Michael Herold

Communication in an Agile FOSS Project: A Socio-Technical Case Study

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

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

Co-Superviser

Dipl-Ing. Matthias Müller, BSc

Institute for Softwaretechnology

Graz, August 2019

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

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

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

Date

Signature

Abstract

The development of Free and Open Source Software (FOSS) involves several challenges. Besides the required technical experience, a contributor needs to have a profound domain expertise as well as a comprehensive knowledge about the project's guidelines, rules and standards. These requirements constitute potential barriers for newcomers. To facilitate a smooth project entry, it is necessary to identify and understand the challenges contributors are facing while placing their first contributions. This thesis offers insights into the collaborative software development process of an agile FOSS project, presenting the case of Catrobat. A multiple case study was conducted to qualitatively analyze the contributions of newcomers within the scope of the Google Summer of Code program. Potential hurdles were identified and propositions were made by investigating multiple qualitative and quantitative data sets collected through the newcomers' socio-technical interactions with the community. Based on these findings, multiple expert interviews were conducted and triangulated with the outcomes of the case study. The final results indicate that newcomers at Catrobat strongly depend on guidance from the community because of (a) the project's strict requirements about software testing, (b) an insufficient awareness about internal guidelines; and, (c) a lack of domain expertise. The results suggest that frequent and direct communication as well as pair programming and synchronous code reviews have a positive effect to a newcomer's contribution. This thesis provides a set of recommendations to lower the entrance barriers at Catrobat.

Kurzfassung

Die Entwicklung von Free and Open Source Software (FOSS) bringt einige Herausforderungen mit sich. Mitwirkende eines FOSS Projektes benötigen neben der vorausgesetzten technischen Erfahrung auch ein tiefgreifendes Domänenwissen sowie umfangreiche Kenntnisse von projektinternen Richtlinien, Regeln und Standards. Diese Voraussetzungen stellen mögliche Hürden für Neuanfänger dar. Um einen reibungslosen Projekteintritt zu ermöglichen, ist es notwendig, die anfänglichen Probleme der Neulinge zu erkennen und zu verstehen. Diese Arbeit gewährt Einsicht in den kollaborativen Softwareentwicklungsprozess eines agilen FOSS Projektes und stellt in diesem Rahmen das Catrobat Projekt vor. Eine mehrfache Fallstudie wurde durchgeführt um das Mitwirken von Neuanfängern im Rahmen des Google Summer of Code Programms qualitativ zu untersuchen. Durch die Analyse von mehreren qualitativen und quantitativen Datensätzen, gesammelt durch die soziotechnischen Interaktionen zwischen den Einsteigern und der Community, wurden potentielle Hürden identifiziert und Behauptungen aufgestellt. Basierend auf diesen Erkenntnissen wurden mehrere ExpertInnen-Interviews geführt und mit den Ergebnissen der Fallstudie trianguliert. Die daraus entstehenden Resultate weisen darauf hin, dass Einsteiger bei Catrobat stark von der Betreuung der Community abhängig sind, unter anderem aufgrund (a) der strikten Projektvoraussetzungen bezüglich Softwaretests, (b) einem unausreichenden Bewusstsein über interne Richtlinien; und, (c) einem fehlenden Domänenwissen. Die Ergebnisse deuten an, dass synchrone und direkte Kommunikation, Pair-Programming und synchrone Code-Reviews einen positiven Effekt für Neuanfänger haben. Diese Arbeit liefert eine Reihe von Empfehlungen um die Einstiegsbarrieren bei Catrobat zu reduzieren.

Contents

List of Figures

List of Figures

1. Introduction

The development of Free and Open Source Software (FOSS) involves several challenges different to those perceived in conventional software development teams (Crowston, Li, et al., 2007). While FOSS is getting more and more popular throughout the whole industry (GitHub, 2019; Ebert, 2008), a lot of research has been conducted about the collaborative software development practices of the loosely coupled contributors (Scacchi, 2010; Crowston, Li, et al., 2007). Geographical separation, cultural differences and a decentralized team structure bring up new challenges for the collective development process (Steinmacher, Conte, Gerosa, et al., 2015; Herbsleb and Grinter, 1999). Additionally, these challenges are reinforced by the application of agile methods due to its reliance on frequent and open communication (Korkala and Abrahamsson, 2007). Still it is proven that FOSS usually is of high quality considering code quality and other technical aspects (Krogh, Haefliger, et al., 2012). Besides personal interest and motivation, contributors need to have (a) technical experience, (b) domain expertise; and, (c) knowledge about organizational rules in order to find their way into the project (Canfora et al., 2012; Trainer, Chaihirunkarn, and Herbsleb, 2013). This can be a challenging task which relies on a lot of communication and guidance. The socio-technical congruence in software development projects has been researched a lot since the publication of Conway's law (Herbsleb and Grinter, 1999; Scacchi, 2010; Syeed and Hammouda, 2013). Conway, 1968 argues that a software's architecture reflects the organizational structure of its development team. Nevertheless, this field has rarely been studied in the context of FOSS development (Bolici, Howison, and Crowston, 2009). Rather than trying to give evidence for a potential congruence of the communication needs induced by a project's technical components and the actual communication performed within a FOSS community, this thesis focuses on the rationales behind the contributors' communication patterns. To better understand the complex context of this socio-technological environment, the case of Catrobat is presented offering an insight into the collaborative software development process of an agile FOSS project. In this study, potential hurdles showing up during a newcomer's contribution are identified and it is investigated how they can be overcome. Since onboarding is a challenging process for both the newcomers and the community, it is important to streamline this process and to keep the project's entry barriers as low as possible; otherwise contributors might find it too time-consuming to join (Krogh, Spaeth, and Lakhani, 2003). However, the long-term success of FOSS projects strongly relies on the constant acquisition of newcomers who eventually transform to seniors (Jensen, King, and Kuechler, 2011; Steinmacher, Wiese, and Gerosa, 2012). Therefore it is crucial to understand the challenges contributors are facing: *Why, when and how are newcomers interacting with the community? What are they communicating about? What are their main hurdles while placing their first contributions? How can these hurdles be overcome?* This thesis sheds light into these questions investigating the case of Catrobat.

1.1. Outline

The remainder of this thesis is organized as follows: Chapter 2 elucidates the term FOSS, addresses its major influences, outlines how FOSS communities are structured and how development and communication practices are handled. An introduction to Extreme Programming and Kanban, two agile development practices employed at Catrobat, is given in Chapter 3. The Catrobat project, which constitutes the research subject of this thesis, is presented in Chapter 4. The research questions as well as the combined research design are discussed in Chapter 5. In the subsequent chapters, the contributions of three participants of the Google Summer of Code program are thoroughly analyzed as part of the case study (Chapter 6 and 7). To build up a chain of evidence, three semi-conducted expert interviews are evaluated combining diverse perspectives from different areas (Chapter 8). In Chapter 9, the outcomes of the case studies are triangulated with the findings of the interviews and discussed with prevalent literature. Furthermore, recommendations are given to lower the entry barriers for newcomers at Catrobat. Limitations and threats to validity are listed in Chapter 10. To conclude, the outcomes of this thesis are summarized in Chapter 11.

2. Free and Open Source Software (FOSS)

The term "Free and Open Source Software", hereafter referred to as FOSS, is a hybrid phrase influenced by two major movements: the *free software* and the *open source software* movement. In 1985 Richard M. Stallman founded the Free Software Foundation (FSF)[1], a nonprofit organization to support the notion of free software. Stallman refers to free software (FS) as a product that respects the users' freedom by allowing them to *"run, copy, distribute, study, change and improve software"* freely at any time (Stallman, 2002). In order to make use of the freedom to study, change and improve software, a user needs to have access to a software's source code. Different to proprietary or non-free software, the free software label propagates that users must have the freedom to run a program on any kind of computer system, for any purpose without being required to ask or pay for permission. Thus, "free" is referred to the users' liberty and not the product's price.

More than a decade later, in 1998 the Open Source Initiative (OSI)[2] was founded by Eric Raymond and Bruce Perens, two representatives of the aforementioned open source movement. One of the OSI's first tasks was to assess licenses and state compliance criteria for the distribution terms of open source software (OSS). To continue, the OSI is a philosophy and development model to foster the collaborative development of OSS. In his book, Raymond, 1999 refers to this collective method as the *bazaar*, where the development process is done by a large number of equal developers. Development is coordinated over the internet and exposed to the public. The

[1]https://www.fsf.org, visited on 30 May 2019
[2]https://opensource.org, visited on 30 May 2019

author, who was one of the first GNU[3] contributors, credits Linus Torvalds[4] as the inventor of this process. In a case study of the Fetchmail[5] project, the writer lists the advantages of the bazaar model and claims that it is more successful than the traditional model of developing software in small, hierarchical teams – referred to as the cathedral model. Raymond states that *"given enough eyeballs, all bugs are shallow"* and refers to this as the *Linus's Law* (Raymond, 1999), underpinning the necessity to disclose source code to as many developers as possible.

Although the four freedoms of free software and the ten criteria of open source advocate similar principles in terms of licenses, both movements underlie different values. While the former can be interpreted as a social philosophy, the later has a more practical goal and focuses on a collaborative development methodology. To combine both values, the hybrid terms "Free and Open Source Software" (FOSS) and "Free/Libre Open Source Software" (FLOSS) were introduced to emphasize that free relates to *"free speech, not free beer"* (Stallman, 2002). In the remainder of this thesis, the term FOSS is being used without considering the controversy of the aforementioned movements and its underlying values.

Nowadays FOSS is widely used, both in industry and for private use. According to GitHub's yearly report for 2018, there are more than 31 million developers and more than 96 million repositories on Github[6], one of the world's largest FOSS development platform (GitHub, 2019). To name a few popular examples, Microsoft's Visual Studio Code[7], Facebook's React-Native[8] and Tensorflow[9] are the top three FOSS projects having the largest number of contributors as shown in the report. Among the most successful and best-known FOSS products all-time are the Apache web server[10],

[3]GNU is an operating system that is free software. https://www.gnu.org, visited on 30 May 2019

[4]Linus Torvalds is the creator and leader of the Linux kernel. https://www.kernel.org, visited on 30 May 2019

[5]http://www.fetchmail.info, visited on 30 May 2019

[6]https://github.com, visited on 31 May 2019

[7]https://github.com/Microsoft/vscode, visited on 31 May 2019

[8]https://github.com/facebook/react-native, visited on 31 May 2019

[9]https://github.com/tensorflow/tensorflow, visited on 31 May 2019

[10]https://httpd.apache.org, visited on 31 May 2019

the Mozilla browser[11], the GNU C compiler[12] and the MySQL database management system[13] (Fitzgerald, 2006).

2.1. Roles

Different to conventional closed source software projects, Ye and Kishida, 2003 show that in FOSS projects there is a role transformation allowing contributors to transform to different roles depending on the phase of contribution. Figure 2.1 illustrates the different roles, as originally proposed by Ye and Kishida, 2003. A community is built around the *Project Leader* who initiated the project and is responsible for communicating the project's vision and mission. The development process is guided and coordinated by *Core Members* who have been in the project for a long time and made a multitude of contributions to the code base. While their main task is to support other contributors and serve as advisors for key decisions, the majority of the development is done by *Active Developers*. Similarly, *Peripheral Developers* are making selective contributions to new features on an irregular basis. Having no clear overall picture, *Bug Fixers* are primarily focussing on parts of the code base necessary to implement bug fixes reported by either themselves or other community members, like *Bug Reporters*. These reporters do not know the source code and exclusively serve as testers to submit potential bugs. In contrast to members who are directly or indirectly involved in the development process, *Readers* profit from the source code by trying to learn from the ideas and patterns of the implementation which is usually of high quality. The final software product is then used by so-called *Passive Users*, who use the software similar to traditional closed source software.

Identifying all users as potential developers, the role of a user can transform from a passive role to a more active role, moving towards the core of the model as illustrated in Figure 2.1. According to the authors, the existence and proportion of different roles varies across different FOSS communities.

[11]https://developer.mozilla.org, visited on 31 May 2019
[12]https://gcc.gnu.org, visited on 31 May 2019
[13]https://www.mysql.com, visited on 31 May 2019

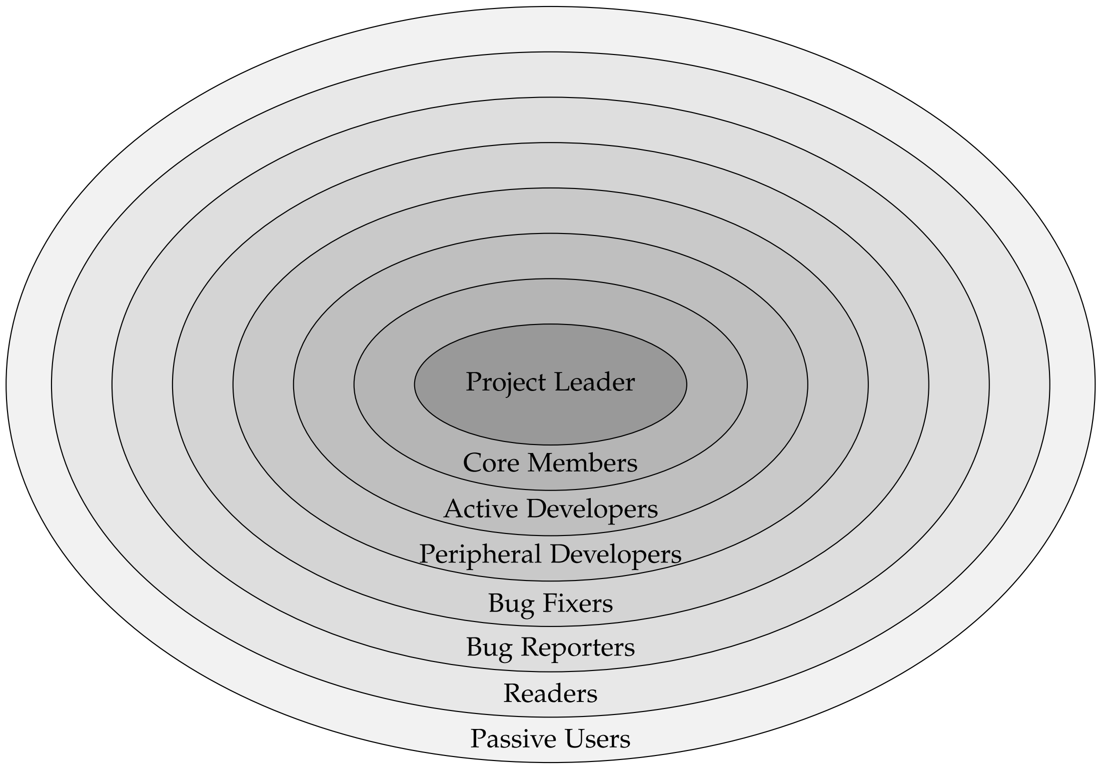


Figure 2.1.: Roles in FOSS communities (Ye and Kishida, 2003)

The number of participants increases from inner to outer layers resulting in the largest group of people carrying out the role of *Passive Users*. Contributors at the core of the model have more influence to key decisions and the development of the project than members at outer layers (Ye and Kishida, 2003).

Underpinning the transformation towards the core, Hippel and Krogh, 2003, Scacchi, 2002, and Scacchi, 2005 state that FOSS is regularly developed by the same people who use it. Fitzgerald, 2006 found out that traditional FOSS projects tend to be horizontal infrastructural systems – for example operating systems, web server, database management systems and compilers – where developers were unexceptionally users of the software in development, thus involving both a profound domain knowledge as well as a strong technical expertise.

2.2. Community structure

Centralization refers to the number of people who contribute to the code base. In a highly centralized project, the majority of code is written by a few contributors, whereas decentralization would result in an equal distribution of development. Additionally, in hierarchical projects a few members would have more authority over the code than others (Crowston and Howison, 2006).

In a case study of the Apache Foundation, Crowston and Howison, 2006 suggest that non-hierarchical and decentralized structures are favored compared to hierarchical and centralized structures *"because they are more robust to personality disputes and the withdrawal of individuals at the centre or top of the hierarchy"*. On the other hand, research has revealed that a multitude of contributions are made by only a small amount of people (Ye and Kishida, 2003; David, Waterman, and Arora, 2003). In another case study, Mockus, Fielding, and Herbsleb, 2002 found out that 80% of the new functionality is created by only 10-15 developers.

To enable a successful collaboration of all persons involved in the community, it is important that all contributors are communicating with each other. In their research, Crowston and Shamshurin, 2017 argue that community interactions are related to a project's success. They suggest that successful projects have a larger amount of communication which is almost evenly divided between the core (*Project Leader*, *Core Members* and *Active Developers*) and peripheral users (all other roles except *Readers* and *Passive Users*).

2.3. Development practices

Typically, source code is stored at a public software repository using a Concurrent Versions System (CVS). According to Fogel and Bar, 1999, a CVS has two main functions: (a) to keep record; and, (b) to enable collaboration.

Record keeping involves the maintenance of a commit history to allow the restoring of source code to a previous version at any time. To continue, source code can be modified by multiple developers following the *copy-modify-merge* development model as suggest by Fogel and Bar, 1999. A simplified version of this process looks as follows:

1. A developer requests a local working copy from the CVS.
2. The local working copy can be freely modified.
3. When the local changes are complete, the working copy can be committed to the CVS along with a message indicating what has changed.

A survey of eleven successful FOSS projects (Halloran and Scherlis, 2002) revealed that in relation to CVS, there is a distinction between two different kinds of developers: developers with commit privileges and developers without commit privileges. Commits by developers without privileges need to be reviewed by developers with commit privileges. Furthermore, all projects under study have automated nightly builds ensuring that the source code still compiles after changes have been committed. To continue, most projects have regression tests ensuring that the changes do not affect the existing behavior. The central point of development constitutes a public issue tracking tool where users can post bugs and feature requests. While duplicate bug reports are common, bug reports typically need to be accepted by core members before they become visible to others. Having a list of issues publicly available, a contributor then starts with the development of a preferred ticket. When the work is done, the changes are submitted to the CVS.

2.4. Communication

To increase awareness of the geographically distributed developers, Halloran and Scherlis, 2002 found out that *tool mediation* plays a critical role during development. While communication is solely done by the means of computer supported tools, an organization is able to build an organizational memory embodying a large extent of knowledge which would not be possible for traditional synchronous face-to-face conversations. Jensen, King, and Kuechler, 2011 discovered that mailing lists constitute the heart of all communications and discussions. In mailing lists messages are broadcasted to all subscribers making the development work transparent to increase the team awareness (Yamauchi et al., 2000). Messages are being archived and can be accessed at any time. Krogh, Spaeth, and Lakhani, 2003, Jensen, King, and Kuechler, 2011 and Ducheneaut, 2005 show that mailing lists offer a frequent starting point for newcomers who may spend weeks and months silently observing the community.

In his book, Fogel, 2009 recommends the use of different communication channels. Mailing lists and forums serve as archivable platform for asynchronous technical discussions and announcements. Real-time chat rooms, like Internet Relay Chat (IRC)[14] are employed to ask questions and get instant feedback from the community. This is in accordance with a study of the GNOME[15] project (Poo-Caamaño et al., 2017) which presents that their main communication channels are mailing lists and IRC. Additionally, GNOME's community is interacting using the issue tracking system, a wiki platform, developers' blogs and face-to-face meetings at conferences.

2.5. Motivation

While some FOSS projects are supported by commercial companies providing paid contributors, most organizations are depending on the contributions of volunteers (David, Waterman, and Arora, 2003; Dinh-Trong and Bieman, 2004). Therefore it is important to understand the motivation that drives these volunteers to contribute during their spare time.

According to Ryan and Deci, 2000, motivation moves people to do something. In their study it is argued that people have both different amounts and different kinds of motivation. Motivation is divided into an *intrinsic* and an *extrinsic* form. The former involves that an activity is done solely because of the satisfaction to do so rather than profiting from some supplementary consequences. On the other hand, extrinsic motivation leads to the performance of tasks in order to obtain some additional benefits. Thus, an

[14]Internet Relay Chat (IRC) is a text-based, instant messaging system
[15]A free and open-source desktop environment for Unix-like operating systems

activity is not done because of enjoyment but to benefit from a separable outcome, like receiving an external reward (Ryan and Deci, 2000).

A survey conducted by Lakhani and Wolf, 2003 shows that motivation of FOSS contributors is either

- enjoyment-related intrinsic because development is experienced as a pleasing and creative exercise,
- community-related intrinsic because it is perceived to have obligations with the community; or,
- extrinsic because contributors are getting paid for their contribution.

To continue, Ye and Kishida, 2003 demonstrate that volunteers who are simultaneously developing and using the software (see Section 2.1) are frequently getting involved due to the need for additional functionality. It is argued that one of the major driving forces that motivate contributors in FOSS projects is *learning*. By contributing to open source, developers learn from high-quality source code as well as from sharing knowledge with other developers. This complies with the findings of Yamauchi et al., 2000 who investigated the mailing list of the FreeBSD Newconfig project[16] and found out that questions were answered almost 2.7 times more often than new questions were placed, thus providing the community with a lot of valuable feedback and additional information.

2.6. Community engagements

As suggested by Jensen, King, and Kuechler, 2011 and Steinmacher, Wiese, and Gerosa, 2012, the success of FOSS projects strongly relies on a constant influx of newcomers. In order to engage a large number of people to contribute to existing FOSS communities, various community and coding engagement events are organized by multiple associations. Among the largest events are the Rails Girls Sommer of Code[17] where women receive a three-month scholarship to work on FOSS projects, the Google Summer

[16]http://www.jp.freebsd.org/newconfig/project.html, visited on 30 May 2019
[17]https://railsgirlssummerofcode.org, visited on 23 May 2019

of Code (GSoC)[18] and the Google Code-in[19] program. These programs typically offer a sponsorship to students to promote contributions to FOSS projects during the summer. Students get assigned to a mentor who guides them through their first steps into the community, the development process and the upcoming tasks.

2.6.1. Google Summer of Code

The GSoC program is the largest community engagement event and was initiated by Google in the year 2005. Since then this annual program increased strongly, having over 14,000 students from 118 countries accepted in the year 2018, as mentioned on their website[20]. Upon many other goals, the program wants to inspire young students to take part in the open source development and thus help FOSS communities to find and retain newcomers. As stated on their website, the main goals of the programs are:

- "Get more open source code written and released for the benefit of all."
- "Inspire young developers to begin participating in open source development."
- "Help open source projects identify and bring in new developers."
- "Provide students the opportunity to do work related to their academic pursuits during the summer: *flip bits, not burgers*."
- "Give students more exposure to real-world software development (for example, distributed development and version control, software licensing issues, and mailing list etiquette)."

Typically the three-month program lasts from the end of May to the end of August. Students from accredited universities can submit up to three proposals to up to 217 participating FOSS organizations (as of 2018). Once accepted by an organization, the program consists of a *community bonding phase* which serves as a starting point to get to know the people behind a community. Thereupon the actual development starts with the *coding period*.

[18]https://summerofcode.withgoogle.com, visited on 23 May 2019
[19]https://codein.withgoogle.com, visited on 23 May 2019
[20]https://google.github.io/gsocguides/student/index, visited on 23 May 2019

During all phases the student is paired with one or several mentors who serve as a central point for guidance and who introduce the student to domain- and project-related areas (Trainer, Chaihirunkarn, and Herbsleb, 2013).

This offers a great opportunity for organizations to engage newcomers to become long-term contributors. While a study by Trainer, Chaihirunkarn, Kalyanasundaram, et al., 2014 found out that around 18% of the GSoC students are becoming mentors in subsequent years, Silva et al., 2017 argue that around 64% of the students do not contribute longer than one month after the program has finished.

2.6.2. Google Code-in

Founded by Google in 2010, the main purpose of this annual program is to engage pre-university students with the age of 13-17 to become part of an open source community. According to their archive[21], 27 FOSS organizations participated in the year 2018 enabling the completion of 15,323 tasks. During this program each organization provides a large list of short tasks which should take no longer than 5 hours to complete. Similar to GSoC, organizations provide mentors who provide the participants with feedback and guidance and evaluate their work. Depending on their evaluation, participants can win multiple prizes provided by Google. Organizations, on the other hand, have the opportunity to introduce the next-generation students to their FOSS projects.

2.7. Entrance barriers

First-time contributors are frequently imposed to various barriers that hinder them from contributing. For some newcomers this can even lead to the abandonment of open source development as a whole (Steinmacher, Conte, Gerosa, et al., 2015). In order to counteract these barriers it is important to

[21]https://codein.withgoogle.com/archive/, visited on 23 May 2019

understand what hindrances are experienced while placing a first contribution. In their study, Steinmacher, Silva, et al., 2015 classified these barriers into five categories:

1. social interaction
2. newcomers' previous knowledge
3. finding a way to start
4. documentation
5. technical hurdles

The first category represents the largest, appearing in 60% of all cases. As explained in their study, problems within this category mainly arise because of (a) *a lack of social interaction with project members*, (b) *receiving an improper answer*; and, (c) *not receiving a (timely) answer* from the community.

