Ajdin Vihric, BSc

Process Refactoring - Reintroduction of Agile Project Management Practices into an Agile (Distributed) Free Open Source Project

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Dipl-Ing. Dr.techn. Christian Schindler

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

Graz, May 2018

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.

Graz, ________________________ ____________________________

Date Signature

Abstract

Distributed teams have become a common practice in the field of software engineering. Free Open Source Software (FOSS) contributed to this effect as its developers community is decentralized. With the success of the Linux operating system the developers' interest to participate in FOSS projects and subsequently its community increased. Working with distributed teams required new management frameworks that are capable of combining new software development practices with traditional project management. Within this context several agile frameworks emerged over the last two decades. The present thesis studies the current research on agile frameworks and analyzes three commonly used agile project management frameworks in detail. It furthermore outlines practical applications of these methods with a special focus on the Catrobat project. Catrobat is a FOSS project founded at Graz University of Technology. Currently (2018) the project has more than 500 contributors and several released smartphone applications. The thesis concludes agile management practices within Catrobat and outlines possible improvements for future work.

Contents

List of Abbreviations

AG	Apache Group
APK	Android application package
ASD	Adaptive Software Development
ASF	Apache Software Foundation
DoD	Definition of Done
DSDM	Dynamic Systems Development Method
FDD	Feature Driven Development
FFS	Five Focusing Steps
FLOSS	Free/Libre Open Source Software
FOSS	Free Open Source Software
FSF	Free Software Foundation
GSoC	Google Summer of Code
IRC	Internet Relay Chat
LDAP	Local Directory Access Protocol
OOPSLA	Object Oriented Programming Systems, Languages, and Applications
OSI	Open Source Initiative
PO	Product Owner
RUP	Rational Unified Process
SM	ScrumMaster
TDD	Test Driven Development
TPS	Toyota Production System
WIP	Work in Progress
XP	eXtreme Programming

List of Figures

List of Tables

1 Introduction

Free Open Source Software (FOSS) is gaining ever more popularity and importance in today's IT world. Many systems, companies, projects and individuals are using free software or contributing to FOSS projects. Furthermore distributed teams, which are common in FOSS projects, require different management as well as development methods. Therefore new frameworks for software development and project management were born with increasing growth of FOSS. Amongst a variety of frameworks three popular ones distilled as the most widely used: Scrum, Kanban and eXtreme Programming (XP). A detailed description and analysis for using techniques out of these frameworks in a FOSS project are conducted below. This thesis analyzes practices of agile project management in the Catrobat FOSS project. Catrobat was founded at Graz University of Technology. It is a visual programming language designed for smartphones and tablets. The target group are children and teenagers. Although the idea of free software initiated by Richard Stallman is thirty years old, in-depth research on FOSS is small. In the case of combining a FOSS project in an educational environment the research is even sparse.

The main part of this thesis is an historical reappraisal of the Catrobat project. In particular agile methods and practices inside a sub-team within Catrobat are analyzed. Further an analysis of the ticketing system (Jira) and statistical data on various aspects around tickets is conducted. The analysis is complemented with an improvement process. In this process solutions and recommendations to further improve Catrobat's situation are given. Some solutions are already being implemented in the project. In the final section a reflection of FOSS and challenges in such are made. A special focus is laid on Catrobat. Therefore solutions evaluated in the improvement process are summarized. The thesis is concluded with a future outlook for the Catrobat project.

1.1 Thesis Outline

This thesis analyzes methods and processes of a distributed Free Open Source Software (FOSS) project in an educational environment. It is structured to give an overview of agile project management and software development frameworks. After that a concrete example of a FOSS project that started as a university project is given. Section 2 discusses a literature review on the field of FOSS in distributed teams. Section 3 explains agile software development methods and their benefits. The subsequent section 3.1 introduces *The Agile Manifesto*, which is an important rule set for agile software development. Sections 3.2 and 3.3 explain the Scrum and Kanban frameworks in detail. Section 3.4 explains eXtreme Programming (XP), a framework focused on development techniques, highlights the main differences to project management focused frameworks like Scrum and Kanban. Section 3.5 concludes with a detailed summary on FOSS. A practical analysis of a process introduction in a FOSS project is discussed in section 4. First a historical appraisal of the Catrobat project is done in sections 4.2 and 4.3. Afterwards an explorative analysis with Jira[1] is presented in section 4.4. Solutions and suggestions for findings of section 4.4 are described in section 4.5. Section 5 summarizes the findings and proposed improvements for a successful process refactoring. The final section 6 reflects on challenges that a FOSS project can face and gives possible outlooks for future work on this field. In particular challenges of the Catrobat FOSS projects are reflected.

[1] Tracking and planning software by Atlassian - https://www.atlassian.com/software/jira

2 Related Work

There are many books and papers describing the pro and cons of FOSS. While many give a rough overview how a FOSS project works, only a small hand of researchers do an in-depth analysis how the collaboration within FOSS projects is exactly done (Bonaccorsi and Rossi, 2003). Crowston and Howison, 2006 for exampleanalyze hierarchical structures, centralization and social structures in FOSS teams. A hierarchical structure usually consists of a bottom up system. This means that clear roles and division of labor exist in such a system. However, in FOSS teams this situation is slightly changed. Hierarchies are less strict in a FOSS project, where open development is encouraged. Basically, anyone who is willing to help can contribute to the code base. Crowston and Howison, 2006 write about development centralization, which means "who [actually] writes the code". According to their analysis of the *Apache httpd* project, about fifteen developers (the core team) produce 80 percent of the code base. Mockus, Fielding, and Herbsleb, 2002, Ghosh and Prakash, 2000 and Koch and Schneider, 2002 come to similar conclusions as observed by Bonaccorsi and Rossi, 2003. Now one may ask how come that all the volunteers are developing for Open Source projects without financial remuneration. Bonaccorsi and Rossi, 2003 try to give an answer to this question. They analyze three main questions in their work. First why developers contribute to Open Source projects. Second, how does the coordination of several hundred of people distributed around the globe work. Third, the reasons why Open Source software is becoming more popular, even though proprietary software dominates the market. To answer the first question it is necessary to know what kind of persons the contributing developers are. Bonaccorsi and Rossi, 2003 identify three groups of people using Open Source software. The first and largest group are the users of the software. This group does not contribute any code, but helps to propagate the software. The second group consists of developers, who contribute small portions in their free time. However the real success and increased popularity of Open Source software has its roots somewhere else. This third group consists of developers, who emerged from the hacker culture. The best example is given in Raymond, 1999's analysis of the Linux operating system. In his analysis Raymond defines what he calls "Linus' law". Linus' law means that a problem is solved quicker when more developers work on it. Linux is developed by a large number of people distributed all around the world. Many of these developers are hackers, who are often called the "real programmers". In the 1980's, the beginning of the hacker culture, programmers came from the engineering and physics fields. Bonaccorsi and Rossi, 2003 say that many developers see the Open Source community similar to academic research. In research sharing is a core part to gain valuable feedback and hence recognition in the research community. The same applies in Open Source. Some developers as well as authors (Ullman, 1998) describe Open Source as an art. That is one reason why Open Source software is often written elegantly. The developers who are committed try to write efficient and elegant code. The second research question comes to the result, similarly like Crowston and Howison, 2006, that hierarchies in Open Source projects do exist. Nevertheless, these hierarchies are less strict than in proprietary projects. The third research question concludes that the success of Open Source in large parts comes from companies. Many companies realized that they can benefit from adopting and developing Open Source software. Bonaccorsi and Rossi, 2003 call that a hybrid business model. Companies can use Open Source software in their proprietary products like some libraries. In return the companies contribute to Open Source projects, as they have a direct benefit of doing so. Crowston and Howison, 2006 studied distributed teams that have been successfully applied to different large and complex projects. In this context reference is made to the projects *Madefast* (Cutkosky, Tenenbaum, and Glicksman, 1996), a project for collaboration over the internet, and the software development performed in the Linux kernel described by Moon and Sproull, 2000. Crowston and Howison, 2006 furthermore highlight possible problems of software development arising in these projects which are exacerbated within distributed environments. Within this context, reference is made to Armstrong and Cole, 2002 who outlined possible techniques that counter these problems. Exemplary, project managers may consider using practices that promote mutual understanding. This is exactly done in the frameworks described in Section 3. Mockus, Fielding, and Herbsleb, 2002 state that Open Source development has the potential to compete and in some cases even replace traditional development methods like the Waterfall model described by Royce, 1987. Within this context they analyze the two Open Source projects Apache and Mozilla. To draw conclusions they compared these projects to five commercial projects and among each other. Apache was the de facto standard software for web servers on the internet when Mockus, Fielding, and Herbsleb, 2002's paper was published. This continues to be the case today, but several competitors like nginx[1] and Microsoft[2] have caught up. According to Netcraft[3] nginx could overtake both, Microsoft and Apache in the next few years. The fact that Apache is free software has certainly contributed to its success. The Apache Group (AG), the initial name of the organization around the Apache web server, started with eight people. Each of the eight founders were volunteers with a regular job. Due to limited time they could dedicate for the project, a development process that supports asynchronous communication was required. In this case a mailing list was the system of choice. In 1999 the AG was incorporated to the Apache Software Foundation (ASF) and hosts now more than 350 Open Source projects (*The Apache Software Foundation (ASF)*), while the Apache HTTP web server is the most popular one. Mozilla, the second Open Source project, differs slightly however. Unlike in Apache, where a release manager for a certain release is a volunteer from the core team, in Mozilla there are rather predefined roles. This has two main reasons. First Mozilla is a much larger project, consisting of many sub projects. The sub projects are called modules. Every owner of a module is responsible for reviewing and

[1] https://nginx.org/

[2] https://www.iis.net/

[3] https://news.netcraft.com/archives/2018/01/19/january-2018-web-server-survey.html - accessed on 2018-04-28

approving new code for the module. Second, Mozilla, as being a so-called hybrid project, is predestined to implement a mixture of open source and commercial development processes. A hybrid project has commercial and Open Source workflows. Mockus, Fielding, and Herbsleb, 2002 furthermore analyze the development processes of Apache, Mozilla and the five commercial projects that they chose for their study. The commercial development process has a well-defined sequence of actions. A change request for a feature or problem passes a meticulous design process. Afterwards it is assigned to a developer in form of a modification request. The assignments are done by a supervisor, who delegates work according to developer availability and skills. Within Apache there is no single development process that defines the stages a code change passes through. However, some similarity to the commercial process can be observed. A problem is first discovered. Subsequenly a solution is being worked out. After that the changes are presented to other core developers for review. When the review process was successful, the changes are committed to the code repository. In contrast to the commercial case the work is not being assigned to developers like in the commercial case. Developers in Open Source projects tend to pick certain problems they want to work on. Usually these problems involve code sections they are familiar with or are experts in the field (Mockus, Fielding, and Herbsleb, 2002, p.10). The Mozilla development process is a mixture of commercial and Open Source style, like mentioned previously. Most of the core developers at Mozilla work full time and are paid (Mockus, Fielding, and Herbsleb, 2002, p.28). Therefore, a more defined process like in the commercial projects is in place. Reviews pass through two stages. First the owner of a module must approve a code change. Second a group of so-called super-reviewers (Mockus, Fielding, and Herbsleb, 2002, p.25) inspect the changes for ramifications to the entire system. Bug reports and feature request are handled via Bugzilla[4]. Reporters need to set up an account at Bugzilla in order to create a request. Additionally, there is a group in the Mozilla team who specialize in reporting defects (Mockus, Fielding, and Herbsleb, 2002, p.33), which can be compared to a testing team in commercial projects. Mockus, Fielding, and Herbsleb, 2002 develop seven

[4] A bug tracking tool - https://www.bugzilla.org/

hypotheses in their work, which are not further specified except for one:

- *Hypothesis 6: In successful open source developments, the developers will also be users of the software.*

They conclude that this hypothesis is especially true in the case of Mozilla. The developers of the Firefox browser, a Mozilla software, are very likely also users of that software. Hence, they can be considered as domain experts.

3 Frameworks for Agile Project Management

The term *Agile*, related to software development, first appeared in 2001 in the *Agile Manifesto* (*The Agile Manifesto*). The manifesto is the result of a meeting, whose participants were advocates of different software development processes. Details of the manifesto are discussed in Section 3.1. Software development in general consists of several different parts. The parts are grouped, among other things, into activities, models, methods and practices. Some practices like Test Driven Development (TDD) and Pair Programing are discussed in the forthcoming sections. Methods are structured in frameworks. Details of three specific frameworks are discussed in the upcoming sections. The term *Project Management* refers to the classic definition:

> "A project is a planned program of work that requires a definitive amount of time, effort, and planning to complete. Projects have goals and objectives and often must be completed in some fixed period of time and within a certain budget."
>
> — Layton, 2012

On the other hand the term Agile Project Management refers to Agile Software Development. There are many Agile software frameworks, but the focus is set to the most popular and widely used, which are Scrum, Kanban and XP. Scrum and Kanban focus on project management and continuous delivery of work. XP has its focus on practical software development skills, like *Pair Programming*, *Continuous integration* and *Testing*. Other agile frameworks including Crystal, Dynamic Systems Development

Method (DSDM), Feature Driven Development (FDD), Rational Unified Process (RUP), Adaptive Software Development (ASD) are described in detail in Abrahamsson et al., 2002. There are also frameworks that combine principles from different frameworks like Scrumban (Ladas, 2009), taking the best from Scrum and Kanban.

3.1 The Agile Manifesto

The Agile Manifesto was signed by seventeen people (Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland and Dave Thomas), who were "[r]epresentatives from Extreme Programming, SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, Pragmatic Programming, and others [...]" (*The Agile Manifesto*), of a meeting that took place in February 2001 in Utah. The manifesto consists of the following twelve principles (*The Agile Manifesto*):

> "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."
>
> "Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage."
>
> "Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale."
>
> "Business people and developers must work together daily throughout the project."

> “Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.”

> “The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.”

> “Working software is the primary measure of progress.”

> “Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.”

> “Continuous attention to technical excellence and good design enhances agility.”

> “Simplicity - the art of maximizing the amount of work not done - is essential.”

> “The best architectures, requirements, and designs emerge from self-organizing teams.”

> “At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.”

The meeting in Utah was not coincidental. It was incubated at an earlier meeting in Oergon a year before. Participants of that meeting were discussing about *light* software development methodologies. However nothing concrete was decided. Robert C. Martin finally kicked-off the Utah-meeting by sending an email and suggesting a conference to take place in February of 2001 (*The Agile Manifesto*). The resulting *manifesto* was then created in Utah and serves as a central pillar for agile software development.

3.2 Scrum

Mary Poppendieck describes Scrum in Schwaber, 2004's book with the following simple example: Consider traveling with an aircraft and a car. In an aircraft flight there are certain procedures and rules you have to follow to reach your destination. Starting with the pilot checking all precautions necessary before an aircraft departure and ending with the predefined landing runway. All the actions in between the flight are planned as well. For instance the flight duration and speed. Whereas when you travel by car, the only rules you have to follow are the traffic rules. You are are free to choose which route you will take to reach your destination. Time is also on your own disposal, you can stop the car at any time and take a pause. Scrum is like the car. It provides a small amount of rules and tools, but it does not provide every single step in the process of reaching the goal. Ken Schwaber and Jeff Sutherland presented Scrum at the Object Oriented Programming Systems, Languages, and Applications (OOPSLA) conference in 1995 (Schwaber, 1995) as a new project management tool for object oriented programming. Since then the global Scrum community, as well as the framework's popularity, grew constantly. According to Schwaber, 2004, p.1 Scrum is based on industrial process control theory. Scrum is a contemporary framework for managing complex projects and is most widely used in software development. The following section discusses the framework in more detail. Schwaber, 1995 and Lacey, 2012 serve as the primary literature for this section.

