Requirements Management in an Open Source Software Project – Empirical Case Study

Petri Mäkinen

MSc Thesis
University of Helsinki
Department of Computer Science

Helsinki 24.9.2017

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta – Fakultet –Faculty	Laitos – Institution – Department
Faculty of Science	Department of Computer Science

Tekijä – Författare – Author

Petri Mäkinen

Työn nimi – Arbetets titel – Title

Requirements Management in an Open Source Software Project – Empirical Case Study

Oppiaine – Läroämne – Subject

Computer Science

Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
Pro gradu	24.09.2017	77 pages

Tiivistelmä – Referat – Abstract

In the past few decades there has been increasing interest towards open innovation both among academia and in businesses. Especially software intensive companies face rapid technological change, which forces them to seek new sources of innovation. Companies can do this by using various open innovation approaches where they can share their knowledge and resources or utilize the knowledge and resources of outsiders ranging from other companies to individual developers.

Open source software (OSS) is a blooming open innovation strategy used by a growing number of companies. OSS communities can have a large number of members, making the requirements management process challenging. This thesis aims to build an understanding of the requirements management process in a company that is leading an actively developed open source project. The studied OSS community doesn't only have individual developers, but many companies participate in it as well.

The thesis explores first open innovation, open source software development, and requirements engineering with means of a literature review. The goal of the literature review is to investigate open innovation and requirements management in OSS context. The literature review also provides a theoretical background for studying the open innovation and the requirements management process in the case company called Qt Company.

The Qt Company leads an OSS project, which is the subject of this study. The project's requirements management process was studied with the help of several information sources. These included interviews of Qt's employees. The interviews were conducted in a research project called OpenReq. Additional information was gathered from the company's websites and the project's requirements management system. To verify the results, we first studied a few issues from the requirements management system to see whether they followed our conceptual model built based on the interviews, and illustrated as a swimlane diagram. Finally, we had a follow up interview with an employee from Qt Company (Qt's community manager) to verify the results, and to correct any inaccuracies or misunderstandings.

We found Qt Company using both inbound and outbound open innovation approaches in the studied OSS project. The project, and the community around it has many similarities to the OSS community descriptions found in the literature. For example, as often in OSS, the requirements in the studied project are unstructured. The requirements management process in the studied OSS project was found to include 12 different stakeholders. Also a diagram summarizing the whole requirements management process is constructed based on the interviews and presented at the results section of the thesis.

Avainsanat – Nyckelord – Keywords

Open innovation, requirements management, open source software

Säilytyspaikka – Förvaringställe – Where deposited

Muita tietoja – Övriga uppgifter – Additional information

Table of Contents

1 Introduction 1
1.1 Research Questions 1
1.2 Scope 2
1.3 Research Methodology 3
1.4 Thesis Structure 3
2 Related Literature 5
2.1 Innovation 7
2.1.1 Definition of Innovation 7
2.1.2 Open vs Closed Innovation 8
2.2 Open Innovation Strategies 11
2.2.1 Outside-in (inbound) 12
2.2.2 Inside-out (outbound) 13
2.2.3 Coupled Process 14
2.2.4 Open Source Software 15
2.3 Online Communities as Platforms for Open Innovation 20
2.3.1 Community Roles 21
2.3.2 Competition and Cooperation 25
2.4 Requirements Engineering for Open Source Software 27
2.5 Summary of Findings from the Literature 31
3 Case Study: Qt Software's Requirements Management Process 33
3.1 Case Study Design 33
3.2 The Qt Company 35
3.3 Context of the Requirements Management Process 36
3.4 Roles of Individuals in Qt Software's Open Source Community 38
4 Empirical Results 41
4.1 Stakeholders and their actions 42
4.2 Workflow of a Requirement 49
4.3 Sample Issues 54
5 Discussion 59
5.1 Implications of the Empirical Results 62
5.2 Comparison of the Theoretical and Empirical Findings 64
6 Conclusion 68
References 72
Appendices 76

1 Introduction

During the few past decades the Internet, proliferation of personal computers, and other technological advancements have changed the competitive position for many companies. Facing the ever-increasing competition, organizations are looking for new ways to produce novel products and services that would satisfy the needs and demands of their customers. Thus, it is not a surprise that open innovation has gotten a growing attention both in the academic world as well as among businesses. Open innovation refers both to looking for ideas and other inputs from outside of the company, and to increasing the knowledge outflows from the company to the marketplace to gain advantages over competitors.

One approach to open innovation is open source software, which is produced in sometimes big online communities. While OSS can be produced in a closed community, often the communities are open for anyone to join. Because of this, there can be many community members from various backgrounds. Some of the community members may use the software for their hobby, while others may work for a company using the OSS software as a critical component in their business. It seems that in this kind of an environment requirements management, for example, the tasks of understanding and prioritizing the requirements can be challenging. In this thesis, we will first investigate open innovation and requirements management in OSS context through the literature, and then investigate those, in the case company called The Qt Company, focusing especially in the requirements management.

1.1 Research Questions

As mentioned above, open innovation and requirements management are central themes in this thesis, and both a literature review and an empirical case study are used to answer research questions about them. The case company for this thesis is The Qt Company, which produces an open source cross-platform application framework in collaboration with a lively open source community. This framework is also called Qt, or Qt software in this study to make a clear distinction to the company name.

The Qt Company has customers with diverse needs from using Qt software to build user interfaces in medical devices (https://www.qt.io/built-qt-medec-medical-devices/), to

building cross platform marine navigation and information systems (https://www.qt.io/case-navico/), making it a good subject to study the requirements management process in an open innovation context.

The first goal in this study is to review the literature to learn what does open innovation mean, and how it can be practiced in organizations developing open source software. We will also investigate requirements engineering, and how it differs in OSS projects and more traditional software projects. Our first research question and its sub questions are as follows:

- RQ1: What is open innovation with OSS based on literature?
    a) What kinds of approaches can be used for implementing open innovation?
    b) How can the OSS development model be used as an open innovation strategy?
    c) How is requirements engineering different in OSS compared to other software projects?

After we have described how open innovation, OSS and requirements engineering are described in the literature, we will report a case study, which is aimed at answering the following research questions and their sub questions.