In order to allow newcomers to readily start contributing and to enable a constant acquisition of new community members, it is important to counteract these entrance barriers. This is a necessary step in order to ensure the long-term success of a FOSS project (Jensen, King, and Kuechler, 2011).

2.8. Demographics

A survey of 1,588 FOSS developers conducted by researchers of the Stanford University illustrate that the majority of FOSS developers are living in Western Europe (52.7%), North America (27.1%), Russia and Eastern Europe (7.6%) (David, Waterman, and Arora, 2003). Their findings comply with Jensen, King, and Kuechler, 2011 who revealed that almost all FOSS developers are male (98.4%), leaving the female part to a minority comprising only 1.6%. While half of the contributors are between 23 and 33 years old, the age ranges from 11 to 69 (David, Waterman, and Arora, 2003).

Even though most projects understood the importance of diversity within their communities (Poo-Caamaño et al., 2017), Nafus, Leach, and Krieger, 2011 conducted a survey on gender among FOSS developers and found out that *"half of the woman observed or experienced discriminative behaviour against women, but only about one out of ten men had the same perception."*. They argue that discriminative behavior is often not perceived as such by men, mostly expressed in the sense of "jokes". To foster diversity, many FOSS projects nowadays focus on attracting and retaining women (Jensen, King, and Kuechler, 2011).

3. Agile software development

In the context of software engineering, the term *agile* became widespread when Beck, Beedle, et al., 2001 met to discuss their similar beliefs and attitudes about software craftsmanship. In the "Manifesto for Agile Software Development" they expressed their common values and beliefs in writing. Following four basic values, agile software development is based on the use of light but effective project rules as well as human- and communication-oriented rules (Cockburn, 2001), as listed by Beck, Beedle, et al., 2001:

1. **Individuals and interactions** over processes and tools
2. **Working software** over comprehensive documentation
3. **Customer collaboration** over contract negotiation
4. **Responding to change** over following a plan

Based on these ideals, the manifesto states twelve principles which put the human in the centre as the most important ingredient for success. Martin, 2006 states that principles and rules are important, but *"it's the people who make them work"*. The sixth principle which is especially important when it comes to team communication underpins the necessity of direct, face-to-face conversations as *"the most efficient and effective method of conveying information to and within a development team"* (Beck, Beedle, et al., 2001). Additionally, agile software development embraces change as emphasized in the second principle which advocates to *"welcome changing requirements, even late in development"*. Thus, the term *agile* refers to the flexibility and the ability to constantly react to changing environments to gain a decisive competitive advantage. Pursuing these values and principles, the literature lists multiple successful agile software development methodologies, including Extreme Programming (XP), Scrum, Lean Software Development (LD), Feature-Driven Development (FDD), Adaptive Software Development (ASD) and Crystal (Chow and Cao, 2008). In the remainder of this section, two important approaches will be discussed.

3.1. Extreme Programming (XP)

Extreme Programming (XP) as introduced by Beck and Gamma, 2000 is a collection of practices designed to develop quality software in teams and on time. XP focuses on team work, early and continuous feedback, progressive planning approaches, flexible scheduling depending on business changes and on an evolutionary design process. As proposed by Beck and Gamma, 2000, XP reduces project risk, is open to changes, increases productivity and, among other things, makes fun. On the other hand, XP counteracts problems related to communication, unclear requirements and rapidly changing environments. This is done with the help of various activities, practices and variables, as described in the subsequent paragraphs.

3.1.1. Variables and values

In XP there are two basic groups of people: the customer and the development team. According to the authors, it is most important that these groups have a close cooperation and efficient communication during all phases of development (Beck and Gamma, 2000). To continue, XP embodies four project variables: (1) *cost*, (2) *time*, (3) *quality*; and, (4) *scope*. At the beginning of a project, the customer has to pick the value of any three of these variables which constitute the most important ones according to the customer's perspective. The developers then assign the value of the remaining fourth variable which is a flexible variable and can be adjusted during various releases. In order to accomplish this, Beck and Gamma, 2000 emphasize that XP focuses on four important values:

- **Communication**. In many software projects, bad and infrequent communication is the main cause of problems. Thus, XP proposes an efficient, frequent communication within the development team as well as between the customer and the developers.

- **Simplicity**. XP emphasizes the importance to develop simple today and pay attention to changes whenever the situation requires it.

- **Feedback**. The customer needs to provide a constant feedback of the current state of the project to foster continuous improvement and learning.

- **Courage**. It is essential that the team is motivated, open to learn new things, open for critics and always tries to give its best.

3.1.2. Activities and practices

One of the basic activities in an XP project is *listening*. By listening to the customer, the developers gain domain knowledge and business expertise within the customer's business domain. This is an essential element of this methodology since it enables the developers to better understand the customer's requirements. Besides other important activities like coding, testing and designing, XP is based on 12 principles. In the subsequent paragraphs the most important principles are listed as explained by Beck and Gamma, 2000:

The Planning Game. Business and technical experts regularly meet to estimate new features according to their business value and according to their technical effort and dependencies. As a result, the scope of the next release is provided by prioritizing features according to their costs and benefits.

Small releases. During all phases of development, XP offers two strategies: (a) frequent iterations of small releases; and, (b) focus on the most important functionalities first. In this way both risk and costs can be reduced since changing requirements cost less money in early stages of the development process.

Collective ownership. The code base is not split into several parts each maintained by individual programmers. Instead, the whole developer team is responsible for the whole code base allowing all parts to be changed by everyone. In order to enable this, every engineer needs to be familiar with all parts of the system. Consequently, this practice is ensuring that knowledge is not getting lost when core developers are leaving. This problem is also known as the truck factor, reflecting *"the number of people on your team who have to be hit with a truck before the project is in serious trouble"* (Bowler, 2019).

Pair programming. This practice involves two developers working collaboratively on one machine. One person (the driver) has control over the computer and writes code while the other is observing (the observer or navigator). After a certain amount of time, the roles are periodically switched (Williams et al., 2000). Pair programming provides an opportunity to effectively transfer knowledge within a team and to foster team building (Cockburn and Williams, 2001). Thus, this practice is counteracting the aforementioned truck factor problem. Williams et al., 2000 found out that through pair programming software can be produced with higher quality, among other things because coding standards are followed more accurately and design quality is improved.

Testing. To save time for potential debugging and to gain confidence in code changes, tests are written to guarantee that changes do not break the system. This is an essential part to foster collective code ownership and to enable continuous refactoring. XP also allows customers to write functionality tests in the sense of stories. As a result of testing, the software becomes more reliable and trust-worthy by both the customer and the developers. From a developers' point of view, Test-Driven Development (TDD) plays a fundamental role. TDD involves the writing of unit tests prior to the implementation of the actual functionality (Martin, 2006). At the beginning tests fail because of the functionality under test does not exist. Thereupon, the missing functionality is implemented until the previously failed test passes. At this point, the developer begins to refactor code in order to increase readability and improve design. When the tests still pass, the developer is confident that the refactoring did not break the system. In this way code is designed to be testable and modularity is increased because code is written in a fashion which tends to be much less coupled (Martin, 2006; Martin, 2008).

In addition to these practices, XP uses a *metaphor* for every project, focuses on *simple design* decisions, embraces continuous *refactoring* and requires the compliance to a predefined *coding standard*. Finally, importance is attached to the presence of an *on-site customer* as well as to a *40 hour week*.

3.2. Kanban

kanban is a compound Japanese term consisting of the words *kan*, meaning "signal", and *ban* which translates to "card" (Epping, 2011; Anderson and Carmichael, 2016). The term was first introduced by Taiichi Ohno at the Toyota Motor Company (Sugimori et al., 1977). In this context, Kanban is considered as a production control system for just-in-time manufacturing (JIT) designed for making the full use of the employees' capabilities. Anderson and Carmichael, 2016 summarize Kanban as *"a method that shows us how our work works"*. A shared understanding of work is established in order to facilitate continuous improvement and to become more predictable. Kanban, as a method for managing and improving knowledge-based services, first occurred in the context of software engineering in the year 2004 when Microsoft employed this method for the first time (Epping, 2011). Since then Kanban has been successfully employed in a large number of agile software projects.

3.2.1. Practices

According to Anderson and Carmichael, 2016, Kanban is based on six general practices which all involve the transparency of work and rules, continuous improvement as well as an ongoing learning process. In this section these practices are briefly introduced.

Visualize. To understand the current state of a system, the work and the process it goes through, also known as workflow, needs to be visualized on a *Kanban board*. In this way, information about ongoing work as well as information about potential bottlenecks and blockades are accessible to the whole team. The design of such a board is not limited, yet it is commonly visualized as a two-dimensional grid representing every step of the workflow process as a column. Work items correspond to cards in the grid whereas the items' respective column represents the current work progress. An example of a digital depiction of a Kanban board can be found in Appendix C. Furthermore, it is essential that work is limited for every workflow step, as mentioned below.

Limit work in progress. Columns in the aforementioned board are limited to a certain amount of items, thus limiting the overall work in progress (WIP). To facilitate this, tickets are "pulled" from one column to another rather than "pushed" when the work is done. According to Anderson and Carmichael, 2016, too much partially completed work is wasteful as it increases the lead time since it can not be used in releases yet. To enable WIP limits, tasks need to be prioritized. As a consequence, lead time decreases and predictability increases (Epping, 2011; Ahmad, Markkula, and Oivo, 2013; Anderson and Carmichael, 2016).

Manage flow. Processes should be optimized in such a way that work can flow as fast as possible through all stages of the system. In order to maximize this flow, it is important to measure and quantify the flow's value. This is frequently done with the help of the *cost of delay*. This figure represents a delivery's lost value when the implementation of certain work items is delayed or postponed to a subsequent release (Anderson and Carmichael, 2016). Work items need to be managed in order to facilitate a constant and maximized flow.

Make policies explicit. To enable a continuous improvement, it is essential that the team understands all processes and its underlying rules and policies. Kanban is trying to make policies transparent by allowing the team to jointly decide on their rules, for example to define a policy stating which criteria must be met in order to pull work items to their next step (Epping, 2011). As a result, team communication is fostered to increase awareness and understanding. Additionally, transparent policies enable a technical discussion to question or update certain policies, hence facilitating continuous improvement (Anderson and Carmichael, 2016).

Implement feedback loops. Feedback loops are kept short to quickly adapt processes and to foster constant learning (Ahmad, Markkula, and Oivo, 2013). Feedback, both on an intra-team as well as on a project level, is provided at regular meetings and periodical reviews (Anderson and Carmichael, 2016).

Improve collaboratively, evolve experimentally. Kanban fosters change. Different to other approaches where the desired outcome of change processes is well defined, Kanban does not offer fixed endpoints since *"perfection in an ever-changing fitness landscape is unattainable"* (Anderson and Carmichael, 2016). Consequently, Kanban emphasizes the necessity to continuously implement new approaches and to constantly reverse changes which are considered to be ineffective.

By following these practices, Ahmad, Markkula, and Oivo, 2013 found out that projects benefit, among other things, from improved communication, improved software quality and increased productivity.

4. Catrobat: an agile FOSS project

Catrobat[1] is a non-profit FOSS project initiated by Wolfgang Slany, a professor at the Institute of Software Technology at Graz University of Technology. Founded in 2010, the main goal is to develop a visual programming language for mobile devices. Inspired by Scratch[2], a further visual programming language developed by the Lifelong-Kindergarten-Group at the MIT Media Lab, Catrobat enables children and teenagers to create and execute programs in a visual "LEGO-style" solely using their smartphones (Slany, 2012). By doing so, the project has an educational purpose allowing young people to foster conceptual thinking and learn programming in a visual way, thus becoming creators instead of being consumers.

Programs built with the Catrobat language can be easily created and shared using a mobile development environment which is freely available by downloading an app named "Pocket Code". The app is available for Android and iOS devices and can be downloaded from the Google Play Store[3] and Apple's App Store[4]. With the help of Pocket Code, Catrobat programs can be developed without the need of a computer. A large number of pre-defined commands are available in the means of Bricks which are block shapes, similar to Lego bricks, used to create code. Additionally, the device's camera and numerous device sensors (like the compass, or the acceleration and inclination sensor) can be easily utilized in the users' programs. To get a better understanding, Figure 4.1 shows an example of various Bricks. Once a program has been created, Pocket Code allows its user to upload and share their creative work with others using a community website. This integral

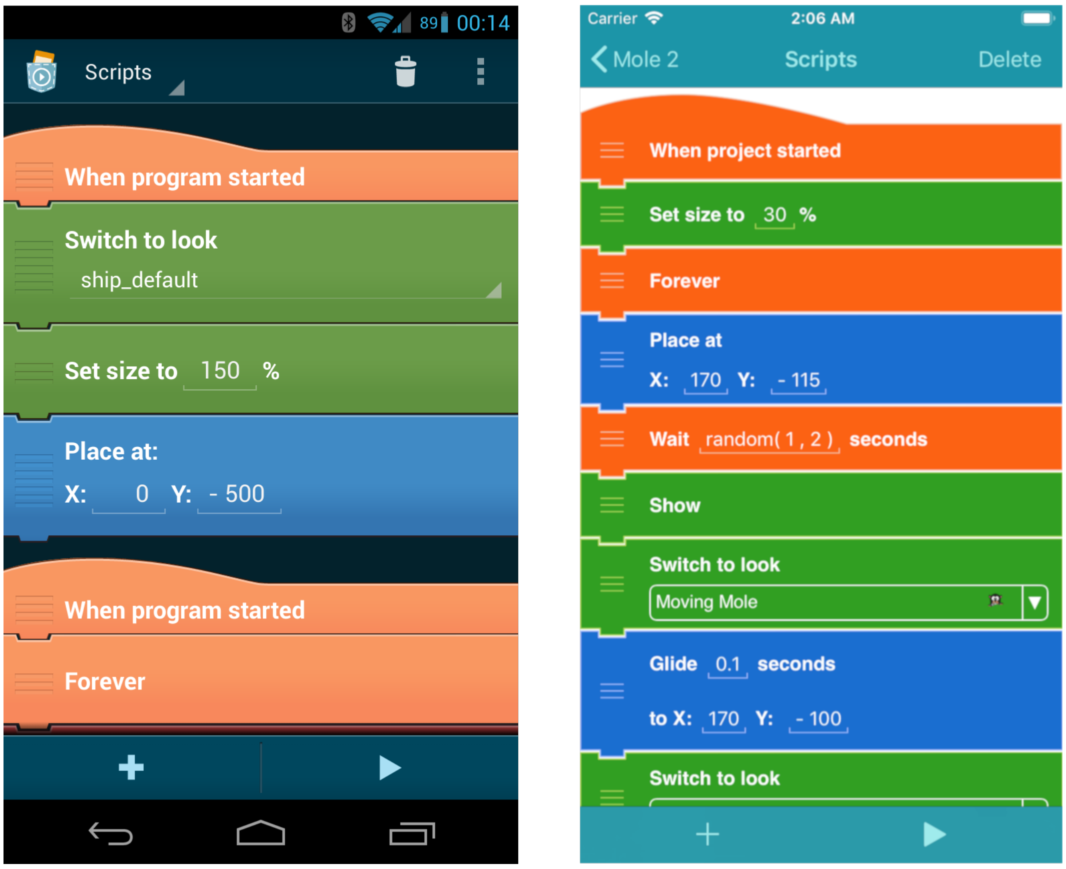
[1]https://www.catrobat.org, visited on 2 June 2019

[2]https://scratch.mit.edu, visited on 2 June 2019

[3]https://play.google.com/store/apps/details?id=org.catrobat.catroid, visited on 2 June 2019

[4]https://itunes.apple.com/app/pocket-code/id1117935892, visited on 2 June 2019

part of the project enables teenagers to creatively express themselves and allows them to learn from each other by studying each other's code and by getting inspired by new ideas. Being motivated by someone else's idea, a project can be downloaded and enhanced with additional functionality – this feature is known as "remixing".



```
When program started
Switch to look
  ship_default
Set size to 150 %
Place at:
X: 0 Y: - 500
When program started
Forever
```

```
When project started
Set size to 30 %
Forever
Place at
X: 170 Y: - 115
Wait random( 1 , 2 ) seconds
Show
Switch to look
  Moving Mole
Glide 0.1 seconds
to X: 170 Y: - 100
Switch to look
```

Figure 4.1.: Pocket Code on Android (left) and iOS (right).

In the practical part of this thesis, contributions to the Catrobat project will be investigated in close details. To get a better overall pricture of the project, the remainder of this section gives insights into Catrobat's structure, contributors and development practices.

4.1. Motivation and educational aspects

On the one hand, the demand for software systems is vastly increasing throughout numerous parts of our society (Slany, 2012). On the other hand, the high complexity of software is less and less understood by people not affiliated with computer science. Raising this discrepancy, the National Academies of Sciences, 2018 found out that the outsized number of computer science jobs grows much faster than the number of graduates enrolled to a computer science bachelor program. To counteract this problem, it is important to foster computational thinking from an early age and to motivate children to get inspired by the world of software engineering. Catrobat is making its contribution to this problem by providing children with a platform to create and share their own games. Games are published under an Open Source and Creative Commons (CC) license allowing to get downloaded and modified by other people than the author. This exchange of ideas not only enables teenagers to learn from each other, but also gives an insight into the open source philosophy.

But that's not all: as listed in Section 4.3, Catrobat's development team mainly consists of university students, thus trying to introduce students to open source as well. This two-sided educational effect represents a unique setting and offers benefits for all parties. A survey conducted by Müller, Schindler, and Slany, 2019 revealed that there are strong indicators that university students are benefitting from their contribution at Catrobat in their later careers. One reason for this is that students have to deal with professional development tools, high-quality code and collaborate with experienced developers.

Catrobat sets its focus on the mobile device market, a market with a tremendous growth. Smartphones are available at low-prices enabling children from all over the world, starting from a young age to have access to their own devices (Slany, 2012). Since Pocket Code runs on almost every modern smartphone and does not require computer access, the app is being used by a large and diverse group of users of different age and from a large variety of different countries, including developing countries.

4.2. Community structure

The non-hierarchical organizational structure at Catrobat promotes innovational thinking and enables a rapid flow of information between the contributors. The project is managed by Professor Wolfgang Slany and a small management board of highly involved contributors. Development is carried out by multiple teams each concentrating on a special field of expertise. Figure 4.2 illustrates the organigram based on the internal team structure as of June 2019. The nature of open source code enables a contribution to almost every part of the project. Contributors are not limited to be part of one single team but are allowed to freely shift to different activities. Since the majority of contributors are connected to the university and voluntarily contribute as part of their bachelor's projects and master's thesis, there is a high fluctuation within the community. This complies with the findings of Müller, Schindler, and Slany, 2019, who found out that a great number of contributors stay in the community for less than one year only. Thus, to ensure the long-term success of the project, several challenges are faced.

Like in other FOSS projects, there needs to be a constant influx of newcomers (Jensen, King, and Kuechler, 2011) to ensure a constant size of the community and the long-term success of the project. To continue, information and organizational rules need to be communicated effectively to enable a knowledge transfer between contributors ensuring that knowledge is not getting lost when core developers are leaving. In conventional software projects this problem is also known as the truck factor, as discussed in Chapter 3.

4.2.1. Roles

To apply the roles discussed in Section 2.1 to the Catrobat community, this section provides a list of roles available throughout all teams. Although the roles at Catrobat deviate from those originally proposed by Ye and Kishida, 2003, there is a similar role transformation allowing all contributors to transform to different roles depending on their phase of contribution. Figure 4.3 illustrates the different roles whereas the contributors' influence

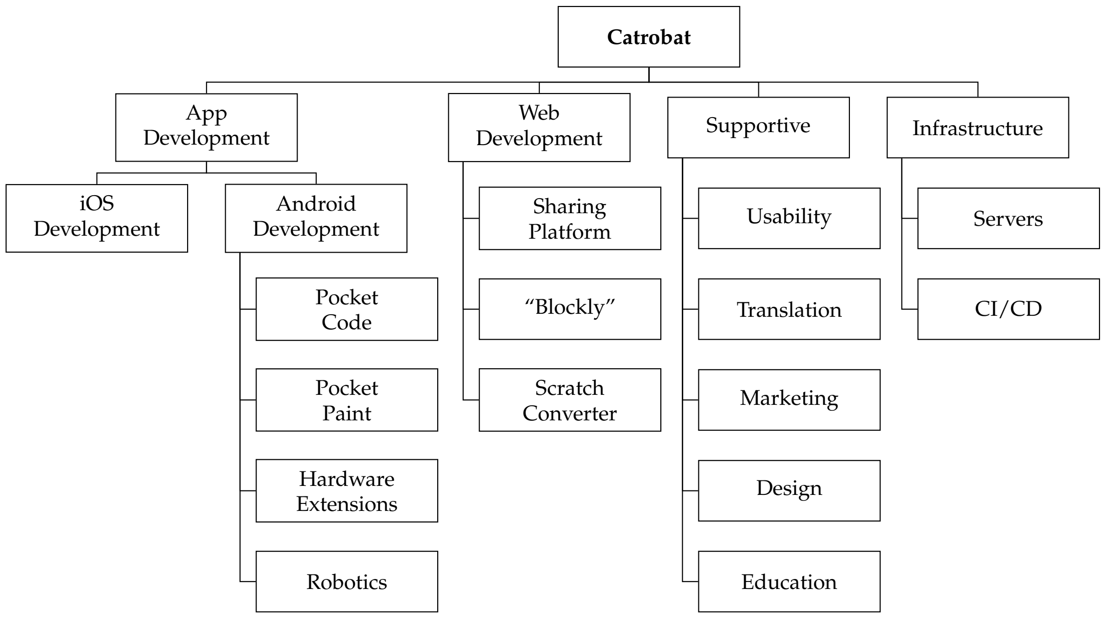


Figure 4.2.: Team structure of Catrobat as of June 2019 (based on Müller, Schindler, and Slany, 2019)

increases when moving towards the core and the number of people increases when moving away from the core. In the project, the following roles are identified throughout all teams:

- **Project Head**: The head of the project is the founder who is responsible for communicating the project's vision and mission. The project head has the most influence.

- **Product Owners**: A board of a few people highly involved into the project. Contributors carrying out this role are responsible for the project's organization and strategic management.

- **Coordinators**: Each team has one coordinator who acts as an information disseminator and represents a bridge between the developer team and the management board.

- **Senior Contributors**: Contributors practicing this role have a pertinent technical expertise and a thorough understanding of the project's overall picture. They are responsible for development, code acceptance and for introducing newcomers.

- **Active Contributors**: This role represents the majority of all contributors within the community. Their main task is code development in the form of bug fixing and the development of new features.

- **Peripheral Contributors**: Contributors who are no active part of the community but occasionally contribute by opening a pull request and/or creating a bug ticket are called peripheral contributors.

- **Passive Users**: Instead of contributing, people of this role are mainly using the software. This role primarily consists of children.

Different to other FOSS projects, where a role transformation is possible between all roles within the community (Ye and Kishida, 2003; Hippel and Krogh, 2003; Scacchi, 2002), there is usually no transformation between Passive Users and other roles observed within the Catrobat community, thus clearly separating developers and users.

4.3. Contributors and Demographics

The hybrid nature of this project allows both student of the Graz University of Technology and external developers to contribute. As of 2 May 2019, Catrobat's main code repository consists of contributions from 203 different developers. Figure 4.4 illustrates that more than 80% of the contributors were working from Austria at the time of the study. Additionally it is shown that altogether 84.24% were affiliated with the Graz University of Technology. From now on this subset is referred to as *internal contributors* in contrast to the other 15.76% which are named *external contributors*. The data was manually collected from the *develop* branch of the *Catroid* repository[5] and assigned to countries by evaluating the contributors' profile page on

[5]https://github.com/Catrobat/Catroid

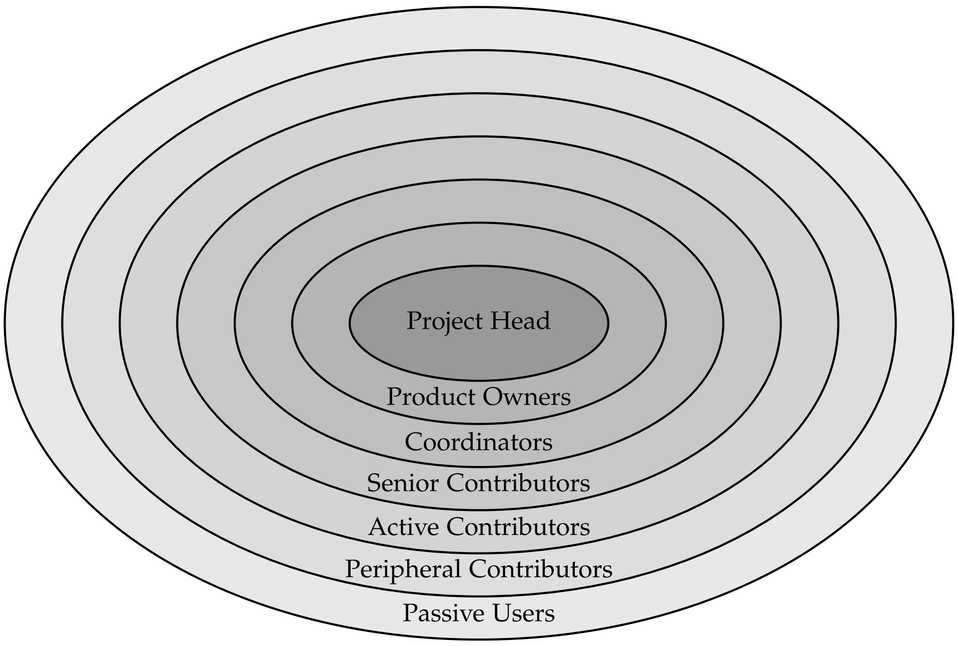


Figure 4.3.: Roles in the Catrobat community

GitHub and Catrobat's internal Confluence[6] page using the contributors' email addresses and full names if available. Contributors who could not be allocated to a specific country were assigned to the country named *Other*. Contributors affiliated with the Graz University of Technology were either having an email address ending with *tugraz.at* or were having an entrance in the project's Confluence page confirming the affiliation.

