).start();
    }
}

```

We can identify two major differences if we compare both test classes:

- While stubs need classes like `EngineStub`, mocks rely on `Mockito.mock(<SomeClass>.class)` calls
- Tests with mocks use `verify(...)` while tests with stubs use `assert(...)`

[Martin Fowler](#) talks a lot more about mocks and stubs, which is why we are focusing on some general practices instead. The following two statements only mention mocks, but these principles can also be applied to stubs!

Don't Mock What You Don't Own

Let's look at an example which does **not** comply with this statement:

```

public void testDirectory() {
    File directory = Mockito.mock(File.class);
    when(directory.isDirectory()).thenReturn(true);
    FileWorker worker = new FileWorker();

    assertEquals("This is a directory!", worker.callFile(directory));
}

public void testFile() {
    File directory = Mockito.mock(File.class);
    when(directory.isDirectory()).thenReturn(false);
    FileWorker worker = new FileWorker();

    assertEquals("This is a file!", worker.callFile(directory));
}

```

Note: JUnit 4 offers some *tricks* to limit the amount of duplication we have in cases like these, but they are really clunky. That's why we are sticking with the above style for now.

These tests should be rather straightforward. We expect either *"This is a directory!"* or *"This is a file!"* if we call `worker.callFile(...)`. The issue with this snippet is that we are using a mock of `File`. The [Mockito Wiki](#) mentions a few reasons why this is a problem.

We've already talked about splitting our code into three zones (happy zone, demilitarized zone, bad outside world) in a previous post. Using this approach avoids mocking types that we don't own in the first place!

Mocks Which Return Other Mocks

Whenever we create test doubles which return test doubles, we should take a step back and think about what we are doing. Let's look at an

example using one of these previously mentioned (nasty) service locators:

```
public void testUseService() {
    ServiceLocator locator = Mockito.mock(ServiceLocator.class);
    FancyService service = Mockito.mock(FancyService.class);
    // potential more FancyService configuration code here
    when(locator.getService("FancyService")).thenReturn(service);
    ServiceUser user = new ServiceUser(locator);

    user.doWork();

    verify(service).someServiceMethod();
}
```

This test creates a service locator mock which contains yet another service mock. We provide the locator to `ServiceUser` and verify that it calls the `someServiceMethod()`. This doesn't seem too bad, but nesting mocks can become complicated very quickly. We could get around this nested approach simply by having `ServiceUser` take an instance of `FancyService` instead of `ServiceLocator`.

Nesting test doubles cannot always be avoided, e.g. we need them when we pass some factory class instance to a constructor or a method:

```
SomeObject object = Mockito.mock(SomeObject.class);
SomeObjectFactory factory = Mockito.mock(SomeObjectFactory.class);
when(factory.createSomeObject()).thenReturn(object);
```

This code block looks very similar to the service locator example, but in this case there's not much we can do about it.

Conclusion

Here's the takeaway from working with test doubles:

- Don't mock what you don't own
- Be careful when using mocks which return mocks

Resources

- [Mockito](#)
- [Martin Fowler](#)
- [Don't mock what you don't own](#)

[Testing] #9 Wrapping It All Up

Here's a summary of the most important topics we have talked about regarding testing. Keep these points in mind to ease your unit testing experience:

Use Dependency Injection

Think hard about where you want to place your `new` operations. Learn to love Dependency Injection. It's our friend. We can't get rid of using `new` statements in our code, but separating object creation and object usage helps us to decrease coupling which in turn makes it easier to introduce test doubles.

Avoid Global State

Dealing with global state is frustrating. Singletons may seem like a good idea, but they will haunt us for a very very long time. Any kind of global state is a hard dependency which we cannot easily change in a testing environment. This makes tests slow and error prone in very weird ways (e.g. a test fails in a full test run, while it works fine if run on its own).

Expand The Happy Zone

Remember the three zones:

- **The happy zone:** Code that our team has written
- **The bad outside world:** Code that was written by other (non-team) people
- **The demilitarized zone:** Code that was written by us which taps into the bad outside world

Keep the demilitarized zone as small as possible and don't ever call the outside world from within the happy zone! This approach speeds up test execution and makes it very easy to put a class in a testing environment without any outside dependencies (e.g. file system, network or databases).