- RQ2: What open innovation approaches are used in the Qt OSS project?
- RQ3: What is the requirements management process in the Qt OSS project like?
    a) Who are the stakeholders/actors?
    b) What actions do they perform?
    c) What is the flow of the requirements management process, i.e. in what order do the actions occur?

Question RQ3 is divided in three parts. The goal is to first find the actors and their actions of Qt software's requirements management process. When the actors have been identified, we will study the process as a whole and visualize it in a diagram.

1.2 Scope

While Qt software's requirements management process is the main subject under study in this thesis, we are focusing on a specific part of it. The Qt Company leads an open source project to develop its application development framework, and the requirements

for it are handled in a public requirements management system called Jira. However, The Qt Company has also private projects for some of its customers. Viewing and interacting with the requirements in the private projects requires special permissions, and those requirements are thus not visible to the open source community. In this study, we exclude these requirements from our analysis, and focus on the requirements visible for everyone.

Also the public side in Jira has several projects. There is, for example, a project for The Qt Company's website, another for a framework for creating cross-platform installers, and so on. In this thesis, we focus on open source software called Qt, which is a cross-platform application development framework.

Furthermore, we are mostly focusing on requirements that materialize into individual tasks (also called issues) in the public requirement management system. Thus, we are not analyzing the bigger, strategic requirements of the business, which are handled by different people in different systems. Finally, we restrict our analysis on the main issue types that are used in the system, namely bugs and suggestions.

The issue type can be chosen by the creator of the issue. Suggestions usually describe new features or functionality, and bugs existing features that are not working in some situation. In the Qt software's requirements management process, the same process is followed to manage both suggestions and bugs.

1.3 Research Methodology

The research methods in this study are a literature review and a descriptive case study. A literature review was conducted to gain understanding about open innovation and requirements engineering practices in OSS context.

These same issues were then studied with means of a case study where the Qt software's requirements management process was studied. Information for the case study was collected from interviews with The Qt Company's employees as well as from publicly available online sources, e.g. The Qt company's website.

1.4 Thesis Structure

The rest of the thesis is structured as follows. In section 2, open innovation, open source software and requirements management are discussed with help of related literature both to find answers to RQ1, and to provide theoretical background and context for the

empirical part of the thesis.

In section 3, the case company and the case study's design are described. We will also discuss a higher-level context of Qt software's requirements management process, and the roles in the Qt software's open source community in this section. Thus, section 3 provides both background for the following section as well as some answers to RQ2.

The main results of the case study answering RQ3, are reported in section 4. We first describe the actors of the requirements management process along with their actions, and then present the process' flow. Section 5 provides discussion about the results, and finally section 6 concludes the thesis.

2 Related Literature

In this section, we will investigate literature related to open innovation, open source software, online communities, and requirements management. The goal of the literature review is to answer research question RQ1. As the case company and the case OSS project are also examples of open innovation, the literature review will also help us to understand what kind of issues the case company may be facing with its OSS project.

Articles having relevant information with regards of this study's research questions were searched in online databases, for example, Google Scholar and Scopus, and filtered first by their title, abstract and year of publication. If the article was published in the past 15 years, i.e. earliest on 2002, and if the title and/or abstract of an article seemed promising, the rest of the article was read and included if it really had relevant information for this study. When relevant articles were found, more articles were identified by forward and backward snowballing. Figure 1 shows how the number of studies about open innovation has been growing in Scopus database over the past twenty years.

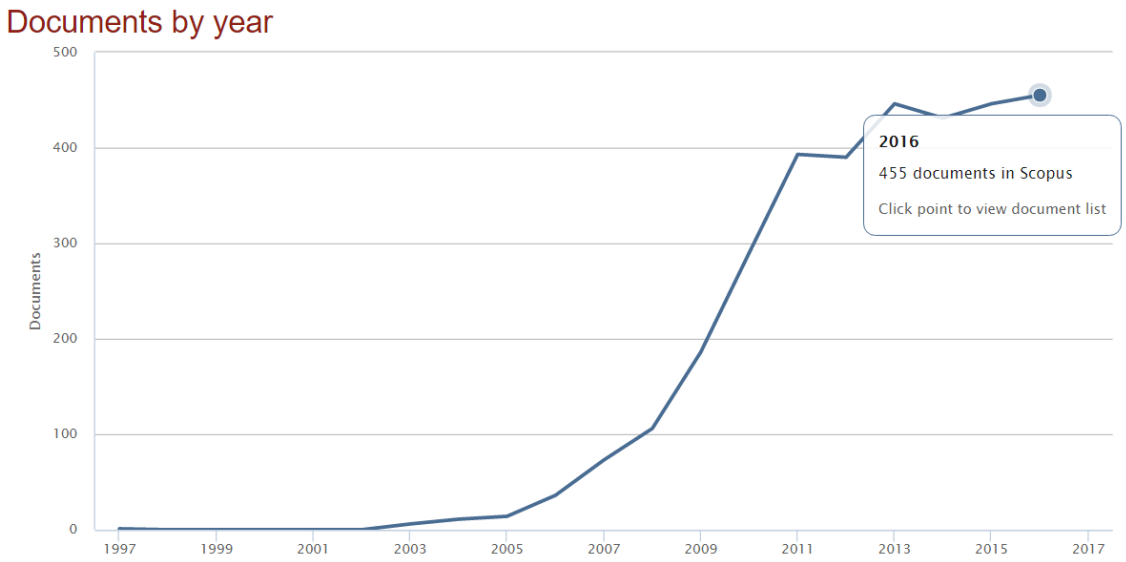

Figure 1 Appearance of open innovation research over time [1]

As can be seen from Figure 1, research about open innovation has been increasing almost every year since early 2000s. The web service scopus.com has 3371 documents (articles, conference papers, book chapters, etc.) having the term open innovation in their title, abstract or keywords. Figure 1 was produced with scopus' analyze functionality after searching with a search string TITLE-ABS-KEY("open innovation"), i.e. searching for

documents having the term “open innovation” in their title, abstract or keywords.