Including active and inactive members of all sub-teams, the Catrobat development team involves more than 370 members, as listed on the credits page[7].

[6]A collaboration software published by Atlassian, the company behind Jira
[7]https://developer.catrobat.org/credits, visited on 2 June 2019

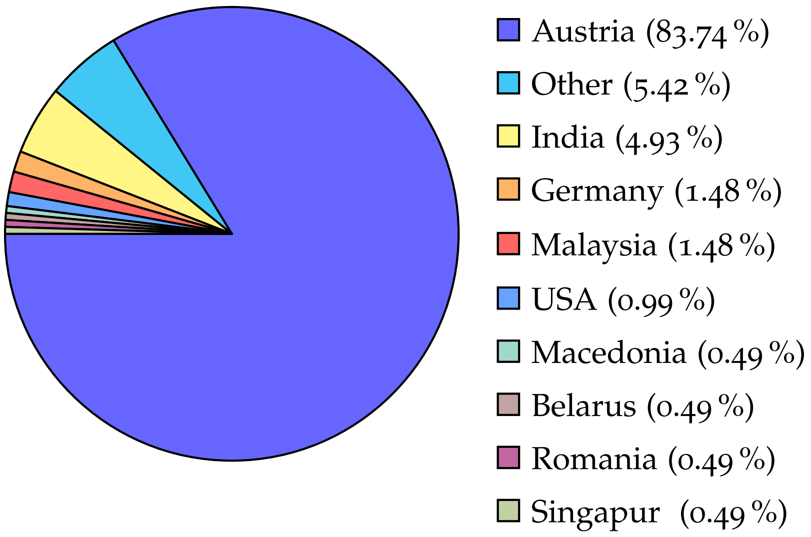


Figure 4.4.: Contributors of the *Catroid* repository grouped by country (rounded to two decimals)

4.4. Development practices

At Catrobat development is strongly influenced by agile development methods. Kanban (see Chapter 3) is used to improve software productivity, increase team awareness and to quickly react to a rapidly changing environment. Additionally, Extreme Programming (XP) is employed to foster collaboration and collective code ownership and to improve communication, knowledge transfer and code quality. Furthermore, TDD (Beck, 2003) is used to get an instance feedback of newly written code, to create a detail specification of what's required from the code and to ensure that the code is still functionally correct after refactoring. While the details of these agile methods are illustrated in Chapter 3, this section gives an insight into the development tools and processes utilized at Catrobat, as summarized in the project's README file[8]:

```
If you want to contribute we suggest that you start with forking
```

[8]https://github.com/Catrobat/Catroid/blob/develop/README.md, visited on 14 June 2019

```
our repository and browse the code. Then you can look at our
Issue-Tracker and start with fixing one ticket. We strictly use
Test-Driven Development and Clean Code, so first read everything
you can about these development methods. Code developed in a
different style will not be accepted.  After you've created a
pull request we will review your code and do a full testrun
on your branch.
```

4.4.1. Issue tracking

Jira[9] is used as an issue tracking and agile project management tool. The platform can be accessed publicly with a limited functionality allowing visitors to view all open tickets. A ticket can have two different types: (a) a *Bug* indicating an error or a failure; or, (b) a *Story* describing a new feature in an informal, natural language. Each ticket describes all necessary information in order to start development and serves as an acceptance criteria for the code review, as explained in the remainder of this section.

Newly created tickets need to be confirmed by a Product Owner (PO) in order to become visible by others. This ensures that the focus is put on important development tasks first and that the number of tickets currently in development is limited, as suggested by the work in progress (WIP) limit of Kanban. To get a better overall picture of what's currently in development and which tickets are ready for code review, the workflow of each team is visualized in a Kanban board. Jira also provides information about the persons involved during the development and review process providing an important information for newcomers to find potential guidance for detailed questions of a specific topic.

4.4.2. The Planning Game

In order to decide which tickets are visible and ready for development, regular "Planning Games", an XP practice mentioned by Beck and Gamma,

[9]https://www.atlassian.com/software/jira, visited on 14 June 2019

2000, are held at the Graz University of Technology. These meetings can only be attended by local contributors, which can be seen as a limitation. Nonetheless, since most contributors are students from this university (see Section 4.3), the majority of contributors are able to attend these local meetings. As mentioned in Chapter 3, the Planning Game offers an excellent opportunity to vaguely communicate desired requirements and estimated efforts between POs and the developer team, thus combining business priorities and technical estimates (Beck and Gamma, 2000). Finally, this meeting can also serve as an additional aid to clarify unclear requirements from the developer's point of view.

4.4.3. Code repository

The source code of all teams is available on GitHub[10] and can be publicly viewed and copied by everyone. Like most other FOSS projects, Catrobat is using the *copy-modify-merge* development model as introduced in Section 2.3. Thus, although the source code can be freely copied and locally modified by others, it can only be merged back into the project's repository by senior contributors (see Section 4.2.1). Catrobat is using the "Gitflow Workflow" as suggested by Atlassian[11].

4.4.4. Code review

Once local changes are made they need to be submitted for a code review by opening a *pull request* on GitHub. In case of a successful review, the changes are becoming part of the project's code base. If that is not the case, the author of these changes is either requested to make additional changes (in a *change request*) or the whole changes are discarded. Figure 4.5 illustrates a pull request where merging is blocked prior to the submission of a code review. This review process is an essential part of the development process since it ensures both a high code quality and a high customer satisfaction. To further

[10]https://github.com/Catrobat, visited on 14 June 2019

[11]https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow, visited on 14 June 2019

increase quality from a technical and a functional perspective, Catrobat's review process has changed in the last few years. While the review initially merely consisted of a technical check by a senior contributor, the review process has been extended by an additional step: the validation by at least one PO. This supplementary approval guarantees that the code is not only well written and tested from a technical point of view, but also behaves as expected from a domain-related, functional view. Section 4.2.1 outlines that, different to other FOSS projects, there are no developer-users at Catrobat, making the PO review inevitable in order to overcome a potential lack of domain expertise.

From a technical perspective, a pull request needs to confirm to the principles suggested in the book Clean Code by Martin, 2008. Furthermore, every change must be reflected by an according test case. In order to be approved, both the existing tests and the newly created tests need to be passed.

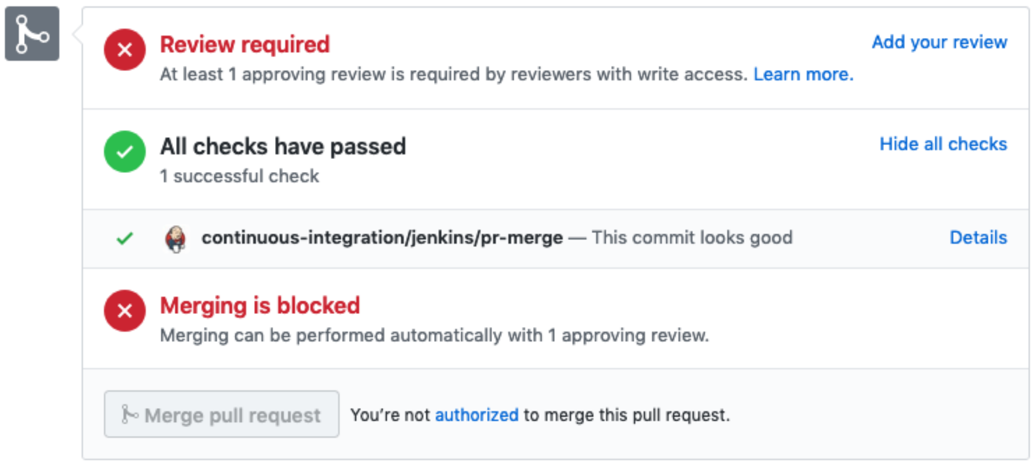
4.4.5. Continuous integration

To speed up the process of testing and code integration, the open source automation server Jenkins[12] is used. After a pull request is submitted, Jenkins checks out the respective changes, builds it and runs the automated test suite. The results are reported back to the web interface of GitHub, as illustrated in Figure 4.5, and serve as a basis for the aforementioned code review. Additionally, the main code base is checked out automatically several times a day to ensure that the software still builds and all tests are passing, even after a successful integration of a pull request.

4.4.6. Test-driven development (TDD)

In addition to the existence of meaningful and readable tests, contributors are advised to employ the *red/green/refactor* pattern as introduced by Beck, 2003. TDD involves the writing of tests prior to the actual code which is being tested. By doing so, developers are more likely to create a detailed

[12]https://jenkins.io, visited on 14 June 2019


Review required

At least 1 approving review is required by reviewers with write access. Learn more.

Add your review

All checks have passed

1 successful check

Hide all checks

✓ continuous-integration/jenkins/pr-merge — This commit looks good

Details

Merging is blocked

Merging can be performed automatically with 1 approving review.

Merge pull request

You're not authorized to merge this pull request.

Figure 4.5.: Test results and required review for a pull request on GitHub

specification of what's required and to write code which solely focuses on this specification and nothing else. Finally, to enable TDD the software design needs to be well thought out prior to the development which usually results in cleaner and better modularized code (Beck, 2003).

4.4.7. Pair programming

Local contributors are profiting from a team room which is provided by the Graz University of Technology. Contributors are highly engaged to meet there to carry out pair programming, an XP practice mentioned by Beck and Gamma, 2000. As explained in Chapter 3, this practice helps to improve the quality of software design and code and offers a unique opportunity to share knowledge, improve communication and foster team building (Cockburn, 2001; Williams et al., 2000). Secondarily, applying this practice can help to reduce the risk of losing key programmers, as explained by the aforementioned truck factor.

4.5. Documentation

Instead of employing a heavyweight, documentation-driven development process, Catrobat makes use of software informalisms as proposed by Scacchi, 2002. Informalisms used in the form of mailing lists, discussion forums, online chats or posts on wiki-like systems are utilized as resources to describe and document all relevant happenings in the project. This process is in agreement with the Manifesto for Agile Software Development which states that *"working software"* is preferred *"over comprehensive documentation"* (Beck, Beedle, et al., 2001). Catrobat was facing some problems in the past where the focus was more on formal, comprehensive documentation. Documentation became outdated, owners and maintainers left the project and there was no clear overview of the content available, as pointed out by Fellhofer, Harzl, and Slany, 2015. This complies with the findings of Steinmacher, Conte, Treude, et al., 2016 who confirm that documentation issues are one of the main entry barriers newcomers are facing in FOSS projects. Nowadays at Catrobat, documentation is primarily written in one of the following forms:

- **READMEs and how-tos** are briefly formulated guides publicly available on GitHub[13], primarily addressed for first-time contributors. These documents serve as rough guides to find out how to get in touch with the community and how to start contributing.

- **Confluence pages** serve as artifacts for meeting notes, blog posts and general information about the project. Additionally, how-to guides are available for code review, pull requests and for how to write clean code. This system is for internal use only and can not be accessed without credentials. However, credentials with limited access can be obtained via self-registration.

- **Software tests** serve as a human-readable basis to understand how code can be used as well as to get an insight of what the code is doing without even looking at it. Instead of writing a comprehensive

[13]An example README is available at (visited on 14 June 2019):
https://github.com/Catrobat/Catroid/blob/develop/README.md

technical documentation about the code base and instead of writing inline documentation in the form of code comments, software tests are used as the only form of technical documentation.

4.6. Communication

At the beginning of the project, communication mainly took place via face-to-face meetings at university (Fellhofer, Harzl, and Slany, 2015). While this approach worked fine for local students, it was not feasible for remote contributors not affiliated with the Graz University of Technology. To counteract this problem, a project-wide IRC channel was introduced open to everyone, as suggested by Fogel, 2009. Since the technology seemed old-fashioned to the contributors and the majority of Catrobat's contributors has never used IRC before, this turned out to be no practicable solution for the long run (Fellhofer, Harzl, and Slany, 2015).

To come up with a better solution, Slack[14] was introduced in 2017 and eventually became the main communication channel for both intra- and cross-team conversations. As of 15 June 2019, Catrobat's Slack channel counts 175 members in 33 public channels. Figure 4.6 illustrates that direct messages (DM) between two contributors are strongly preferred to conversations in public channels. While messages in public channels can be read and answered by all contributors of the community, DM can only be received by the two conversation partners.

Local contributors are benefitting from regular team meetings which take place in the team room at the university. Meetings are held individually for every team but can be joined by all interested parties. Additionally, to strengthen cross-team communication, a biweekly coordinator meeting (*Bi-WeCo*) takes place. At this meeting, each team reports its incidents, updates and achievements of the last two weeks.

[14]A team communication tool available at https://catrobat.slack.com, visited on 15 June 2019

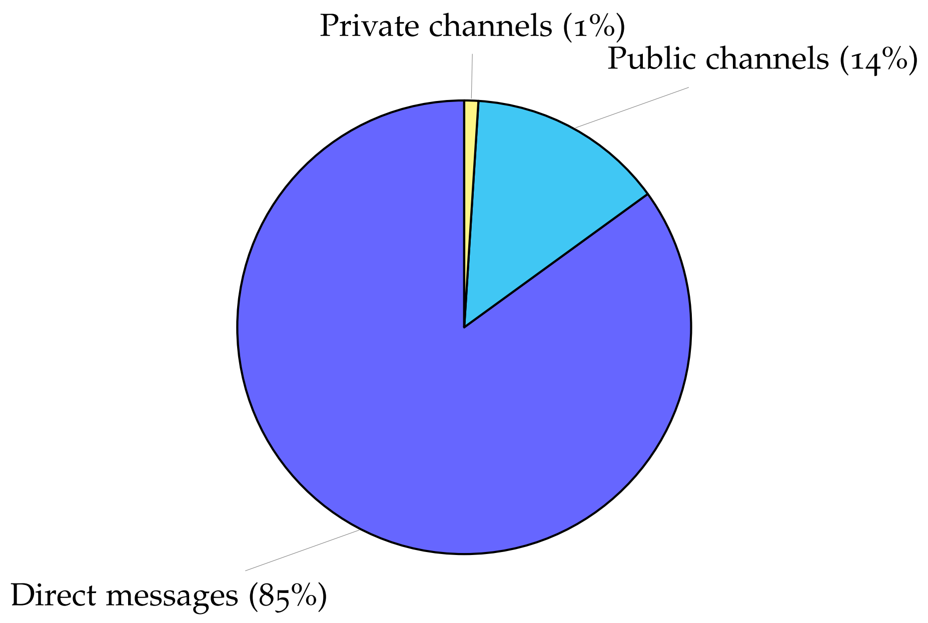


Figure 4.6.: Conversations in Slack grouped by channel (as of 15 June 2019)

5. Research design

This chapter illustrates the research methodology employed in this thesis, as summarized in Figure 5.1. The aim of this research is to (1) find out how and why new contributors are interacting with the community, (2) identify potential challenges showing up during their contributions, (3) investigate how they can be overcome; and, (4) give recommendations of what works best as well as to discuss the outcomes with correlating findings in literature to motivate further research.

To better understand the complex context of this socio-technological environment, a multiple case study was conducted presenting the case of Catrobat. Providing valuable insights into the collaborative software development process of an agile FOSS project, the research subject of this thesis is limited to the Catrobat project. Case studies are useful methods for conducting "how" and "why" research questions whereas the researcher has little control over the investigated, contemporary events (Yin, 1984). Instead of trying to generalize the results from this case study to a wider population (statistical generalization), the purpose of this approach is to extend and generalize theories (analytic generalization), as suggested by Yin, 1984 and Strauss and Corbin, 1990.

As illustrated in Section 6.2, multiple data from different sources is collected resulting in both qualitative and quantitative data to get a more holistic understanding of the subject under study (Yin, 1984). Grounded theory (Glaser and Strauss, 1967) is used to iteratively analyze the qualitative data. Different to the initial variant developed by Glaser and Strauss, 1967 where the research question is developed from the emerging codes during data collection and no literature should be reviewed prior to data analysis, this study integrates the Straussian approach (Strauss and Corbin, 1990) which adheres to the processes applied in the case study methodology. The integration of grounded theory as a method for the case study methodology is a combination commonly used in information system research (Halaweh, Fidler, and McRobb, 2008). As proposed by Yin, 1984, the researcher starts with a literature review and a set of propositions in order to be able to focus on a certain kind of information during the data collection. This complies with grounded theory developed by Strauss and Corbin, 1990 which suggests that the research should be started with a literature review and a vague definition of the research questions which can be further developed by emerging ideas.

While analyzing the case studies, conceptual categories are identified using open, axial and selective coding (Strauss and Corbin, 1990). Open coding was conducted multiple times on each of the qualitative data sets until no more new codes emerged, thus theoretical saturation was reached (Glaser and Strauss, 1967). A repetitive application of axial coding resulted in a refinement of codes and categories and the identification of relationships between these. Selective coding finally led to a predominant core category and several sub-categories which is represented by the formulation of propositions. Different to hypotheses, which require measured relationships, Pandit, 1996 emphasizes that propositions require conceptual relationships which complies with the application of grounded theory. The case study outcomes and the resulting propositions are listed in Chapter 7.

To build up a "chain of evidence", as suggested by Yin, 1984, three semi-structured interviews are conducted with long-term experts of the Catrobat FOSS project. This chain combines the perspective of both internal and external contributors, as suggested in Section 4.3. The outcomes are then compared with the generated theory resulting from the case studies. A blend of open-ended and specific, closed-ended questions helps to ensure that in addition to expected information, new and unexpected types of information can be collected (Seaman, 1999). The experts have a detailed knowledge about the project and a profound experience in collaborating with other contributors making them qualified to precisely respond to complex questions. Consequently, this research method is combining new points of views with prevalent outcomes from the case studies. The interview guideline, which is enclosed in Appendix F, ensures to address the most significant parts of the propositions under study. On the other hand, the open-ended questions were designed to enable an interactive conversation between the interviewee and the interviewer to get an insight into new and unexpected points of views (Turner III, 2010). The interview transcriptions are analyzed using the aforementioned coding procedure suggested by Glaser and Strauss, 1967. Bogner, 2014 prefers this procedure for theory-generating interviews rather than using qualitative content analysis, as proposed by Mayring, 2000, which is a popular method frequently applied when gathering facts (Bogner, 2014).

The outcomes from the interviews are then cross-validated against the propositions to confirm, deny and extend theory and to capture different perspectives. As mentioned by Flick, 2004, the observation of the subject under study from different point of views is referred to as "triangulation". To summarize, Figure 5.1 depicts the approach followed in this thesis. Finally, the cross-validated results are discussed in Chapter 9.

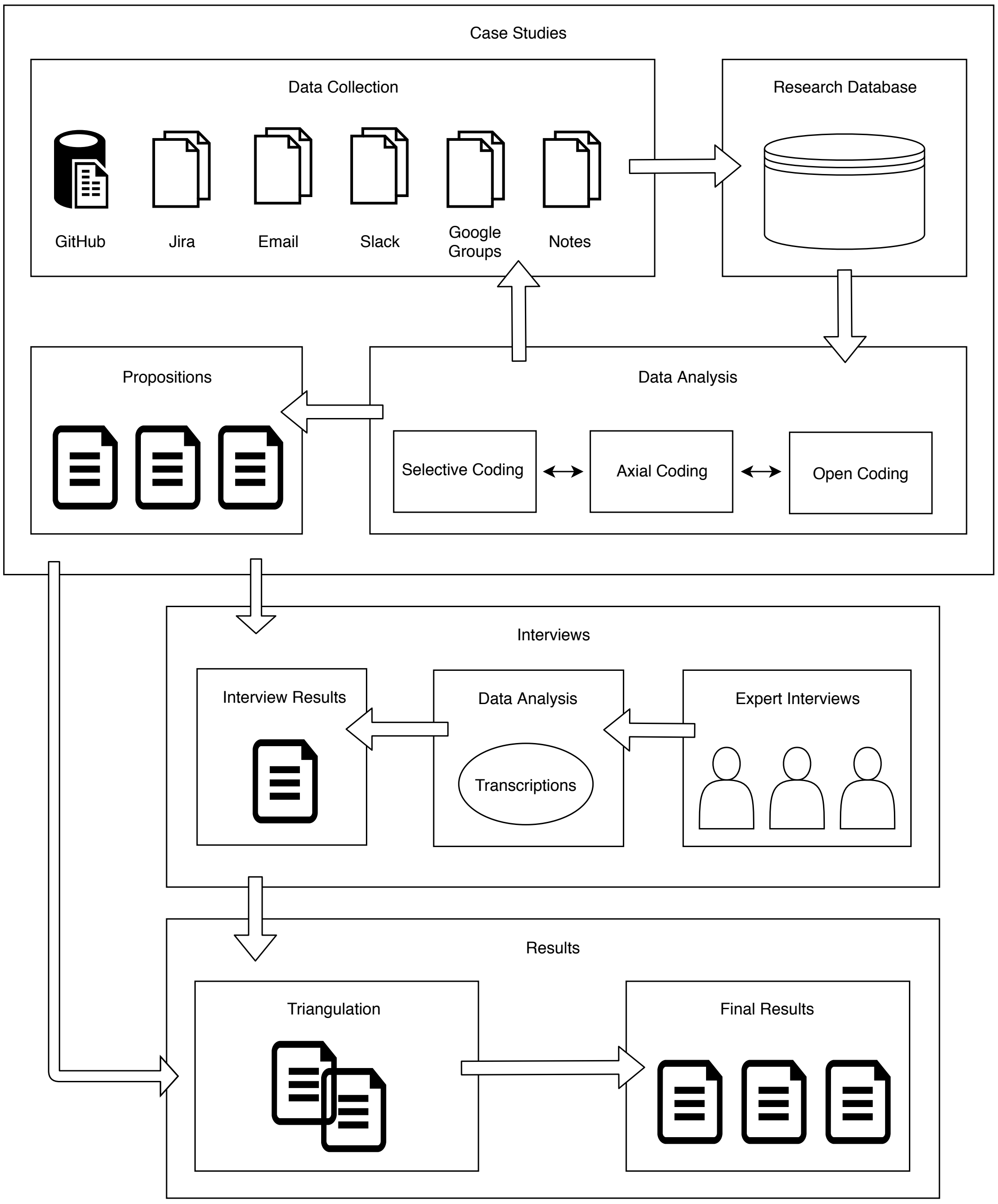


Figure 5.1.: The research design applied in this thesis

6. Case studies

The research subject of this study is limited to the Catrobat FOSS project. All data, observations and assumptions being made are exclusively based on investigations observed at the Catrobat project, or to be more precisely, the iOS[1] subproject.

6.1. Units of analysis

In order to get insights into the different phases of contribution (first interaction, onboarding, development, code review and integration), the contributions of three participants of the GSoC program is thoroughly analyzed, as summarized in Table 6.1. The GSoC program proves to be particularly helpful for this study since it provides a possibility to observe the perceptions of first-time contributors for a certain amount of time (at least three months). As mentioned in Chapter 2, the program is open to students enrolled in an accredited university program and usually starts in the middle of May and lasts until the end of August. The educational context of this program reflects a common setup in the Catrobat project since most contributors are students from the Graz University of Technology (see Section 4.3), offering a wide range of different experience levels. None of the participants did have any connections either to the Catrobat community nor to the Graz University of Technology prior to the GSoC program. A mentor was assigned to each contributor who guided them through all phases of contribution and served as a central point of contact for questions, feedback and guidance of any kind. In the remainder of this chapter, the participants' contributions as well as their interactions with their mentors and the community were

[1]https://github.com/Catrobat/Catty

	Case A	Case B	Case C
Year of contribution	2017	2017	2018
Technical experience	High	Moderate	Low
Contributions to other FOSS projects	Yes	Yes	No
Pull requests submitted	16	25	47
Pull requests merged into code base	56.25%	84.00%	78.22%
LOC merged into code base	+3,620	+7,931	+10,458
	-6,471	-974	-1,359
Interactions per week	4.3	12.2	146.4
Average length of message	453	426	82
Asynchronous communication	Yes	Yes	Yes
Synchronous communication	No	No	Yes
Voice chat	No	No	Yes
Pair programming	No	No	Yes

Table 6.1.: Summary of case studies

investigated in close details. Finally the findings of the case studies (from now on referred to as *Case A*, *Case B* and *Case C*) are discussed with related literature and propositions are formulated.