Keep Code In Tests Relevant

Patterns like Test Data Builders or Test Data Factories (Object Mothers) help us to keep our tests short. Irrelevant information should be hidden as good as possible so that we can get all the important information as quickly as possible while reading a test.

Remember The Anatomy Of A Test

- Assembly
- Act
- Assert

Don't Try To Test Private Components

Most of the time there's an alternative to testing private methods or fields. Often this means that we have to introduce another class.

Stop Digging Into Classes

```
public void doStuff(SomeService service) {
    String name = someService.getElements().getCurrentElement().getName();

    // ...
}
```

Digging into a parameter makes it hard to put a method in a testing environment. Instead do this:

```
public void doStuff(String name) {  
    // ...  
}
```

Mocking

Remember two things when working with mocks:

- Think hard if a mock should return another mock. This could be an indicator of digging into a collaborator (see above)!
- Don't mock what you don't own. That's what the "demilitarized zone" is for. Mock these classes instead.

That's it

Alright, that's all I've got for you. I hope this helps. If you still have any questions, feel free to contact me! :)

[Testing] The Deep Synergy Between Testability And Good Design

I just stumbled over an interesting talk by [Michael Feathers](#). In it he talks about two observations:

- Improving design makes it easier to write tests
- Changing code so that we can create tests does not necessarily mean that we improve its design

Read that again, because the difference is subtle but critical: Having a test suite does not mean that a code base is well designed.

He also gives an overview of several testing pains and explains why they are problematic. Here's an incomplete list:

- **Hidden components:** You wish you could get access to private components while writing your test
- **Setting up the test is hard:** Creating an instance of your class under test or calling one of its methods involves an excessive amount of components
- **Leaky State:** Tests may change the result of other tests. They may fail if executed together, but work fine if run alone
- **Mocking is hard:** You end up writing mocks that return mocks that return mocks or you find it very hard to create a test double for a particular class
- **No access:** You cannot easily assert the behavior of a method because you have no access to its end result
- **Domino effect:** Many unit tests fail after performing code changes

Check out the video to learn more!

Appendix H.

Roadmap to Improve Pocket Code's Test Suite

This is an exported version of the Confluence article summarizing the roadmap presented in Section [5.5.2](#). Because of the export this text lacks all contained links.

[Testing] Roadmap: Refactoring And Redesigning Topics To Improve Pocket Code's Testing Situation

This post offers a closer look at the "Getting Practical" sections of the [testing article series](#).

Here is again a quick overview of the mentioned topics:

1. Use dependency injection
 - Reduce global state (Singleton)
 - Hide external dependencies behind wrappers
2. Test structure
 - Upgrade from JUnit version 3 to version 4 to use its improved Exception testing
 - Use the Create, Act, Assert structure when writing tests
3. Focus on Relevant Details
 - Use test doubles (e.g. mocking frameworks)
 - Use creational design patterns to create objects in tests

Where To Start?

The above list already outlines the order of how I would deal with these topics. Introducing dependency injection while also removing global state is the most important, but also most time consuming task. Starting with any other task would most likely lead to extra work.

How Long Will This Take?

Doing all these changes could take weeks or even months. Here's why:

- Introducing dependency injection is a complex and time consuming task. Some existing singletons are used all over the place, which makes it hard to identify potential side effects
- The current test suite is incomplete, which could make it hard to verify changes
- Pocket Code's senior member count is currently rather low
- The normal development cycle would be effected by larger refactoring and redesigning work

Appendix I.

How Git Handles Commits

This section explains how the proposed single branch approach in Chapter 6.4.2 can contain the same information as a two branch approach.

Commits and branches in git internally function like this:

- Each commit knows its predecessor but not its successor commit through a pointer
- A branch is yet another pointer which points on a commit
- A commit is "lost" if it has no incoming pointer (either through another commit or through a branch)

This means that after merging one branch A into another branch B, the commit history of A is now reachable by branch B as well (Git, 2016). Figure I.1 summarizes these technical details in an example: While the branch "feature#38" only has access to three commits, the "master" branch has access to all four commits after the merge. Deleting the branch "feature#38" only removes its pointer, which has no further impact on the commit graph as "master" still points to all other commits.