3.2.1 Structure of Scrum

Scrum is structured into roles, meetings or events and artifacts. In Scrum there are the following core responsibilities: Product Owner (PO), ScrumMaster (SM) and the development team. The PO acts as the representer of wishes and expectations of customers and stakeholders. Furthermore the PO decides about the development process like what and when is something developed and is also responsible for the success or failure of a project. The

SM is responsible to protect the development team from distractions and acts as a middle-person between the PO, customers, stakeholders and the development team. If the team gets stuck at any point, the SM is the first person to provide help. The PO and SM are working with the development team. The development team brings the Product Owners vision to life, by developing, testing and optimizing the code. This development process is done in so called *Sprints*. A Sprint is basically a task list that has to be completed until the next release. Depending on various factors such as team experience or project size, the typical Sprint length is between one and four weeks. The Scrum Guide (*Scrum Guide*) implies that during a Sprint:

- "No changes are made that would endanger the Sprint goal."
- "Quality goals do not decrease."
- "Scope may be clarified and renegotiated between the PO and development team as more is learned."

Another important pillar of the Scrum framework are meetings of which there are four kinds, namely planning meetings, the Daily Scrum meetings, Sprint review meetings and Sprint retrospective meetings. Planning meetings, also called Sprint Planning, take place on the first day of each Sprint and are time boxed to a maximum of eight hours for a four week Sprint, shorter Sprints usually have a shorter time boxing. The plan is created collaboratively by the whole Scrum team. The Sprint Planning serves to answer two main questions (*Scrum Guide*):

1. "What amount of work can be done in the time frame of the upcoming Sprint?"
2. "How to get the Sprint's work done?"

The Daily Scrum meetings usually take place on a daily basis for about fifteen minutes and serve for team synchronization, as well as a quick planning session for the next twenty four hours. Daily Scrums are held at the same time and place to reduce complexity. Participants of the meeting explain the following (*Scrum Guide*):

- "What did I do yesterday that helped the development team meet the Sprint goal?"

- "What will I do today to help the development team meet the Sprint Goal?"
- "Do I see any impediment that prevents me or the development team from meeting the Sprint goal?"

The SM, who helps the team to keep the fifteen minute time frame, also ensures that only development team members participate in the Daily Scrum. The *Scrum Guide* summarizes the main advantages of the Daily Scrum. Due to an improvement in communication, the development team is capable of eliminating dispensable meetings. The project gains efficiency by quick decision making. Subsuming, the Daily Scrum improves the overall development team's knowledge. The Sprint Review and Sprint Retrospective take place on the last day of a Sprint. Sprint Reviews are an opportunity for customers to review the work so far and ask for changes, but they also give overall feedback to the team. As a result the Product Backlog is revised further, as the team gains more detailed knowledge. On the other hand the Retrospective serves the team to further improve their work by analyzing what went well and wrong in the Sprint. Primarily only the development team and the SM participate in the Retrospective. When there is need the PO can be invited to the Retrospective. In Scrum there are three kinds of artifacts which are crucial to success according to Lacey, 2012. First there is the Product Backlog which contains everything that needs to be done in order to successfully complete a project. It is constantly optimized by the PO who prioritizes and orders tasks. The Product Backlog displays the tasks with the attributes *description*, *order*, *estimate* and *value*. The description states the work that has to be done for this task. The order indicates how important a task is. The estimate gives an estimated effort needed to complete a task. The value depends on the usage of a product. A feature that has already been released and generates additional tickets enhancing or correcting the feature have more value than other tickets for instance. Cohn, 2009 and Pichler, 2010 use different attributes for the Product Backlog, that are summarized under the acronym DEEP. *Detailed Appropriately* is the first attribute. Tickets that are planned to be worked on the next Sprint need to contain enough detail to be delivered. *Estimated* is the second attribute. Tickets at the top of the Backlog are estimated more precise than the ones at the lower end.

Emergent is the third attribute and describes that the Backlog is changing accordingly to requirements. *Prioritized* means that the valuable tickets are at the top and less valuable at the bottom of the Backlog. Therefore the team will deliver maximum value by finishing tickets from the top of the Backlog. The Sprint Backlog is the second artifact type and is a list of tasks that are going to be addressed in the upcoming Sprint. It is constantly being refined during the Sprint, as the team gets to know more details about the work to be done. Even removing tasks is not uncommon in this phase, as the team is solely responsible for the Sprint Backlog. The third kind of artifact according to Lacey, 2012 is the Sprint Burn-Down chart. It is a graphical representation of the remaining work of the current Sprint (see Figure 3.1).

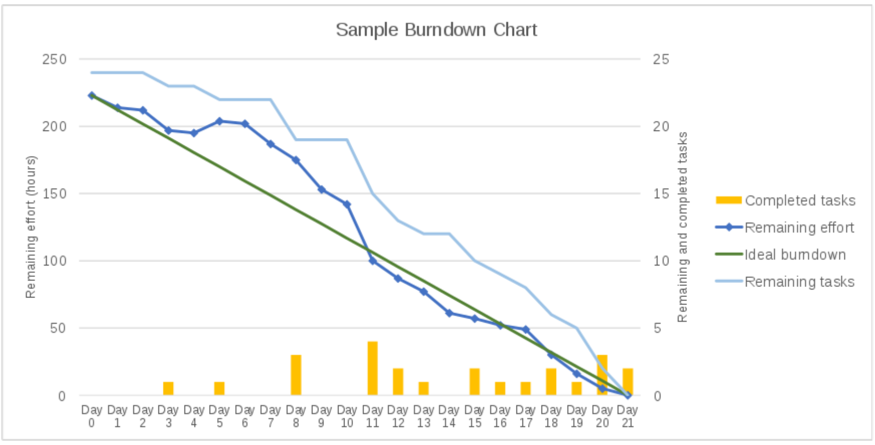


Figure 3.1: A sample Burn-Down chart (adapted from Wikipedia[1])

Figure 3.1 displays the remaining days on the x-axis and the remaining hours on the y-axis. During the Sprint Planning the ideal Burn-Down is plotted. Now each day the SM or a team member updates the remaining hours at the end of the day. Comparing the ideal and real Burn-Down values gives a good overview if the team is on track to meet the Sprint Goal. Even

[1] *https : //en.wikipedia.org/wiki/Burn_down_chart* - accessed on 2017-10-29

if Sprint Burn-Downs are a useful tool, not all consider it as a core part of Scrum as Lacey, 2012 says in his book. Schwaber and Sutherland (*Scrum Guide*) for instance talk about the Product Increment. The Product Increment is a sum of all Product Backlog items completed during a Sprint as well as the value of all previous Sprints. At the end of a Sprint the Increment must be complete, which means that it has to meet the team's Definition of Done (DoD).

Definition of Done (DoD) - The DoD is a checklist of activities that need to be fulfilled. A product (software) is done when all parts of the DoD are finished. Nevertheless a DoD is not static. With changing requirements and/or organizational structure the DoD needs to be adapted accordingly. For the Catrobat project the DoD currently consists of eight parts. New code must meet the following requirements:

1. Tests are available that cover the new functionality
2. Does not break existing tests or functionality
3. Should be branched off the development branch
4. Ticket is in *Ready for Code* review column in Jira
5. A pull request has been made at GitHub
6. A green (successful) test run on the Jenkins CI system
7. Commit guidelines[1] are fulfilled
8. Pull request has passed code review

3.3 Kanban

The Kanban system was originally developed in the automotive industry by Toyota and is often referred to as the Toyota Production System (TPS). As of different customization wishes the classic Ford system, which introduced the division of labor, was not suitable anymore. Taiichi Ohno, former vice president of Toyota Motor Corporation, therefore developed the kanban

[1] Catrobat commit guidelines on GitHub - https://github.com/Catrobat/Catroid/wiki/Commit-Message-Guidelines

technique that latter became the TPS and introduced the Just-in-Time production (*Toyota Production System*). The goal is to minimize production to a low level and only produce what is needed at the time it is needed. *Kan* means signal and *ban* means card, thus *kanban* is the signal card that a new production or work can be performed. In Kanban everything is about to achieve *Kaizen*, which is the Japanese expression for continuous improvement. David J. Anderson was the first who adapted the traditional Kanban system to the needs in software development. According to Anderson, 2010, p.13 in software development the signal cards represent work items, rather than a sign to pull more work. In this thesis Kanban is always referring to software development. One important rule when implementing or using a Kanban system is to start small with existing processes and slowly, but continuously, improve the production flow or in other words *gain Kaizen*. This evolutionary change is the biggest difference to other agile methods (Anderson, 2010; Leopold and Kaltenecker, 2013). Kanban has four characteristic elements according to Epping, 2011, although these characteristics are not an official commitment:

Pull principle - Tasks are not being pushed throughout the value chain, but rather pulled by the development team. This has two main advantages. First the team is not overloaded with work and second the team is being supported to work self organized. Visually this process can be displayed on a Kanban board. Reference is made to Section 3.3.1, where Kanban board is explained in detail.

Work in Progress (WIP) limits serve two main purposes. First, to avoid work overload and second, to shorten the task cycle time in the value chain. Furthermore the average completion rate is increased as task switching is avoided or at least minimized when WIP limits are in place. Depending on the maturity level of an organization the WIP limits should be chosen accordingly. A more mature organization can set the WIP limit to one per developer. Organizations that use Kanban for the first time may use a higher limit of two or three per developer in order to avoid big sufferings from the so called J-curve effect. The J-curve effect describes the phenomenon of capability drop and recovery when implementing a change initiative (*The J-Curve*

Effect). From a management‘s point of view the depth of the J-curve should be minimized.
The biggest challenge is to come out of the curve‘s dip. Stakeholders often expect things to move according to the dotted line as displayed in Figure 3.2. In reality the opposite takes place, no matter how well the management planned to avoid the J-curve. Viney, 2005 identified that people experiencing the J-curve effect, go through three phases. The first phase is optimism. Almost everyone is excited about a new system at the start. Soon after that the second phase, down slope, takes place. This phase is characterized by shock, denial, anger and bargaining. The key to success is to overcome this phase as quick as possible. Otherwise stakeholders may withdraw their support, which can lead to project failure. Therefore management shall take people‘s anger serious and work with them on a solution. On the other hand, management also needs to calm stakeholders and explain to them, that this situation is normal in a change initiative. At success phase three, upslope, is reached. Phase three comprises of adapting, testing and acceptance.

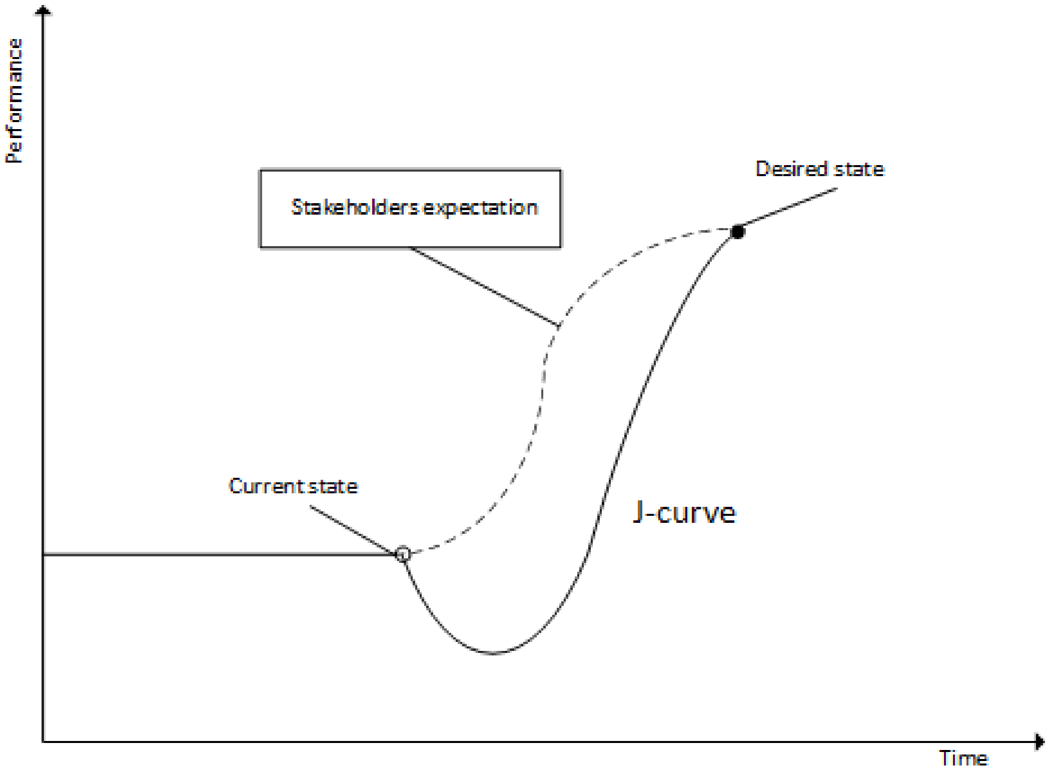


Figure 3.2: J-curve (adapted from Viney, 2005)

Transparent Information - Transparent means that every person in a team profits from information related to a task. All information is disclosed on the Kanban board and displays the following (Epping, 2011, p.58):

- The phases of the value chain, every task passes.
- Tasks in the different phases.
- People working on a task.
- WIP limits in every phase.
- A project index, that indicates the work progress.

Continuous improvement - At the beginning in Kanban everything is about Kaizen, the continuous improvement. This means to constantly adapt to project specific parameters. It is important to involve every team member in this process. Every improvement passes five steps, known as the Five Focusing Steps (FFS) by Goldratt. The FFS emerged from Goldratt's *Theory of Constraints* published in Goldratt and Cox, 1984. Anderson, 2010, p.189 states the FFS as follows:

1. "Identify the biggest bottleneck."
2. "Decide how to exploit the bottleneck."
3. "Subordinate every other decision to the decision made in step two."
4. "Remedy the bottleneck."
5. "Start over at step one."

3.3.1 Kanban board

A Kanban board is a visual representation of open tasks that need to be done. In practice, it is easy to create and supports transparent information. A whiteboard is the best tool to draw a Kanban board (see Figure 3.3), as it is visible to everyone in a room and supports the Daily Standup meetings very well. Every participant of the meeting can simply put or take sticky notes to or from the board, depending if the developer is taking a task to work on or finished a task. The board itself has several columns that can be designed as it fits the needs of the development team. A simple partition usually consist of columns containing *Product Backlog*, *In Development* and *Done* (see Figure 3.3 as an example).