6.2. Data collection

While investigating the contributions of each case, a research database was compiled consisting of quantitative and qualitative data. Seaman, 1999 states that the combination of social and technical aspects in software engineering perfectly qualifies the mix of qualitative and quantitative data to benefit from the strengths of both. Quantitative data was collected from the source code repository on GitHub[2] and from the issue tracking and project management software Jira[3]. Qualitative data was collected from observations, notes, and communication records obtained from (1) email messages (2) Slack messages

[2]A web-based version control service using Git (https://github.com/Catrobat)

[3]A project management and issue tracking tool used by Catrobat (https://jira.catrob.at)

(3) comments on Jira tickets (4) comments on GitHub pull requests; and, (5) messages on Google Groups.

After collecting the qualitative data, each communication record between the contributor and the community has been manually labelled with the following properties:

- *Name of the sender*. The sender was either directly extracted from the header of an email message, or, for all messages other than email, the name was extracted from the username.

- *Name of the recipient*. The recipient was extracted similar to the sender. Messages not addressed directly to a single person, were labelled with the recipient *Community*.

- *Link to Jira ticket (optional)*. Each interaction was manually assigned to a task. On GitHub and Jira the name of the ticket was explicitly listed which was also valid for a large number of messages. If a message could not be explicitly assigned to a ticket, this link was omitted.

- *Link to pull request on GitHub (optional)*. The link to a pull request on GitHub was established either through the Jira ticket or extracted from the text, if directly mentioned.

- *Parent interaction (optional)*. Interactions were connected to a parent interaction to reconstruct threads, if available, to gain a better understanding of the context.

Quantitative data has been extracted from Jira (number of tickets, time to close a ticket) and GitHub (lines of code added and deleted, as well as time to close a pull request). Combining all data in a single database enables the simultaneous analysis of the contributors' communication pattern and their contribution to the source code. A detailed representation of the research database can be found in Figure 6.1.

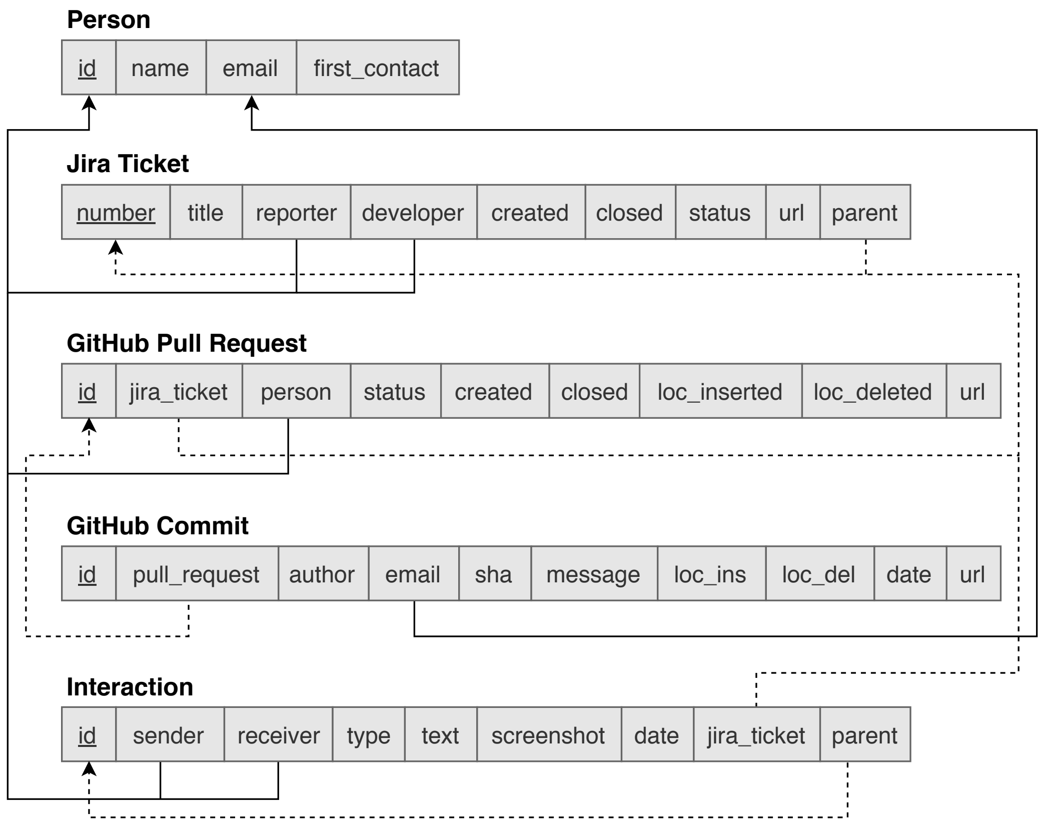


Figure 6.1.: Structure of the evidentiary case study database

6.3. Case A – technical experience meets infrequent asynchronous communication

The subject of the first case study is the contribution of a participant of the GSoC program representing the case of a technical experienced newcomer trying to find a way into the project by communicating on an infrequent and solely asynchronous basis. At the time of the study, the contributor already had a founded technical knowledge about software engineering in general as well as a pertinent experience with the technology utilized in the project. The concept of block-based programming including Catrobat's visual programming language was not familiar to the student.

An interaction between the contributor and the mentor, who were separated by one timezone, is defined as an exchange of email messages, comments on Jira tickets as well as comments on GitHub pull requests. During this program 15 Jira tickets were in development, allowing a total of 8 tickets (53.3%) to be merged into the project's code repository.

6.3.1. First contact

The first contact with Catrobat's community took place 52 days prior to Google's announcement of the accepted students' names. The student commented on a Jira ticket asking to get permission to start working on a ticket:

> *"I would like to work on this issue. Could you please give me permissions to move issues on the board."*[4]

After receiving the proper permissions, two patches fixing two small bugs were submitted. The domain expertise required to fix these two tickets was kept to a minimum, making it possible to start contribution in a timely manner and without further guidance. Derived from this, it can be said that the student was able to autonomously find a task to start with and autonomously setup the required local working environment.

6.3.2. Communication

The communication between the mentor and the mentee took place on an infrequent, asynchronous basis averaging to 4.3 interactions per week during the community bonding and coding period. The average number of characters per message amounts to 435. During that period 14% of all interactions occurred in the week before the first evaluation and 44% took place one week before the final submission. As illustrated in Figure 6.2, around two third of all conversations were being held on Jira using ticket comments, slightly more than one quarter using email messages

[4]https://jira.catrob.at/browse/IOS-431, visited on 16 May 2019

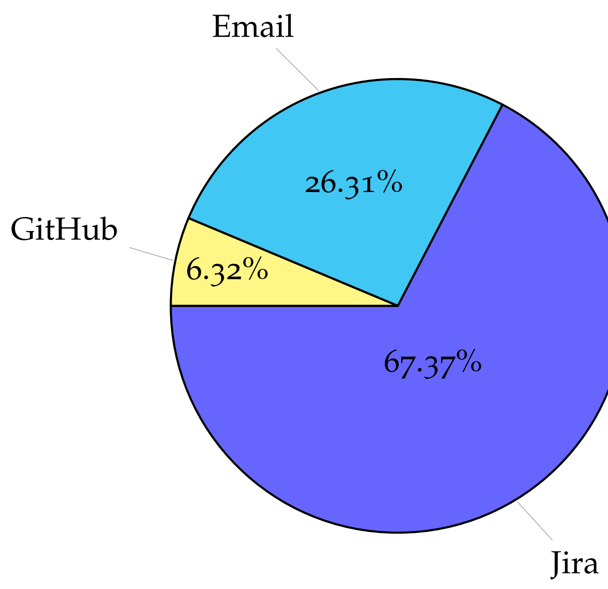


Figure 6.2.: Communication channels used in Case A (rounded to two decimals)

and 6% were exchanged using GitHub's commenting functionality for pull requests.

Public communication channels (Jira and GitHub) were mainly employed to communicate change requests for submitted pull requests, to refine requirements and to explain missing domain knowledge. In contrast to that, private email messaging served to clarify organizational aspects, to give guidance about further contributions and to address problems related to communication and the schedule of the GSoC program.

Interactions were solely observed between the student and the mentor – any kind of public communication between the student and other community members has not been recorded. Due to the distance between the mentee and the mentor there was no face-to-face meeting. A project-wide Internet Relay Chat (IRC) channel was offered open to everyone interested. Since the technology seemed old-fashioned to the contributors (Fellhofer, Harzl, and Slany, 2015) and was rarely used at the time of the case study, it was not taken advantage of.

6.3.3. Contribution

The primary contribution consisted of the implementation of a new feature which was a fairly complex task with a lot of interdependencies throughout the whole app. Beyond that, propositions for refactoring tasks were made to increase the internal code quality and design.

A transition diagram shown in Figure 6.3 illustrates the different types of events during development. The diamonds represent the begin and the end of the development of an issue described in a Jira ticket. The circular and square-shaped nodes represent an interaction between the contributor and the community while the edges are labelled with the transition probability. *Submission* indicates the opening of a new pull request on GitHub, whereas *Accepted* and *Declined* signal the closing of the stated. The square labelled *Change Request* reflect the appeal to make changes to the submission, for example because of unexpected behavior, missing functionality, bugs or missing project guideline compliances. The node *Interaction* represents any task-related communication between the contributor and other members of the community other than a change request. For instance, this type of interaction serves to clarify any uncertainties, to give feedback to a pull request, or to get synchronized on the progress of work initiated by either the contributor or the community. An example wording for change requests is illustrated in Appendix B.

The high probability (0.77) between *Start* and *Submission* indicates that there is a trend to start development and open pull requests before interacting with the community to, for instance, coordinate action plans or ask for guidance. A pull request for slightly more than three-quarter of all tickets was submitted without asking any questions about technical and/or functional requirements. Half of the submitted patches were instantly accepted without any change requests. While this half mainly consisted of bug tickets, it was necessary to pass on a large number of change requests when it comes to the implementation of new features requiring a comprehensive domain knowledge. A closer look at the interactions between the contributor and the community, either before or after the hand-in of a pull request, revealed the predominance of the following categories discussed in the remainder of this section.

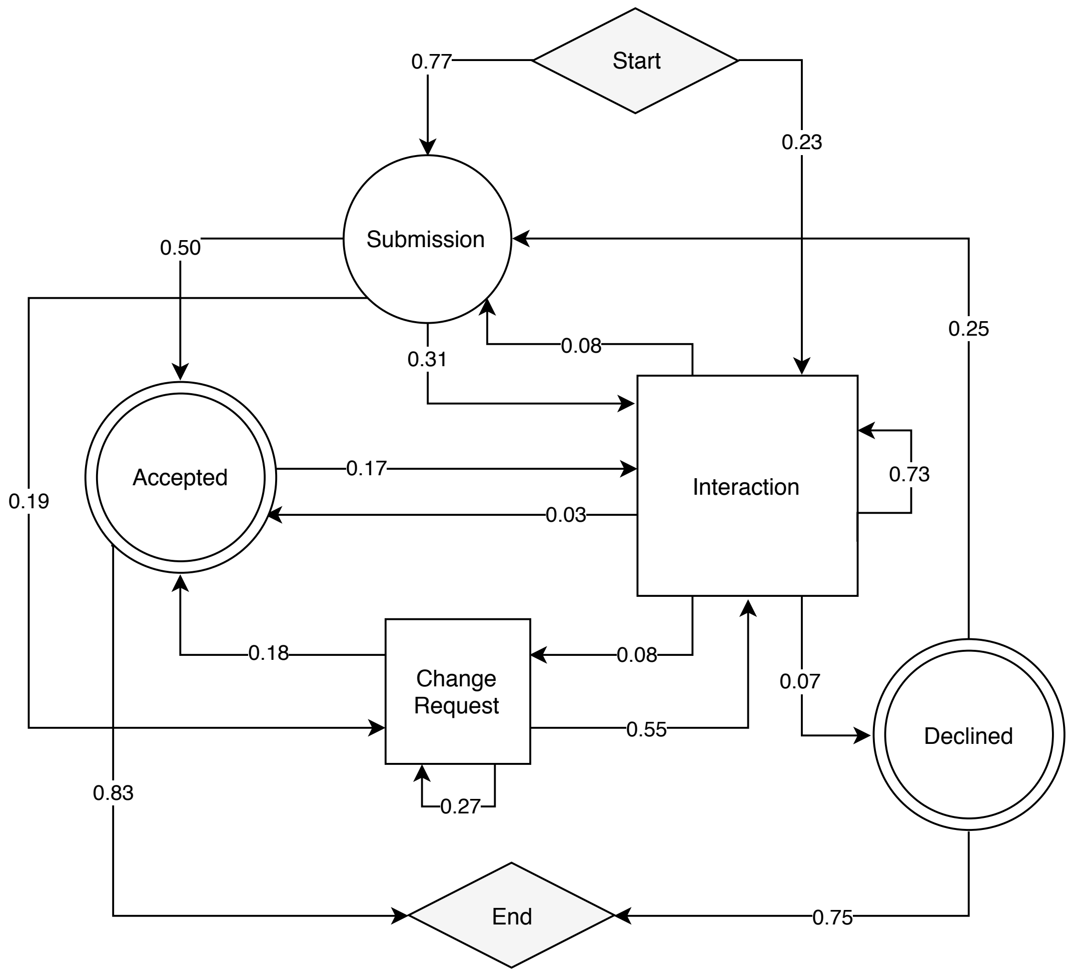


Figure 6.3.: Communication pattern during development in Case A

A. Requirements and domain knowledge

Slightly more than one quarter of all communications (27,15%) was about unclear requirements, requirement refinements and change requests related to misunderstood requirements since it was not clear *"how exactly [this feature] should behave"*. In contrast to a pertinent technical experience, there was a lack of domain knowledge leading to misunderstandings in how the visual programming language is expected to behave and how it is being used. Although the anticipated functionality can be perceived in the Android version of Pocket Code, the need for documentation has been addressed asking whether *"any documentation about this except [Jira Issue]"* exists. As a result of lacking domain knowledge and lacking documentation there has been a large number of interactions leading to potential idle times due to its asynchronous form: While the contributor was dependent on further guidance from the community, the development could not be continued prior to receiving an answer.

B. Change requests

The combination of a misunderstood requirements and the tendency to make a pull request before communicating any action plans (see Section 6.3.3) turned out to be an issue. Since the submission did not always have the expected behavior and most of the time did not comply to the project's guidelines, a lot of change requests needed to be communicated, comprising more than 15% all of interactions. This was done with the help of plain text, screenshots, pseudo code and stack traces. Figure 6.4 shows an example where pseudo code was exchanged in order to provide the relevant context for a change request. While this worked fine for simple bug tickets, this was a cumbersome process involving a large amount of communication when trying to clarify complex, domain related issues.

Change requests were mainly necessary due to unexpected behavior, app crashes and code refactoring. Additionally, many changes needed to be made because of compliance problems with the project's development guidelines, for instance because of missing tests. When repeatedly asking

```
Scene1:
- increment global var "global" by 5
- increment object var "local" by 5
- show object var "local" -> 5
- show global var "global" -> 5
- start "Scene2"
```

Figure 6.4.: An example of pseudo code used in a change request message

for multiple change requests negative attitude was observed resulting in short answers and unresponsiveness.

C. Technical discussion

Discussions about code related issues, like the advantages and disadvantages of different implementation strategies and design decisions were observed. Unexpected technical hurdles arose, requiring more effort than expected to be put into the completion of tasks, for instance because *"it was really difficult [...] to work on this"*, thus causing *"many new problems [...] during development"*.

According to the contributor, changes to the code base were inducing too many unexpected side effects, which resulted in unforeseen problems. Autonomous propositions for code refactoring were made by the contributor suggesting to *"write cleaner/simpler/safer code"* in order to increase the attractiveness for other open source developers since *"all these minuses push open-source developers away"*.

D. Lack of awareness

While a lot of refactoring and restructuring was done during this case study, concurrent actions by other members of the community and upcoming plans were not communicated well enough. Subsequently, a lot of effort had to be put into the resolving of conflicts since there was no communication between the participant and other developers. To give an example, one code reviewer from the community commented on a JIRA ticket:

> *"This will cause terrible headache when merging the master. Reason for that is because in the meantime lists have been introduced (like variables which can hold several values like a list). What is the reason for removing this class?"*[5]

E. Feedback, status update and toolchain

The remaining interactions (around 30%) mainly addressed feedback on pull requests, status updates and organizational topics related to the GSoC program. Occasionally there was a short discussion about issues related to git[6] or other matters related to the toolchain (for example Jira). While technical and organizational guidance was offered mainly at the beginning of the contribution, the contributor did not ask for any technical help during the development.

6.3.4. Summary

The participant in this case study already had a pertinent technical experience and preferred to contribute autonomously keeping the communication with the community to a minimum. For tasks requiring a low level of domain knowledge, like bugs, it was possible to individually make a contribution without any guidance. There has been the desire to refactor code in order to try to make the code base more attractive for further contributors. Guidance was needed for the implementation of new features, mainly due to vague requirements, a lacking domain knowledge and unclear specifications. Consequently, the need for documentation has been addressed multiple times. To continue, the trend to make pull requests prior to interacting with the community (for example to coordinate action plans), as illustrated in Figure 6.3, and the infrequent and asynchronous nature of interactions between the mentor and the mentee resulted in multiple change requests

[5]https://jira.catrob.at/browse/CATTY-5, visited on 16 June 2019
[6]A free and open source version control system used by GitHub

and a large amount of conflicts. As a result, there was a lag behind the schedule of Google's program and not all submissions became part of the code repository. Although it was promised to continue contributing after the end of the program, the last interaction so far took place 22 days after the final submission[7].

6.4. Case B – moderate technical experience meets frequent asynchronous communication

This case was documented to illustrate the collaboration of a contributor with a moderate technical experience interacting with the community on a frequent but asynchronous basis. While the student was already familiar with FOSS due to multiple contributions to other projects on GitHub, the concept of visual programming languages and the usage of Pocket Code were not familiar.

Interactions between the participant of the GSoC program and other members of the Catrobat community, who had a time difference of 3.5 hours, took place using entirely asynchronous forms of communication. Exchanged messages were recorded on GitHub comments, Jira tickets, Google Groups[8] as well as using private email messages.

6.4.1. First contact

It was asked for permissions to start contribution on a ticket 65 days prior to Google's official announcement of the accepted students:

> *"Can someone please assign this to me? Also, all I have to do is push the changes to my fork and let it be known here right?"*[9]

[7]As reviewed by 10 May 2019
[8]A service hosted by Google providing discussion groups similar to mailing lists
[9]https://jira.catrob.at/browse/IOS-451, visited on 16 May 2019

The contributor was able to autonomously setup the local working environment and, after becoming the assignee of the ticket, start working on a task. Although the ticket did not require any domain knowledge and was mainly concentrating on code related issues, an autonomous completion of this ticket was not possible. While the contributor was waiting for guidance from the community, it was asked to simultaneously start working on other tickets. This second approach to get in touch with Catrobat's community did not succeed since the contributor's requests remained unanswered. Possibly driven by the incentives to participate in Google's program at some point in the future, the student persistently continued to contribute to other tickets, even without receiving an answer from the community.

6.4.2. Communication

Finding it difficult to find out how to get in touch, the demand for various synchronous communication tools was addressed:

> *"Since the IRC channel is inactive and there is no other service like gitter or slack being used, any and all communication would be either over email, or directly on JIRA itself, right?"*

At the time of the case study, Catrobat did not offer services like Slack or Gitter to the public (see Chapter 4). IRC was set up, but as mentioned in Section 6.3.2 it was rarely used, making it an unattractive alternative.

During the community bonding and coding period an average of 12.2 interactions between the contributor and the community were recorded each week whereas the average message length amounts to 414 characters. As illustrated in Figure 6.5, slightly more than 6% were exchanged using GitHub's commenting functionality, more than three-quarter of the conversation took place on Jira whereas only 15% were transferred via email messages. Google Groups has been utilized with a frequency rate of less than one percent solely for affairs related to the GSoC program.

The majority of change requests and domain related issues were communicated on public channels (Jira and GitHub) whereas email was primarily used for organizational matters. Although there was a constant interaction

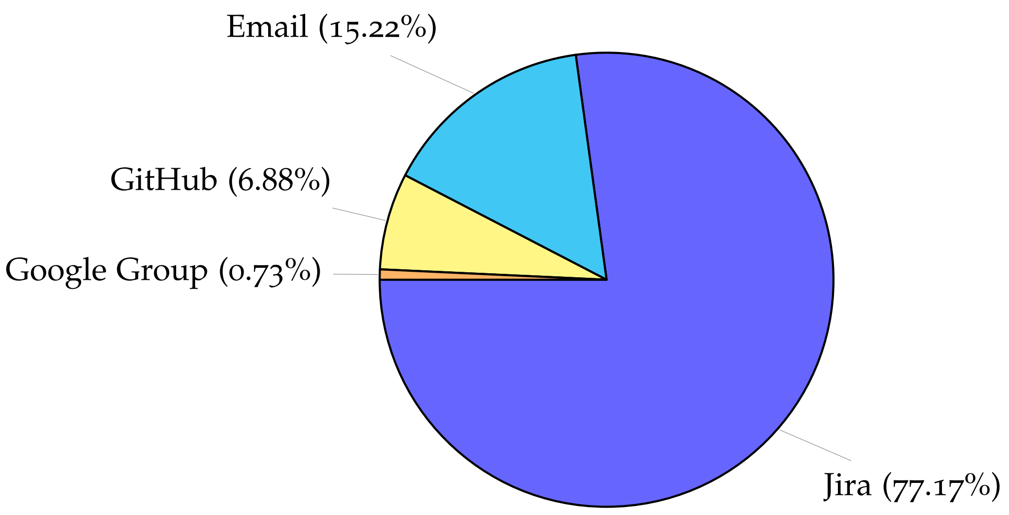


Figure 6.5.: Communication channels used in Case B (rounded to two decimals)

between the contributor and the community, around 20% percent of all communications took place one week before the final submission.

6.4.3. Contribution

As part of the GSoC program, 25 pull requests were submitted allowing a total of 21 (84%) to become part of the main code repository. During this case primarily new Bricks were implemented requiring a large extent of domain knowledge to understand how each Brick should behave and/or interact with other Bricks of the programming language.

Figure 6.6 depicts the different types of events which appeared during the contribution. While depending on support from the community, for more than 80% of all submissions there was an interaction prior to the delivery of a pull request. The low transition probability (0.04) between *Submission* and *Accepted* indicate that a lot of communication was necessary before a submission became part of the code repository. On the other hand, the small transition probability (0.01) to *Declined* demonstrates that there is a high chance that pull requests were accepted after interacting with the community. The resulting interactions have been recorded and investigated in the subsequent paragraphs of this section.