Appendix I. How Git Handles Commits

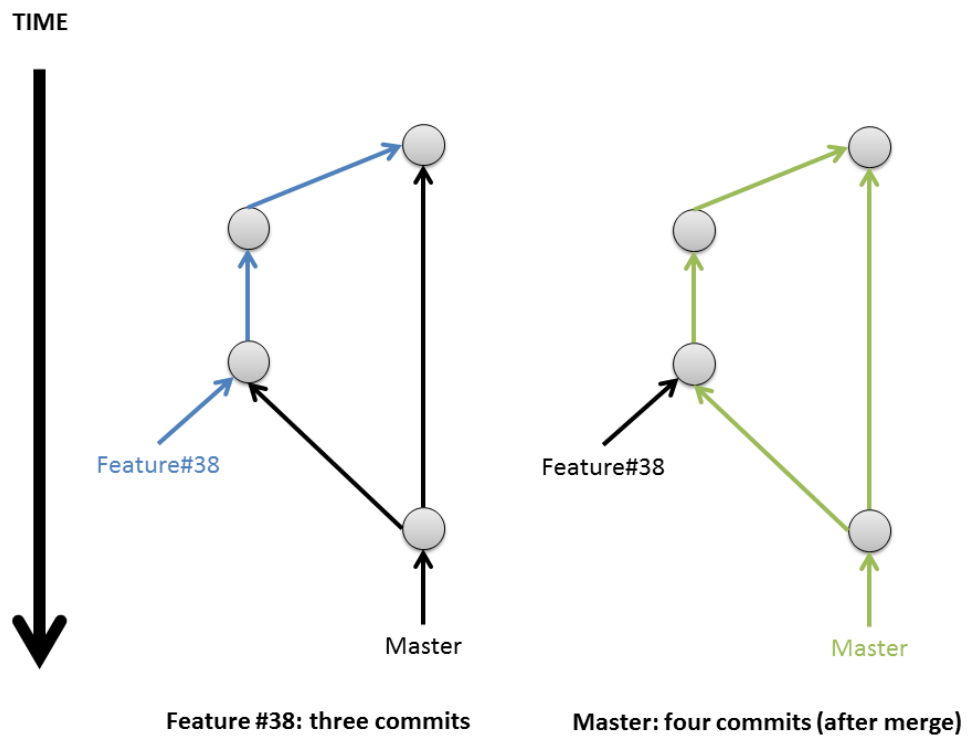


Figure I.1.: View on how commits and branches are handled by git internally

Appendix J.

Confluence Decision Log: Git Workflow

This is an exported version of the Confluence decision log concerning the proposed git workflow change mentioned in Section [6.4.2](#). Due to the export this text lacks all contained links.

Git Workflow: Remove Master Branch

Status	NOT STARTED
Stakeholders	Thomas Schranz Thomas Hirsch Robert Painsi Thomas Mauerhofer Bernadette Spieler
Outcome	
Due date	
Owner	Florian Winkelbauer

Background

We have had problems in the past with our git workflow concerning the develop and the master branch: Scenarios occurred in which we could not merge the content of the develop branch into the master branch.