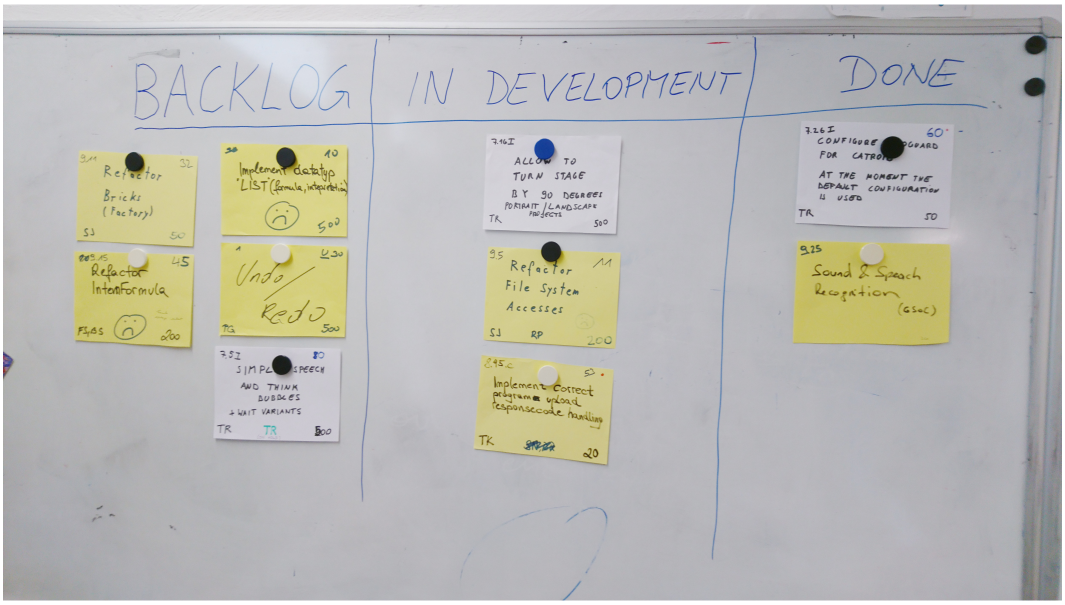
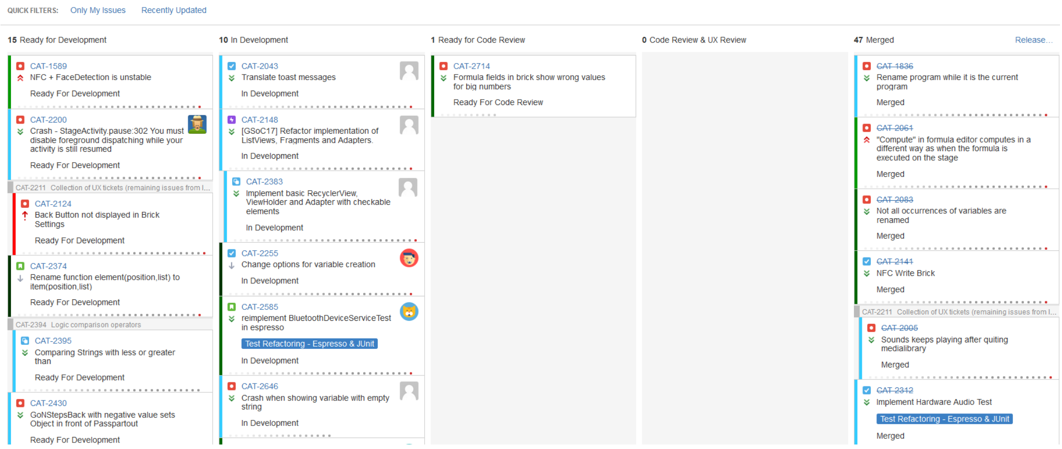

BACKLOG
IN DEVELOPMENT
DONE
Refactor Bricks (Factory)
Implement datatyp 'LIST' (formula, interpretation)
Refactor InternFormula
Undo / Redo
SIMPL SPEECH AND THINK BUBBLES + WAIT VARIANTS
ALLOW TO TURN STAGE BY 90 DEGREES PORTRAIT / LANDSCAPE PROJECTS
Refactor File System Accesses
Implement correct program upload responsecode handling
CONFIGURE BOARD FOR CATROID AT THE MOMENT THE DEFAULT CONFIGURATION IS USED
Sound & Speach Recognition (GSoC)

Figure 3.3: Kanban board from Catrobat project

In nowadays globally connected businesses it is very common that development teams work in distributed areas around the globe. Classical whiteboards are not useful in such scenarios. Therefore there are also electronic tools that model a Kanban board.

Such a tool can be seen in Figure 3.4. Electronic tools have a main advantage over a whiteboard, as they allow to generate reports and metrics out of the data, which is useful for management. Furthermore they allow an easy communication between all stakeholders.



QUICK FILTERS: Only My Issues Recently Updated

15 Ready for Development

CAT-1589
NFC + FaceDetection is unstable
Ready For Development

CAT-2200
Crash - StageActivity.pause:302 You must disable foreground dispatching while your activity is still resumed
Ready For Development

CAT-2211 Collection of UX tickets (remaining issues from l...

CAT-2124
Back Button not displayed in Brick Settings
Ready For Development

CAT-2374
Rename function element(position,list) to item(position,list)
Ready For Development

CAT-2394 Logic comparison operators

CAT-2395
Comparing Strings with less or greater than
Ready For Development

CAT-2430
GoNStepsBack with negative value sets Object in front of Passpartout
Ready For Development

10 In Development

CAT-2043
Translate toast messages
In Development

CAT-2148
[GSoC17] Refactor implementation of ListViews, Fragments and Adapters.
In Development

CAT-2383
Implement basic RecyclerView, ViewHolder and Adapter with checkable elements
In Development

CAT-2255
Change options for variable creation
In Development

CAT-2585
reimplement BluetoothDeviceServiceTest in espresso
Test Refactoring - Espresso & JUnit
In Development

CAT-2646
Crash when showing variable with empty string
In Development

1 Ready for Code Review

CAT-2714
Formula fields in brick show wrong values for big numbers
Ready For Code Review

0 Code Review & UX Review

47 Merged Release...

~~CAT-1836~~
Rename program while it is the current program
Merged

~~CAT-2061~~
"Compute" in formula editor computes in a different way as when the formula is executed on the stage
Merged

~~CAT-2083~~
Not all occurrences of variables are renamed
Merged

~~CAT-2141~~
NFC Write Brick
Merged

CAT-2211 Collection of UX tickets (remaining issues from l...

~~CAT-2005~~
Sounds keeps playing after quiting medialibrary
Merged

~~CAT-2312~~
Implement Hardware Audio Test
Test Refactoring - Espresso & JUnit
Merged

Figure 3.4: Jira Kanban board from Catrobat project

3.4 eXtreme Programming

> “[eXtreme Programming (XP)] is a lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop software.”
>
> — Beck, 1999

The framework was created by Kent Beck and published in his book *Extreme Programming Explained: Embrace Change* (Beck, 1999). According to Beck, 1999 there are four variables in software development

Cost - A balance for spending the right amount of money has to be found. Too much money is as bad as too little. Customer requirements will not be solvable with limited investments. On the other hand an excessive budget might tempt the team to not focus on what the customer really needs.

Time - Similar to cost the right amount of time until production delivery needs to be found. With too little time all other variables suffer. Too much time can hinder a project to get valuable feedback out of production, which is the most valuable aspect according to Beck, 1999.

Quality - There are two types of quality, external and internal quality. External quality is measured by customers and internal quality is measured by programmers. Sacrificing internal quality for reducing time to market will back fire sooner or later. In the worst case the software gets unmaintainable or too expensive (Beck, 1999).

Scope - Beck, 1999 describes this as the most important variable in software development. The key is not to do too much or in other words, only do as much as is needed. This may sound as a lazy approach, but if the customer changes mind, which happens quite often, the amount of useless work is limited. A side effect of this approach is that (quality) products are produced on time. Projects sometimes require to reduce the scope, either out of time or money issues. Dropping functionality is a common strategy for reducing scope. In order not to upset the customer, XP offers two strategies. The first strategy are estimates, which become more accurate over time and practice. Giving the customer more accurate estimates helps them to decide going on with the project as planned or not. The second strategy is to implement the most important features, from a customer's view, first. So even if the project gets cut off or runs out of funding, the customer has at least a basic functionality delivered.

XP has four core values and twelve derived principles. The four values are:

Communication is a key when multiple people work on a project. Sometimes people do not communicate clearly with each other, e.g., a programmer does not tell the team about an important design change or a manager does not ask the right questions, thus important information is not shared during a meeting. XP tries to counter bad communication by requiring testing, pair programming and estimating, which forces the team to communicate with each other.

Simplicity - Sometimes simple solutions are more valuable than complex ones. Ironically simplicity is not an easy task to perform (Beck, 1999). It is hard to do a task without thinking forward including possible implications of that task. For instance a rather simple programming solution to a problem would take only a fraction of time, than a complex one. Besides that a complex solution might be discarded as requirements change quickly, thus more time and work would be thrown away. XP's approach is to encourage simplicity. Beck, 1999 says that "[s]implicity and communication have a [...] mutually supporting relationship", which means that more communication leads to see things clearer and thus make them simpler.

Feedback is an important supplement for *communication*. The more feedback there is the easier the communication gets. A programmer can get two kinds of feedback. First, from the system itself. When code is changed the system, respectively existing tests, return immediate feedback if the change was successful or not. Second, from testers and customers, who write functional tests.

Courage - Sometimes it is better to start from scratch or try new things, like throwing away code or refactoring. To do such things it needs people with courage. However the first three values need to be in place, in order that courage becomes valuable. Otherwise it would be "[...] plain hacking [...]" (Beck, 1999)

All of the four values described above are useless, if there is no mutual respect within a team. Thus respect can be seen as the fifth value, which is more important than all previous values combined. Another important pillar of XP are the twelve principles, often referred as the twelve practices (Beck, 1999):

Planning game - A quick and rough planning for the next scope of the release cycle.

Small releases - As soon as a requirement set is finished, release a new version.

Metaphor - A guideline for developing a system that every developer can easily understand.

Simple design - Keep the design simple. Complex parts should be simplified if possible.

Testing - Every code fragment should have a corresponding test that ensures the correctness and functionality of that fragment.

Refactoring - Developers constantly improve existing code by simplifying it or removing duplicates.

Pair programming - Code is written by two developers on one machine.

Collective ownership - Any developer can work on any part of the code.

Continuous integration - Integrate and build the system several times a day. Ideally this is done as soon as a task is completed.

40 hour week - This is a rule of thumb. The goal is not to have tired developers, as this results in lower quality code.

On-site customer - A real person who is available at all times for questions and setting priorities.

Coding standards - Same code style supports collaboration and makes it easier to recruit new developers.

3.5 Free Open Source Software

The term Free Open Source Software (FOSS) is a composition of two movements, namely the Free Software community and the Open Source community. The two terms may be perceived as being the same, but there are significant differences. Free software is a social movement that started in 1984 and Open Source is a development methodology. The Free Software Foundation (FSF) is the main driver behind the Free Software community. Richard M. Stallman, one of the most popular persons from the FSF, defines free software with four principles (Stallman and Gay, 2002, p.43):

- "The freedom to run the program, for any purpose."
- "The freedom to study how the program works, and adapt it to your needs. (Access to the source code is a precondition for this.)"
- "The freedom to redistribute copies."
- "The freedom to improve the program, and release your improvements to the public, so that the whole community benefits. (Access to the source code is a precondition for this.)"

Many people consider free software only to be free in the sense of cost, but quite the opposite is true. It is legitimate to sell copies of free software and charge money for it. According to Stallman and Gay, 2002 the word *free* means the freedom to do basically anything with the software, like free speech allows you to say whatever you like. Therefore you are allowed to use, copy, modify and redistribute free software however you like. There are of course some restrictions, but only to ensure that the main principles are not hurt. For instance it is not allowed to to add restrictions that would undermine one of the existing principles. What began in the late 1990s with the emergence of Linux became the Open Source Initiative (OSI) in 1998, also referred to as the Open Source movement by Stallman and Gay, 2002. The Open Source community has similar definitions as the Free Software community, but they are less strict in some terms like licensing. Software is considered Open Source by the OSI when the source code of a particular software is publicly available. However executables of such software are mostly provided by one entity, who also digitally signs the executable. A modification of the executable may not be possible, as it was digitally signed, which would not be classified as free software by the FSF. Both movements have a thing in common, which is that the source code of software must be publicly available. These differences in details are confusing even for experts in this area. Therefore people began to use the term FOSS, also referred to as Free/Libre Open Source Software (FLOSS), to indicate that the software is both free and Open Source. In fact FOSS has become so popular that even for profit organizations start to switch certain projects to Open Source or contribute to other Open Source projects. Microsoft has announced[2] in late September of 2017 that it is joining the OSI as a premium sponsor. Furthermore Microsoft has even made some very popular products Open Source like *.NET Core*[3] and *Visual Studio Code*[4]. Although these products do not have the same rich features as the commercial pendants, the commitment of companies like Microsoft in the Open Source community is laudable. Another important tool that has to be mentioned when we talk about FOSS is GitHub. GitHub is an online platform for hosting code repositories of all kinds. Since the service is free for FOSS projects it has become the de facto standard for hosting and managing code repositories of many FOSS projects, including *.NET Core* and *Catrobat*. The only drawback of GitHub is that private repositories are only available as a paid plan. Since Open Source projects are meant to be open to the public, this is not a problem. For those who support and follow the principles of the FSF, GitHub is not an option. Luckily there are two other application which meet the FSF's criteria[5]: GitLab[6] and Savannah[7].

[2] Microsoft announcing accession to OSI - https://open.microsoft.com/2017/09/26/microsoft-joins-open-source-initiative/ - accessed on 2018-01-15

[3] https://blogs.msdn.microsoft.com/dotnet/2014/11/12/net-core-is-open-source/ - accessed on 2018-01-15

[4] https://code.visualstudio.com/

[5] Blogpost by FSF announcing ethical evaluations of code-hosting services - https://www.fsf.org/news/gnu-releases-ethical-evaluations-of-code-hosting-services - accessed on 2018-01-15

[6] https://about.gitlab.com/

[7] https://savannah.gnu.org/

4 Process Introduction into the Catrobat FOSS project

Catrobat started as a Free Open Source Software (FOSS) project initiated and managed by Professor Wolfgang Slany at Graz University of Technology at the Institute of Software Technology in 2010. Catrobat is a visual programming language (VPL) designed for smartphones and tablets. It is inspired by Scratch[1], another VPL, that is developed by the Lifelong Kindergarten Group at the MIT Media Lab (*Catrobat developer site*). The idea of Catrobat is to teach children and teenagers basic concepts in programming. This is achieved with a Lego-style approach (Slany, 2012). Pocket Code is the Android implementation of the Catrobat language. Also known as Catroid internally, it allows to run, modify and create Catrobat programs. Besides the Android version of Pocket Code an iOS and HTML5 version are in development. Unlike in Scratch, programming with Pocket Code does not require to have a computer. All actions can be performed on a smartphone or tablet.

The following section introduces the agile methods used in Catrobat are going to be explained. Fellhofer, Harzl, and Slany, 2015 and my own experience as a Catrobat member serve as the main literature within this section. Catrobat acts as an umbrella organization that contains several teams. In total there are eight different teams, segregated in development, infrastructure and design. Six out of eight teams are development focused, two are maintaining infrastructure and one team is responsible for all design and useability. The development teams are split into Android, iOS, HTML5 and web. It is important to know that the Catrobat project is developing actually

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

two smartphone apps. The first app is *Pocket Code*, which is inspired by the Scratch programming language. The second app is *Pocket Paint*, which originally was designed as a supplement for Pocket Code. With Pocket Paint users can draw and edit images. These images can further be used in Pocket Code to build a program. The popularity amongst users of Pocket Paint soon made the development nearly as important as Pocket Code. While Pocket Paint as a standalone image editor has been successful according to user feedback and reviews on Google Play[2], Pocket Code struggled to keep up for some time. Currently both apps have an average rating of 4.0 on Googe Play[3]. Besides being far more complex than Pocket Paint and thus having more bugs, the greatest drawback of Pocket Code is the missing integration of Pocket Paint. In the starting days of Pocket Paint Android development with use of *Intents* was very popular. Therefore the decision was made to release Pocket Paint as a standalone app.