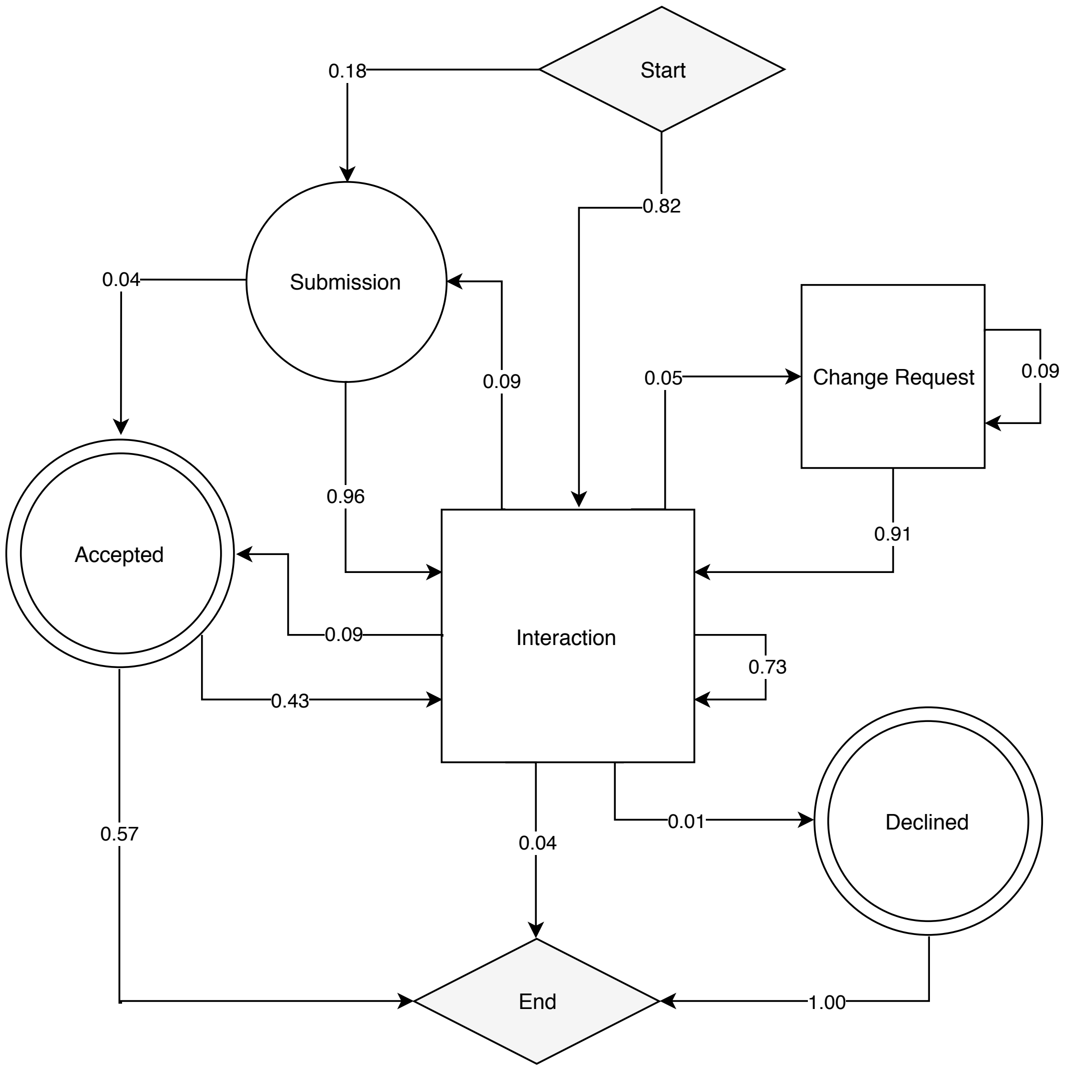


Figure 6.6.: Communication pattern during development in Case B

A. Technical discussion

One major category comprising around 20% of all conversations was dealing with discussions about different implementation strategies, asking the community for suggestions and recommendations on which one to follow. Unexpected hurdles arose during development, possibly due to the code base being perceived as not intuitive and too complicated:

> *"Phew, this one took a LONG time. [...] There were way too many complicated constraints, and the scroll view's scrollable content height was not being inferred properly."*[10]

B. Guidance for further contribution

During this case a new theme emerged, frequently asking for hints and guidance on how to start or continue with the development of a task. While this pre-action communication was causing less domain-related problems during code reviews, it was resulting in a potential cause for delay since it was not practicable to continue development without further community-driven support. The purpose for these interactions were mainly arising from issues related to the project's strict guidelines about testing:

> *"But can't really figure out how to get it working. Is it feasible to test this way? How else should I go about testing this?"*[11]

> *"I do understand what this is about, but can't seem to be able to figure out what exactly is the right way to go about fixing this. Could you please let me know what I should do here?"*[12]

Around 17% of all communication subjects can be categorized into this or a similar type of interaction where guidance was necessary. While waiting for further instructions and hints from the community, the desire to simultaneously *"assign more tickets [...] to work on"* was expressed by the contributor multiple times. Delayed answers were perceived as hindrance

[10]https://jira.catrob.at/browse/IOS-509, visited on 16 June 2019
[11]https://jira.catrob.at/browse/IOS-496, visited on 16 June 2019
[12]https://jira.catrob.at/browse/IOS-200, visited on 16 June 2019

resulting in the contributor to repeatedly ask the same question on different communication channels.

C. Requirements and domain knowledge

Explained by the tendency to discuss action plans prior to performing actions (see Figure 6.4.3), a lot of requirement- and domain-related interactions were recorded before the actual development was initiated, for example because the contributor was *"unable to understand what exactly this brick does"*.

The development was not commenced until most uncertainties were eliminated resulting in a large amount of interactions and standby times due to its asynchronous nature. Having to accomplish reverse engineering[13] tasks, the already existing implementation of the Android version of Pocket Code served as a model to better understand expected behavior:

> *"Could you suggest a script / series of bricks I could try on the android app that could better help me understand the functioning of this brick?"*[14]

D. Feedback and status update

The largest category (22.5 % of all topics) covers regular status updates mainly communicated proactively by the contributor to both update the community on the current state of the development and to ask for feedback about recent implementations. No coordination problems were perceived which could possibly be a positive side effect of frequent communication and multiple feedback loops.

[13]An existing functionality is deconstructed to reveal details about its implementation and behavior

[14]https://jira.catrob.at/browse/IOS-502, visited on 16 June 2019

E. Change requests and toolchain

The remaining interactions were dealing with questions and problems related to the code repository and the issue tracking system (Jira). As a result of immediate feedback and the coordination of action plans prior to starting development, only less 10% of all affairs cover requests to make changes to pull requests.

6.4.4. Summary

The contributor of this case study preferred to interact with the community on a frequent basis: action plans, unclear requirements and design decisions were discussed prior to the start of development. Since the code base was not always perceived as self-explaining, multiple unexpected technical hurdles were observed. As a result of both technical and domain related issues, there was a frequent call for guidance. Issues related to a lacking domain knowledge were solved with the help of a lot of communication. To be more independent from the community, the ambition was expressed to simultaneously work on multiple tasks.

As a result of frequent status updates, immediate feedback and the coordination of action plans, only less than 10% of all affairs cover the conversation of change requests. Communication issues and coordination problems were not observed. After Google's program has finished, the participant was planning to continue contributing whereas the last interaction so far was recorded 23 days after the final submission[15].

6.5. Case C – low technical experience meets intense synchronous communication

The subject of this case study is another contribution of a participant of the GSoC program, which was documented one year after the first two case

[15]As reviewed by 10 May 2019

studies. In contrast to the majority of first-time contributors, the student was already experienced with the concept of visual programming languages. While the student already had a profound experience using Scratch, the use of the Catrobat programming language was not familiar. Having only little experience in programming itself, Catrobat was the first FOSS project a contribution was made to.

The interaction between the contributor and other members of the Catrobat community, who were separated by only one timezone, took place on an intense basis using both asynchronous and synchronous forms of text-based communication. Additionally, voice chat and desktop sharing was employed to enable the use of remote pair programming. At the time of this case, a project-wide Slack workspace[16] has been introduced which was already in use by a majority of Catrobat's contributors. Data reflecting the interactions between the contributor and the community was collected using Jira, GitHub, Slack, private emails and pair programming protocols.

6.5.1. First contact

After submitting a proposal for the GSoC stipend, the mentor got in touch with the potential candidate 15 days before the announcement of the accepted students. Different to last year's candidates, the student did not make a contribution prior to the start of the program. A phone interview revealed that the contributor was not able to autonomously setup the required local working environment, thus not being able to start contributing. After investigating this issue, it turned out that there was a wrong directory listed in the according README[17] section which was causing the setup to fail.

[16]A shared place where community members can communicate and collaborate on various channels

[17]A text file in the repository to tell other people about the project and how to use it

6.5.2. Communication

During the community bonding period and the period until the first evaluation, email was used to exchange information whereas desktop sharing and voice calls were regularly employed to give guidance to code related issues. During that period the need for a synchronous form of text-based communication was frequently addressed by the contributor:

> *"First of all, regarding Slack, if it is not available, I am fine with using Facebook Messenger or Whats App. I know it is less professional, but any chat is slightly better than these long and formal emails."*

Although a project-wide Slack workspace was already available at that time, it could only be instantly joined by contributors having an email address ending with *tugraz.at*. Thus, external contributors need a separate invitation in order to join the project's workspace. After the participant has received such an invitation, communication increased drastically. Whereas there was a constant interaction between the mentor and the mentee, around 17% of all interactions took place one week before the second evaluation and more than 24% of all messages were exchanged after the final submission. During the community bonding and coding period an average of 146.38 interactions between the contributor and the community were observed each week. Due to the predominant form of synchronous communications, the average message length is lower than in the other cases, amounting to 82 characters. Additionally, around 28 hours were spent on voice chat, remote desktop and pair programming sessions. According to collaboration notes, the mentor and the mentee agreed on having two remote session per week during the coding period.

As Figure 6.7 illustrates, messages were mainly exchanged using Slack (95.95%). The category labelled *Other* consisted of interactions using Google Hangouts[18], Skype[19], Stride[20], TeamViewer[21] and Zoom[22].

[18] A communication platform enabling text-based messaging, video and voice chat
[19] An application providing text, video and voice chat
[20] A collaboration tool for text, video and voice chatting and desktop sharing
[21] A software enabling desktop sharing and remote control
[22] A remote collaboration platform providing voice and video chat and desktop sharing

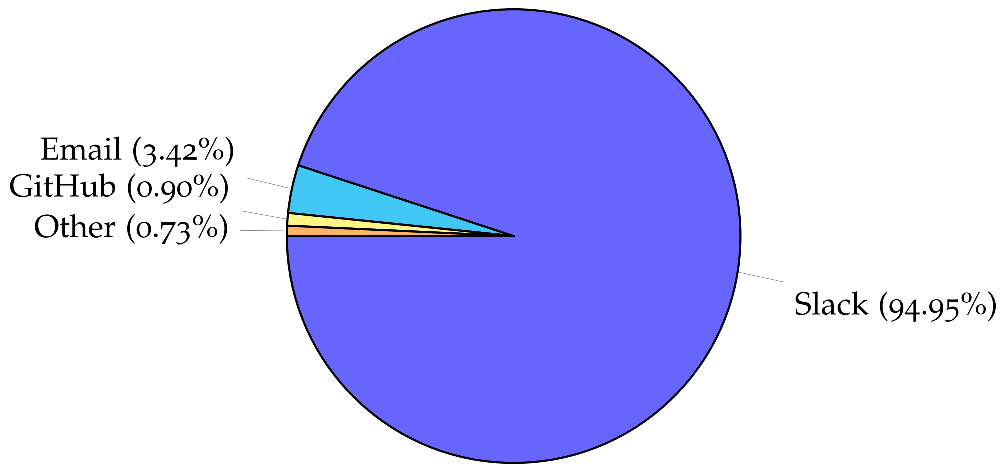


Figure 6.7.: Communication channels used in Case C (rounded to two decimals)

6.5.3. Contribution

The main contribution consisted of the implementation of multiple tasks requiring a large amount of detailed, domain-related expertise. The contributor was working on 37 tickets, allowing a total of 35 tickets (94.59%) to be merged into the code repository.

Figure 6.8 demonstrates that there was a constant interaction with the community prior to the start of the development. This can be derived from the transition probability (1.0) of the edge connecting the nodes *Start* and *Interaction*, outlining the strong dependency on community-driven guidance. In the remainder of this section the communication between the contributor and the community was investigated in depth.

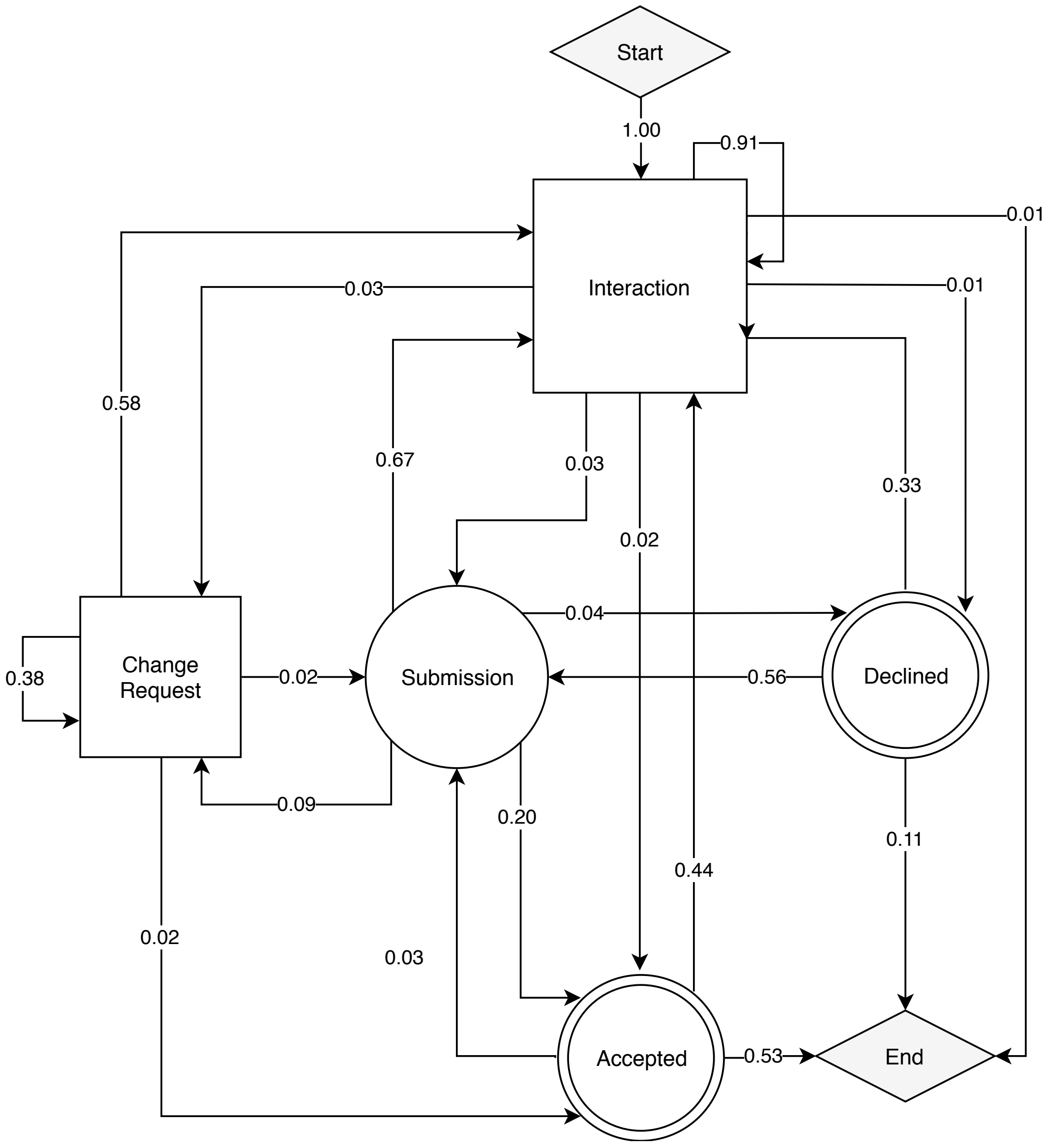


Figure 6.8.: Communication pattern during development in Case C

A. Technical discussion and support

Slightly less than one third of all types of interactions (30.6%) were of technical nature, mainly initiated by the contributor. Questions related to Xcode[23], TDD and the code base came up during development. Since the student did not have any experience with code testing in general, several technical questions arose:

> *"I can't test anything because I don't know how to do the setup (the code snippet I sent you). And what is it that I have to do? Do I have to test if the x is exactly as specified in the CG point?"*

While trying to support the contributor and give understanding to code related issues, remote desktop sharing and voice chatting sessions were initiated between the mentor and the mentee. Clarifying technical issues related to the code base proved to work better when performing pair programming sessions rather than text-based chatting. During remote desktop sharing sessions the contributor's screen was visible to the mentor enabling both sides to see and control the integrated development environment (IDE) running on the contributor's computer. For the role of the navigator (see Chapter 3) a tool for indicating which position on the screen the conversation is about has proven to be very helpful. Without such a functionality, it was more time consuming to make it clear which position at the code to look at using solely verbal communication (for example *"Please look at line number 123 in the file called Implementation.swift"*).

Table 6.2 summarizes the different software tools employed during the collaboration. Zoom and Slack turned out to be the student's favorite applications, among other things because on both applications an annotation tool was available enabling the navigator to draw and write on the driver's screen. Given that Zoom offers more features for its annotation tool, the contributor was suggesting to continue working with this application: *"why not zoom? [...] please, let's use zoom! [...] LETS MOVE TO ZOOM"*.

Furthermore, design decisions and implementation details were discussed before and during the development making the contributor more confident to start and continue the implementation of a certain task:

[23] An integrated development environment (IDE) provided by Apple

	Zoom	TeamViewer	Stride	Slack
Plan	Free	Free	Standard	Standard
Remote Control	Yes	Yes	Yes	Yes[a]
Annotation Tool	Yes	No	No	Yes
Shared clipboard	No	Yes	No	No
Switch between screens	Yes	Yes	No	Yes

[a] This feature has been removed in July 2019 (Slack, 2019)

Table 6.2.: Summary of collaboration tools employed in Case C

> *"I will continue once I have your feedback, I do not want to keep writing wrong things until the end and then having to modify. :D"*

B. Guidance for further contribution

In around 15% of all conversation topics, it was asked for hints on how to continue or start with the development of a certain task. In those cases, the contributor otherwise was not able to autonomously continue with the implementation. Simple tickets were recommended on how to get familiar with the code base. Guidance was necessary especially in the following areas:

- IDE and toolchain
- Compilation errors
- Find a task to start with
- Hints on where to start debugging
- Hints on how to test

The lacking experience in these areas were counteracted with a lot of communication, mainly using pair programming sessions which were scheduled regularly and initiated on demand:

> *"I did commit. But I think I might have cloned the branch in a wrong way... we need a Zoom Call."*

C. Toolchain

Problems related to git were observed on a regular basis. A total of 307 interactions were related to issues with the versioning system mainly resulting from merge conflicts with the base branch or an improper usage. While most of the issues were resolved within a few minutes during pair programming sessions, these affairs were causing demotivation for the contributor.

D. Community bonding

A new theme emerged for the first time covering conversations about the Catrobat community in general not related to the GSoC program. Questions about other members, their roles and responsibilities were observed trying to get a broader understanding of what is happening in the project. The contributor started to become a part of the community and tried to get in touch with other members from other teams. An offer was provided to promote the Catrobat project on social media platforms in order to help to reach a wider audience. Additionally a blog post about the contributor's personal experience at Catrobat was published on a social media platform, followed by a presentation at the contributor's home university to encourage other students to become part of open source communities. Other developers were contacted to get implementation details of certain features and it was helped to make contact between new contributors and the community almost one year after the program has finished.

E. Requirements and change requests

An in-depth understanding of domain-related details was acquired through reverse engineering as well as interactions with other community members. Change requests were passed on mainly caused by inaccurate and unexpected behavior (slightly more than 3% of all interaction topics).

6.5.4. Summary

Voice chat and desktop sharing were employed to enable remote pair programming to (a) counteract issues related to programming and the toolchain, (b) guide the contributor through the code base; and, (c) give an introduction to the concept of testing and TDD. While the student already had a sound experience with visual programming languages, there was only little experience with FOSS projects and programming in general, making pair programming sessions inevitable. In the period between pair programming sessions, the use of long and formal emails was not well accepted by the contributor stressing the need for a synchronous form of communication, like for example Slack. After switching to Slack, the communication frequency increased drastically (having up to 548 interactions per week).

Development consistently started with an interaction clarifying requirements, code related issues or the coordination of action plans (see Section 6.5.3). Contribution was not possible without further guidance, underscoring the necessity for both mentorship and frequent communication. Response time was significant in order to enable a continuous development. To counteract issues related to high response times, multiple tasks were chosen to simultaneously work on.

During this case a new theme emerged, comprising community-related conversations not associated with Google's program. The contributor wanted to get in touch with other community members and tried to encourage other students to contribute to FOSS projects.

After the final submission, the student was planning to continue to contribute to the Catrobat project whereas the last interaction with the community was recorded 253 day after the end of the program (compared to 22 days in *Case A* and 23 days in *Case B*)[24].

[24]As reviewed by 10 May 2019

7. Case study outcomes and propositions

The contribution of three volunteers with different levels of technical and domain-related experience was examined in close details. The communication between the contributor and the community took place on diverging frequencies (ranging from low to intense) whereas the form of communication ranged from formal emails and comments on GitHub to voice chatting and pair programming. In this section the findings of all three case studies are compared with each other drawing cross-case conclusions. Similar results (literal replication) and contrasting results (theoretical replication) are employed to develop a cross-case report (Yin, 2017) which is used to generate theory for the expert interviews (see Chapter 8). In the remainder of this chapter the outcomes of the case studies are listed in the form of propositions.

7.1. Find a task to start with

Proposition 1. *In the case of Catrobat, first-time contributors tend to start with technical bug fixes keeping the required domain-specific knowledge to a minimum.*

Contributors with a pertinent technical experience did not have any problems setting up the required environment and were able to autonomously start development and open their first pull requests (*Case A* and *Case B*). With the help of the issue tracking system Jira it was possible to autonomously find a task to start with. Rather than starting with domain-specific affairs related to the Catrobat language, small bug fixes were autonomously chosen in all three cases, keeping the required domain knowledge to a minimum in order to foster a swift start.

7.2. Guidance from the community

Proposition 2. *In the case of Catrobat, newcomers need guidance from the community in order to make a contribution to domain-related tickets (e.g. the implementation of new features).*

While an autonomous contribution worked well for technical experienced newcomers implementing refactoring tasks and bug fixes, it was causing severe problems when autonomously developing new features requiring a large extent of domain knowledge (*Case A* and *Case B*). As a result of misinterpreted requirements, lacking documentation and infrequent communication, the submitted changes did not behave as expected involving a large number of change requests subsequent to the submission. To get a better understanding of the requirements and the project's guidelines, the need for documentation has been addressed, asking whether *"any documentation about this except [Jira Issue]"* exists (*Case A*).

The contributor of *Case B* was taking up contact with the community in more than 80% of all submissions whereas the contributor of *Case C* was always (100%) proactively interacting with the community before starting development. Along with domain-related affairs, in *Case B* and *Case C* there was a constant discussion about software testing and TDD, two important internal project requirements (see Chapter 4). Due to the diverse levels of experience and seniority, either a lack of domain knowledge, a lack of technical experience or a combination of both has been observed. Moreover, newcomers need to collect knowledge about organizational rules (coding guidelines, quality standards, organizational workflows, et cetera), making a large number of interactions with the community unavoidable and underpinning the necessity to help newcomers to find their way into the project.

7.3. Communication

Proposition 3. *In the case of Catrobat, frequent and direct communication has positive effects to a newcomer's contribution. Low frequency and high response times have a negative impact.*

To overcome the aforementioned difficulties a lot of communication was necessary. To discuss technical details and refine unclear requirements a wide range of questions were asked using a large number of asynchronous emails (*Case A*, *Case B* and *Case C*). While this worked well for simple tasks and bug fixes, in complex situations, a more direct contact, like the use of synchronous voice chatting in combination with desktop sharing was inevitable, especially when trying to clarify complicated technical issues (*Case C*). Deferred communication and delayed answers were seized as a problem by both the mentor (*Case A*) and the contributors (*Case B*, *Case C*) who were not able to continue development without receiving an answer from the community.

Infrequent communication turned out to be an issue in *Case A* where misinterpreted requirements were discovered too late causing many conflicts and a long delay in development. In contrast, frequent communication and a constant exchange of pre-action plans facilitated the reduction of required change requests (compare 15.83% in *Case A* and 9.38% in *Case B* with 3.16% in *Case C*). The necessity for a synchronous chat-based form of communication was addressed in *Case C* stating that *"any chat is slightly better than these long and formal emails"*. After switching to Slack as the main communication channel, the interaction frequency increased tremendously averaging to 146.4 interactions per week compared to 4.3 weekly interactions in *Case A* and 12.2 interactions per week in *Case B*. On the other hand, the average length of a message decreased to 82 characters compared to 453 in *Case A* and 426 in *Case B*. As a result, the average response time decreased from 11.89 hours when using email to 0.29 hours when using Slack (*Case C*).

7.4. Community bonding

Proposition 4. *In the case of Catrobat, frequent and direct communication fosters community bonding.*

During the third case study a new category emerged for the first time representing community-related discussions not affiliated with the GSoC program. While this category did not show up in *Case A* and *Case B*, this could be a positive side effect of a frequent, synchronous and direct form of communication. Contact with other members was established and it was tried to engage other developers to start contributing to FOSS projects. While the contributor and the mentor in *Case C* are still in contact one year after the GSoC program, the last interactions with the community and the contributor of *Case A* (*Case B*) were recorded 22 (23) days after the end of the program. It has been observed that direct and frequent communication potentially has a positive effect on creating strong ties with the community.

7.5. Technical hurdles

Proposition 5. *In the case of Catrobat, unexpected technical hurdles and the project's strict guidelines foster demotivation and raise the entrance barrier for newcomers.*

Issues related to the code base were remarked in the first and second case study arising from (a) bad code quality, (b) complex and outdated code; and, (c) unexpected hurdles resulting from high interdependencies and low modularity. In all three case studies, more time than expected was needed for development, among other things because the code base was perceived as not intuitive and too complicated. Additionally, the software architecture made it difficult to comply to the project's strict guidelines about testing, especially when not having much experience in the area of software testing at all. Demotivation was observed as a result of passing on a large number of change requests, offering a possible reason to discontinue contribution. As observed in *Case A*, the desire to write *"write cleaner/simpler/safer code"* has been noted in order to increase the attractiveness for future contributors since *"all these minuses push open-source developers away"*. Due to the limitation of this thesis, technical aspects of software quality are not being discussed. However, this has already been investigated by Schranz et al., 2019.

7.6. Agile software development

Proposition 6. *In the case of Catrobat, newcomers want to work on multiple tickets simultaneously while waiting for guidance from the community.*

In *Case A*, *Case B* and *Case C* the need to simultaneously work on multiple tickets was repeatedly stressed while waiting for guidance, feedback or code reviews in order to overcome potential idle times resulting from asynchronous communications:

> *"Meanwhile, please assign more tickets for me to work on?"* (*Case B*) [1]

As previously mentioned in Section 7.2, contribution was strongly dependent on community-driven guidance, representing a possible restraining factor, especially when response times are high. In those cases, a continuous development was not feasible which led to multiple waiting times. The contributors in all three cases overcame this challenge by simultaneously working on multiple tickets while waiting for an answer from the community.