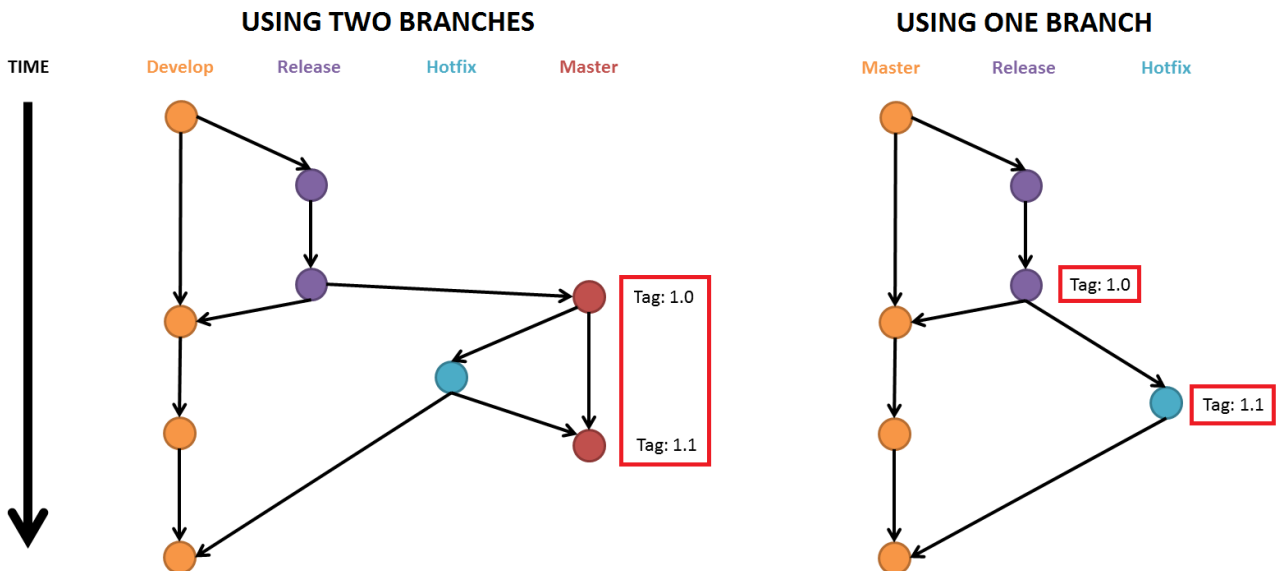
I have suggested a workflow change which only uses one branch instead of two. This change has not been implemented yet as past problems were neither frequent nor severe. The purpose of this decision log is to summarize all information concerning this topic in case it becomes relevant in the future. The idea for this topic came from this [blog post](#).

Current State: Two Branches

Our current git workflow is based on [Gitflow](#) which is why we are using two branches to separate ongoing development and finished releases. The advantage of the current approach is that we have a single branch (master) in which each commit represents a releasable state of Pocket Code. The disadvantage is that any hotfix has to be integrated into two branches, which was the reason for the above mentioned merge problems between master and develop: We don't know exactly why these merge issues occurred, but we are certain that this situation was the result of integrating a hotfix through a rebase.

Alternative: One Branch

The picture below shows the difference between the current and the proposed workflow:



On the left side each commit on the master branch is a new release, while on the right side this information is scattered in the commit tree. Overall both contain the same information. Git offers commands which can [list tagged commits](#) only, which could make a one branch approach even more convenient to use.

Action items

If we decide to change the workflow to only use a single branch we have to:

- Remove the master branch on GitHub, **or** merge all content of the develop branch in the master branch and delete the develop branch instead
- For all Jenkins jobs which are used for the release version of Pocket Code: change the git checkout step to use the latest **tagged** commit
- Educate team members about the change
- Change the relevant Confluence reading material to reflect this change

Bibliography

- Atkinson, R. and J. Flient (2001). "Accessing Hidden and Hard-to-reach Populations: Snowball Research Strategies." In: *Social Research Update* 33 (cit. on p. 25).
- Atlassian (2016a). *Kanban Board*. [Online; accessed: 06.04 16:33]. URL: <https://confluence.atlassian.com/agile/glossary/kanban-board> (cit. on p. 10).
- Atlassian (2016b). *Software Development and Collaboration Tools*. [Online; accessed 23.03.2016 17:20]. URL: <https://atlassian.com> (cit. on p. 11).
- Aurum, Aybuke, Håkan Petersson, and Claes Wohlin (2002). "State-of-the-art: software inspections after 25 years." In: *Softw. Test. Verif. Reliab.* 12.3, pp. 133–154. ISSN: 1099-1689. URL: <http://dx.doi.org/10.1002/stvr.243> (cit. on pp. 31, 36).
- Bacchelli, A. and C. Bird (2013). "Expectations, outcomes, and challenges of modern code review." In: *Software Engineering (ICSE), 2013 35th International Conference on*, pp. 712–721. DOI: [10.1109/ICSE.2013.6606617](https://doi.org/10.1109/ICSE.2013.6606617) (cit. on p. 34).
- Beck, K. and C. Andres (2004). *Extreme Programming Explained: Embrace Change: Embracing Change*. Addison-Wesley Professional (cit. on pp. 7, 8).