As we can see, the amount of research has surged after 2003, when now-famous Chesbrough’s article about open innovation was published. Furthermore, according to the number of articles in Scopus, interest in the subject seems to be still increasing, although at a decreasing rate since 2011. Scopus has 455 open innovation documents published in 2016, whereas in 2003 there were only 6 (3 of which were authored by Chesbrough). Open innovation had existed in some form a long time before 2003, Chesbrough’s works about open innovation in early 2000 gave a label to a bunch of activities that organizations were already doing [2].

The interest is not purely academic either, both companies and even nations and regions are paying growing attention to open innovation activities. One prominent example of a large initiative in the industry is High Tech Campus Eindhoven (www.hightechcampus.com) in the Netherlands, where around 10 000 researchers, developers and entrepreneurs work to develop new technologies and products [3]. According to the campus’ website [4] the companies at the campus are responsible for almost 40% of the Dutch patent applications. At the campus, many companies including Philips, IBM, NXP and Intel, share their skills, knowledge and R&D facilities to be more innovative [4]. Van de Vrande et al. (2009) also found in their survey of manufacturing and service SMEs that they were increasing their open innovation activities [5], however this study’s sample only included companies who were systematically innovating.

With regards to the interest on a regional level, for example the European Commission recognizes the importance of open innovation, and has done many publications about the subject. Even though EU is the biggest producer of scientific knowledge in the world, often this scientific knowledge is not realized as an application in the market, and the technologies developed in Europe are commercialized elsewhere [3]. European Commission actually has an open innovation policy consisting of three pillars: reforming the regulatory environment, boosting private investment in research & innovation, and maximizing impacts. The two first ones are quite self-explanatory, and the last one – maximizing impacts – refers to making it easier for projects to access funding from different sources and being able to identify the best innovations that should get funding [3].

Benefits from open innovation have been reported for example by Procter and Gamble

which was able to make their R&D more efficient by 60% with an open innovation approach while also increasing the success rate of the products by 50% [6]. One major benefit of moving to more open innovation practices, where the users' or customers' ideas and views are utilized, is that it helps the organization to better understand its customers. Often the new products developed by manufacturers are commercial failures, and according to research it seems to be because the manufacturers don't understand their users [7]. On the other hand, if the organization is too open, there is a risk that it can lose control or some of its core competences [6].

In the remaining of this section, we will first discuss innovation, and its definition, and how open innovation differs from the way organizations have traditionally innovated. We will also talk about open innovation strategies including open source software. Finally, we will conclude the literature review with a discussion about online communities and requirements management.

2.1 Innovation

The goal of this section is to understand what innovation and open innovation are. We will first discuss the definition of innovation in itself, and the move on to describing open innovation.

2.1.1 Definition of Innovation

Innovation is a word used often in everyday contexts, but it is also a word having a plethora of different definitions depending on the person defining it, and in the context the word is used in. Also in the open innovation literature, there is disagreement or perhaps even confusion of the meaning [8]. In 2009 Baregheh et al. stated that there are several definitions for innovation, from varying perspectives of different disciplines, but there is no clear, authoritative definition for it [9]. The study by Baregheh et al. was an extensive literature review where they found more than 60 definitions from multiple disciplines. They did a content analysis on the found definitions to come up with the first cross-disciplinary definition of innovation [9]. The definition they came up with was:

"*Innovation is the multi-stage process whereby organizations transform ideas into new/improved products, service or processes, in order to advance, compete and differentiate themselves successfully in their marketplace."* [9]

Creating new or improved products might be the first thing to come into mind when thinking about innovation, but innovation can as well be about processes, which is highlighted by the definition. In the field of software development, the new processes can be related, for example, to the software development process. The expression "multi-stage process" in the definition, refers to innovation not being a discrete act, but a continuous process. Finally, the definition shows the objective most organizations have for innovating: they want to become better and more competitive against their competitors.

In the innovation process, first an idea or invention is created, and then it is commercialized [10], so compared to invention, the term innovation has a business aspect embedded in it. Innovations can be classified in many ways. Often, they are classified with regards to how disruptive they are, i.e. whether the innovation is radical or incremental. However, in this study we are more interested in the distinction between open and closed innovation, which will be discussed more in the following subsection.

2.1.2 Open vs Closed Innovation

The open innovation concept was introduced by Chesbrough [11, 12]. One main difference between Chesbrough's conceptualization of open innovation and the earlier literature about the topic was that Chesbrough's definition requires the innovations to be aligned with the company's business model [8].

Often open innovation is contrasted to "closed innovation" where the innovations are created inside an organization by internal R&D effort. In the closed - or vertically integrated (e.g. [10]) - innovation model, companies use various methods including patents and copyrights to prevent others from copying their innovations [7]. While open and closed innovation are contrasted with each other, there is an agreement in the literature that the choice between open and closed innovation is not a binary one, but rather a continuum [13].

As already mentioned, innovations can be both about products and services as well as about the processes of producing and selling the products and services. Open innovation might be more suitable for product/service innovations. While there are several process innovations that have been developed with knowledge originally from outside of the organization, process innovations require more knowledge about many kinds of internal issues in the organization, and this knowledge might not be available outside the organization [2].

Most companies' innovation processes have always used external resources [13, 2], so integrating customers or suppliers in the innovation process is not a new idea [14, 2]. However, innovation has been mostly manufacturer centric [7]. Big corporations have innovated in large R&D labs adding to potential competitors' entry barriers [5].

Recently however, as we saw earlier, the academic interest towards open innovation has been increasing. Also in practice, it has become necessary to open up the innovation processes [2]. For example, market institutions including venture capital, intellectual property rights, and technology standards have improved, helping organizations to exchange ideas [13]. New technologies have also created new ways of collaboration and coordination for people and organizations in different geographical locations [13].

In many industries, the competition is tougher than ever. In recent decades, computers, Internet, smart phones and other digital devices have become available to the mainstream population, and various kinds of software have become increasingly important for organizations in their daily processes. Also, the increased mobility of the workforce, shortened product life cycles, widely available venture capital and highly distributed knowledge increase the importance of open innovation [5]. Due to these developments, organizations are nowadays able to innovate faster and more efficiently, but they also must do so to stay ahead of competition.

While open innovation has gotten increased attention lately, it does not mean that all companies would have an open innovation process. Van de Vrande et al. studied open innovation in manufacturing and service SMEs 2009, and found that the medium sized companies (100 - 499 employees) were more likely to engage in open innovation activities than their smaller counterparts (10 – 99 employees) [5].