> "[An intent[4]]provides a facility for performing late runtime binding between the code in different applications. Its most significant use is in the launching of activities, where it can be thought of as the glue between activities."
>
> — Android developers website

Users regularly complain about having to install two applications in order to use all features of Pocket Code. A desired target is to merge the two apps. However this is a challenging task. The code bases of Pocket Code and Pocket Paint are too different to pass a smooth and automatic code merge. A lot of manual work is necessary to include Pocket Paint into Pocket Code and not breaking existing functionality. Nevertheless, the Pocket Paint team already started to work on these tasks with spring 2018.

[2] https://play.google.com/store/apps/details?id=org.catrobat.paintroid - accessed on 2018-05-15

[3] https://play.google.com/store/apps/details?id=org.catrobat.catroid - accessed on 2018-05-15

[4] https://developer.android.com/reference/android/content/Intent - accessed on 2018-05-15

4.1 Problem statement

The current situation in Catrobat, and Catroid in particular, is the broad amount of code and complexity of processes. Furthermore despite being a FOSS project the majority of contributors are students. Thus many of the students leave the project after a dedicated amount of time. The fact that hardly anyone is a professional programmer when joining the project, leads to significant amount of time spent in training. While having implemented some techniques from Kanban and XP in the past, the current situation gives room for further improvements. The first two sections deal with Catrobat's use of agile development methods in the period from 2010 until 2017. In Section 4.4 I will show with the aid of statistical data from Jira that the average ticket age is too high. Additionally the statistics will point out data that may have been misconfigured or exhibit a failure in Jira's internal database. Generally it is very difficult to plan a release based on such old data. In Section 4.5 a comparison between the current workflow and the new workflow is made. The new workflow is currently being worked out by the also newly created PO board. Section 4.5 covers the PO board as well.

4.2 Agile methods in Catrobat (2010-2014)

Back in 2010 when the project was launched, only five contributors were part of it. Over the years the number of contributors grew and with them also organizational issues arose (Fellhofer, Harzl, and Slany, 2015). At the beginning, mostly students from Graz University of Technology participated in the project. With increasing number of contributors and popularity of the project, external contributors joined Catrobat. Until 2014 the number of contributors had grown to an impressive number of 130 people (Fellhofer, Harzl, and Slany, 2015, p.13). Noteworthy not all members of the project are developers. Some are specifically responsible for design and usability, others for marketing. This is quite important as the majority of developers are not users of Pocket Code. Some problems that Fellhofer, Harzl, and

Slany, 2015 describe include documentation and communication channels which were capable of improvement. Communication was not really an issue while the project consisted only of students from Graz University of Technology. However, it became an issue in 2011 when Catrobat applied to participate in Google Summer of Code (GSoC)[5] for the first time. Face to face conversations were not possible with externals. Google additionally required the project to have an Internet Relay Chat (IRC) channel, which is often an essential part in FOSS projects. This was the initial start when the project management decided to change the overall project infrastructure and communication practices. In the following subsection a detailed summary of changes in the Catrobat project will be discussed, that Fellhofer, Harzl, and Slany, 2015 describe. The changes in infrastructure and communication include switching from a manual user management to a centralized one, introduction of new tools for communication and code change tracking and use of agile methods.

User management - No central user management existed prior to the projects change initiative. All accounts needed for participating in the project were created on demand. Additionally, every tool had its own user management. Some services were used with shared accounts. With increasing number of participants, the manual user/account management was not feasible anymore. The missing accountability due to shared accounts was also a problem. In order to solve these problems, the management decided to introduce Local Directory Access Protocol (LDAP). LDAP allows to manage users and access rights in a central database. Fortunately, almost all tools and services in use had a build-in LDAP support. GitHub[6] and Crowdin[7] were the only exceptions.

Communication - As mentioned previously face to face communication was practiced in the beginning of the project. The fact that the project is allowed to use a dedicated room at the university, supported the face to

[5] A global program focused on bringing more student developers into Open Source software development - https://summerofcode.withgoogle.com/

[6] An online version control system and hosting service - https://github.com/

[7] An online collaborative translation service - https://crowdin.com/

face communication approach. Members meet, discuss and code in that dedicated room. However, with increasing number of members, soon the room was too small for the whole team. The initial solution was to use Instant Messaging. Skype[8] was the tool of choice. Nevertheless, Skype came with several drawbacks. First a lot of members who joined the group chat were passive users. Another problem was that new members needed an invitation to join the group. Furthermore the main language used in the group chat was German. Non-German speaking contributors had a clear disadvantage, as they could not follow the conversations. Therefore IRC channels were introduced, with the promising attempt to improve overall communication. The Skype chat was terminated and English was compulsory to be used in IRC. Additionally, an IRC bouncer was installed to keep the history of messages. Users could read back a conversation until the last log-out session. However, IRC was not as successful as initially expected. One reason was that it was deemed as old fashioned by many members, especially students from Graz University of Technology. Being an almost 30-year-old technology, many of the young students never used IRC. It was also quite complicated to set it up correctly. Users had to create a separate account at freenode[9]. In addition, an authentication at the project bouncer was necessary. Both services needed different credentials. After some time however, members got used to IRC.

Agile methods - At an early stage of Catrobat Kanban in combination with XP and TDD were chosen as agile development methods. Nevertheless, there have been some drawbacks. For Kanban the whiteboards in the project room were used. This became problematic when the project grew and several sub-teams emerged. Every team needed a separate Kanban board. The room size soon became too small to serve every team with a board. Another problem was the issue tracking of user stories and bug reports. Although GitHub was used as a source code repository, the bug reporting was scattered to GitHub, the local

[8] An instant messaging software - https://www.skype.com/

[9] An IRC hosting network - https://freenode.net/

Kanban board and a Google group[10]. Hence an asynchrony was unavoidable. The resolution to the problem was to switch to an online agile environment. Several software solutions were available, but they had to meet certain criteria. The most important criteria were LDAP support and a rights management. For a detailed list of criteria and tools refer to Fellhofer, Harzl, and Slany, 2015, p.18. Jira in combination with Confluence[11] were chosen, as these tools best met the criteria. Jira was tested as a pilot during GSoC 2013. The trial period revealed useful information for customization of the workflow. After the successful pilot it was decided to roll out Jira to the whole project. Confluence was introduced at a later stage. Reasons are given below.

Documentation management - The documentation was incomplete. At the beginning of the project with five people a lot of decisions were communicated orally. With increasing project size and increasing number of members documentation became essential. The problem of different communication channels remained. Documents were distributed through: Google documents, Dropbox[12] and a complex wiki[13] system. Another problem was a missing central user and document management. When members left the project, the document ownership was not transferred yielding inclomplete information. Therefore, a central content collaboration management system with customizable access rights was needed. As mentioned previously Confluence was chosen as a solution. The focus of the new system was laid on simple application.

[10] https://groups.google.com/forum/#!forum/catrobat

[11] Content collaboration software by Atlassian https://www.atlassian.com/software/confluence

[12] An online file hosting service - https://dropbox.com

[13] A website on which users collaboratively modify content and structure directly from the web browser (definition from Wikipedia)

4.3 Agile methods in Catrobat (2014-2017)

In summer of 2013 I joined Catrobat. Several teams were up to choose from, where I decided to join the Android team. This was and still is the largest team in Catrobat. While the Android team is actually split into several sub teams, the other development teams are not. Moreover the Android development is devided into Pocket Code and Pocket Paint, as mentioned earlier. Pocket Code has had a lot of sub teams of which some have been merged completely. A portion of sub teams are discontinued as development stopped. Table 4.1 shows all sub teams and their current status.

Sub-team	Status
Drone	merged
Livewallpaper	discontinued
Lego mindstorms (NXT)	merged
Albert robot	discontinued
Physics	merged
Arduino	merged
Musicdroid	in development
Virtual gamepad	discontinued
Chromecast	merged
NFC (Near FIeld Communication)	merged
Sony Xperia Play	discontinued
Tutorial game	discontinued
Catroid 3d	discontinued
Android application package (APK) generator	in development
Rasperino	merged
Scratch2Catrobat	in development
Phiro robot	merged
Catblocks	in development
No One Left Behind (NOLB)	merged

Table 4.1: List of sub teams and their status

While the Android version of Pocket Code and Pocket Paint have apps in the Google Play Store since 2013, the iOS version is currently in development. The Windows Phone team has been disbanded due to discontinuation of the Windows Phone operating system by Microsoft[14]. The HTML5 team is developing a player for the web browser, which is integrated into Pocket Code's sharing website[15]. The player can load some programs for a tryout in the browser. Unfortunately programs that use the sensors of a smartphone cannot be simulated in the browser. The sharing website is a central part of Pocket Code. Pocket Code would not be as successful as it is without the sharing website. Users of Pocket Code can download new programs, remix them and upload their own creations to the sharing website. This is one reason why the contribution of the web team is important. The web team is responsible for developing and maintaining all functionality around the sharing website. Users have to create an account in order to upload own programs. The download of programs however is available without any registration or account. Registration for a new account requires a user name, an email address and a password. Alternatively users can register an account via Facebook or Google Plus. Details about third party registration can be found in Jaindl, 2016's work. Development in Catrobat follows the principle of Test Driven Development (TDD). TDD prescribes that first a test has to be written for new code functionality. Logically this test fails on the first run, as no code is available that meets the test criteria. The developer then implements the new code. In the subsequent test-run the previously developed test-case should pass. To use TDD in practice, the project uses a Jenkins[16] server for Continuous Integration and Continuous Delivery. The web and Jenkins teams are part of the infrastructure in Catrobat. Developers can trigger complete test-runs or single test-cases on the Jenkins server. While a complete test-run can take up to 40 minutes, a single test-run is finished in a couple of minutes. Furthermore automatic test-runs are performed regularly on Jenkins. Finally the Useability (UX) and Design (UI)

[14] https://redmondmag.com/blogs/the-schwartz-report/2017/10/no-more-windows-phone.aspx - accessed on 2018-05-15

[15] https://share.catrob.at

[16] https://jenkins.catrob.at

team round up Catrobat's team constellation. They provide other teams with mockups for new icons, colors and overall design. Another important task are field tests with teenagers in schools, which result in valuable feedback for further development. This is very critical, as the developers may not always be users of Pocket Code. Mockus, Fielding, and Herbsleb, 2002's hypothesis 6 in Section 2 is hardly met in Catrobat's case. Therefore the feedback the development team receives from field studies is quite essential.

I started as as novice in Android programming. The biggest advantage at that time were the people already working in the project. Many experienced developers were active and helped to gain knowledge of the overall system. At that time the project used different tools and methods than today. Pair Programming, one of XP's principles, was used commonly. Therefore a senior developer programmed with a beginner on simple tasks. GitHub was a key part around code management. It served as the code repository as well as a bug and issue tracker. With increasing number of project members and sub-teams GitHub soon became unsuited as a tool. Planning a project was very limited due to the lack of an electronic Scrum or Kanban board. The only tools GitHub provided at that time were labels and milestones. These are very limited tools when things like release plans, cross team organization or a certain workflow need to be realized. This lack of functionality within GitHub led to a non ideal situation. All planning and decision making was being made offline and analog. Every team had their own Kanban boards in form of whiteboards in the project's room. For a new release a Planning Game was held. In this process new work tickets have been created, as well as a review of existing ones. However in order to enable contributors to work on tickets outside of the project room, an electronic solution was needed. Therefore all tickets were recreated at GitHub. This was not only double work, but also led to asynchrony between the offline and online Backlog. As details regarding a ticket often appear during the development phase, the result was that adjustments were often done only at GitHub. A lot of Planning Game work thereby became obsolete. The majority of developers solely regarded on information that was on GitHub. Another non ideal situation was the rights management for the repository at GitHub. Every developer had write access. Furthermore new developers and contributors needed to be added manually to the repository. This circumstance was not a problem with GitHub itself, but rather with the organization of the Catrobat project. Like described in section 4.2 at the beginning only a handful developers were part of the project. Giving all of them write access was an obvious decision. As the project's contributor size grew however, the rights management had been unaffected. Usually FOSS projects have a different workflow, especially when distributed, non core developers, contribute code. The majority of FOSS projects (at GitHub) uses a forking workflow. Therefore a repository is forked (a copy is made) and contributions back to the original repository are made via pull requests. This workflow has two advantages. First, contributors do not need write access to the original repository. Hence possible damage by unskilled developers is avoided. Second, if the original repository is abandoned or discontinued by the owner, the fork can still exist[17]. A good example is Veracrypt[18], a fork of the popular Truecrypt encryption software. Truecrypt was Open Source software, but the developers decided to discontinue the project for unknown reasons. The atypical Git[19] workflow in Catrobat compared to other FOSS projects did not state a problem until August 2014. At that time the Catroid team and the code base reached a critical size. In order to streamline the commit habits of developers, commit guidelines had been introduced. One requirement was that commits had to be kept to an absolute minimum, ideally only one commit. However this is rather unlikely to happen in real development processes. Therefore developers had to merge all their commits to a single one. An additional requirement was to preserve the local and remote commit history. Git being a very powerful tool, offers such an operation, which is called *rebase*. Technically it represents a rewrite of Git's history. Git creates a hash value for every command (commit, push, checkout, ...) and keeps track of every change that was made. However in the meantime changes in the main branch and other branches most likely already happened. If a developer tries to push a rebased history, Git would not allow the command to succeed. It provides a protective measure to avoid

[17] Assuming the license of the project allows it

[18] https://www.veracrypt.fr/en/Home.html

[19] Git is a a decentralized versioning system

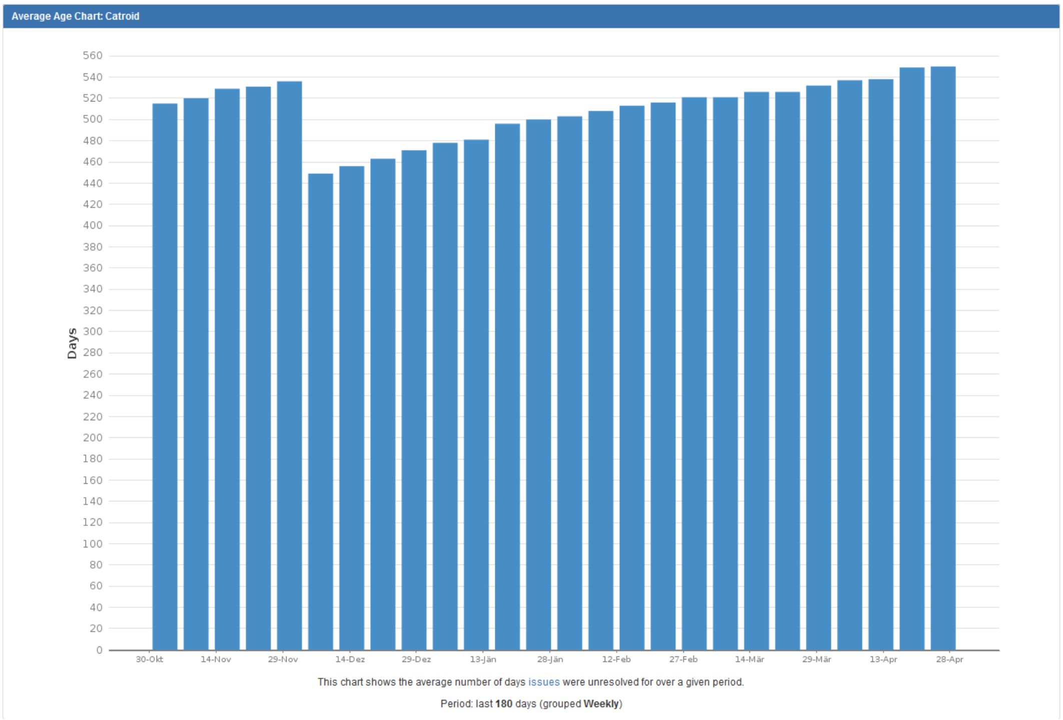
conflicts with other developers that are working on the same repository. However a developer can force the push command. Now the force push command can damage quite a lot, depending on the Git configuration and the individual situation it is used in. Prior to Git version 2.0[20] the default behavior was matching. That means that all local branches that match the same name at the remote server, will be pushed. One developer who was working on an issue had exactly this old default Git configuration. A rebased branch needed to be force pushed to the remote repository. Unfortunately that developer had checked out the master branch, which was not up to date with the remote origin master. The result was a broken master branch of the Catroid project. Not only was the work and Git history of two days gone, but also the contributions of sub-teams of Catroid. Weeks were needed to recover from this incident and to repair the repository. This was the tipping point when the Catrobat management decided to switch to a forking workflow in order to prevent similar incidents in the future.