- Beck, Kent (2002). *Test Driven Development: By Example*. Addison-Wesley Professional (cit. on pp. 3, 7, 8).
- Bernhart, M. and T. Grechenig (2013). "On the understanding of programs with continuous code reviews." In: *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pp. 192–198. DOI: [10.1109/ICPC.2013.6613847](https://doi.org/10.1109/ICPC.2013.6613847) (cit. on p. 33).
- Bosu, A., M. Greiler, and C. Bird (2015). "Characteristics of Useful Code Reviews: An Empirical Study at Microsoft." In: *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pp. 146–156. DOI: [10.1109/MSR.2015.21](https://doi.org/10.1109/MSR.2015.21) (cit. on pp. 3, 31, 33, 86).

Bibliography

- Brykczynski, Bill (1999). "A Survey of Software Inspection Checklists." In: *SIGSOFT Softw. Eng. Notes* 24.1, pp. 82–. ISSN: 0163-5948. DOI: 10.1145/308769.308798. URL: <http://doi.acm.org/10.1145/308769.308798> (cit. on p. 38).
- Catrobat (2016). *Catrobat*. [Online; accessed 04.06.2016 14:19]. URL: <http://www.catrobat.org/> (cit. on p. 5).
- Christian Hofer, BSc (2014). "Behaviour Driven Web Development." MA thesis. Graz University of Technology (cit. on p. 92).
- Cucumber (2016). *Cucumber*. [Online; accessed 08.12.2016 16:13]. URL: <https://cucumber.io/> (cit. on p. 92).
- Driessen, Vincent (2010). *A successful Git branching model*. [Online; accessed 14.04.2016 11:04]. URL: <http://nvie.com/posts/a-successful-git-branching-model/> (cit. on pp. 3, 66).
- Driessen, Vincent (2012). *Gitflow Extensions*. [Online; accessed 23.10.2016 13:39]. URL: <https://github.com/nvie/gitflow> (cit. on p. 78).
- Dunsmore, A., M. Roper, and M. Wood (2002). "Further investigations into the development and evaluation of reading techniques for object-oriented code inspection." In: *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pp. 47–57. DOI: 10.1145/581348.581349 (cit. on p. 35).
- Dunsmore, A., M. Roper, and M. Wood (2003). "The development and evaluation of three diverse techniques for object-oriented code inspection." In: *Software Engineering, IEEE Transactions on* 29.8, pp. 677–686. ISSN: 0098-5589. DOI: 10.1109/TSE.2003.1223643 (cit. on p. 37).
- Espresso (2016). *Espresso*. [Online; accessed 10.11.2016 13:49]. URL: <https://google.github.io/android-testing-support-library/> (cit. on p. 20).
- Fagan, M. E. (1976). "Design and Code Inspections to Reduce Errors in Program Development." In: *IBM Syst. J.* 15.3, pp. 182–211. ISSN: 0018-8670. DOI: 10.1147/sj.153.0182. URL: <http://dx.doi.org/10.1147/sj.153.0182> (cit. on p. 31).
- Fagerholm, Fabian et al. (2014). "The Role of Mentoring and Project Characteristics for Onboarding in Open Source Software Projects." In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '14. Torino, Italy: ACM, 55:1–55:10. ISBN: 978-1-4503-2774-9. DOI: 10.1145/2652524.2652540. URL: <http://doi.acm.org/10.1145/2652524.2652540> (cit. on p. 3).

- Feathers, Michael (2010). *The Deep Synergy Between Testability and Good Design*. [Online; accessed 24.11.2016 11:45]. URL: <https://www.youtube.com/watch?v=4cVZvoFGJTU> (cit. on pp. 57, 63).
- Fowler, Martin (1999). *Refactoring: Improving the Design of Existing Code*. Addison Wesley (cit. on p. 7).
- Fowler, Martin (2010). *Feature Toggles*. [Online; accessed 04.06.2016 17:21]. URL: <http://martinfowler.com/bliki/FeatureToggle.html> (cit. on p. 76).
- Git (2016). *Git*. [Online; accessed 22.03.2016 19:53]. URL: <https://git-scm.com/> (cit. on pp. 13, 15, 165).