Open innovation is a broad concept, but "The core research questions in open innovation research are how and when firms can commercialize the innovations of others and commercialize their valuable innovations through others" [10]. Commercializing innovations of others is often called inbound innovation, and commercializing one's innovations through others is called outbound innovation. Both of these open innovation strategies will be discussed further in the following subsection.

Many different perspectives to open innovation can be recognized from the research, for example, knowledge sourcing, crowd sourcing, distributed problem solving, inter-organizational alliances, licensing agreements, collaboration with and within

communities, and crowds or networks of individuals [12].

In the context of software companies, open source software is an often cited, and studied example of open innovation. Open source software is software that has been published under an open source license, i.e. license that allows anyone to view, use, and modify the software's source code [15, 7].

Whether the software is closed or open source, there are significant differences between the software industry and many of the more traditional industries. For a software vendor, most of the costs of new products come from shipping the first copy [16]. The costs of producing more once created software, i.e. duplicating it, are almost zero as are the costs of transporting the software [17, 16], which has followed from the advancements in ICT developments [17]. Information products, like software, can be developed and diffused by user communities without needing the manufacturer, while in industries where physical products are produced, the manufacturers are often the only ones producing in large enough scale to make the production and shipping profitable [7].

In addition to the low transportation and duplication costs of software, producing the first copy is probably also significantly cheaper than before. Within the few past decades, personal computers have become commodity items that a big proportion of the population can afford to have, especially in countries where the standard of living is relatively high. The performance of computers has also increased so much that even an inexpensive laptop can be used to develop almost any kind of software.

There also exist nowadays many APIs (Application Programming Interface), and other readymade frameworks and libraries that can cut down the development time of a software. The APIs, frameworks and libraries can be thought of as building blocks of software, designed to do common tasks in a generalized way, so that they can be easily used in various applications. The tasks performed by these building blocks can be for example saving data into a database or drawing nice looking graphs.

Because of the Internet, and search engines like Google, finding documentation of the APIs, frameworks, libraries as well as the programming languages themselves is also easier and faster than before the Internet. There are even developer-focused websites such as Stackoverflow (https://stackoverflow.com/), which make it easier to find existing solutions to commonly encountered problems. All these developments make it easier to produce software, lowering the entry barriers for new entrants in the market, thus making

the competition for the existing players harder.

Considering the increasing popularity of open source software, it is also possible to find a sophisticated ready-made software to use as a base of a new offering. When the license allows it, anyone can enter the market and start selling open source software [16]. The low entry barriers of producing software together with the – close to zero – marginal costs, mean that the competitive position of a software intensive organization is quite different compared to an organization producing physical goods, where the bulk of daily operations and effort goes to producing more units of the existing products. In comparison, in a software company, a big portion of daily activities would go to designing, and implementing new or improved features. We can even say that in case of software, innovation and production are essentially the same thing, since software is an information product [15].

For these reasons, it is increasingly important for organizations to widen their approach from traditional closed innovation process to a more open approach. In the following subsection, we will present different open innovation strategies that an organization can use to have a more open innovation process.

2.2 Open Innovation Strategies

Three core open innovation processes have been identified in the literature. These processes are applicable to companies no matter whether they produce software, physical goods, or something else. The processes, identified by Gassmann and Enkel, are named inside-out, outside-in and coupled process [14]. These three approaches have since been widely referred to in the literature. When a company chooses one of the three processes as its approach, it can still use aspects of the other processes. In fact, Gassmann and Enkel found that the companies they studied had one of the three processes as their primary approach, but were also using elements of the others.

The identified open innovation processes are called inside-out, outside-in and coupled process. In the following subsections, we will discuss inside-out, outside-in and the couple process further. Open source software also presents a profound choice between an open and closed approach for a software intensive company. Therefore, we will here consider it as a separate approach to open innovation, even though software companies (or other companies utilizing software), can also use the core open innovation processes

presented by Gassmann and Enkel.

2.2.1 Outside-in (inbound)

One of the core open innovation processes presented by Gassmann and Enkel was the outside-in, or inbound process. In the outside-in approach the organization is using external ideas, resources and knowledge contributed by various stakeholders — customers or suppliers for example [14]. According to a recent literature review [8], the inbound activities have been studied much more compared to the outbound and coupled activities.

The outside-in process can also be called technology exploration (as in e.g. [5]). The outside-in process can be further divided into two different strategies, *sourcing* and *acquiring* [13]. Sourcing refers to using external resources, like ideas and knowledge, from suppliers, customers, competitors, etc. for the innovation process. In the acquiring process, the external innovation inputs are purchased from the market place [13]. The outside-in process is usually the dominant observed open innovation process [18, 2]. Some examples of inbound open innovation activities include customer involvement, external networking, external participation, outsourcing R&D and inward IP licensing [5].

In the context of open source software, the sourcing process is important. Developers in the community are donating their skills, time and knowledge for the project. For a company leading such a project, the direct benefit of advancing the project is clear. In addition, since the open source developers are usually also users of the software [19, 20], the company can gain significant insights about the needs of its customers. There can also be knowledge inflows related to technical skills from the external developers providing important learning opportunities for the company's employees.

It is only possible for an organization to benefit from external innovations if it manages to identify them, understand them, and combine them with its internal innovation processes, so that it can turn it into a suitable product for itself [21]. A recent review on inbound open innovation [8], reviewed 151 articles published between 2003 and 2010, which were discussing inbound or coupled open innovation process. According to this review, the analyzed studies were often lacking a comprehensive view that would consider the broad view of the open innovation process. Many of the studies focused on obtaining the innovations, ignoring things like integrating and commercializing them,

which are also needed for the organization to actually profit from the innovations.

To fix this oversight West and Bogers [8] developed a model illustrated in Figure 2. It highlights that it is not enough to obtain the innovation, but that it needs to also be integrated and commercialized. Integrating an innovation means here that the organization must have the technical capabilities to assimilate the innovations as well as overcoming any cultural barriers in the organizations [8].