[1] https://jira.catrob.at/browse/IOS-493, visited on 13 May 2019

8. Interviews

As a secondary research method, semi-structured interviews were conducted with three long-term experts in the area of FOSS, GSoC and the Catrobat project in general. Rather than gathering facts, expert interviews are a useful method for reconstructing subjective interpretations for specific research topics (Bogner, 2014). All experts have been in the Catrobat project for multiple years and have introduced and guided both many co-located newcomers as well as several external contributors within the scope of GSoC and/or similar programs. With their profound experience in collaborating with other community members they provide a valuable input for this thesis. As a consequence, new perspectives are combined with prevalent outcomes from the case studies, building up a "chain of evidence", as suggested by Yin, 1984. An interview guideline (Bogner, 2014) was employed to ensure that the most significant parts of the propositions under study are addressed. On the other hand, the guideline, which is enclosed in Appendix F, was designed to enable an interactive conversation between the interviewee and the interviewer. The mixture of open- and closed-ended questions enable the collection of new and unexpected types of data along with expected information (Seaman, 1999; Turner III, 2010). The interview transcriptions are analyzed using a coding procedure as illustrated in Chapter 5. Finally, the outcomes of this analysis are presented in the remainder of this section.

8.1. Interview outcomes

The experts (in the remainder of this section referred to as E1, E2 and E3) have been in the Catrobat project for at least four years and have a lot of experience in the areas of onboarding, code reviews, project management and collaborating with other contributors. To get an insight info different perspectives, the interview partners were thoroughly selected from different areas within the project to have both overlapping and divergent fields of responsibility. Therefore it is crucial to emphasize that the experts might have different mindsets and diverging expectations regarding the project. This needs to be considered when trying to draw conclusions, as detailed in Chapter 10.

8.1.1. Communication

When collaborating with other contributors, the experts have tried various strategies and different forms of communication. Prior to the introduction of a project-wide Slack channel, E1 preferred to communicate using email messages, comments on Jira tickets and comments on GitHub pull requests, because *"that were practically the only channels to communicate with people. Because at that time, only the IRC channel was available but it was rarely used, not even by the internal students"* (E1). In contrast to that, E2 prefers to employ Slack for both text and voice chatting as well as remote desktop sharing. Instant messaging is strongly preferred in comparison to email because *"personally, I think that writing emails would result in much more overhead in contrast to instant messengers"* (E2). On the other hand, E3 prefers to directly get in touch with the contributors using face-to-face communication when ever possible: *"solely communicating remotely in a written, asynchronous form is not that efficient compared to situations where people sit next to each other directly explaining things in person"* (E3). If face-to-face contact is not possible, the expert encourages to directly get in touch with each other using any form of synchronous communication. The expert stresses the importance to simultaneously speak with each other.

To continue, E2 believes that when introducing newcomers, communication should be as frequent and direct as possible. The expert once introduced a couple of newcomers holding a remote meeting once a week. It is emphasizes that this was far too infrequent and that this was not sufficient in order to familiarize with the project, especially because of its high complexity and its large scale. Technical hurdles, mainly resulting from insufficient modularization were mentioned as one of the reasons why it is probably not feasible to *"sit down for just two hours a week and complete a task. If so, this is only possible for small bug fixes"* (E2). To counteract this issue, the contributors started to meet face-to-face and performed on-site pair programming. Several advantages were mentioned in comparison to the remote collaboration, among other things because *"interpersonal matters helped to break down social barriers"* (E2) and that the contributors were more likely to ask questions whenever something was not clear to them. From a technical point of view, the interviewee states that there was almost no difference between the remote and co-located collaboration:

> *"When technology is available which allows to remotely share a screen and to annotate parts of the screen [using a virtual pen], then it is not vital to sit next to each other. [...] Maybe sometimes it can be even better, because you do not have to switch mice and you can draw on the screen."* (E2)

8.1.2. Project guidelines

When it comes to the main problems during the contribution, E1 recalls that the application of TDD and testing in general were one of the biggest hurdles. According to the expert, students often do not have any technical expertise in these areas at all. This is in accordance with the experience of E2 who notes that writing tests is difficult for newcomers, because some contributors *"do not even know what a test is, or what JUnit is or what an instrumented test is"* (E2). To continue, it is stressed that sometimes tests are skipped simply because contributors claim that *"this [feature] is difficult to test"* (E3). This turns out to be a great challenge, especially because on the one hand the contributors' technical experience is so diverse and on the other hand the project's expectations are very high:

> *"One of the biggest challenges for newcomers is that we are taking automated tests so seriously. There's no one, no matter where you look at, there's no one who takes that so seriously. Not even the Android Developer Guides. [...] If you ask me, that's more like performing research."* (E2)

This is confirmed by E1 who states that the project has several strict guidelines which need to be complied to in order to get code merged into the project's code base. These policies are mainly about software tests, Clean Code (Martin, 2008) and coding standards. It is emphasized that these standards are as important as a well functioning code in order to maintain a project in the long run. In compliance to that it is claimed that *"our project only works because we have that many tests. [...] Without them, we would have no progress at all"* (E3). According to E3, this has also been confirmed by many senior members. Without regression testing it would be mandatory to manually test the whole app after every change which would not be possible due to its high complexity and its large scale. On the other hand, E1 counteracts that these strict guidelines can also lead to demotivation when pull requests need to be changed multiple times and do not get merged for a long period. This behavior was encountered at some of the interviewees' students when it was observed that *"suddenly the student stopped to write long explanations as usual and started to answer in very short sentences"* (E1) or the students even *"do not continue to work on other tickets and are not that responsive anymore"* (E2). Sometimes contributors seem to be a little bit annoyed and try to convince their reviewers to finally merge their changes (E3). In those cases, demotivation was a reaction to technical hurdles and multiple change requests.

8.1.3. Awareness

It is proceeded that the project's internal guidelines can be an unexpected hurdle for external contributors who simply do not know that there are certain standards. To increase the awareness in such cases, direct contact with a pull request's author is usually established by E1 to, for example, *"tell the contributor that the pull request has checkstyle[1] warnings and that checkstyle can be run locally by executing this and that command"*. This is a vital measure, because for external contributors there is no exhaustive list of rules and standards which can be read before opening a pull request. On the one hand, it is emphasized that without documentation or written resources

[1]A static code analysis tool that checks whether Java code adheres to a coding standard. https://checkstyle.sourceforge.io, visited on 12 July 2019

of any kind, there is always the need for people to directly communicate unwritten rules to newcomers which can be a time- and resource-intense procedure. On the other hand, it is stressed that, according to E1, this is far more personal than receiving a large list of rules at the beginning of a contribution. Similar to that, E2 reveals that a large number of contributors are simply not aware of the technical details and quality standards required in this project. It is therefore necessary to raise the awareness of these standards because, according to E2, contributors tend to think that *"code is equal to code, and testing is equal to testing"*, no matter of which quality. It is emphasized that *"this happens often, very often; actually I would say that this is the standard already"* (E2).

Therefore it is necessary to properly onboard and train all newcomers. The high fluctuation and the large number of short-term contributions set a challenge to pass on knowledge to newcomers so that it is possible for them to comply to these guidelines. This is a time-consuming process; therefore it is vital to motivate every contributor to stay in the project for as long as possible (E3). It is emphasized that there is no static hierarchy in this project and that it is important that, ideally, every contributor makes progress to finally become a senior. Nevertheless, it is continued that *"sometimes I have the feeling that seniors always stay seniors and that newcomers always stay newcomers"* (E3).

8.1.4. Code reviews

A few years ago, a long email was received by E1 complaining about the way pull requests are handled in this project. The author of this email was complaining because it took so long until his or her pull request got merged which, according to the external contributor, was unusual for FOSS projects. This was the author's first and last contribution to this project. It is emphasized that there is a potential for improvement when it comes to the review of pull requests, because *"in this project, it is kind of the way that many people are getting discouraged because code reviews are handled in a too negative way"* (E1). Sometimes the only feedback you get for a complex pull request is that *"this needs to be done in this way, that this needs to be done in that way and this needs to be done like that – and so on, and so forth"* (E1).

According to the expert, this can be very demotivating, especially when contributors only hear negative things without being praised for the good and positive parts of their pull requests. It is mentioned that one of the main reasons for this behavior could be that contributors of this project are used to collaborate with students within the Graz University of Technology, and that students possibly tend to address their student colleagues more directly, which should not be done the same way with external contributors. In E2's opinion, contributors can be insulted rapidly when telling them that their changes do not comply to certain standards or that their pull requests can be improved in certain ways, although it is believed that:

> *"Personally, I do not know a single person who does that because he or she wants to cause harm or wants to be evil in any manner. Code reviews are meant to be absolutely constructive, among other things to improve the code quality."* (E2)

When it comes to code reviews, E3 prefers to directly get in touch with the contributors using face-to-face communication when ever possible, because sometimes developers do not realize how unfriendly they appear to other people when writing comments on GitHub as part of their code reviews. According to E3, this happens especially when developers have built an emotional opinion about the code base and its underlying quality. When communicating asynchronously, E3 states that the interpersonal part is getting lost, which can sometimes cause troubles, especially when communicating both *"in a written form and emotionally"*:

> *"Instead of passing on change requests back and forth with insane delays and great frustration, code reviews are finished immediately [when communicating face-to-face] and both parties are happy. Among other things, this is because they directly communicate with another real human being who, instead of giving disparaging comments, turns out to be friendly."* (E3)

8.1.5. Domain expertise

Apart from the technical point of view, it is mentioned that in many cases changes to pull requests are necessary because new features are not behaving as expected or are not perfectly implemented from the users' point of view. According to E3, in those cases sometimes as many as 20 change requests are passed on before the code can be merged. Different to other FOSS projects, Pocket Code is not programmed by the same people who use it, which sometimes causes unawareness of what is important for the users:

> *"In general it is not problematic that our developers are no end users. But sometimes this results in an insufficient problem awareness. [...] Because our developers are not using the app, they never experience the frustration when things do not work."* (E3)

When developing new features, it is stressed by E2 that vague requirements and unclear specifications can be one of the main issues. It is believed that sometimes specifications do not appear to be defined in a sufficiently comprehensive way. When it comes to complex features, the behavior for corner cases[2] is often not specified well enough. The developers' missing domain expertise makes it infeasible to autonomously give meaningful suggestions to undefined behavior. Therefore it is important that requirements are getting defined as detailed as possible. Nevertheless, it is stressed that this process has drastically improved recently (E2).

8.1.6. Guidance from the community

According to E3, newcomers have to learn so many things while making their first contribution that it is almost impossible to accomplish that without further guidance from the community. To prevent situations where a newcomer's contribution could not be continued autonomously, a spreadsheet with multiple bug tickets was provided by E1 in order to enable a steady, uninterrupted contribution. This can be especially helpful when collaborating remotely across several timezones. Bug reports were chosen by the expert because they, according to E2, usually do not require the understanding of the whole code base and they can typically be solved within a short timeframe.

[2]A corner case involves a situation where input parameters or environmental variables are outside of their normal value range. Typically, at a corner case several boundary conditions are met at once.

Similarly, a pool of training tickets has been provided to the newcomers mentored by E2 and E3. This list consisted of easy to fix bug tickets which were explicitly reserved for newcomers. Such a list is important in order to enable a smooth project entry. On the other hand, it is counteracted that, according to E2, chances are high that with such a list people tend to start working on multiple tickets simultaneously before even finishing one single ticket. It is continued that sometimes pull requests are submitted and never respected again even though they are not completed at all:

> *"Sometimes pull requests are handed in and that is it. Test results or similar checks are not even looked at."* (E2)

It is emphasized that, in those situations, it would be much more efficient to carefully look whether all checks in the pull request have passed and hence completely finish one ticket before starting working on other tickets simultaneously.

8.1.7. Software architecture

It is argued by E1 and E2 that some of the technical entry barriers are a result of the project's software architecture which *"sometimes makes things simply untestable"* (E2). The interviewee claims that this is a negative side-effect from shipping new features too early in the past. In those cases, the code quality has been neglected in the favor of new functionalities. As a result, code with moderate to bad quality was merged while it was hoped to be patched at some point later in time. In reality this was rarely done, as stressed by E2, who names the high contributor fluctuation as one of the main reasons for this issue.

8.1.8. Interpersonal communication

Multiple internal and several external contributors have been introduced into the project by the experts. When it comes to the main difference between the co-located and the remote collaboration, it is emphasized by E1 that when being physically collocated, the team spirit, the sense of belonging and the community bonding is much stronger. E1 quotes that *"when we were in the team room we also talked about other things not related to the project [...] and you got to know many other people which was resulting in a more relaxed and personal atmosphere."*. When talking directly to each other on a synchronous basis, E1 mentions that there is almost no difference between a remote and co-located collaboration. Nevertheless, it is suggested to have an as direct contact as possible. A similar experience was shared by E2 who argues that from a technical point of view both forms of collaboration are equally good. The main advantages of a co-located collaboration though is that *"people start to talk about things they would not write [on digital communication channels]"* (E2). As a result you get to know many other people, their experiences and interests and you have the ability to debate about different problems:

> *"Personally, I think this is something really cool. You get to know other people with cool ideas. Especially when communicating with people from other teams. [..] You have the chance to ask them how they would deal with this and that situation. And I can not imagine that there would be a similar communication when using Slack."* (E2)

This complies with the experience of E3, who mentions that when communicating asynchronously on a written basis, the interpersonal component is getting lost. As a result people feel more distanced from each other which can have several negative side effects. Therefore it is suggested to directly communicate with each other using face-to-face meetings, or, if not possible, to use any form of synchronous communication.

9. Findings and recommendations

In this section the propositions resulting from the case studies are triangulated with the outcomes from the expert interviews to confirm, deny and extend theory. The cross-validated outcome is then discussed with findings from related literature. Finally, in Section 9.2, recommendations for the Catrobat FOSS project are given.

9.1. Findings

The experts have underpinned *Proposition 2* in two ways: To start with, it is argued that, at Catrobat, developers sometimes have an insufficient problem awareness when it comes to the end users' perspective (E3). While Hippel and Krogh, 2003, Scacchi, 2002, and Scacchi, 2005 state that FOSS is regularly developed by the same people who use it, this is not the case for the Catrobat project which aims to *"empower children and teenagers to easily create their own program"* (Fellhofer, Harzl, and Slany, 2015). In his research, Fitzgerald, 2006 argues that traditional FOSS projects tend to be horizontal infrastructural systems – for example operating systems, web server, database management systems and compilers – where developers were unexceptionally users of the software in development. As Section 4.3 illustrates, the majority of Catrobat's contributors are students from the Graz University of Technology, hence can not be classified as typical "developer-users". As a result, developers sometimes have a *lack of domain expertise* as characterized by Steinmacher, Conte, Gerosa, et al., 2015. In the study of Krogh, Spaeth, and Lakhani, 2003, it is suggested that the level of difficulty to start *"modifying and coding"* in open source projects is a barrier relative to the developer's profession, technical experience and pre-existing domain knowledge. In order to contribute successfully in the long run a developer needs to be experienced in all areas. This is in accordance with *Case A*, *Case B* and *Case C* where a constant communication about domain-related issues has been observed, among other things arising from from an insufficient problem awareness, as suggested by the experts.

Secondly, it is necessary to comply to the project's strict guidelines about TDD and Clean Code (Martin, 2008) which can be difficult for newcomers, especially when developing complex features. It is argued that, among other things, *"one of the biggest challenges for newcomers is that we are taking automated tests so seriously"* (E2). As emphasized by E3, newcomers have to learn so many things while making their first contribution that it is almost impossible to autonomously accomplish that without further guidance from the community. In literature it is claimed that mentorship can offer an exceptionally important opportunity to give newcomers an understanding of (a) domain-related affairs, (b) technical aspects; and, (c) organizational rules (Canfora et al., 2012; Trainer, Chaihirunkarn, and Herbsleb, 2013). Furthermore, Steinmacher, Conte, Gerosa, et al., 2015 state that newcomers look for mentorship and guidance when starting their contribution which confirms with the outcomes of the case studies and the expert interviews.

To continue, *Proposition 1* and *Proposition 6* were strongly supported by all experts who provided newcomers with a pool of training tickets to help to *find a task to start with*, a technical barrier mentioned by Steinmacher, Conte, Gerosa, et al., 2015. Small and easy to implement bug tickets were chosen and explicitly reserved because they, according to E2, usually do not require the understanding of the whole code base and can typically be solved within a short timeframe. Wang and Sarma, 2011 state that new developers typically start contributing to open source projects by lurking on issue trackers trying to discover potential bugs to start working on. In their study it is argued that it is difficult to find out which bugs are of interest, can be solved with pre-existing knowledge and are no duplicates – a process which takes a substantial amount of time. This barrier is also mentioned by Gousios, Storey, and Bacchelli, 2016 who emphasize the need for a comprehensive task list with recommendations for newcomers. To overcome this challenge at Catrobat, a pool of easy to implement bug tickets were notably marked as training tickets in Jira and explicitly kept free for newcomers by E3. When using Kanban, a popular framework to implement agile software development practices (see Chapter 3), it is recommended to limit work in progress (WIP) per workflow state (Kniberg and Skarin, 2010). While WIP limits offer several advantages for co-located teams – like the predictability of cycle times, the discovery of work blockers or a continuous work capacity (Epping, 2011; Ahmad, Markkula, and Oivo, 2013; Anderson and Carmichael, 2016) – this can lead to several issues for newcomers in FOSS projects. First, it can slow down the newcomers' process to find a task to start with. Furthermore, when WIP limits are low and the community's response times are high, this can be a potential restraining factor while depending on guidance from the community. As suggested by *Proposition 6*, this issue was overcome by simultaneously working on multiple tickets (*Case B* and *Case C*). Thus, it is crucial to offer newcomers the ability to easily find as many tasks as possible to work on and to easily find out whether a ticket can be solved with the contributor's pre-existing knowledge.

Software testing and the project's architecture were mentioned as one of the major technical barriers. Demotivation was observed by both the contributors in *Case A*, *Case B* and *Case C*, as suggested in *Proposition 5*, and the contributors mentored by the experts. It was observed as a reaction to technical hurdles and multiple, long-lasting change requests which were mainly caused due to the non-compliance to certain quality standards, like the existence of software tests. As mentioned in Section 7.1, technical hurdles are reinforced due to the project's high interdependencies and low modularity. A case study by Syeed and Hammouda, 2013 shows that, to a large extent, the communication pattern of the contributors is due to the communication needs induced by the software architecture. Supporting Conway's law (Conway, 1968), the researcher found out that interdependencies and low modularity influence the contributors' communication, thus verifying the socio-technical congruence. Hannebauer, 2016 argues that a higher degree of modularity in FOSS projects lowers technical barriers allowing newcomers to start development as soon as they gained a certain understanding about the modules involved. According to E1 and E2 and the contributor of *Case A*, the software architecture in the Catrobat project is not ideal which *"sometimes makes things simply untestable"* (E2). As observed in *Case A*, the desire to write *"write cleaner/simpler/safer code"* has been noted in order to increase the attractiveness for future contributors since *"all these minuses push open-source developers away"*. This confirms with the literature review by Beecham et al., 2008 which reveals that *"producing poor quality* *software"* is a frequent de-motivator in software engineering. On the other hand, Ye and Kishida, 2003 theorize that *learning* is one of the most frequent reasons why developers contribute to FOSS projects. Thus, it is important that developers can constantly learn from a project and value it from a technical point of view because otherwise they may stop contributing and move on to a different project (Scacchi, 2010).

As suggested by all experts, newcomers are getting discouraged because *"in this project, [...] code reviews are handled in a too negative way"* (E1). It is continued that contributors can be insulted rapidly when telling them that their changes do not comply to certain standards or can be improved in certain ways, although it is believed that no one *"does that because he or she wants to cause harm or wants to be evil in any manner"* (E2). This concurs with the observations of E1 who received a complaint from an external contributor protesting about the way pull requests are handled in this project which, according to the external contributor, was unusual for FOSS projects. This is in accordance with the findings of Gousios, Storey, and Bacchelli, 2016 arguing that contributors should have more empathy towards new contributors when reviewing pull requests. Referring to code reviews, E3 suggests to directly get in touch with the contributors using face-to-face communication when ever possible. When collaborating remotely, it is suggested to use any form of synchronous voice chatting. When using an asynchronous form of communication, like mailing lists or comments on GitHub, the interpersonal component is getting lost and people feel more distanced from each other, which can sometimes have several negative side effects (E3). A case study at Google reveals that the geographical distance between the author and the reviewer is perceived as potential cause of delays and misunderstandings (Sadowski et al., 2018). The researchers found out that communication plays a critical role while performing code reviews. Tone and power, referring to the ability to induce another person to change something, were mentioned as possible source for frustration and demotivation. In their study, Bacchelli and Bird, 2013 recommend that code reviews should be done in-person or at least on a synchronous basis. It is argued that developers have the need for richer forms of communication to overcome the aforementioned issues. This is in accordance with the media richness theory (MRT) by Daft and Lengel, 1986, which illustrates that personal, direct communication is more effective for complex and unclear situations. Communications able to clarify ambiguous concerns, punctually change awareness and give immediate feedback are referred to as rich. As depicted in Appendix G, the richest media is face-to-face followed by telephone whereas letters and impersonal written documents represent the lowest information richness and effectiveness of communication (Daft and Lengel, 1986).

Similar to that, it is claimed that face-to-face contact results in a *"more relaxed and personal atmosphere"* (E1) and that *"people start to talk about things they would not write [on asynchronous communication channels]"* (E2). As a result, the expert mentions that you get to know many other people, their experiences and interests which you would not when communicating on an asynchronous basis. This goes in line with the outcomes of the case study underpinning *Proposition 4* where strong ties with the community were only established when directly communicating on a frequent and synchronous basis. It is crucial for newcomers to build strong ties with the community in order to finally transform to long-term contributors. This transformation is vital for FOSS projects in order to succeed in the long-run (Steinmacher, Conte, Gerosa, et al., 2015; Trainer, Chaihirunkarn, Kalyanasundaram, et al., 2014; Silva et al., 2017; Krogh, Spaeth, and Lakhani, 2003).

The importance of frequent communication is emphasized by E2 as well as E1 who preferred to get in touch with his mentees on a daily basis. Experience has shown that communicating on an infrequent basis, like once a week, is not enough in order to familiarize with the project, especially because of the its high complexity and its large scale (E2). As suggested in *Proposition 3*, this was also observed in the case studies. Since newcomers rely on support of the community (E2, E3, *Case A*, *Case B*, *Case C*), it is important to quickly respond to their questions. Herbsleb and Grinter, 1999 argue that developers strongly rely on informal ad hoc communications to manage exceptions and eliminate uncertainties. The researchers claim that this becomes challenging when being geographically separated, having different timezones and cultural differences. Consistent with Steinmacher, Conte, Gerosa, et al., 2015 who argue that *delayed answers* are a barrier affecting the contributors' motivation, deferred communication was seized as a problem by both the community (*Case A*) and the contributors (*Case B*, *Case C*). This is in line with the findings of Gousios, Storey, and Bacchelli, 2016 who state that poor responsiveness is one of the biggest challenges and frequently generates frustration among the contributors. Jensen, King, and Kuechler, 2011 proved that both receiving an answer and receiving it in a timely manner is fundamental for newcomers to enable a steady contribution. A study of the Freenet[1] project by Krogh, Spaeth, and Lakhani, 2003 illustrates that 10.5% of the participants of their mailing list did not receive a response resulting in not appearing on the same list again, underpinning the necessity for frequent and direct communication.

While the need for documentation has been addressed by the contributors of *Case A*, *Case B* and *Case C*, this has not been addressed by the interviewees. Instead, the existing amount of documentation was considered to be sufficient. Nevertheless, it has been emphasized by E1 and E2 that, in the majority of cases, contributors are not aware of the strict guidelines and unwritten standards required in this project. According to E1, there is no exhaustive list of rules and standards which can be read by external newcomers before opening a pull request which makes a large number of interactions and a large number of change requests unavoidable. Gousios, Storey, and Bacchelli, 2016 reported *project compliance* as one of the major hurdles for newcomers. Knowledge about these unwritten standards can only be passed on to newcomers by communicating with them. Referring to the transfer of knowledge, the high contributor fluctuation and the diverse degrees of experience and seniority set an additional challenge at Catrobat (E3). However, it is crucial for new contributors to become familiar with both the project's technical and social rules and standards (Wang and Sarma, 2011). In their research, Bacchelli and Bird, 2013 argue that code reviews offer a practical ability for a bidirectional knowledge transfer and to increase the awareness of the team. Similar to that, it is claimed that when employing pair programming, knowledge is constantly being passed on between the contributors (Cockburn and Williams, 2001). This complies with E2, E3 and *Proposition 3* where pair programming was preferred when introducing newcomers. From a technical point of view, the experts claim that there is almost no difference between a remote and co-located collaboration, provided that the appropriate tools are available. Nevertheless, several advantages were mentioned in comparison to a remote collaboration, among other things because *"interpersonal matters helped to break down social barriers"* (E2) and that the contributors were more likely to ask questions whenever something was

[1]Freenet is a peer-to-peer platform for communication

not clear to them. In their research, Crowston and Shamshurin, 2017 proved that successful FOSS projects have more communication than unsuccessful projects which correlates with the finding of this thesis (*Proposition 3*).