- GitHub (2016a). *A whole new GitHub Universe: announcing new tools, forums, and features*. [Online; accessed 23.09.2016 10:43]. URL: <https://github.com/blog/2256-a-whole-new-github-universe-announcing-new-tools-forums-and-features> (cit. on p. 41).
- GitHub (2016b). *GitHub*. [Online; accessed 22.03.2016 19:28]. URL: <https://github.com/> (cit. on p. 17).
- GitHub (2016c). *Pocket Code's build.gradle file*. [Online; accessed 18.11.2016 12:01]. URL: <https://github.com/Catrobat/Catroid/blob/develop/catroid/build.gradle> (cit. on p. 85).
- GitHub (2016d). *Pull Request of JIRA Ticket CAT-1717*. [Online; accessed 18.11.2016 11:29]. URL: <https://github.com/Catrobat/Catroid/pull/1873> (cit. on p. 85).
- Google (2015). *Google*. [Online; accessed 04.06.2016 19:56]. URL: <http://googletesting.blogspot.co.at/2015/04/just-say-no-to-more-end-to-end-tests.html> (cit. on pp. 51, 52).
- Gousios, Georgios, Margaret-Anne Storey, and Alberto Bacchelli (2016). "Work Practices and Challenges in Pull-based Development: The Contributor's Perspective." In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, pp. 285–296. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884826. URL: <http://doi.acm.org/10.1145/2884781.2884826> (cit. on p. 74).
- Gousios, G. et al. (2015). "Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective." In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1, pp. 358–368. DOI: 10.1109/ICSE.2015.55 (cit. on p. 73).
- Gradle (2016). *Gradle*. [Online; accessed 17.07.2016 09:43]. URL: <https://gradle.org/> (cit. on p. 20).

Bibliography

- Hatton, Les (2008). "Testing the Value of Checklists in Code Inspections." In: *Software, IEEE* 25.4, pp. 82–88. ISSN: 0740-7459. DOI: [10.1109/MS.2008.100](https://doi.org/10.1109/MS.2008.100) (cit. on pp. 36, 47).
- Hevery, M. (2008). *Guide: Writing Testable Code*. [Online; accessed 01.07.2016 10:01]. URL: <http://misko.hevery.com/code-reviewers-guide/> (cit. on p. 58).
- Hove, S.E. and B. Anda (2005). "Experiences from conducting semi-structured interviews in empirical software engineering research." In: *Software Metrics, 2005. 11th IEEE International Symposium*, pp. 10–23. DOI: [10.1109/METRICS.2005.24](https://doi.org/10.1109/METRICS.2005.24) (cit. on p. 23).
- Humble and Farley (2010). *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison Wesley (cit. on p. 91).
- Jenkins (2016). *Jenkins*. [Online; accessed 27.03.2016 15:36]. URL: <https://jenkins.io/> (cit. on p. 21).
- JUnit (2016). *JUnit*. [Online; accessed 10.11.2016 12:39]. URL: <http://junit.org/junit4/> (cit. on p. 17).
- Kochhar, P. S. et al. (2015). "Understanding the Test Automation Culture of App Developers." In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–10. DOI: [10.1109/ICST.2015.7102609](https://doi.org/10.1109/ICST.2015.7102609) (cit. on p. 55).
- Krusche, Stephan, Mjellma Berisha, and Bernd Bruegge (2016). "Teaching Code Review Management Using Branch Based Workflows." In: *Proceedings of the 38th International Conference on Software Engineering Companion. ICSE '16*. Austin, Texas: ACM, pp. 384–393. ISBN: 978-1-4503-4205-6. DOI: [10.1145/2889160.2889191](https://doi.org/10.1145/2889160.2889191). URL: <http://doi.acm.org/10.1145/2889160.2889191> (cit. on p. 34).
- Manuel Wallner, BSc (2014). "Specification by Example of the Broadcast Mechanism of Catrobat." MA thesis. Graz University of Technology (cit. on p. 92).
- Maximilian Fellner, BSc (2013). "Spezifikation einer visuellen Programmiersprache durch Beispiele." MA thesis. Graz University of Technology (cit. on p. 92).