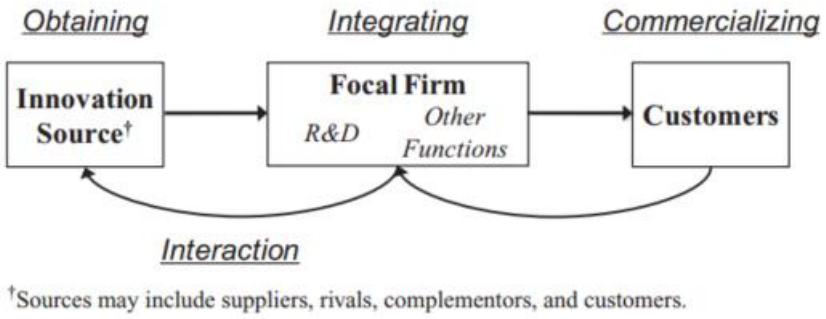

Figure 2 Integrative Model for Leveraging External Sources of Innovation [8]

2.2.2 Inside-out (outbound)

Compared to the outside-in process, inside-out is, in a way the opposite. In this approach ideas are flowing from the organization to outside. Outbound innovation "…refers to external exploitation of internal knowledge" [2], and is also called technology exploitation (e.g. in [5]). The inside-out approach can help an organization to bring its ideas to market faster, set technological standards, and increase its profits [14]. Regarding to standards, also Dahlander and Gann state that "In the literature on standards, for example, it is well-known that being open and focusing less on ownership increases the opportunities to gain interest from other parties." [13].

Like the outside-in, also the inside-out process can be further divided into two different strategies, selling and revealing [13]. Revealing refers to action where an organization is trying to gain some indirect benefits by revealing some of its internal resources. Another option, called selling by Dahlander & Gann, is defined as "*...refers to how firms commercialize their inventions and technologies through selling or licensing out resources developed in other organizations."* [13]. It seems, according to research, that organizations are increasingly licensing out inventions and technologies [13].

One example of the above-mentioned revealing in the OSS context, would be open sourcing a software product previously sold as a closed source offer. An organization

might see this as the best option if they notice that the competition is too far ahead, and the organization seems to be getting a very small piece of the market [16, 22]. On the other hand, the organization(s) leading the market might see it as a threat that the other competitors might open source their products, and decide to respond to the threat by open sourcing their own product first [16].

An organization can make its product open source also in hopes of selling complements. For example, some Linux distributors sell complements in form of training, installation, support services [21] and complementary proprietary products [22]. Dual licensing, where the same product is sold to different target audiences at different prices (e.g. it can be free for hobbyists and cost money for companies), is also an example of the selling complements [21] strategy. For example, MySQL is following the dual license strategy offering a free version for OSS projects, and a paid commercial version [23]. While only a small proportion of the customers (about 1 in a 1000) buy the commercial version, through the high number of users, this means thousands of paid customers [24].

2.2.3 Coupled Process

An organization following primarily the coupled process combines the previous two. With this process, an organization is trying to obtain knowledge from outside (outside-in process) and bring ideas to market (inside-out process) [14]. To successfully do both, the outbound and inbound open innovation organizations join strategic networks where they co-operate with other organizations [14]. It has been found that companies that have an open innovation process, usually do not focus solely on technology exploration (inbound open innovation) or technology exploitation (outbound open innovation), but rather conduct both kinds of activities [5].

One interesting example of the coupled process comes from the video game industry. There companies sometimes publish editing tools that allow the users themselves to create modifications for the game [21, 7]. This strategy is called *donated complements* [21] or *innovation toolkits* [7]. The users can share these game modifications for free online, so they don't directly create revenue for the game developer [21]. However, the company benefits because the modifications extend the game's demand period [21], and because other users cannot play the modification without first buying the actual game [7]. Since the users are continuing the game's success in the market, the company's resources are freed to focus on follow up products [21].

Video game industry is not the only one providing toolkits for users. For example, a company called StataCorp making a proprietary statistics software, allows the users to create new statistical tests for the Stata platform, and includes the most popular tests as modules in the Stata platform [7]. In addition to the professional testing preceding the inclusion of a test, the creator benefits from being publicly credited by the company [7].

Innovation toolkits can help an organization to manage the expectations of the users of the organization's products or services. Often users expect the products, services and solutions they purchase to work right out of the box [7]. However, for customized products the users have created themselves, the expectations are usually lower [7]. A highly visible company using the toolkits is Google. For Android platform, it provides a software development kit (SDK), which anyone can use to make applications for the platform extending its functionality. Also for example, Google sheets allows the user to create scripts that can extend the functionality of the sheets application. While this requires some technical skill, and might not be something an average user would do, Google provides tutorials and documentation, so for someone who already knows how to code, it is relatively easy to get started. Additionally, the users can publish these scripts for other users to see, use and review. In all of these examples, the organization needs to make an initial investment to enable users to create their add-ons, modifications or applications, but once the infrastructure is in place, the organization can mostly enjoy the free labor from their users while at the same time making its users happier with the product.

We have now discussed the three core processes for open innovation. In the next subsection, open source software will be discussed. It can be used in combination with any of the three core processes.

2.2.4 Open Source Software

Software is nowadays everywhere around us in our daily lives. Our laptops, smart TVs, cell phones, cars, etc, all have software written in different languages, produced by different development methodologies, and released under different licenses. Many of these products are produced by companies who want to retain the intellectual property (IP) rights for the software running inside the product. An organization producing software that it owns all the IP rights for, has complete control over the development direction of the product whether it is built by an in-house development team, external

consultants, etc. The source code of the software is normally not revealed to other parties, and even if someone would have access to the code, they would not be allowed to use it because of the license used to publish the software. This type of software can be called *closed source software*. As stated before, open source software is software published under an open source license [15, 7], i.e. a license that allows anyone to view, use, and modify the software's source code. In other words, open source software and closed source software can be thought as being the opposite of each other.

The closed source model seems to make sense. When an organization has invested money to develop a software, it is not directly clear why the organization would decide to release the software's source code as open source software, thus letting the competition to take advantage of the investment. Many notable software products, such as Microsoft Windows are successfully using the closed source approach. However, open source software (OSS), is a feasible alternative as shown by well-known open source products such as Linux and Firefox. Open source software is usually distributed over the Internet, where anyone can download it for free, use it and even inspect and modify the source code [15].