9.2. Recommendations

A set of recommendations are presented in order to help to lower the entrance barriers for newcomers and to streamline the process of handling pull requests and code reviews. The recommendations are based on three basic principles: (a) awareness, (b) transparency; and, (c) frequent and direct communication.

9.2.1. Find a task to start with

To enable a smooth project entry, it is crucial to easily find adequate tasks that can be completed with the contributors' pre-existing knowledge. While a list of tickets can be easily found on Jira, it is currently not possible to immediately determine the level of skills required for development. Although recently the Catroid subproject has started to encode similar things into the tickets' name (see Figure E.1), it is recommended that this essential information should be a required part of the project-wide Jira workflow. An additional required dropdown field should ensure that this information is filled out when tickets transform from the *Issues Pool* to *Ready for Development* (see Figure D.1). This field should represent the level of seniority required for a certain ticket (e.g. starter, medium, advanced) and should be displayed on the Kanban board as well as used in the link on the *How to contribute* section on GitHub[2] as part of Jira's filter query.

Additionally, in the current Jira workflow newly created *bug tickets* are gathered in the *Bug Backlog (Seniors only)* workflow status. While senior contributors can directly move these issues to the *Ready for Development* status, newcomers (internal and external) can not start working on these

[2]https://github.com/Catrobat/Catroid/blob/develop/README.md, visited on 8 August 2019

tickets because of insufficient permissions. Nevertheless, it is important to provide newcomers with a constant list of training tickets. Therefore it is favorable to set high (or no) WIP limits for trainings tickets and move them to *Ready for Development* as soon as possible.

9.2.2. Pull request checklists

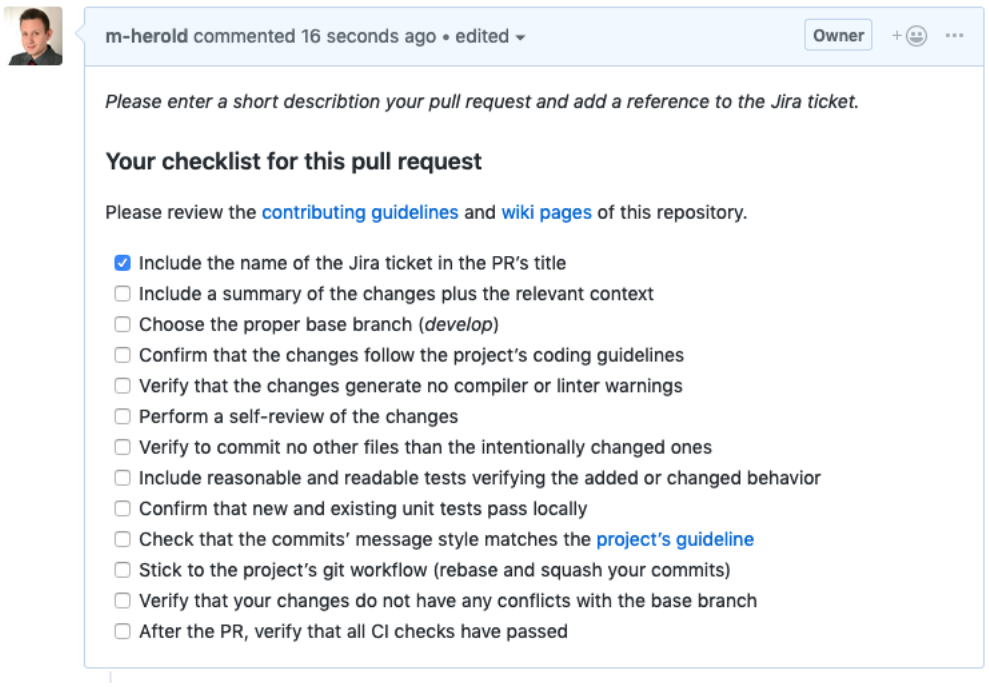
As previously discussed, in many cases contributors are not aware of the guidelines and standards required in this project. As a consequence a large number of change requests need to passed on which increases the *time to merge* and thus increases the *lead time* of a release (see Chapter 3). As a result, this gradually generates frustration and demotivation among both newcomers and code reviewers. To raise awareness and make policies explicit, as suggested in one of the Kanban practices (see Chapter 3), it is recommended to provide contributors with a prominent checklist which can be reviewed before submitting a PR. This can be implemented as a list of checkboxes within a PR template on GitHub[3]. As a result, this list should make sure that developers do not miss anything important and thus remind the author of a PR to:

- Include the name of the Jira ticket in the PR's title
- Include a summary of the changes plus the relevant context
- Choose the proper base branch
- Confirm that the changes follow the project's coding guidelines
- Verify that the changes generate no compiler or linter warnings
- Perform a self-review of the changes
- Verify to commit no other files than the intentionally changed ones
- Include reasonable and readable tests verifying the added or changed behavior
- Confirm that new and existing unit tests pass locally
- Check that the commits' message style matches the project's guideline
- Stick to the project's git workflow (rebase and squash your commits)
- Verify that your changes do not have any conflicts with the base branch

[3]https://help.github.com/en/articles/creating-a-pull-request-template-for-your-repository, visited on 9 August 2019

- After the PR, verify that all CI checks have passed
- Read further details on the wiki pages (*Commit message guidelines*[4] and *Creating a pull request*[5])

To continue, this template can also serve as an additional checklist[6] for code reviewers, lowering the barrier for contributors to initiate or take part in code reviews. The aforementioned list includes issues emerged during the case studies, but is not exhaustive; it needs to be adapted on a regular basis. An example of such a checklist is illustrated in Figure 9.1.



Please enter a short describtion your pull request and add a reference to the Jira ticket.

Your checklist for this pull request

Please review the contributing guidelines and wiki pages of this repository.

- [x] Include the name of the Jira ticket in the PR's title
- [ ] Include a summary of the changes plus the relevant context
- [ ] Choose the proper base branch (*develop*)
- [ ] Confirm that the changes follow the project's coding guidelines
- [ ] Verify that the changes generate no compiler or linter warnings
- [ ] Perform a self-review of the changes
- [ ] Verify to commit no other files than the intentionally changed ones
- [ ] Include reasonable and readable tests verifying the added or changed behavior
- [ ] Confirm that new and existing unit tests pass locally
- [ ] Check that the commits' message style matches the project's guideline
- [ ] Stick to the project's git workflow (rebase and squash your commits)
- [ ] Verify that your changes do not have any conflicts with the base branch
- [ ] After the PR, verify that all CI checks have passed

Figure 9.1.: An example for a pull request template on GitHub

[4]https://github.com/Catrobat/Catroid/wiki/Commit-Message-Guidelines, visited on 9 August 2019

[5]https://github.com/Catrobat/Catroid/wiki/Creating-a-pull-request, visited on 9 August 2019

[6]In addition to https://confluence.catrob.at/display/KNOWHOW/How+to+do+Code+Reviews, visited on 10 August 2019

9.2.3. Code review code of conduct

Code reviews constitute one of the most central parts of collaborative software engineering. As illustrated before, they ensure that changes meet the project's quality standards, allow to transfer knowledge and in many cases represent the first point of contact between a newcomer and the community. Therefore it is important to conduct these reviews as effectively as possible which requires a set of socio-technical skills. As emphasized in the previous section, at Catrobat, there is still room for improvement when it comes to code reviews. Passing on multiple change requests can be a frustrating and demotivating part for both the reviewer and the developer. While the previously mentioned checklist could help to lower the number of change requests in some cases, it is most essential to develop an effective communication and reviewing culture. Therefore it is necessary to raise awareness to:

- *Be constructive.* Make sure that comments have a useful purpose and that comments are about the code and not about the code's author.
- *Give suggestions instead of commands.* Ask questions instead of giving answers. Try to understand the author's motives for choosing a certain implementation.
- *Distinguish personal preferences from project guidelines.* Do not try to make the code look like it was written by the reviewer.
- *Share responsibilities* and avoid selective code ownership. Avoid phrases like "your code" or "my function". Avoid a blame culture.
- *Provide positive feedback.* Try to occasionally provide positive feedback when deserved, but do not exaggerate.
- *Learn from each other.* Celebrate mistakes and take up the opportunity to share knowledge.
- *Be responsive.* Quickly respond to questions and be accessible to clarify misunderstandings. If possible meet in person, initiate a remote desktop sharing conference, or, if not possible, at least communicate on a synchronous basis.

Most importantly, it is fundamental that code reviews are based on technical facts instead of personal preferences and that change requests are communicated in a technical and non-derogatory way. These social values could be outlined together with a code of conduct (CoC) for code reviews. While in literature it is recommended that CoCs should be collaboratively developed by an ethics commission, similarities among communities may plead for the reusage of existing CoCs (Tourani, Adams, and Serebrenik, 2017). The researchers crawled several hundreds FOSS projects and found out that the *Contributor Covenant*[7] offers a practicable template designed for FOSS projects. Like most codes, the *Contributor Covenant* fosters a respectful, welcoming behavior and prosecutes sexist and racist language, harassment and violence. While *"constructive criticism"* plays a central role, it is suggested to be *"respectful of differing viewpoints and experiences"* which offers a very good starting point for developing an effective code review culture. Beyond that, it is crucial that these social value are lived by the community, most importantly by the senior contributors and code reviewers to exemplify and foster a welcoming learning culture.

9.2.4. Natural mentoring culture

Experience of the case studies has shown that mentorship offers a great opportunity for newcomers to find their way into the project. Newcomers can overcome hurdles related to (a) technical experience (like software testing), (b) domain expertise; and, (c) project guidelines. While programs like GSoC offer a one-to-one mentorship, this is not feasible for all newcomers due to the project's limited resources. Therefore it is essential to develop a natural mentoring culture as part of the daily workflow. Contributors should be motivated to do code reviews in person or at least on a synchronous basis when communicating remotely. The results of the case studies have shown that Slack offers a great tool for synchronous communication and a great solution for remote pair programming. Furthermore, it is presented that personal code reviews and pair programming sessions (either remotely or face-to-face) foster a bidirectional knowledge transfer and reduce the occurrence of misunderstandings and delays. In addition to this, a positive influence on community bonding, team awareness, and on the contributors' motivation has been observed. Therefore it is necessary to motivate contributors to perform pair programming sessions as often as possible, which has

[7]https://contributor-covenant.org, visited on 9 August 2019

also been suggested in the XP methodology (see Chapter 3). For internal contributors it is recommended to enforce regular sessions by requiring a certain amount of tickets to be implemented as pairs; ideally starting from the very beginning of the contribution to expedite the onboarding process.

10. Limitations

10.1. Threats to validity

In his book, Yin, 2017 recommends four tests to establish the quality of empirical case studies:

- *Construct validity* examines whether a study investigates what it claims to evaluate. This test ensures that data collected by the researcher is not biased by "subjective" judgements (Yin, 2017). Quantitative and qualitative data (see Section 6.2) from multiple sources were collected to increase the construct validity. The author had no control over the outcomes of the cases since the study was conducted almost one year after the end of the last case. Additionally, triangulation was applied to enable observations from different points of views, as suggested by Flick, 2004. A further tactic to increase construct validity is the use of a draft case study (Yin, 2017) which has not been conducted, representing a potential limitation of this thesis.

- *Internal validity* assures that no other variable expect the one under study is causing the result (Campbell, 1986). Seaman, 1999 emphasizes that measures must be taken by the researcher to ensure that the participants being observed should not constantly be aware of being observed in order to ensure that the inspected behavior is "normal". At the time these case studies were conducted, neither the author nor the contributors were aware of the existence of this thesis, thus having no possibility to intentionally influence their behavior. According to Yin, 2017, internal validity is mainly a concern for explanatory case studies rather than for descriptive case studies (like the one in this thesis).

- *External validity* deals with the problem of arguing whether a study's findings are generalizable to settings other than described in the study (Yin, 1984). As mentioned in Chapter 5, the purpose of this thesis is to extend and generalize theories (analytic generalization) instead of trying to generalize the results from this case study to a wider population (statistical generalization), as suggested by Yin, 1984 and Strauss and Corbin, 1990. Deducted from this it can be said that the findings observed within the Catrobat FOSS project, do not implicitly need to occur in other projects. The propositions in this thesis are solely made in conjunction with the project under study. Furthermore, it is important to state that the interview outcomes are solely based on the expert's personal judgements and thus can not be statistically generalized to all project members. To get a more representative insight into the contributors' views, additional interviews or questionnaires need to be conducted. Finally, to get a more holistic picture of the research topic, it would be necessary to make further research within various other FOSS projects. However, it could be difficult to draw cross-project conclusions due to the projects' diverse organizational structures, guidelines and workflows.

- *Reliability* ensures that an external researcher is able to conduct the same case study over again and to be able to reproduce the same findings. In order to enable this, a detailed description of the research method utilized in this thesis has been stated in Section 5. Data has been compiled into a research database separated from the actual research report which can be shared individually to enable the inspection of the raw data by other people in order to replicate the conclusions of the case study. The collected evidence is stored in a retrievable form as illustrated in Figure 6.1, increasing the reliability of the case study (Yin, 2017). To continue, the expert interviews have been transcribed to enable a separate analysis by additional researchers. Due to the limitations of the thesis, the open, axial and selective coding was not reviewed by other investigators. Although codes have been precisely characterized by properties and keywords, the actual coding decision was solely made by the author of this thesis, offering a possibility to obtain a diverging categorization when performed by other researchers. Whereas in a scientific setup, the coding phase could have been conducted by multiple researchers who mutually agree on the final coding, this can be identified as a limitation of this thesis.

10.2. The third generation of FOSS

Figure 10.1 illustrates a transition model proposed by Yamakami, 2011 where FOSS shifts from a free economy to a cost-benefit economy. FOSS started with an economy where gifts were the basic construct having no traditional economic rules. A contribution was made as a gift, in return to the gift of sharing FOSS with others. In the subsequent years, open source communities became greater, eventually resulting in the evolvement of social norms and ties. The emerging *guild economy* was built upon an association of craftsmen where the community becomes the core of the model. Nowadays this model is shifting towards a cost-benefit economy applying traditional economic rules.

This thesis does not focus on the cost-benefit economy and is not targeting to put the outcome of a contribution in relation to the costs resulting from support and mentoring. Disregarding cost-benefits analyses, it is not questioned whether the mentorship of a contributor is the best investment of time and money or whether the resources necessary for mentorship can be employed differently in a more efficient way.

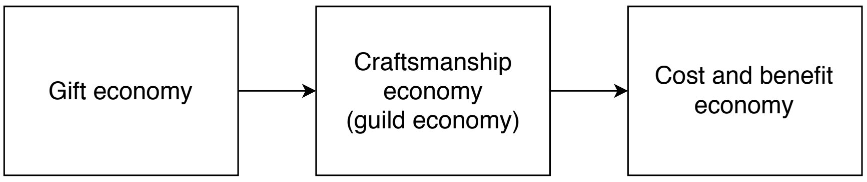


Figure 10.1.: The three generation view of the FOSS economy types (Yamakami, 2011)

11. Conclusion and future work

This thesis qualitatively investigates first-time contributions to the collective software development process of an agile FOSS project. Presenting the case of Catrobat, a multiple case study was conducted to identify the main challenges newcomers have to overcome in order to place their first contributions. The results emerged from qualitative and quantitative data collected from the analysis of multiple contributions within the GSoC program as well as from interviews conducted with long-term experts. Based on the triangulation of the outcomes, recommendations are given to streamline the collaborative software development process at Catrobat and thus lower the project's entry barriers.

11.1. Future work

The case studies elaborated in this thesis could be used as a draft case study for further research, as suggested in Chapter 10. The findings in Chapter 9 could serve as an input for a larger study targeting a greater set of Catrobat's contributors by conducting quantitative research. Additionally, experiments could be conducted on how to efficiently overcome the aforementioned challenges. To continue, further investigations could focus on the differentiation of internal and external contributors and on how contribution can be optimized for both parties. All data, observations and assumptions are exclusively limited to the Catrobat project and do not have to apply for other projects. To get a more holistic understanding of the socio-technical environment, future researchers could compare the case of Catrobat with other FOSS projects.

Bibliography

Ahmad, Muhammad Ovais, Jouni Markkula, and Markku Oivo (2013). "Kanban in software development: A systematic literature review." In: *2013 39th Euromicro conference on software engineering and advanced applications*. IEEE, pp. 9–16 (cit. on pp. 20, 21, 83).

Anderson, David J. and Andy Carmichael (2016). *Essential kanban condensed*. Blue Hole Press (cit. on pp. 19–21, 83).

Bacchelli, Alberto and Christian Bird (2013). "Expectations, outcomes, and challenges of modern code review." In: *Proceedings of the 2013 international conference on software engineering*. IEEE Press, pp. 712–721 (cit. on pp. 84, 86).

Beck, Kent (2003). *Test-driven development: by example*. Addison-Wesley Professional (cit. on pp. 29, 32, 33).

Beck, Kent, Mike Beedle, et al. (2001). "Manifesto for agile software development." In: (cit. on pp. 15, 34).

Beck, Kent and Erich Gamma (2000). *Extreme programming explained: embrace change* (cit. on pp. 16, 17, 30, 31, 33).

Beecham, Sarah et al. (2008). "Motivation in Software Engineering: A systematic literature review." In: *Information and software technology* 50.9-10, pp. 860–878 (cit. on p. 83).

Bogner, Alexander (2014). *Interviews mit Experten: Eine praxisorientierte Einführung (Qualitative Sozialforschung) (German Edition)*. Springer VS. ISBN: 9783531194158 (cit. on pp. 39, 72).

Bolici, Francesco, James Howison, and Kevin Crowston (2009). "Coordination without discussion? Socio-technical congruence and Stigmergy in Free and Open Source Software projects." In: (cit. on p. 1).

Bowler, Michael (2019). *Truck factor*. URL: http://www.agileadvice.com/2005/05/15/agilemanagement/truck-factor (visited on 6 February 2019) (cit. on p. 18).

Campbell, Donald T. (1986). “Relabeling internal and external validity for applied social scientists.” In: *New Directions for Program Evaluation* 1986.31, pp. 67–77 (cit. on p. 93).

Canfora, Gerardo et al. (2012). “Who is going to mentor newcomers in open source projects?” In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, p. 44 (cit. on pp. 1, 82).

Chow, Tsun and Dac-Buu Cao (2008). “A survey study of critical success factors in agile software projects.” In: *Journal of systems and software* 81.6, pp. 961–971 (cit. on p. 16).

Cockburn, Alistair (2001). *Agile Software Development*. Addison-Wesley Professional. ISBN: 0201699699 (cit. on pp. 15, 33).

Cockburn, Alistair and Laurie Williams (2001). “Extreme Programming Examined.” In: ed. by Giancarlo Succi and Michele Marchesi. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. Chap. The Costs and Benefits of Pair Programming, pp. 223–243. ISBN: 0-201-71040-4 (cit. on pp. 18, 86).

Conway, Melvin E. (1968). “How do committees invent.” In: *Datamation* 14.4, pp. 28–31 (cit. on pp. 1, 83).

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 (cit. on p. 7).

Crowston, Kevin, Qing Li, et al. (2007). “Self-organization of teams for free/libre open source software development.” In: *Information and software technology* 49.6, pp. 564–575 (cit. on p. 1).

Crowston, Kevin and Ivan Shamshurin (2017). “Core-periphery communication and the success of free/libre open source software projects.” In: *Journal of Internet Services and Applications* 8.1, p. 10 (cit. on pp. 7, 87).

Daft, Richard L. and Robert H. Lengel (1986). “Organizational Information Requirements, Media Richness, and Structural Design.” In: *Management Science* 32.5, pp. 554–571 (cit. on pp. 84, 85, 115).

David, Paul A., Andrew Waterman, and Seema Arora (2003). “FLOSS-US the free/libre/open source software survey for 2003.” In: (cit. on pp. 7, 9, 13).

Dinh-Trong, Trimg and James M. Bieman (2004). “Open source software development: a case study of FreeBSD.” In: *10th International Symposium on Software Metrics, 2004. Proceedings.* IEEE, pp. 96–105 (cit. on p. 9).

Ducheneaut, Nicolas (2005). “Socialization in an open source software community: A socio-technical analysis.” In: *Computer Supported Cooperative Work (CSCW)* 14.4, pp. 323–368 (cit. on p. 9).

Ebert, Christof (2008). “Open source software in industry.” In: *IEEE Software* 25.3, pp. 52–53 (cit. on p. 1).

Epping, Thomas (2011). *Kanban für die Softwareentwicklung*. Springer-Verlag (cit. on pp. 19, 20, 83).

Fellhofer, Stephan, Annemarie Harzl, and Wolfgang Slany (2015). “Scaling and Internationalizing an Agile FOSS Project: Lessons Learned.” In: *IFIP International Conference on Open Source Systems*. Springer, pp. 13–22 (cit. on pp. 34, 35, 46, 81).

Fitzgerald, Brian (2006). “The transformation of open source software.” In: *MIS quarterly*, pp. 587–598 (cit. on pp. 5, 6, 81).

Flick, Uwe (2004). “Triangulation in qualitative research.” In: *A companion to qualitative research* 3, pp. 178–183 (cit. on pp. 39, 93).

Fogel, Karl (2009). “How To Run A Successful Free Software Project-Producing Open Source Software.” In: (cit. on pp. 9, 35).

Fogel, Karl and Moshe Bar (1999). *Open source development with CVS*. Coriolis Group Books (cit. on pp. 7, 8).

GitHub (2019). *The State of the Octoverse*. URL: `https://octoverse.github.com/` (visited on 31 May 2019) (cit. on pp. 1, 4).

Glaser, Barney and Anselm Strauss (1967). *The discovery of grounded theory: Strategies for qualitative research*. New York: Aldine de Gruyter (cit. on pp. 37–39).

Gousios, Georgios, Margaret-Anne Storey, and Alberto Bacchelli (2016). “Work practices and challenges in pull-based development: the contributor’s perspective.” In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, pp. 285–296 (cit. on pp. 82, 84–86).

Halaweh, Mohanad, Christine Fidler, and Steve McRobb (2008). “Integrating the Grounded Theory Method and Case Study Research Methodology Within IS Research: A Possible ’Road Map’.” In: *ICIS*. Association for Information Systems, p. 165 (cit. on p. 38).

Halloran, Timothy J. and William L. Scherlis (2002). “High quality and open source software practices.” In: *2nd Workshop on Open Source Software Engineering* (cit. on p. 8).

Hannebauer, Christoph (2016). “Contribution Barriers to Open Source Projects.” PhD thesis. University of Duisburg-Essen, Germany (cit. on p. 83).

Herbsleb, James D. and Rebecca E. Grinter (1999). “Architectures, coordination, and distance: Conway’s law and beyond.” In: *IEEE software* 16.5, pp. 63–70 (cit. on pp. 1, 85).

Hippel, Eric von and Georg von Krogh (2003). “Open-Source Software and the ”Private -Collective” Innovation Model: Issues for Organization Science.” In: *Organization Science* 14.2, pp. 208–223 (cit. on pp. 6, 27, 81).

Jensen, Carlos, Scott King, and Victor Kuechler (2011). “Joining free/open source software communities: An analysis of newbies’ first interactions on project mailing lists.” In: *2011 44th Hawaii international conference on system sciences*. IEEE, pp. 1–10 (cit. on pp. 2, 8–10, 13, 14, 25, 86).

Kniberg, Henrik and Mattias Skarin (2010). *Kanban and Scrum-making the most of both*. Lulu.com (cit. on p. 83).

Korkala, Mikko and Pekka Abrahamsson (2007). “Communication in distributed agile development: A case study.” In: *33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007)*. IEEE, pp. 203–210 (cit. on p. 1).

Krogh, Georg von, Stefan Haefliger, et al. (2012). “Carrots and rainbows: Motivation and social practice in open source software development.” In: *MIS quarterly* 36.2, pp. 649–676 (cit. on p. 1).

Krogh, Georg von, Sebastian Spaeth, and Karim R. Lakhani (2003). “Community, joining, and specialization in open source software innovation: a case study.” In: *Research Policy* 32. (visited on 30 September 2013) (cit. on pp. 2, 9, 81, 85, 86).

Lakhani, Karim R. and Robert G. Wolf (2003). “Why hackers do what they do: Understanding motivation and effort in free/open source software projects.” In: (cit. on p. 10).

Martin, Robert C. (2006). *Agile Principles, Patterns, and Practices in C#*. Prentice Hall. ISBN: 0131857258 (cit. on pp. 15, 18).

Martin, Robert C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall. ISBN: 9780132350884 (cit. on pp. 18, 32, 75, 82).

Mayring, Philipp (2000). “Qualitative Content Analysis.” In: *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*. Vol. 1. 2 (cit. on p. 39).

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 (TOSEM)* 11.3, pp. 309–346 (cit. on p. 7).