- Phillips, Shaun, Jonathan Sillito, and Rob Walker (2011). "Branching and Merging: An Investigation into Current Version Control Practices." In: *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering. CHASE '11*. Waikiki, Honolulu, HI, USA:

- ACM, pp. 9–15. ISBN: 978-1-4503-0576-1. DOI: 10.1145/1984642.1984645. URL: <http://doi.acm.org/10.1145/1984642.1984645> (cit. on p. 75).
- Rainsberger, J. B. (2013). *Integrated Test Are A Scam*. [Online; accessed 10.07.2016 11:34]. URL: <https://vimeo.com/80533536> (cit. on p. 58).
- Raphael Sommer, BSc (2016). “Motivation in an Agile, Education, Free and Open Source Software Project.” MA thesis. Graz University of Technology (cit. on pp. 2, 92).
- Robotium (2016). *Robotium*. [Online; accessed 10.11.2016 12:15]. URL: <https://github.com/RobotiumTech/robotium> (cit. on p. 17).
- Rong, Guoping, Jingyi Li, et al. (2012). “The Effect of Checklist in Code Review for Inexperienced Students: An Empirical Study.” In: *Software Engineering Education and Training (CSEET), 2012 IEEE 25th Conference on*, pp. 120–124. DOI: 10.1109/CSEET.2012.22 (cit. on p. 35).
- Rong, Guoping, He Zhang, and Dong Shao (2014). “Investigating Code Reading Techniques for Novice Inspectors: An Industrial Case Study.” In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE '14. London, England, United Kingdom: ACM, 33:1–33:10. ISBN: 978-1-4503-2476-2. DOI: 10.1145/2601248.2601280. URL: <http://doi.acm.org/10.1145/2601248.2601280> (cit. on pp. 35, 36).
- Sato, D.T., H. Corbucci, and M.V. Bravo (2008). “Coding Dojo: An Environment for Learning and Sharing Agile Practices.” In: *Agile, 2008. AGILE '08. Conference*, pp. 459–464. DOI: 10.1109/Agile.2008.11 (cit. on p. 3).
- Scratch (2016). *Scratch*. [Online; accessed 04.06.2016 14:19]. URL: <https://scratch.mit.edu/> (cit. on p. 5).
- SonarQube (2016). *SonarQube*. [Online; accessed 30.12.2016 10:55]. URL: <https://www.sonarqube.org/> (cit. on p. 92).
- Swamidurai, R., B. Dennis, and U. Kannan (2014). “Investigating the impact of peer code review and pair programming on test-driven development.” In: *SOUTHEASTCON 2014, IEEE*, pp. 1–5. DOI: 10.1109/SECON.2014.6950664 (cit. on p. 33).
- Vasilescu, Bogdan et al. (2015). “Quality and Productivity Outcomes Relating to Continuous Integration in GitHub.” In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, pp. 805–816. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786850. URL: <http://doi.acm.org/10.1145/2786805.2786850> (cit. on pp. 55, 63).

Bibliography

- Veen, E. v. d., G. Gousios, and A. Zaidman (2015). "Automatically Prioritizing Pull Requests." In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 357–361. DOI: [10.1109/MSR.2015.40](https://doi.org/10.1109/MSR.2015.40) (cit. on pp. 74, 91).
- Wick, Michael, Daniel Stevenson, and Paul Wagner (2005). "Using Testing and JUnit Across the Curriculum." In: *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '05. St. Louis, Missouri, USA: ACM, pp. 236–240. ISBN: 1-58113-997-7. DOI: [10.1145/1047344.1047427](https://doi.org/10.1145/1047344.1047427). URL: <http://doi.acm.org/10.1145/1047344.1047427> (cit. on p. 56).
- Yang, H. Y., E. Tempero, and H. Melton (2008). "An Empirical Study into Use of Dependency Injection in Java." In: *19th Australian Conference on Software Engineering (aswec 2008)*, pp. 239–247. DOI: [10.1109/ASWEC.2008.4483212](https://doi.org/10.1109/ASWEC.2008.4483212) (cit. on pp. 55, 56).
- Yu, Y. et al. (2015). "Wait for It: Determinants of Pull Request Evaluation Latency on GitHub." In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 367–371. DOI: [10.1109/MSR.2015.42](https://doi.org/10.1109/MSR.2015.42) (cit. on p. 34).