A term closely related to OSS, is FLOSS (Free/Libre and Open Source Software). The word libre, borrowed from French or Spanish, refers to the freedom distinguishing it from free referring to cost of a product as in expression "free beer" [25]. Nowadays OSS is often used to refer to both free, and open source software [26]. According to the Open Source Initiative, a non-profit corporation advocating and educating about benefits of OSS [27], free software and open source software are essentially the same thing, the disagreement lies in how to promote such software [28]. The main difference between FLOSS and OSS is that the term "open source" emphasizes more the practical benefits of the licensing practices used with OSS instead of the moral rightness or importance of ensuring the users' freedom [26]. The term open source software is also newer than free software. Calling a product free software led people to believe that they could not sell it for money, so a new, more business-friendly term "open source" was developed in 1998 [24].

Considering that OSS projects are built by communities where often many of the developers contribute in their spare time, it is remarkable that many OSS projects can compete head to head, or even be ahead of their competitor products developed by corporate giants with vast resources. Apache web server is an often-cited example of a

successful open source software (e.g. [7, 22]), and also one of the most studied FLOSS projects [29]. It is also an example of a project that has been able to withstand competition from giants like Microsoft and Google. The development of the Apache project started 1994 [22], and it's still going strong even though it is now seeing tougher competition than before. In February 2017, Apache was on the second place in market share of web server software [30], as can be seen from Table 1. Also, Apache is still the market leader by a clear margin (45,67% market share versus the runner up nginx's 19,60%) in active sites, which "*counts websites but excludes those that contain automatically generated content such as domain holding pages*" [30].

Developer	January 2017	Percent	February 2017	Percent	Change
Microsoft	821,905,283	45.66%	773,552,454	43.16%	-2.50
Apache	387,211,503	21.51%	374,297,080	20.89%	-0.63
nginx	317,398,317	17.63%	348,025,788	19.42%	1.79
Google	17,933,762	1.00%	18,438,702	1.03%	0.03

Table 1 Web Server Software Market Share February 2017 [30]

Apache web server is also a project whose users are probably quite knowledgeable about using computers and software. According to [22], open source projects like this, where the end users are sophisticated, seem to be the most widely diffused ones. The Qt software, which we will consider in the empirical section of this study, is also meant for developers and thus seems to fit well into this category.

Even though there are several successful open source projects, many popular products are still following the closed source model. Like the above-mentioned Microsoft Windows, some software products are sold for a onetime fee. Many others, on the other hand, follow a subscription model, where the user has to pay recurring fees for example monthly or yearly. Spotify, the music streaming service, and many VPN providers are some example products following the subscription model.

Closed source has not always been the norm though. In the sixties and seventies, it was normal that software was freely shared [22, 7], and it was often produced in "*academic and corporate laboratories*" [7]. At this time, commercial software was an exception, and was normally sold bundled with the computer hardware [15]. The ARPANET, established in 1969, and later the Internet, enabled people to quickly and cheaply share code and other information [26].

However, in the beginning of 1980s some companies started limiting access to code that had previously been free [22, 7]. Some of the reasons that proprietary development started to dominate in 1980s were that the copyright laws were started to be applied on software and the increasing diffusion of personal computers enlarged the separation between software developers and users [15], as many users were now only users of software, not developing it themselves.

As an example of the tendency for proprietary software, some code produced in the MIT's Artificial Intelligence Laboratory was licensed to a commercial company, and people outside of the company (even the ones who had developed the code) were restricted access to the code [7, 26]. In 1983, the Free Software Foundation was founded by Richard Stallman [31], a programmer in MIT's Artificial Intelligence Laboratory who opposed this new development of making program code proprietary [7].

Both the foundation and Richard Stallman are still big proponents of free and open source software. As was briefly mentioned before, "*free*" here does not refer to the price of the software – there is nothing preventing people from selling free software – instead it refers to the freedom of the user of the software [25] to study, share and modify it [31].

Stallman also developed the General Public License (GPL) [7], which is sometimes referred to as "copyleft" as opposed to copyright [7, 32]. Under GPL the program code must be free [32], and any work that uses code released under GPL, must also be released under GPL [33].

The free software movement started licensing practices which are behind the ideas of many other OSS licenses used today [7]. The main difference between different OSS licenses is how much the public property rights can be mixed with the private ones, the GPL being one of the strictest licenses in this regard [15]. GPL was mostly used still in the 1980s, but in the 1990s other licenses started to become more common, and for example Linux was developed with a less restrictive license that allowed the open source code to be combined with proprietary code [22]. While many OSS licenses allow the open source code to be combined with proprietary code and be released as a proprietary product, even the least restrictive OSS licenses ensure that once a software has become open source, the program as a whole cannot be converted back to a private good [15].

Since the developers of open source software do not retain intellectual property rights for the software they have created because of the way the open source licenses have been

designed [15], it is clear that companies trying to make money on open source products face some challenges and need to adjust their business models as compared to the common models of selling copies or subscriptions. It is also an interesting question, why do companies decide to engage in open source efforts when, by definition, the resulting product will be available also for its competitors. According to von Hippel [7], freely revealing an innovation is often the best course of action for an innovating individual or a company. Often the innovator is not the only one who has the necessary information to come up with the innovation, and others having this information may have a lot to gain from revealing the innovation. Thus, it can be difficult to keep the required information as a secret [7].

Furthermore, revealing an innovation is usually a low-cost action and can result in private benefits for the innovator [7]. For example, open sourcing software previously offered as a closed source solution, would cost almost nothing, as the innovator would only need to make the already existing source code available on the Internet, and change the license. Some strategies used by companies who invest in OSS have also been identified in [21], and are listed in Table 2.

Strategy	**Description/Examples**
Pooled R&D	For example, development of the Mozilla project. Individuals and companies participate(d) in the development. Vendors like IBM and Sun focused on their own needs, e.g. making the browser compatible with platforms they were producing. Unlike traditional consortia in that non-members can contribute, and get the same benefits as members.
Spinouts	An OSS project is created out of a company's internal development project. The project can create demand for the company's other products, it can help the company to establish a standard, generate goodwill, attract improvements and complements. A spinout can also make sense when the technology is not yet commercialized or when it is believed that it will be of limited commercial value in the future.
Selling complements	Selling, for example, training, installation or support services.
Donated complements	It is possible for an organization to encourage people to donate their time and effort in building complements for the organization's products. For example, in the video game industry, some companies publish tools that anyone can use to create game modifications to existing games. These mods extend the

	lifetime of the actual game.