Another problem that came up were feature branches (sub projects of Catroid) that were merged back to the main development branch of Catroid (see Table 4.1 for a list of all merged sub project). The corresponding pull request included several thousand lines of code. For a reviewer this may be challenging. As such the pull requests were accepted on behalf of the sub teams confirmations that it is tested thoroughly. Consequently some incompatibilities with core functionality arose sometimes. Reasons are hard to identify, as feature branches exist for a long time beside the main branch. Often the main branch is further developed (by number of commits). Resyncing feature branches with the main branch can be very difficult, sometimes even impossible. Similar to feature branches GSoC development often comes as a single pull request. Winkelbauer, 2016, p.85 describes such a ticket (CAT-1717) that resulted in eight follow up tickets to fix issues in CAT-1717.

[20] https://git-scm.com/docs/git-config#git-config-pushdefault

4.4 Explorative analysis of tickets with Jira

In this section an analysis of statistical data retrieved from the Jira system is conducted. The goal of the analysis is to recognize bottlenecks and give possible advice to improve the situation. A time frame of 180 days (6 months), beginning at 30th of October 2017, is taken into consideration for the analysis. In some cases a longer time frame of 1825 days (5 years) is used to illustrate a long term trend of unresolved tickets. Data with a time frame of 180 days are grouped weekly and data with a time frame of 1825 days are grouped monthly. All graphical images are generated out of the Jira system using Jira plugins.



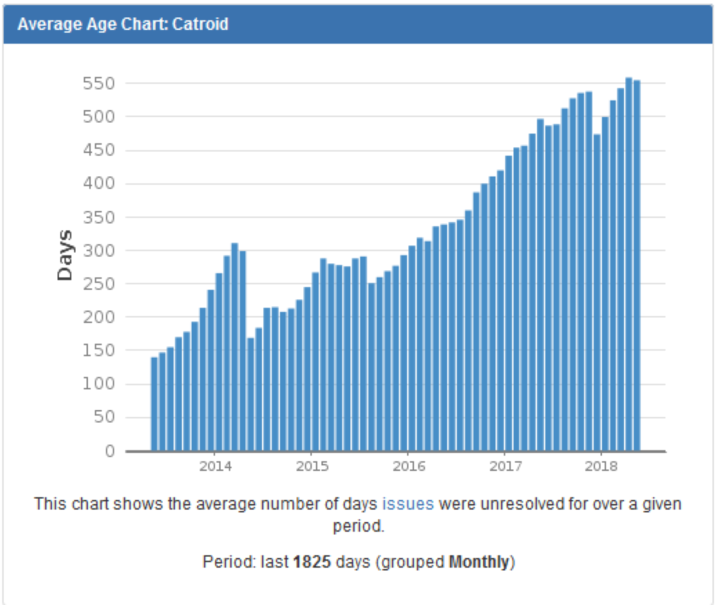
This chart shows the average number of days issues were unresolved for over a given period.

Period: last **180** days (grouped **Weekly**)

Figure 4.1: Average age of unresolved issues

The snapshot (accessed on 28.04.2018) takes into account all tickets available in Jira that have no resolution status. This means a ticket has not reached the

Merged state of the Catrobat workflow. Figure 4.1 shows the average age of unresolved issues is over 400 days for the period of 30th of October 2017 until 28th of April 2018. Apart from a single drop at the beginning of December 2017, Figure 4.1 displays an evident increase in the average of unresolved tickets. In order to illustrate the general development a visualization of unresolved tickets over the last five years has been made and can be seen in Figure 4.2.

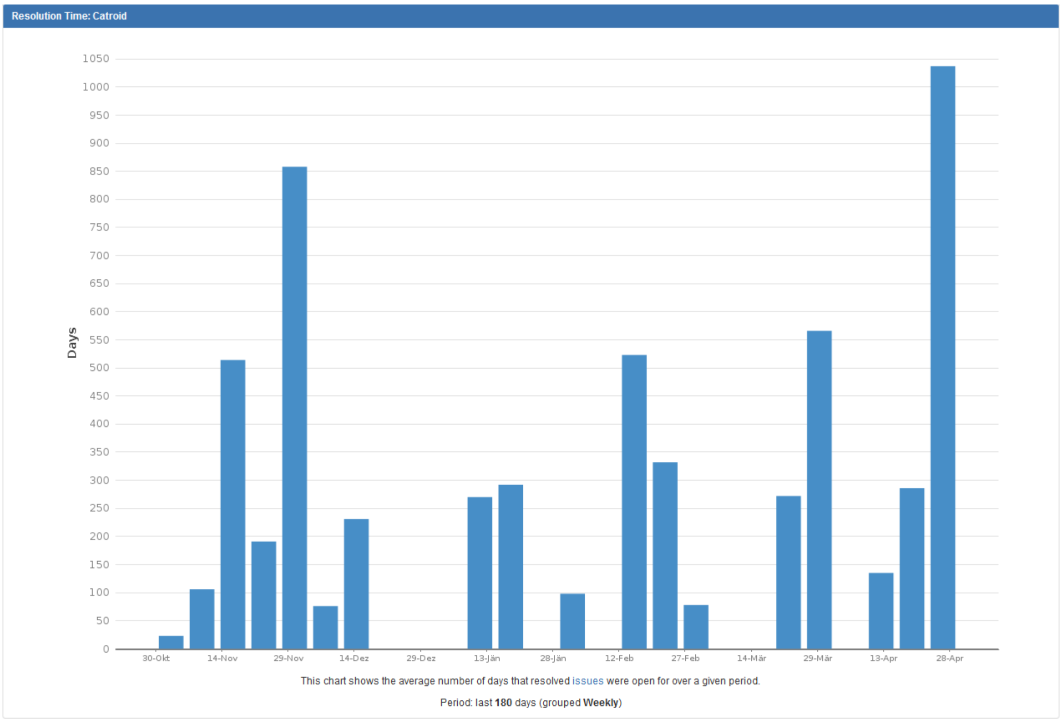


This chart shows the average number of days issues were unresolved for over a given period.

Period: last **1825** days (grouped **Monthly**)

Figure 4.2: Average age of unresolved issues over 5 years

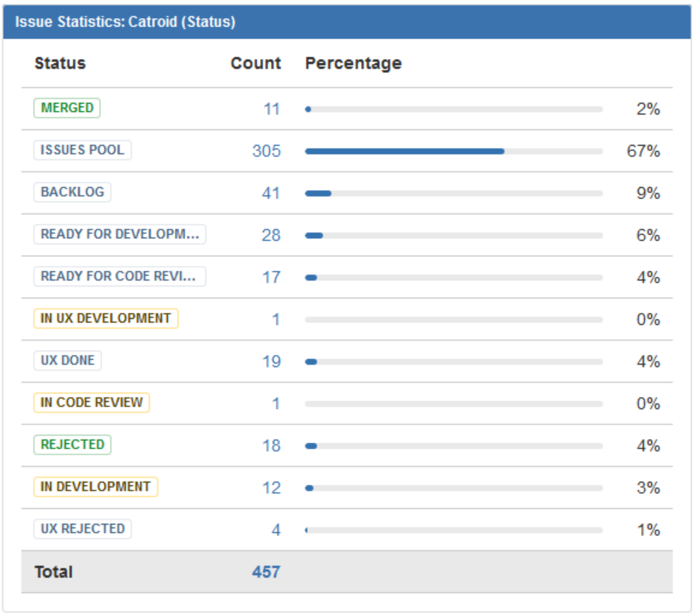
As Figure 4.2 clearly outlines, the average age of unresolved tickets has grown constantly since 2013, except some decreases in several months. One reason for the increase is that very old tickets remain in the *Issues pool*, although they have a creation date several years ago. For the calculation of the average resolution age of tickets the whole lifetime from creation date to resolution date is counted. The current prioritization of tickets is done into *trivial, minor, major, critical* and *blocker*. The creation date of tickets is not taken into account in the prioritization process resulting in old tickets remaining in the *Issues pool*. Therefore the average age of unresolved tickets increases, but does not represent the real resolution time of tickets. In contrast to Figure 4.1, Figure 4.3 shows the average age of resolved issues for a given period. Figure 4.3 has higher amounts in some cases compared to Figure 4.1. However the data has to be examined with caution. While data in Figure 4.1 is increasing over time, Figure 4.3 depends highly on the working time of developers. The week of December 29th and March 14th for instance show no data. This may indicate that no tickets have been resolved in that time frame. Considering that the majority of developers are students, this seems plausible. The time frame falls into Christmas holidays and semester holidays respectively. The high valued average mirrors very old, often several years, tickets. Moreover the circumstance that 67% of issues is still in the *Issues pool* increases the average age of unresolved issues (see Figure 4.4).



This chart shows the average number of days that resolved issues were open for over a given period.

Period: last **180** days (grouped **Weekly**)

Figure 4.3: Resolution time of resolved issues

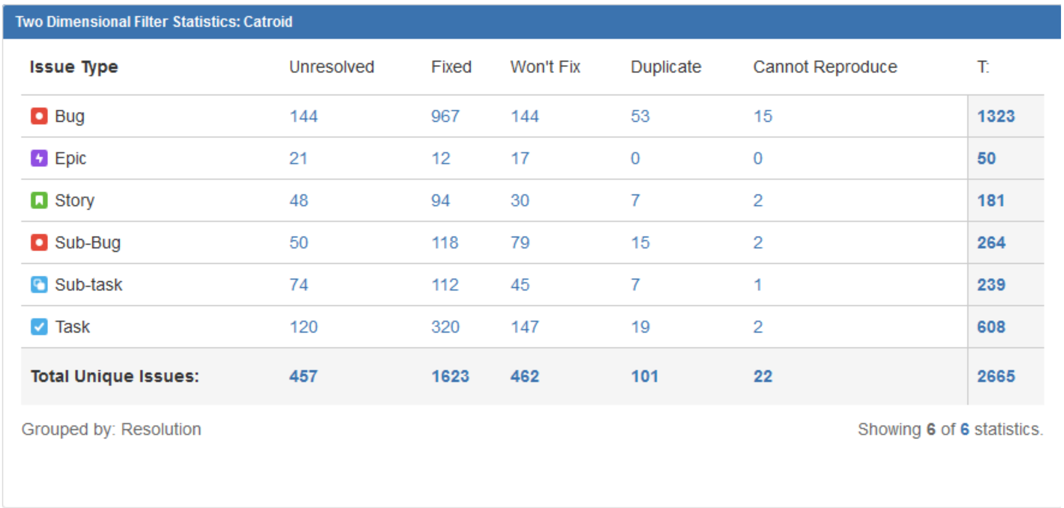


Issue Statistics: Catroid (Status)

Status	Count	Percentage
MERGED	11	2%
ISSUES POOL	305	67%
BACKLOG	41	9%
READY FOR DEVELOPM...	28	6%
READY FOR CODE REVI...	17	4%
IN UX DEVELOPMENT	1	0%
UX DONE	19	4%
IN CODE REVIEW	1	0%
REJECTED	18	4%
IN DEVELOPMENT	12	3%
UX REJECTED	4	1%
Total	457	

Figure 4.4: Issue statistics of Catroid

Figure 4.5 shows the total amount of issues in project Catroid. Column one lists the issue types. In total there are six different types. A *bug* is a flaw in the code that needs to be corrected, like a function that calculates wrong results. A sub-bug represents a similar flaw like it's parent bug, but with minor differences. In order to keep the problem description of a bug ticket compact, sub-bug tickets are helpful to split the work of larger bug issue tickets. The issue type *Epic* represent a major change that needs to be split into several smaller tickets. Epics are used to plan and structure big changes by visualizing them in the workflow as such. A *Story* is a special kind of issue type. It represents a customer or user need, thus it can be called a system requirement. Stories, often called user stories, are the smallest unit of work in agile frameworks. User stories usually do not contain details about how the desired requirements shall be accomplished. Finally issue types task and sub-task represent work items that need to be done. The type sub-task is not necessarily a child node of a task. In Catrobat's case a sub-task is mostly part of an epic ticket. Column 2 of Figure 4.5 shows all



Two Dimensional Filter Statistics: Catroid

Issue Type	Unresolved	Fixed	Won't Fix	Duplicate	Cannot Reproduce	T:
Bug	144	967	144	53	15	1323
Epic	21	12	17	0	0	50
Story	48	94	30	7	2	181
Sub-Bug	50	118	79	15	2	264
Sub-task	74	112	45	7	1	239
Task	120	320	147	19	2	608
Total Unique Issues:	**457**	**1623**	**462**	**101**	**22**	**2665**

Grouped by: Resolution

Showing **6** of **6** statistics.

Figure 4.5: Issue types grouped by resolution

unresolved issues. Out of 457 unresolved issues 261 are older than one year (see Figure 4.6). This is 57,11%. Moreover a significant amount (67%) of the 261 issues is still in the Issues pool, which can be seen in Figure 4.6.