Müller, Matthias, Christian Schindler, and Wolfgang Slany (2019). “Engaging Students in Open Source: Establishing FOSS Development at a University.” In: *Proceedings of the 52nd Hawaii International Conference on System Sciences* (cit. on pp. 24–26).

Nafus, Dawn, James Leach, and Bernhard Krieger (2011). “Deliverable D16: Gender: Integrated Report of Findings.” In: *Free/Libre/Open Source Software: Policy Support.* (Cit. on p. 13).

National Academies of Sciences (2018). *Assessing and responding to the growth of computer science undergraduate enrollments*. National Academies Press (cit. on p. 24).

Pandit, Naresh R. (1996). *The creation of theory: a practical examination of the grounded theory method*. Manchester Business School (cit. on p. 38).

Poo-Caamaño, Germán et al. (2017). “Herding cats in a FOSS ecosystem: a tale of communication and coordination for release management.” In: *Journal of Internet Services and Applications* 8.1, p. 12 (cit. on pp. 9, 13).

Raymond, Eric (1999). “The cathedral and the bazaar.” In: *Knowledge, Technology & Policy* 12.3, pp. 23–49 (cit. on pp. 3, 4).

Ryan, Richard M. and Edward L. Deci (2000). “Intrinsic and extrinsic motivations: Classic definitions and new directions.” In: *Contemporary educational psychology* 25.1, pp. 54–67 (cit. on pp. 9, 10).

Sadowski, Caitlin et al. (2018). “Modern code review: a case study at google.” In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, pp. 181–190 (cit. on p. 84).

Scacchi, Walt (2002). “Understanding the requirements for developing open source software systems.” In: *IEE Proceedings - Software* 149.1, pp. 24–39 (cit. on pp. 6, 27, 34, 81).

Scacchi, Walt (2005). “Socio-technical interaction networks in free/open source software development processes.” In: *Software process modeling*. Springer, pp. 1–27 (cit. on pp. 6, 81).

Scacchi, Walt (2010). “Collaboration practices and affordances in free/open source software development.” In: *Collaborative software engineering*. Springer, pp. 307–327 (cit. on pp. 1, 84).

Schranz, Thomas et al. (2019). “Contributors’ Impact on a FOSS Project’s Quality.” In: *Proceedings of the 2Nd ACM SIGSOFT International Workshop on Software Qualities and Their Dependencies*. SQUADE 2019. Tallinn, Estonia: ACM, pp. 35–38 (cit. on p. 71).

Seaman, Carolyn B. (1999). “Qualitative methods in empirical studies of software engineering.” In: *IEEE Transactions on software engineering* 25.4, pp. 557–572 (cit. on pp. 38, 42, 72, 93).

Silva, Jefferson et al. (2017). “How long and how much: What to expect from Summer of Code participants?” In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, pp. 69–79 (cit. on pp. 12, 85).

Slack (2019). *Removal of remote screen control in Slack Calls*. URL: https://get.slack.help/hc/en-%20us/articles/360022908874-Removal-of-remote-screen-control-in-Slack-Calls (visited on 23 August 2019) (cit. on p. 64).

Slany, Wolfgang (2012). “A mobile visual programming system for Android smartphones and tablets.” In: *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, pp. 265–266 (cit. on pp. 22, 24).

Stallman, Richard (2002). *Free software, free society: Selected essays of Richard M. Stallman*. Lulu. com (cit. on pp. 3, 4).

Steinmacher, Igor, Tayana Conte, Marco Aurélio Gerosa, et al. (2015). “Social barriers faced by newcomers placing their first contribution in open source software projects.” In: *Proceedings of the 18th ACM conference on Computer supported cooperative work & social computing*. ACM, pp. 1379–1392 (cit. on pp. 1, 12, 81, 82, 85).

Steinmacher, Igor, Tayana Conte, Christoph Treude, et al. (2016). “Overcoming open source project entry barriers with a portal for newcomers.” In: *Proceedings of the 38th International Conference on Software Engineering*. ACM, pp. 273–284 (cit. on p. 34).

Steinmacher, Igor, Marco Aurelio Graciotto Silva, et al. (2015). “A systematic literature review on the barriers faced by newcomers to open source software projects.” In: *Information and Software Technology* 59, pp. 67–85 (cit. on p. 13).

Steinmacher, Igor, Igor Scaliante Wiese, and Marco Aurélio Gerosa (2012). “Recommending mentors to software project newcomers.” In: *2012 Third*

International Workshop on Recommendation Systems for Software Engineering (RSSE). IEEE, pp. 63–67 (cit. on pp. 2, 10).

Strauss, Anselm and Juliet Corbin (1990). *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Sage Publications (cit. on pp. 37, 38, 94).

Sugimori, Y. et al. (1977). "Toyota production system and kanban system materialization of just-in-time and respect-for-human system." In: *The international journal of production research* 15.6, pp. 553–564 (cit. on p. 19).

Syeed, M.M. Mahbubul and Imed Hammouda (2013). "Socio-technical congruence in OSS projects: Exploring Conway's law in FreeBSD." In: *IFIP International Conference on Open Source Systems*. Springer, pp. 109–126 (cit. on pp. 1, 83).

Tourani, Parastou, Bram Adams, and Alexander Serebrenik (2017). "Code of conduct in open source projects." In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, pp. 24–33 (cit. on p. 91).

Trainer, Erik, Chalalai Chaihirunkarn, and James D. Herbsleb (2013). "The Big Effects of Short-term Efforts: Mentorship and Code Integration in Open Source Scientific Software." In: *Journal of Open Research Software* (cit. on pp. 1, 12, 82).

Trainer, Erik, Chalalai Chaihirunkarn, Arun Kalyanasundaram, et al. (2014). "Community code engagements: summer of code & hackathons for community building in scientific software." In: *Proceedings of the 18th International Conference on Supporting Group Work*. ACM, pp. 111–121 (cit. on pp. 12, 85).

Turner III, Daniel W. (2010). "Qualitative interview design: A practical guide for novice investigators." In: *The qualitative report* 15.3, pp. 754–760 (cit. on pp. 39, 72).

Wang, Jianguo and Anita Sarma (2011). "Which bug should I fix: helping new developers onboard a new project." In: *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, pp. 76–79 (cit. on pp. 82, 86).

Williams, Laurie et al. (2000). "Strengthening the case for pair programming." In: *IEEE software* 17.4, pp. 19–25 (cit. on pp. 18, 33).

Yamakami, Toshihiko (2011). "The Third Generation of OSS: A Three-Stage Evolution from Gift to Commerce-Economy." In: *Open Source Systems: Grounding Research*. Ed. by Scott A. Hissam et al. Berlin, Heidelberg:

Springer Berlin Heidelberg, pp. 368–378. ISBN: 978-3-642-24418-6 (cit. on p. 95).

Yamauchi, Yutaka et al. (2000). "Collaboration with Lean Media: how open-source software succeeds." In: *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. ACM, pp. 329–338 (cit. on pp. 9, 10).

Ye, Yunwen and Kouichi Kishida (2003). "Toward an understanding of the motivation Open Source Software developers." In: *Proceedings of the 25th international conference on software engineering*. IEEE Computer Society, pp. 419–429 (cit. on pp. 5–7, 10, 25, 27, 84).

Yin, Robert K. (1984). *Case Study Research: Design and Methods*. Applied social research methods series. Sage Publications, Beverly Hills, London, New Delhi (cit. on pp. 37, 38, 72, 94).

Yin, Robert K. (2017). *Case Study Research and Applications: Design and Methods*. SAGE Publications, Inc. ISBN: 1506336167 (cit. on pp. 67, 93, 94).

Appendix A.

List of abbreviations

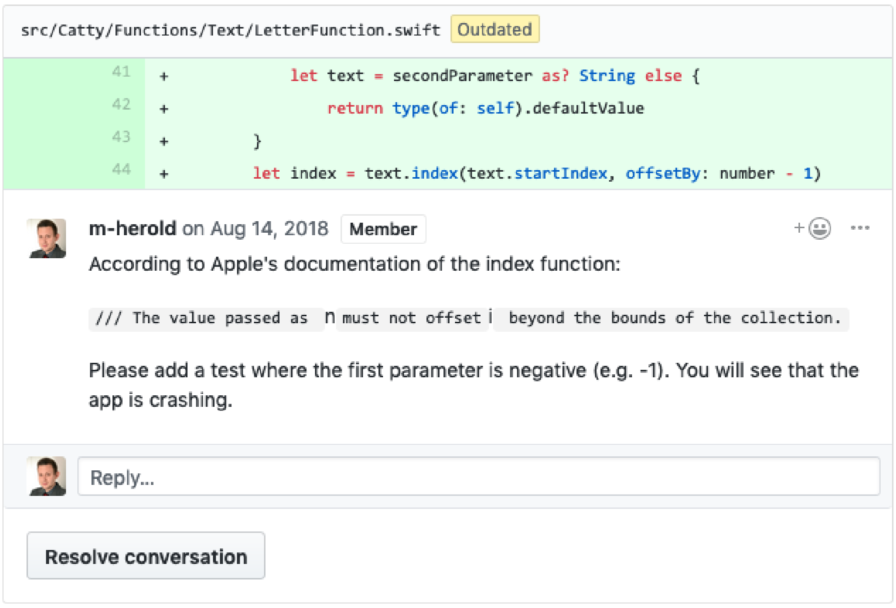
ASD	Adaptive software development
CoC	Code of conduct
CVS	Concurrent versions system
DM	Direct message
FDD	Feature-driven development
FS	Free software
FSF	Free Software Foundation
FOSS	Free and open source software
GSoC	Google Summer of Code
IDE	Integrated development environment
IRC	Internet Relay Chat
JIT	Just-in-time
LD	Lean software development
MRT	Media richness theory
OSI	Open Source Initiative
OSS	Open source software
PO	Product owner

PR	Pull request
WIP	Work in progress
XP	Extreme programming

Appendix B.

Change request on GitHub

Figure B.1 illustrates an example of a change request made to a submission on GitHub[1] using GitHub's code review functionality.



src/Catty/Functions/Text/LetterFunction.swift Outdated

```
+            let text = secondParameter as? String else {
+                return type(of: self).defaultValue
+        }
+        let index = text.index(text.startIndex, offsetBy: number - 1)
```

m-herold on Aug 14, 2018 Member

According to Apple's documentation of the index function:

```
/// The value passed as n must not offset i beyond the bounds of the collection.
```

Please add a test where the first parameter is negative (e.g. -1). You will see that the app is crashing.

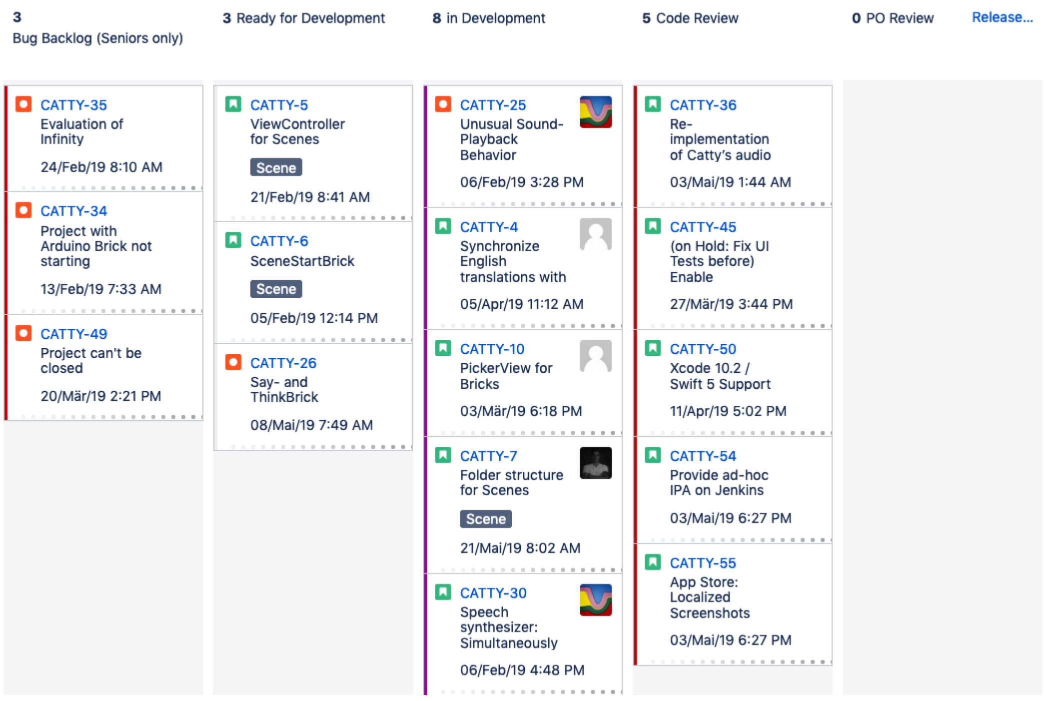
Figure B.1.: An example of a change request on GitHub

[1]https://github.com/Catrobat/Catty/pull/1001, visited on 8 August 2019

Appendix C.

Kanban board

Figure C.1 represents a digital kanban board as visualized by Catrobat's Jira[1].



3
Bug Backlog (Seniors only)

CATTY-35
Evaluation of Infinity
24/Feb/19 8:10 AM

CATTY-34
Project with Arduino Brick not starting
13/Feb/19 7:33 AM

CATTY-49
Project can't be closed
20/Mär/19 2:21 PM

3 Ready for Development

CATTY-5
ViewController for Scenes
Scene
21/Feb/19 8:41 AM

CATTY-6
SceneStartBrick
Scene
05/Feb/19 12:14 PM

CATTY-26
Say- and ThinkBrick
08/Mai/19 7:49 AM

8 in Development

CATTY-25
Unusual Sound-Playback Behavior
06/Feb/19 3:28 PM

CATTY-4
Synchronize English translations with
05/Apr/19 11:12 AM

CATTY-10
PickerView for Bricks
03/Mär/19 6:18 PM

CATTY-7
Folder structure for Scenes
Scene
21/Mai/19 8:02 AM

CATTY-30
Speech synthesizer: Simultaneously
06/Feb/19 4:48 PM

5 Code Review

CATTY-36
Re-implementation of Catty's audio
03/Mai/19 1:44 AM

CATTY-45
(on Hold: Fix UI Tests before) Enable
27/Mär/19 3:44 PM

CATTY-50
Xcode 10.2 / Swift 5 Support
11/Apr/19 5:02 PM

CATTY-54
Provide ad-hoc IPA on Jenkins
03/Mai/19 6:27 PM

CATTY-55
App Store: Localized Screenshots
03/Mai/19 6:27 PM

0 PO Review

Release...

Figure C.1.: An example of a kanban board

[1]https://jira.catrob.at/secure/RapidBoard.jspa?rapidView=127, visited on 21 June 2019

Appendix D.

Jira workflow

The Jira workflow employed in all sub-projects of the Catrobat foundation is illustrated in Figure D.1 (as of 8 August 2019).

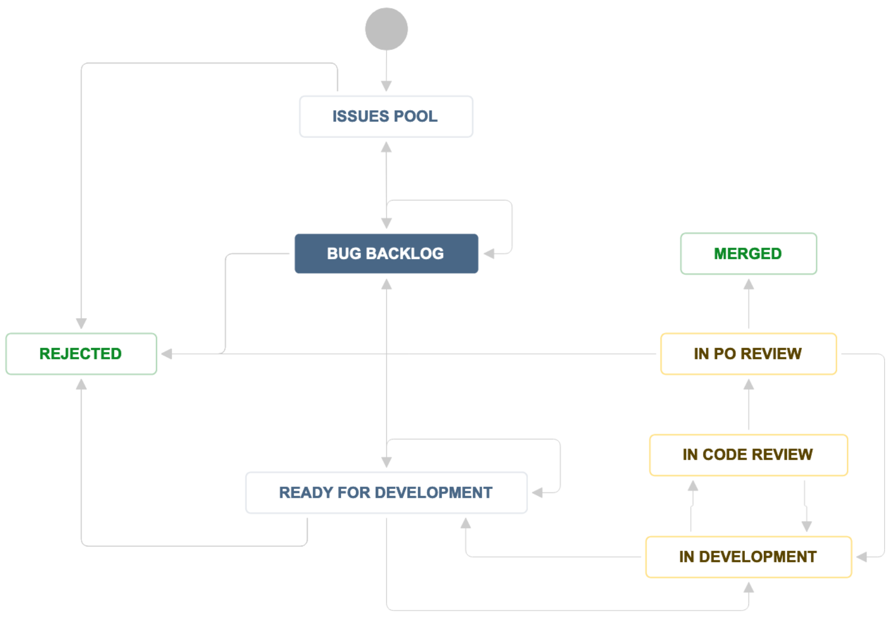
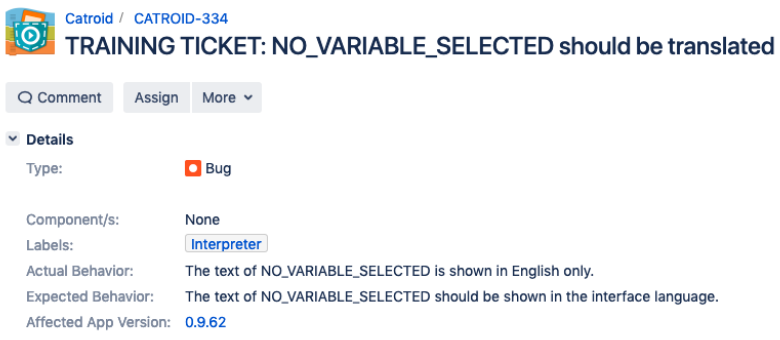


Figure D.1.: The Jira workflow at Catrobat

Appendix E.

Training ticket

Figure E.1 shows an example for a training ticket in Jira[1].



Catroid / CATROID-334

TRAINING TICKET: NO_VARIABLE_SELECTED should be translated

Details

Type:	Bug
Component/s:	None
Labels:	Interpreter
Actual Behavior:	The text of NO_VARIABLE_SELECTED is shown in English only.
Expected Behavior:	The text of NO_VARIABLE_SELECTED should be shown in the interface language.
Affected App Version:	0.9.62

Figure E.1.: An example of a training ticket in Jira

[1]https://jira.catrob.at/browse/CATROID-334, visited on 8 August 2019)

Appendix F.

Interview guideline

This section illustrates the original version of the interview guideline in German.

Interview Leitfaden

Communication in an Agile FOSS Project: A Socio-Technical Case Study

---

1. Einführung

Vorstellung des Forschungsthemas und Interviewablaufs, 1 Minute (Interviewer)

- Danke für die Bereitschaft zum Interview
- Einführung in das Forschungsthema
    - Zusammenarbeit und Kommunikation mit “neuen Contributors”
    - Anfängliche Probleme
        - welche Fragen stellen sie, welche Probleme gibt es?
- Erklärung des Interviewablaufs
    - Zeitlicher Rahmen (ca. 45 Minuten)
    - Erläuterung warum der Interviewleitfaden notwendig ist
    - Offene Antworten wünschenswert
    - Zwischenfragen können jederzeit gestellt werden
    - Es sind auch kritische Aussagen möglich und wünschenswert
- Anonymität zusichern
- Erlaubnis für Aufnahme erbeten

[Aufnahmegerät starten]

---

2. Allgemeine Einstiegsfragen

Einstiegsfragen zum “Aufwärmen”, 2 Minuten

- Können Sie zum Einstieg schildern, was Ihre Aufgabe bei Catrobat ist oder war?

---

3. Betreuung von Contributors

Relevanz für das Forschungsthema, 4 Minuten

- Haben Sie in den letzten Jahren Newcomers in das Projekt eingeführt bzw. diese bei ihrer “Contribution” betreut - wie es beispielsweise bei Programmen wie dem “Google Summer of Code” verlangt wird?

- Wie würden Sie deren Erfahrung - vor Ihrer Betreuung - im Hinblick auf
    - …mobile App Entwicklung einschätzen?
    - …visuelle Programmiersprachen, insbesondere Block-basierende Programmiersprachen einschätzen?
- Waren die Programmiersprachen Scratch oder Catrobat bereits bekannt?

---

4. Kommunikation

Informationsaustausch und Zusammenarbeit, 4 Minuten

- Wie oft haben Sie mit den Contributors kommuniziert?
- Wie bzw. über welche Kanäle haben Sie kommuniziert?
- Über welche Themen wurden hauptsächlich kommuniziert?

---

5. Probleme mit der Kommunikation

Synchron vs. asynchron, Response time, 6 Minuten

- Sind hinsichtlich der Kommunikation irgendwelche Probleme aufgetreten?
- Hat es einen Unterschied zwischen verschiedenen Kommunikationsformen gegeben?
    - *beispielsweise Voice-Chats und Desktop-sharing im Vergleich zu Text-Chats*
- Haben Sie den Contributor jemals persönlich getroffen?
    - *falls “Ja” -> GO TO 5a*
    - *falls “Nein” -> GO TO 6*

5a. Vorort vs. Remote

Unterschiede, Vor- und Nachteile, 3 Minuten

- Wenn Sie sich an die Zusammenarbeit zurückerinnern, hat es Unterschiede in der Zusammenarbeit und Kommunikation vor Ort und über das Internet gegeben?

---

6. Anfängliche Probleme

Probleme beim Projekteinstieg, 6 Minuten

- Können Sie einige Hürden nennen, die es den Neulingen besonders schwer gemacht haben in das Projekt einzusteigen?
- Können Sie sich noch an den ersten Pull-Request erinnern?

    - Hat es sich hier um einen einfachen Bugfix gehandelt oder war das bereits eine Implementierung eines neuen Features?
- Wie ist der Contributor mit den vorhandenen Dokumentationen zurechtgekommen?

---

7. Probleme während der Contribution

Probleme während der Implementierung, Change Requests, Code-Base, 12 Minuten

- Gab es Probleme, die wiederkehrend aufgetreten sind?
- Gab es Situationen in denen der Newcomer nicht mehr selbstständig weiter arbeiten konnte und auf Hilfe von der Community angewiesen war?
- Wurden PRs in der Regel sofort akzeptiert oder waren mehrere Change-Requests notwendig?
    - Falls “Ja”: worüber ging es hauptsächlich in diesen Change Requests?
- Inwiefern haben sich technische Probleme auf die Motivation ausgewirkt?

---

8. Community bonding

Soziale Ebene, 3 Minuten

- Wäre es Ihnen aufgefallen, dass der Contributor auch mit anderen Mitgliedern der Catrobat Community in Kontakt getreten ist?
- Ist der Contributor nach wie vor aktiv tätig?
    - falls “Nein”: wie lange war er nach dem Program (bspw. GSoC) noch aktiv?

Zusatzfragen (optional)

- Was wurde gemacht währenddessen der Contributor auf Hilfe der Community gewartet hat und selbstständig nicht mehr weiterarbeiten konnte?
- Wurde parallel an mehreren Tickets gearbeitet?
- Haben Sie Remote-Desktop Sessions oder Pair-Programming betrieben?
- Gab es irgendwelche gröbere Probleme, die eventuell sogar zum Ausschluss aus der Community führten?
- Hatte es Probleme mit der Verständlichkeit der Code-Base gegeben?
    - Falls “Ja”: inwiefern hat sich das auf die Motivation ausgewirkt?
- Wurde die Kommunikation eher von Ihnen oder dem Contributor initiiert?
- Gab es kulturelle Unterschiede, die einen Einfluss auf die Kommunikation bzw. Zusammenarbeit im Allgemeinen hatten?

Appendix G.

Media richness theory

Figure G.1 depicts the effectiveness and richness of different communication media, as suggested by Daft and Lengel, 1986.

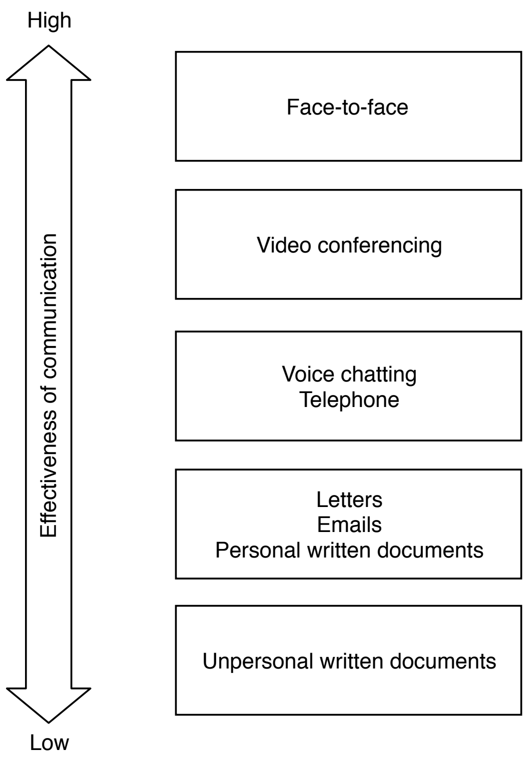

High
Effectiveness of communication
Low
Face-to-face
Video conferencing
Voice chatting
Telephone
Letters
Emails
Personal written documents
Unpersonal written documents

Figure G.1.: The richness of different communication media (Daft and Lengel, 1986)