Table 2 Strategies for companies engaging in OSS [21]

Fitzgerald has also recognized two OSS strategies [24], which in Table 2's categorization both fall in the selling complements category. The first strategy is called *value-added-service-enabling* model, and means selling support services as described in Table 2. Another strategy noted by Fitzgerald, is called *loss-leader/market-creating* model [24]. In that strategy, the idea is to distribute a product for free, boosting a market for another product. An example given by Fitzgerald is the OSS called Sendmail, which has another version (Sendmail Pro) that offers more functionality, and is sold for a fee.

In this subsection, we have looked into open source software, including its definition, history and some strategies organizations can use to benefit from OSS. The topic of the following subsection is online communities, which are highly relevant with regards of OSS, since OSS projects are often developed by online communities.

2.3 Online Communities as Platforms for Open Innovation

In this subsection, we will discuss *online communities*. A simple definition for an online community could be that it is *a group of people or organizations connected through the Internet, for example by email and websites, and sharing an interest or a common goal*. This definition is inspired by definitions of *innovation communities* in the literature.

According to Von Hippel, innovation communities include individuals or firms who can be connected by means of information transfer, e.g. electronic, face-to-face, or other communication methods [7]. Often these communities have tools and infrastructure helping the community members to develop and test their innovations more quickly [7]. Another definition for an innovation community has been developed by West and Lakhani, who define it as "…a voluntary association of actors, typically lacking in a priori common organizational affiliation (i.e. not working for the same firm) but united by a shared instrumental goal—in this case, creating, adapting, adopting or disseminating innovations." [34].

The communities around open source software projects are a good example of online communities, as well as highly relevant for our study. In OSS communities, the users of the software are often also the most active contributors [35], and developers are also users [19]. Since the developers in OSS projects are often also users, and since creating

software is for a big part about innovating, OSS is an interesting example of a user innovation community. In user communities, companies may be especially interested in identifying so called lead users who are "…*at the leading edge of an important market trend(s), and so are currently experiencing needs that will later be experienced by many users in that market"* [7] and "*...anticipate relatively high benefits from obtaining a solution to their needs, and so may innovate"* [7]. It seems likely that in an open source context, the lead users who have the needed skills to develop the project would be members of the developer community. However, an organization may want to consider how it could identify less technically skilled lead users and make it easy for those users to contribute in the project.

Open source communities are also an example of an electronic network of practice. An electronic network of practice is defined as "…a self-organizing, open activity system focused on a shared practice that exists primarily through computer-mediated communication" [36]. With the term, *self-organizing open activity system,* Wasko and Faraj refer to the participation being voluntary and open for people who are interested in solving problems common to the network's practice together with the other participants. Often in this kind of a network the participants don't know each other [36], which also applies to OSS networks or communities.

2.3.1 Community Roles

Often in an online community some of its members are contributing, and participating much more actively than others. Therefore, making a distinction about who is a member of an innovation community, may not always be very straightforward. The borders of the community might be unclear because the differences between a core member and a member on the community's periphery may be bigger than the differences between a periphery member and a nonmember [34]. Different kinds of community members are summarized in Table 3.

Role	Description
Core member	Actively contributing in the community. May have more rights/permissions than other members for example in case of OSS communities.
Member	Contributing in the community, but less actively than a core member. Some members can be active very seldom, making it difficult to distinguish them from non-members
Periphery member	A member that is sometimes contributing in the community, but so rarely, that he is thought to be in the periphery of the community

Non-member	Not a member of the community, i.e. not contributing in it

Table 3 Membership Roles in Innovation Community according to [34]

An OSS community can include for example highly skilled developers who are developing the open source project as their hobby, and less technical users who are providing for example bug reports and feature requests. According to [19], source code and bug reports are in fact the most common contributions in an OSS project, some other examples being documentation and test cases.

If an OSS project is started by a single developer, he has rights to do anything with the project, and since there is only one member in the community, it does not have any structure or hierarchy. The sole developer can commit all the code he wants and determine the direction of the project. However, some other developers may join the project, and when time goes by some of them may decrease their contributions or stop contributing altogether and other, new developers may join the project. If the community becomes bigger, there may be a need to establish different roles with different permissions. Figure 3 shows a hierarchical community structure with various roles. Some projects, for example Apache have developers on even more levels than shown in Figure 3, and have a strict hierarchy [19]. On the other hand, in some projects, the community structure can be very loose, and all the developers can be on the same level [19].

The OSS projects differ from each other also in how they are being led. Some projects, e.g. Linux have a single, undisputed leader, and in others, like the Apache project, there is a committee that tries to achieve consensus when there is a disagreement [22].

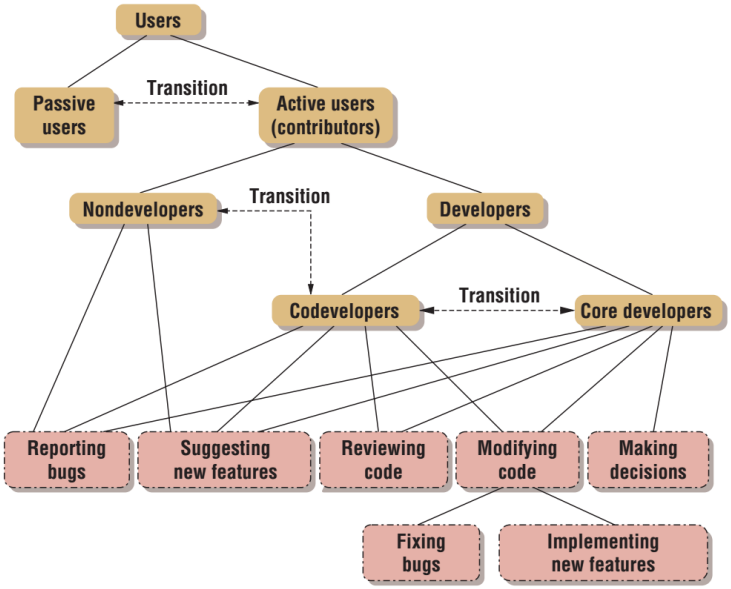

Figure 3 A Possible Community Structure for an Open Source Software Project [19]

Figure 3 depicts a community structure with multiple levels of developers, active and passive users and some of the activities performed by the developers and non-developers. The distinction between passive and active users means that the passive users are only using the software, while active users are somehow contributing in it (e.g. by writing documentation). There are, however, many more roles that can be present in an OSS community. For example, [37] distinguished 8 roles including passive user, reader, bug reporter, bug fixer, peripheral developer, active developer, core member and project leader.