When examining the unresolved tickets some irregularities come to light. See Figure 4.4 for details. 11 tickets out of the 457 happen to be *Merged*. Thus their status should not be *unresolved*, but rather *resolved*. Another 4 tickets have been remaining in state *UX Rejected* and yet another 19 tickets in *UX Done*. These tickets are in fact unresolved, but were never moved back to the Backlog. This is a drawback of the old workflow (see Figure 4.9). All UX related tickets are handled in a separate Kanban board in Jira. This can lead to "forgotten" tickets, which increases the total amount of unresolved tickets. Another problem is given by tickets with the state *Rejected*, which are also listed as unresolved. There are a total of 18 tickets with that state. The correct state should be *Rejected* with resolution *Won't fix*, as seen in Figure 4.5 column 4. In conclusion the correct amount of unresolved tickets would be 428 [457 - 11(Merged) - 18(Rejected)]. Column 3 of Figure 4.5 shows tickets that have been fixed. The resolution of fixed tickets is *Merged*. Column 5 shows tickets that have been classified as duplicates. Out of 101 tickets with

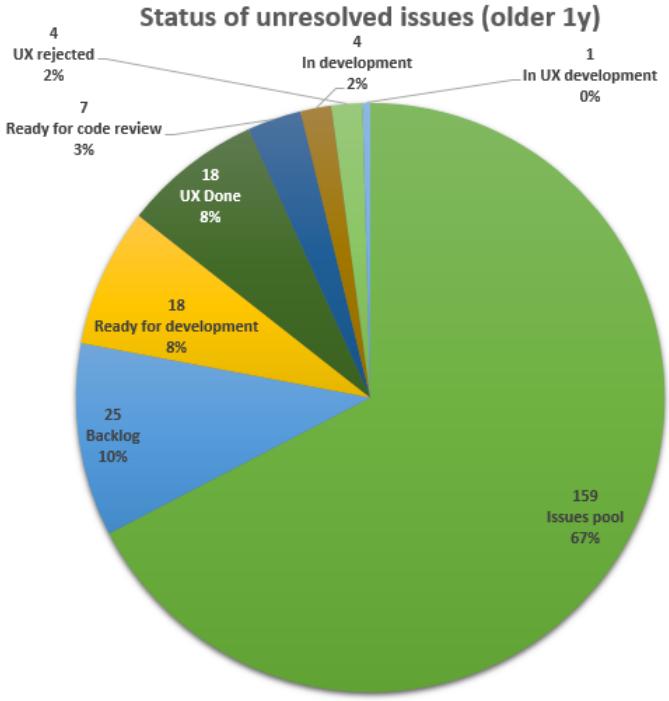


Figure 4.6: Issues older than 1 year

the resolution *Duplicate* only 59 have a status *Duplicate* as well. The other 42 tickets have been wrongly assigned the status *Merged*. This should not be possible according to the current workflow, which is described in section 4.5. Furthermore the 42 tickets have a value set for field *Fix version*[21], which may be the reason for the status to be *Merged*. This may indicate either a failure in the database system of Jira or a manual change that led to a distortion. Column 6 is the last resolution state shown in Figure 4.5. First of all *Cannot reproduce* is neither a resolution nor a status that is available in the current workflow. Moreover all 22 tickets have a state *Merged*. Thus the correct resolution should be *Fixed*. Nevertheless, when examining the tickets however, only two tickets have a commit on GitHub (which were declined). All other tickets have no commit that is recognized by Jira. These facts yield the assumption that the resolution (Cannot reproduce) is indeed correct,

[21] Example ticket - https://jira.catrob.at/browse/CAT-1479 - accessed on 2018-05-15

but the status may be wrong. The reason for the wrong status however can not be verified from existing data. Finally column 7 of Figure 4.5 list the total amount of tickets for each issue type, as well as the accumulated total of all tickets in the project. Figure 4.5 shows furthermore that a total of 1323 out of 2665 tickets are bugs, which equals 49,64%. If sub-bugs are taken into account as well, the total accumulated number of bugs rises to 1587 or 59,55%. This is a relatively large percentage, which may lead to an assumption that the code base needs to be strengthened.

Another fact that data from Jira shows is that certain tickets have been worked on even when they never left the Issues pool. Table 4.2 shows a list of all tickets that have a git commit, which has been merged into the main development branch. Similar incidents should be avoided. When following a given workflow with no exceptions such a situation would not come up.

CAT-2656
CAT-2640
CAT-2630
CAT-2553
CAT-2525
CAT-2494
CAT-2491
CAT-2475
CAT-2400
CAT-1274

Table 4.2: List of tickets still in Issues pool that have been worked on and merged into develop branch

The majority of distributed FOSS projects uses mailing lists for communication. The mailing lists are also used as a discussion platform for work tickets or patches of code that developers share. In Catrobat however, communication is often done face to face in the project room or in corresponding Slack[22] channels. Catrobat uses Slack as a replacement for IRC since 2017.

[22] A cloud based collaboration tool - https://slack.com/

Only in rare cases a discussion about a specific ticket is done inside of Jira. Figure 4.7 shows the correlation between days open, number of comments, and participants of tickets in Jira. The chart in Figure 4.7 displays the first 200 tickets of project Catroid for a time frame of twelve weeks. It allows a general statement about all tickets. Every bubble represents one ticket within Jira. The size of the bubbles indicates the number of participants involved in this ticket. Participants can interact via the commenting function in Jira. Therefore the x-axis of Figure 4.7 displays the amount of comments. The majority of tickets has no comments at all and therefore only one participant, the creator (also called reporter) of the ticket. A smaller fraction of tickets has some comments (1-6). Only very few tickets have more than six comments. The big red bubble in the upper right corner represents a ticket that has recently been commented a lot. On the y-axis of Figure 4.7 the number of elapsed days a ticket has been opened is shown. The increasing

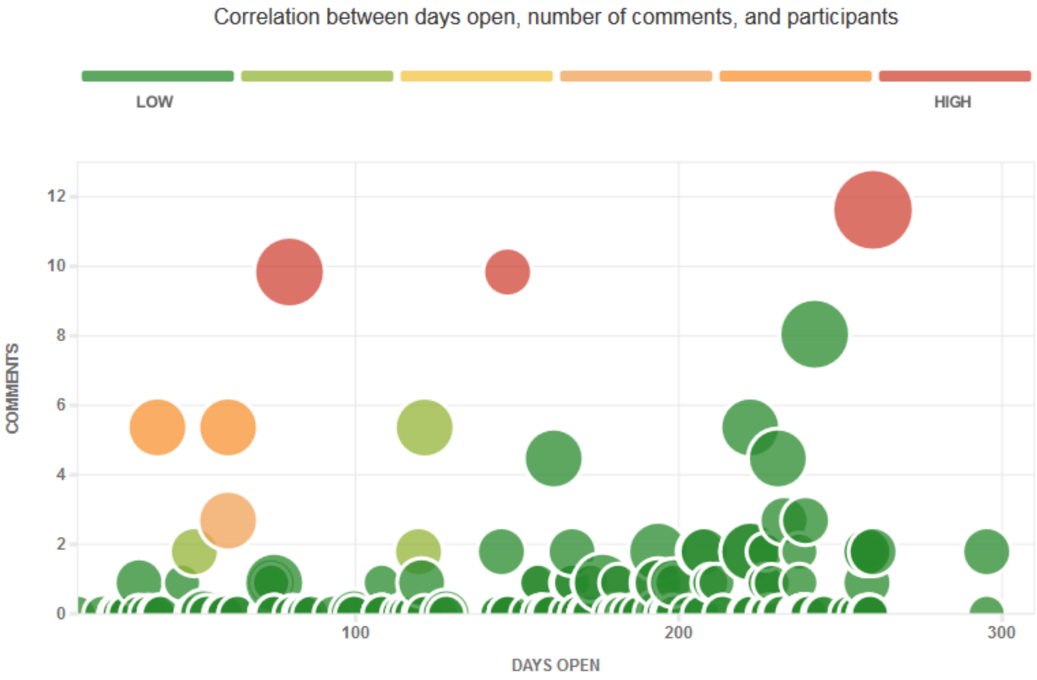


Figure 4.7: Comment activity of tickets in Jira

trend shown by Figures 4.1 and 4.2 appears again in Figure 4.7: A significant amount of tickets is very old. Commenting is one way developers can interact with tickets and provide feedback to others. Another way developers can support a ticket is by voting for it. The intention is that tickets with more votes are more likely to be scheduled for the next release by the PO. Unfortunately the voting feature is less used than comments in Jira, which can be seen in Figure 4.8. The size of the bubbles indicate the number of

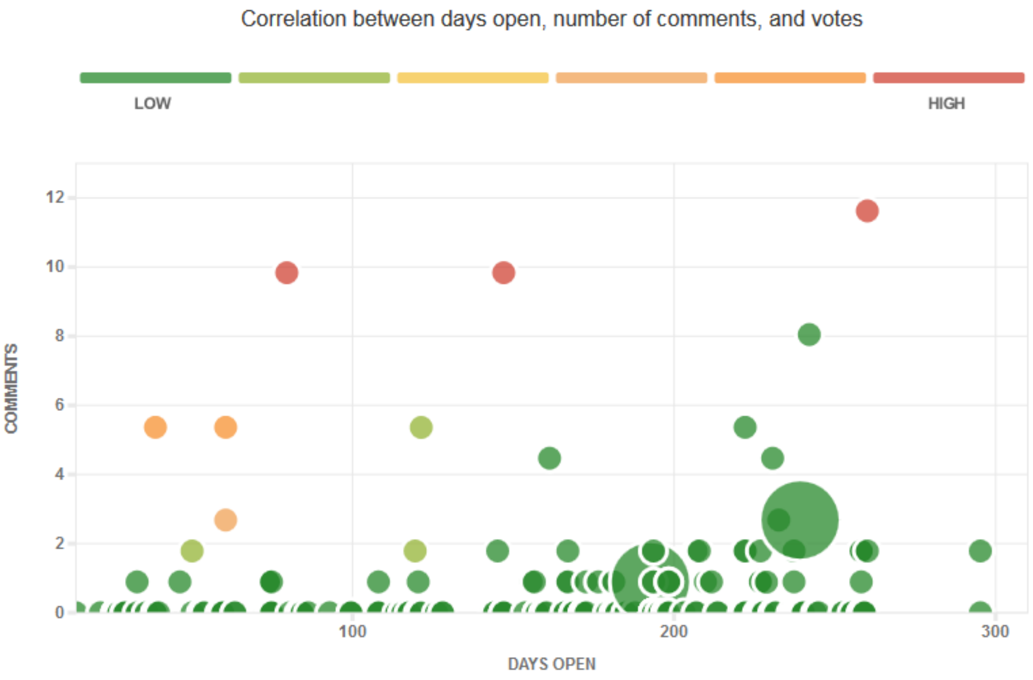


Figure 4.8: Voting activity of tickets in Jira

votes per ticket. Only 2 out of 200 tickets in the time frame considered have a vote (1 in both cases).

4.5 Improvement process

This section describes possible improvements to the presented issues stated in sections 4.2, 4.3 and 4.4. Some of these solutions are already being executed. Others are still in development.

4.5.1 Workflow

Figure 4.9 displays the current Catrobat workflow in Jira. Currently every team except for the web team uses the old/current workflow. The web team has already adapted the new workflow. It is planned that all teams switch to the new workflow, but since it is still in refinement it would be infeasible to introduce it to a larger team like Catroid. Catroid is currently undertaking major changes. Besides refactoring a large portion of the code base, the Catroid team is participating in GSoC. The current workflow appears overloaded when compared to the new workflow in Figure 4.10. This is confirmed by examining the different states. UX related states (*UX Backlog*, *In Development*, *UX Rejected*, *UX Done*) are part of the current workflow. However the UX team uses a separate Kanban board. This can lead to tickets that are not transferred back to the Backlog, like discussed previously. The new workflow reduces the UX related states to a single one. Furthermore only one Kanban board is now used. Another redundant state in the current workflow is *Duplicate*. The resolution of a duplicated ticket is the same as for *Rejected*. Therefore the states can be reduced to *Rejected*. It is still possible to mention that a ticket was rejected because it was a duplicate in the comment section of a ticket. Yet another two states are not present in the new workflow, namely *Ready for Code review* and *Ready for UX review*. These are intermediate states. The PO board decided that there is no need to have a representation of these states in the new workflow. Nevertheless a new state is introduced in the new workflow: *PO review*. This is the final state before a ticket can be merged (*Ready for Merge* is the equivalent in the current workflow). The PO board can evaluate if the tickets meets all criteria to be closed or needs changes. In the case that changes are necessary, the ticket is sent back to development. The PO review can be compared to Sprint review meetings in Scrum. Another change in the new workflow are less transitions between the states. First the transitions of not existing states are gone. Second in the new workflow back transitions have only been implemented with added value like transitions from *Issues pool* to *Backlog* and vice versa. Other transitions were removed like *Rejected* to *Issues pool* (when a ticket is rejected there is usually no need to reopen it) and *Merged* to *Issues pool* (same as for rejected). If for some reason a ticket needs to be reopened, then a new ticket referencing to the old can be made. This prevents possible overload in the workflow.

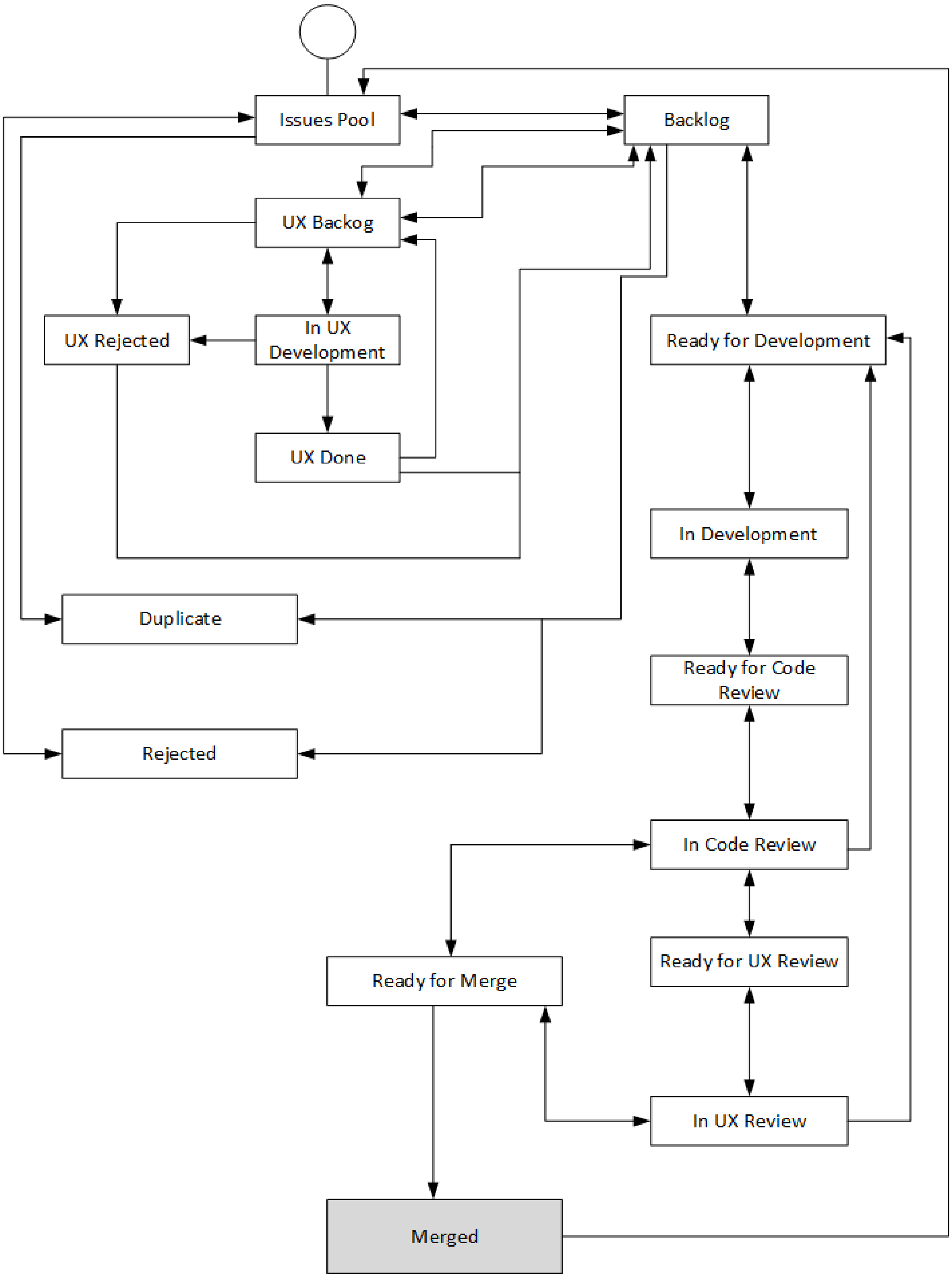


Figure 4.9: Old Catrobat workflow

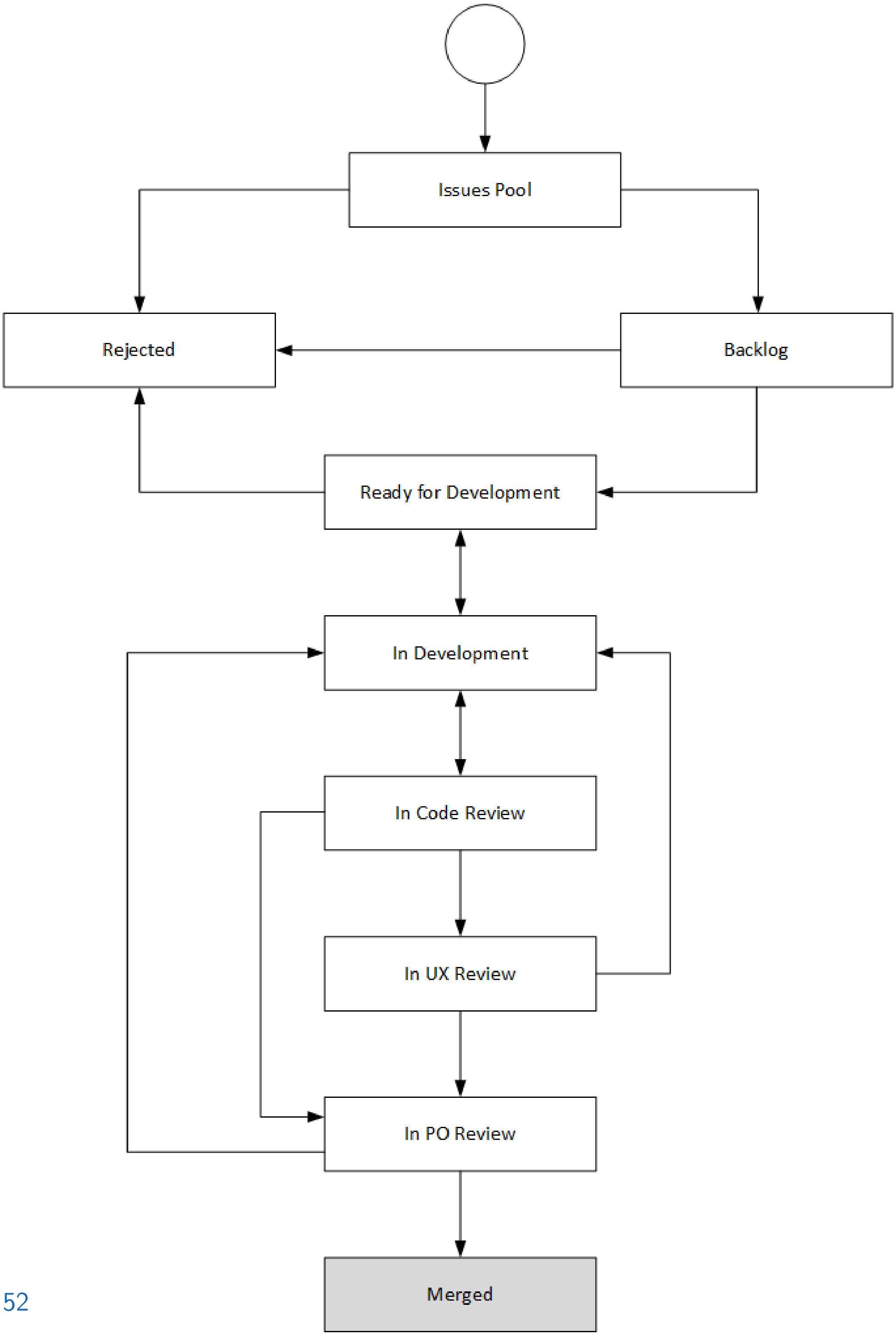


Figure 4.10: New Catrobat workflow

4.5.2 Product Owner board

The Product Owner (PO) board was created out of necessity for general management related purposes of the Catrobat project. Members include Professor Slany amongst three others. The idea of the board is to improve communication project wide, due to a weakness in communication between different teams. Another reason a PO board is necessary is the size Catrobat has grown to. While development and management decisions are becoming more team independent, a central steering board is needed. Due to the fact that team managers (coordinators) are in most cases students, it makes sense to have a general management group. In section 4.4 several aspects were shown that indicate a superior number of tickets is causing slow throughput of development. In addition many tickets do not have a proper problem description or get created without analyzing existing tickets, which causes duplicates. To improve the situation the PO board is meant to review tickets that are planned for development and reject others. In particular a prioritization is planned to be established. Unlike the current prioritization like described in section 4.4 the PO board intends to prioritize tickets with the DEEP (see Cohn, 2009; Pichler, 2010) approach. Furthermore the PO board is also responsible for verifying if a ticket has been solved as intended. It represents the final step before a ticket is merged and therefore closed (see also Figure 4.10). However this circumstance could lead to a bottleneck, as members of the PO board only have limited time. A possible solution could be to involve experienced senior members of the project in the PO review state of the workflow. The reporting tools in Jira can help the PO board in decision making. The current situation produces only limited reports due to missing requirements. In order to improve the situation two steps are needed as basic prerequisites. At first an estimation of tickets would be helpful to enable forecast reports in Jira. Second a ticket cleanup is unavoidable. Leopold and Kaltenecker, 2013 suggest to regularly clean the Backlog. As an example they suggest to discard all tickets older than six months that have not been pulled for work. In the case of Catroid six months would be a too short time frame. A reasonable time frame would be one or one and a half year. This would result in 180 tickets for a one year period and 252 tickets for one and a half year. A more radical, but often more efficient way is to discard all existing tickets and start with a fresh ticket pool. Moreover the PO board could consider to hold retrospective meetings with the development team. Retrospective meetings take place soon after a release and before the next planning meeting. Retrospectives are an excellent opportunity to review what went well and which parts did not succeed as planned. Thereby the team gets more experienced in estimating and delivering the planned release cycles. Furthermore the PO board also learns valuable information about the team's performance. Thus a more accurate future planning is possible.

4.5.3 Work in Progress limits

A Work in Progress (WIP) limit is very effective way to reveal bottlenecks in a workflow. Anderson, 2010, p.115 states it would be a mistake not to have WIP limits. In simpler words a WIP limit serves to restrict the concurrent work that can be done in a queue. Limiting WIP has also positive side effects. Developers are more focused if they have to do only one job at a time. When working on multiple tasks at the same time there is always switching cost, which increases the throughput time. A typical limit is the amount of developers that are working on the project. Though development teams that never worked with WIP limits should be granted a higher limit of two or three per developer. The Catroid core team currently consists of 20 members. This would result in a WIP limit of 20, if the usual limit of one is assigned. Considering that the Catroid team is unfamiliar wit WIP limits, the actual limit would be even higher. Therefore a split in the responsibilities may be reasonable. Considering Kent Beck's advice not to use XP with teams bigger than ten people (Beck, 1999), the team structure in Catroid would need a revision. A possible team constellation could be to have two teams with ten members each. One team could be focused on fixing bugs and implementing new functionality. The other team could refactor code and tests. After a predefined time frame the teams swap roles. However any improvement process should always have the goal to support developers.

Hence an introduction of WIP limits into a team that never worked with comparable constraints has to be carefully planned. Developers need to be convinced that these techniques can improve their work as well as the overall success of a project. Concurrently the management needs to be aware that the development team will need some time to incorporate the new techniques. Furthermore the team coordinator would have the task to keep the team motivated in In the short run WIP limits could slow down a project at first. Though in the long run a project most likely benefits from WIP limits by improving the throughput rate. When the throughput is improved the lead time is automatically improved as well. This can be proven by Little's law. Little's law is based on the queueing theory, a mathematical sub-field of probability theory. Hence the average number of items (tickets) in a queueing system is denoted as L. L is depends on the average number of tickets arriving into the system (λ) and the average time (w) a ticket is unresolved. This is expressed in Little's law formula:

$$L = \lambda \cdot w \tag{4.1}$$

Equation (4.1) can be adapted slightly to match Kanban's terms. Denoting L as WIP, λ as the throughput rate (TR) and w as the lead time (LT) yields to the expression:

$$WIP = TR \cdot LT \tag{4.2}$$

Transforming Equation (4.2) yields the following representation for the average lead time:

$$LT = \frac{WIP}{TR} \tag{4.3}$$

A project manager can easily calculate the lead time with Equation (4.3). Furthermore it can be proven that the lead time increases when higher WIP limits are chosen. Anderson, 2010, p.28 draws the same conclusion in his analysis of two teams inside the Motorola company. To illustrate this, the following example is chosen: Assuming a WIP value of 4 and an average TR of 2 leads to a LT of 2 by applying Equation (4.3). Increasing the WIP to 8, LT is increased to 4. In order to shorten the lead time there are two options. Either the throughput rate is increased or the WIP is decreased. Most projects tend to increase the throughput rate by hiring more developers, using additional tools or requiring team members to work overtime. The fact that most of these actions fail can be explained with Brook's law:

> "Adding manpower to a late software project makes it later."
>
> — Jr., 1995, p.14

The above example illustrates that WIP limits can help to shorten the lead time of a project. However in software development unpredictable things can occur that require to either exceed the WIP limits or reconsider them. Both options are allowed in Kanban, but should not be the rule. Applying WIP limits to Catrobat in general and Catroid specifically needs certain requirements. First a throughput rate needs to be calculated. This however is not feasible with current data in Jira, as the average age of tickets is too high. Furthermore no real planning of releases takes place, which leads to a random ticket processing. The PO board should therefore plan a release together with the development team, including estimates for tickets. It is important that all stakeholders stick to a given process in order to get useful results and be able to calculate the throughput rate. The second requirement is to approximate a possible WIP limits by taking into account the amount of developers available and an approximate lead time from previous releases. With these components several small release cycles can be tested. The results can be taken into account for establishing real WIP limits for the development team.

4.5.4 Improvement summary

As described in Section 4.2 the early stages of the Catrobat project heavily used some XP techniques. In particular this included Pair Programming, Planning Games, Continuous Integration and Testing. By the time the project gained growth, some of these techniques were pushed into the background. As a consequence new problems occurred as described in Section 4.4. To conquer the problems a re-introduction of XP techniques may be effective. Especially the use of Pair Programming and Planning Games is essential to gain success. In the special case of Catrobat being a FOSS with a majority of students as contributors the following suggestion may be fruitful. By introducing a Code Day, developers could actively learn and practice Pair Programming. For further advise and examples how to introduce Pair Programming into a project reference is made to Williams and Kessler, 2002. In the case of Planning Games it is important to stick to the already made decisions made in it. Otherwise a Planning Game exhibits limited success. Another important task is to re-invent estimates for tickets. This is actually a part of the Planning Game. The advantage gained by this process is the ability to generate forecast reports in Jira. Finally, retrospective meetings offer the possibility to review the work and enable a more accurate future planning process.

5 Conclusion

In section 4 a history of Catrobat's development methods has been conducted. The historical reappraisal shows how the project evolved over time and with it also new challenges arose. At the beginning of the Catrobat project with a handful of contributors management was not an issue. With growing number of contributors and popularity of the project challenges appeared. One of the main challenges was user management. Therefore Catrobat's management decided to introduce a more effective way to handle user management. LDAP proved to be right choice for this requirement. However the use of electronic tools proved not always to be the best solution, like the use of IRC. Communication is generally a challenge in a large distributed environment. Important decisions need to be documented properly in order to ascertain basis of decision-making in the future. Section 4.2 and 4.3 describe the use of specific agile methods used in Catrobat. Notably the Catrobat work within the period from 2010 until 2014 was more agile than the period from 2014 until 2017. This unexpected conclusion is based on the present data in Jira. In practice, the acceptance criteria for pull requests was very straight forward. For new code, tests had to exist and pass the test suite on Jenkins. In general new code should not break existing tests and functionality. The pull request were reviewed by several senior members, who gave valuable feedback and improvement comments on GitHub. The second important factor is that prior to Jira, Planning Games included an estimation process. Developers assigned estimates to each ticket in the range from 1 to 500. The range included the values: 1, 5, 10, 20, 50, 100, 200, 500. 1 represents a simple task like correcting a typo in a string. Values 5 to 50 represent work that a single developer can accomplish by themselves. Depending on the complexity of the task a higher or lower estimation is given. For instance rewriting a core functionality may have a value of 50, as several code areas need to be modified. Values from 100 onwards are meant to be split up into smaller tasks. Generally an estimation above 100 represents a major modification of the code base or a summary of different smaller tasks belonging to the same category. In Jira a ticket of type *Epic* would have an estimation between 100 and 500. With the introduction of Jira many developers were uncertain what estimation value a ticket shall be assigned. The estimation range was too broad. As a result the estimate field in Jira became optional to provide. The current situation in Catrobat is that estimation of tickets is not part of the current workflow. This results in difficult release planning processes which exacerbate a forecast on the completion of tickets for future releases. Furthermore the majority of report functionality within Jira can not be utilized without any estimates. The solution is to re-introduce estimates into the project. In order to avoid confusion, the estimation should be worked out with the development teams and the PO board. A simple estimation model like Kniberg and Skarin, 2010 suggest using t-shirt sizes may be effective. However it might take several releases in order for the team to give approximately right estimates. This is aggravated with fluctuation of team members and availability during the university semester. The reappraisal furthermore shows that former agile methods like Pair Programming are not practiced anymore. While Catrobat as a whole and Catroid in particular is growing, the number of senior members is limited. Due to the limited contribution time of students, only a handful of senior members remain who are able to perform code reviews.

In section 4.4 the work in progress and throughput of tickets in Jira were analyzed. The results can be summarized in an abundance of tickets. More than 59% of tickets are of type bug or sub-bug. This leads to an assumption that core functionalities of the code base need a refactoring. A refactoring of all test cases is currently in process. Furthermore the analysis showed that 57,11% of all unresolved tickets are older than one year. This explains the high values of resolution time of tickets. Since the resolution time of a ticket is measured from creation date, all old tickets influence the average resolution time. Furthermore Figure 4.4 shows that 67% (305 out of 457) of currently unresolved tickets are in *Issues pool*. This means they are not considered for current development. All tickets that are considered for development are in the *Backlog*. Another interesting fact that the explorative analysis showed, are wrong resolution states. There are in sum 29 wrong unresolved states. 11 have the status *Merged* and 18 have the status *Rejected*. The status *Duplicate* is only assigned to 59 out of 101 tickets with resolution *Duplicate*. Finally 22 tickets that have been classified with a resolution *Cannot reproduce*, raises questions how this is possible. There is no resolution or status *Cannot reproduce* in the workflow. The irregularities in wrong states and resolutions may indicate a failure in Jira's internal database or a manual change that caused the distortion. Unfortunately no data confirming either of the assumptions could be found. Another finding of the analysis is that 22 tickets have a status (Cannot reproduce) that is neither available in the current nor in the new workflow. The fact that 59,55% of all tickets is a bug or sub-bug, according to Jira's data, leads to an assumption that the code base needs to be strengthened. Communication regarding tickets is very sparse in Catroid, although Jira offers easy to use tools like comments and votes.

Section 4.5 discusses possible solutions to problems identified in previous sections. First a comparison of the old, but still in use by most projects, workflow with the new workflow is made. As the old workflow is basically used by every team, it will be further called the current workflow. The current workflow (see Figure 4.9) has several states and transitions that have been removed compared to the new workflow (see Figure 4.10). The first change that catches the eye are the missing UX related states. Only UX review is still available in the new workflow. Considering that all UX related issues have been handled in a separate Kanban board inside Jira, the elimination of the UX board is a good thing. Thereby "forgotten" tickets shall not be possible with the new workflow. Unnecessary transitions were also removed in order to have a cleaner workflow. After all the workflow should not hinder a developer. Having less states and transitions helps achieving exactly that. Noteworthy the new workflow is still in refinement and therefore only used by the web team.

Subsection 4.5.2 describes the PO board that has been created recently. Due to communication and organizational improvement need the PO board will have the role of a general management unit. The PO board will have a central role in the new workflow as described earlier. Ticket planning and reviewing is a core task of the board. Furthermore the *PO review* status in the new workflow represents the final stage before a ticket is resolved. However this stage could possibly lead to a bottleneck. Therefore it is advised that senior members are involved at this stage. Another finding is that in order to generate forecast reports in Jira, estimation of tickets would be necessary. A recommendation regarding tickets abundance is to either purge all tickets older than a key date (e.g. one year) or start with a fresh ticket pool. The PO board should further encourage retrospective meetings where valuable information for future planning can be obtained. Moreover the development team gains experience estimating release cycles.

In subsection 4.5.3 the advantages of using WIP limits is discussed. Further simple examples illustrate that WIP can help to optimize the lead time. Little's law can be used to mathematically prove that this statement holds. Establishing the right value for a WIP limit always has to take into account the project, company or environment for its intended use. If a development team delivers satisfying results and no sections require improvement then WIP limits are not necessary. For Catrobat and Catroid in particular WIP limits would improve the current situation with high probability. However the current team size and circumstance that no throughput rate can be calculated make a WIP limit introduction not feasible. Smaller team size or a separation into two teams with different responsibilities may be a possible solution. Furthermore several smaller release cycles would be necessary to gain information about the throughput rate.

6 Meta reflection and Future work

Free Open Source Software (FOSS) is enjoying great popularity. Emerged from the hackers' culture the success of FOSS really rose exponentially with Linux. Since the initial release of Linux in 1990 a variety of different products and services based on the Linux operating system emerged. Further a lot of research has been conducted in the field of FOSS and distributed teams. Raymond, 1999's definition of "Linus' law" was a starting point from where a lot of today's methodologies and frameworks followed. Notably *Scrum Guide*'s Scrum, Beck, 1999's eXtreme Programming (XP) and Anderson, 2010's Kanban are agile software development frameworks that are most widely used. All these frameworks, besides many others, follow the principles of the Agile Manifesto. Nevertheless research on the field of distributed teams and their exact functioning is rather small, according to Bonaccorsi and Rossi, 2003. Moreover research about FOSS in combination of an educational environment is even smaller. This thesis analyzes agile project management practices in an educational FOSS project named *Catrobat*. First a historical reappraisal is conducted where practices, methods, challenges and solutions are discussed. The reappraisal is segmented into two periods. The first period is from 2010 until 2014 and the second from 2014 until 2017.Concluding from the results of the reappraisal shows that the period from 2010 until 2014 in some points was more agile than the ensuing period. A possible explanation is that the Catrobat experienced a massive growth in the last years. Therefore new requirements and challenges appeared. The fact that the entire development and management is based on free contributions is crucial for the project. In the subsection 4.4 an analysis of statistical data from Jira is performed. The main points that can be taken out from the explorative analysis are as follows:

- There is an abundance of tickets in the system.
- A large portion of tickets is very old.

This leads to an increase in the average age of tickets and thereby an adverse starting point for planning. Consequently solutions are presented that can help to improve the current situation and become more agile as a project.

Keeping in mind that Catrobat is founded at an university, run by volunteers and has no for profit intentions, it is remarkable what the project has achieved so far. Nevertheless Catrobat faces challenges which will be crucial for future development. However the fact that volunteers around the world are willing to help keeps the motivation up.

This thesis analyzed agile practices and frameworks in general, with a special focus on the Catrobat FOSS project. However a more practical approach of applying these practices could have been done more profound. The introduction of WIP limits may have been a good case for research. In particular a test pilot with a very small team could have been conducted. The involved stakeholders could have been interviewed for an analysis if the intended practices make sense. Furthermore the evaluation of Jira tickets could have been used to review Jira's configuration for possible failures.

6.1 Outlook for the future

As described previously Catrobat is facing challenges. This thesis discussed some of these challenges and suggests fruitful solutions. The historical reappraisal showed that techniques from agile methodologies can help the project, especially principles from XP. Pair Programming and Planning Games are essential in this case. However it is even more important to keep information and decisions that are devised from plannings. Catrobat already uses Confluence as a collaboration and documentation tool. A digitization of planning and decision making inside Confluence is therefore recommended. Section 4.5 showed that specific management initiatives and refactoring processes are already taking place. A result of this thesis is an introduction of WIP limits as a potential improvement method for Catrobat. A pilot phase where the feasibility and added value is tested is recommended. This may represent a possible topic for future research. Further a focus on automation of technical processes in order to reduce overhead may be beneficial to the project. In particular the automatic signing and releasing a finished application into the Google Play store may be an area of application. Depending on the funding of the Catrobat project it is advised to establish a real core team of developers. This means that developers work full time and are paid, like in Mozilla's case described by Mockus, Fielding, and Herbsleb, 2002. The biggest advantage could be to focus on the development of the project and to decrease the training time.

Bibliography

Abrahamsson, P. et al. (2002). *Agile software development methods - Review and analysis*. Tech. rep. 478. VTT PUBLICATIONS. URL: http://www.vtt.fi/inf/pdf/publications/2002/P478.pdf (cit. on p. 10).

Anderson, David J. *The J-Curve Effect*. Accessed on 2017-10-22. URL: https://edu.leankanban.com/blog/organizational-maturity-j-curve-effect (cit. on p. 17).

Anderson, David J. (2010). *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press. 280 pp. ISBN: 978-0-984-52140-1 (cit. on pp. 17, 20, 54, 55, 63).

Apache. *The Apache Software Foundation (ASF)*. Accessed on 2017-10-22. URL: https://www.apache.org/foundation/ (cit. on p. 5).

Armstrong, David J and Paul Cole (2002). "Managing distances and differences in geographically distributed work groups." In: *Distributed work*, pp. 167–186 (cit. on p. 5).

Beck, Kent (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional (cit. on pp. 23–25, 54, 63).

Beck, Kent et al. *The Agile Manifesto*. Accessed on 2017-10-22. URL: http://agilemanifesto.org/principles.html (cit. on pp. 9–11).

Bonaccorsi, Andrea and Cristina Rossi (2003). "Why Open Source software can succeed." In: *Research Policy* 32.7, pp. 1243–1258. ISSN: 0048-7333. DOI: 10.1016/s0048-7333(03)00051-9 (cit. on pp. 3, 4, 63).

Catrobat. *Catrobat developer site*. URL: developer.catrobat.org (cit. on p. 29).

Cobb, Charles G. (2011). *Making Sense of Agile Project Management*. John Wiley & Sons, Inc. DOI: 10.1002/9781118085950.

Cohn, Mike (2009). *Make the Product Backlog DEEP*. Accessed on 2018-05-09. URL: https://www.mountaingoatsoftware.com/blog/make-the-product-backlog-deep (cit. on pp. 14, 53).

Conaldi, Guido and Alessandro Lomi (2013). "The dual network structure of organizational problem solving: A case study on open source software development." In: *Social Networks* 35.2. Special Issue on Advances in Two-mode Social Networks, pp. 237–250. ISSN: 0378-8733. DOI: 10.1016/j.socnet.2012.12.003.

Crowston, Kevin and James Howison (2006). "Hierarchy and centralization in free and open source software team communications." In: *Knowledge, Technology & Policy* 18.4, pp. 65–85. ISSN: 1874-6314. DOI: 10.1007/s12130-006-1004-8 (cit. on pp. 3, 4).

Cutkosky, Mark R., Jay M. Tenenbaum, and Jay Glicksman (1996). "Madefast: Collaborative Engineering over the Internet." In: *Communications of the ACM* 39.9, pp. 78–87. ISSN: 0001-0782. DOI: 10.1145/234215.234474 (cit. on p. 4).

Epping, Thomas (2011). *Kanban für die Softwareentwicklung*. Springer Berlin Heidelberg. ISBN: 978-3-642-22594-9. DOI: 10.1007/978-3-642-22595-6 (cit. on pp. 17, 20).

Fellhofer, Stephan, Annemarie Harzl, and Wolfgang Slany (2015). "Scaling and Internationalizing an Agile FOSS Project: Lessons Learned." In: *Open Source Systems: Adoption and Impact*. Ed. by Ernesto Damiani et al. Cham: Springer International Publishing, pp. 13–22. ISBN: 978-3-319-17837-0. DOI: 10.1007/978-3-319-17837-0_2 (cit. on pp. 29, 31, 32, 34).

Fielding, Roy T. (1999). "Shared leadership in the Apache project." In: *Communications of the ACM* 42.4, pp. 42–43. ISSN: 0001-0782. DOI: 10.1145/299157.299167.

Ghosh, Rishab Aiyer and Vipul Ved Prakash (2000). "The Orbiten free software survey." In: *First Monday* 5.7. DOI: 10.5210/fm.v5i7.769 (cit. on p. 3).

Goldratt, Eliyahu M. and Jeff Cox (1984). *The goal a process of ongoing improvement*. English. Great Barrington, Mass.: North River Press (cit. on p. 20).

Jaindl, Stefan (2016). “Social media software integration for the symfony web framework and Android and iOS versions of the catrobat project.” MA thesis. Graz University of Technology. URL: http://diglib.tugraz.at/download.php?id=5990d20f5c7b6&location=aleph (cit. on p. 36).

Jr., Frederick P. Brooks (1995). *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley Professional. ISBN: 978-0-201-83595-3. URL: http://proquest.techbus.safaribooksonline.de/book/software-engineering-and-development/project-management/0201835959 (cit. on p. 56).

Kniberg, Henrik and Mattias Skarin (2010). *Kanban and Scrum-making the most of both*. Lulu.com (cit. on p. 60).

Koch, Stefan and Georg Schneider (2002). “Effort, co-operation and co-ordination in an open source software project: GNOME.” In: *Information Systems Journal* 12.1, pp. 27–42 (cit. on p. 3).

Lacey, Mitch (2012). *The Scrum Field Guide: Practical Advice for Your First Year*. 1st. Addison-Wesley Professional. ISBN: 978-0-321-55415-4 (cit. on pp. 12, 14–16).

Ladas, Corey (2009). *Scrumban - Essays on Kanban Systems for Lean Software Development*. Raleigh, North Carolina: MODUS COOPERANDI PR. 180 pp. ISBN: 978-0-578-00214-9 (cit. on p. 10).

Layton, Mark C. (2012). *Agile Project Management for Dummies*. New York, NY, USA: John Wiley & Sons, Inc. ISBN: 978-1-118-02624-3. URL: http://proquest.techbus.safaribooksonline.de/book/software-engineering-and-development/project-management/9781118235850 (cit. on p. 9).

Leopold, Klaus and Siegfried Kaltenecker (2013). *Kanban in der IT*. 2nd. Carl Hanser Verlag. ISBN: 978-3-446-43826-2 (cit. on pp. 17, 53).

Mockus, Audris, Roy T Fielding, and James D Herbsleb (2002). “Two case studies of open source software development: Apache and Mozilla.” In: *ACM Transactions on Software Engineering and Methodology* 11.3, pp. 309–346. ISSN: 1049-331X. DOI: 10.1145/567793.567795 (cit. on pp. 3, 5, 6, 37, 65).

Moon, Jae Yun and Lee Sproull (2000). “Essence of distributed work: The case of the Linux kernel.” In: *First Monday* 5.11. ISSN: 13960466. DOI: 10.5210/fm.v5i11.801 (cit. on p. 4).

Pichler, Roman (2010). *Agile Product Management with Scrum: Creating Products That Customers Love*. 1st. Addison-Wesley Professional. ISBN: 978-0-321-60578-8 (cit. on pp. 14, 53).

Raymond, Eric (1999). “The cathedral and the bazaar.” In: *Knowledge, Technology & Policy* 12.3, pp. 23–49. ISSN: 1874-6314. DOI: 10.1007/s12130-999-1026-0 (cit. on pp. 4, 63).

Royce, Winston W (1987). “Managing the development of large software systems: concepts and techniques.” In: *Proceedings of the 9th international conference on Software Engineering*. IEEE Computer Society Press, pp. 328–338 (cit. on p. 5).

Schwaber, Ken (1995). “SCRUM Development Process.” In: *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 117–134 (cit. on p. 12).

Schwaber, Ken (2004). *Agile Project Management With Scrum*. Redmond, WA, USA: Microsoft Press. ISBN: 0-735-61993-X (cit. on p. 12).

Schwaber, Ken and Jeff Sutherland. *Scrum Guide*. Accessed on 2017-10-22. URL: http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf (cit. on pp. 13, 14, 16, 63).

Slany, W. (2012). “A mobile visual programming system for Android smartphones and tablets.” In: *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 265–266. DOI: 10.1109/VLHCC.2012.6344546 (cit. on p. 29).

Stallman, Richard M. and Joshua. Gay (2002). *Free software, free society : selected essays of Richard M. Stallman*. English. Boston, Mass. : GNU Press, Free Software Foundation (cit. on p. 26).

Toyota. *Toyota Production System*. Accessed on 2017-10-22. URL: http://www.toyota-global.com/company/vision_philosophy/toyota_production_system/just-in-time.html (cit. on p. 17).

Ullman, Ellen (1998). “The dumbing down of programming.” In: *21st Salon* 13 (cit. on p. 4).

Viney, David (2005). *The intranet portal guide: How to make the business case for a corporate portal, then successfully deliver*. Mercury Web Publishing (cit. on pp. 18, 19).

Williams, Laurie and Robert Kessler (2002). *Pair programming illuminated*. Addison-Wesley Longman Publishing Co., Inc. 288 pp. ISBN: 978-0-201-74576-4 (cit. on p. 57).

Winkelbauer, Florian (2016). “Tackling software quality problems in a free and open source software project.” MA thesis. Graz University of Technology. URL: http://diglib.tugraz.at/download.php?id=5988e7a223fe4&location=aleph (cit. on p. 39).