Compared to regular developers, the core developers (named core members in [37]) have more power [19], and guide and coordinate the project's development [37]. In addition to the roles in the Figure 3, developers can also be called committers if they have commit rights over the code, i.e. they can decide what code gets added to the software. The committers usually have a high visibility in the project, and they are faster at solving problems as compared to other members [16]. From the discussed roles, by far most people usually belong to the passive users group, for example in Apache project 99% of the users are passive [37].

Joining an OSS project as a new community member is usually not as easy as going to the project's website and starting to contribute. Depending on the project, it may be easier to contribute to some modules than to others. For example, Von Krogh, Spaeth and Lakhani studied project called Freenet and found that the contribution barriers were lower

for projects that were modularized so that the contributor could contribute to a specific module without needing to understand how the other modules worked [38]. Also, projects where the contributor could freely choose the programming language for the contribution had lower contribution barriers [38]. Usually FOSS projects are well modularized to allow developer contributions to participate by focusing only on a specific subsystem [24].

On the other hand, in the Freenet study, it was found that some modules had higher contribution barriers because they required difficult algorithms or for example advanced knowledge of mathematics. New project members of Freenet were found to be lurking, i.e. observing others' messages, often from a few weeks to many months before feeling able to contribute in the technical discussions [38].

As can be observed from Figure 3, transitions can happen between the different community roles. When a community member actively contributes and gains respect of the community, they may be invited in the developer group, where they have more rights over the code [19]. In some projects, the developers can further advance to be core developers [19].

Innovation communities are often created and organized by a company trying to align the community with the company's objectives [34]. This applies to OSS communities as well. While some projects are built entirely by the community, others are backed by a commercial company. Company backed OSS projects can be called *commercial open source*, and the community driven OSS projects *community open source* [16]. The main difference between these is that in commercial open source there is a single entity, a corporation for example, that has all the decision power about the project. Usually in the commercial open source projects the product is free for nonprofit users, and sometimes even for commercial users. Normally a company owning such a project would make money by selling support services [16].

Commercial OSS projects can help a smaller company to innovate for example by giving the company access to many people who can collaborate with the company in the software development [35]. In general, though, open innovation is being done more in larger companies [2, 18], but many SMEs are changing their innovation process to be more open [18].

The degree of company involvement can vary greatly from one project to another. A

company can be *hosting* a project, *supporting* it or *collaborating* in one [35]. When a company is hosting a project, it manages the IP, e.g. the company can require every contributor to sign a contributor license agreement, where the contributors grant the copyright of the contributions to the company. In a project where the company is in a supporting role, it does not manage IP, but only provides support, for example by having its developers working in the project [35]. The supporting model has also been recognized by [22], who call it "*living symbiotically off an open source project*". Finally, in a collaboration model the company manages the IP together with the community and can help in developing the community [35].

As mentioned earlier, in many OSS projects the community members can gain more rights over the code by making valuable contributions and gaining the community's respect [19]. In company sponsored projects, if the company is hosting a project or collaborating in it, the company can give commit rights (right to add and modify software code in the version control system used to manage code contributions) for its employees, while the employees of a company in a supporting role, have to follow the same rules as the other developers in the project [35].

Another way to classify projects where an organization is involved, is to look at the relationship between the community and the organization. Dahlander and Magnusson (2005) proposed three kinds of relationships based on their study of open source software projects [39]. The relationship types identified were *symbiotic, commensalistic,* and *parasitic.* An organization following the symbiotic relationship model focuses on the well-being of the organization as well as the community. An organization with a commensalistic approach, however, would try to benefit from the community. While the organization would not actively develop the community or its resources, it would not cause harm to the community either. Finally, an organization with a parasitic approach would just try to benefit from the community without considering how its actions affect the community [39]. Dahlander and Magnusson argue that this approach would not be sustainable, and it was not observed in their study.

2.3.2 Competition and Cooperation

One question that has been studied a lot, is why do people contribute in innovation communities. For example, as we have mentioned before, in the case of open source software, the licenses are designed so that the developers do not have intellectual property

rights on the code they have written. Still most of the contributors in OSS are professional programmers with experience [26]. Contributing in an OSS project also incurs significant opportunity costs for the contributors. For example, a student contributing in an OSS project might graduate later [22].

It would seem probable that the innovation communities would have a big free-riding problem. For example, in the case of open source software, why would one decide to contribute instead of just benefiting from the contributions of others for no cost for oneself? Indeed, most of the people downloading open source software are free riders, i.e. users who use the software, but don't participate in developing it further. In OSS context, free riders are actually seen as an asset since they still increase the market share of the product, may help set standards, and might contribute in ways that do not involve coding, for example by reporting bugs [26].

Developer motivation is often divided into intrinsic and extrinsic motivation, e.g. [15]. Extrinsic motivation is often related to monetary compensation, but can be about other things, such as the benefits gained from reputation or extended functionality [15]. Majority of the contributors are actually motivated by having either a personal or a business use case for the code they are developing [26]. Even though the chance of getting helpful comments from others after publishing code online, might not be very high, it may still be worth it because the cost of publishing the code is low [15]. By contributing in a project, a developer can of course gain reputation in the project, and advance in the project's hierarchy as presented in the previous section. However, there can also be broader reputation gains among employers or in the venture capital market [15, 22]. It has also been suggested that peer recognition is seen important by itself, and because of it, developers like to work on projects that attract a big number of other developers [22].

A developer has intrinsic motivation when he values the work he is doing for its own sake, for example enjoys solving the technical problems an OSS project presents. Often programmers enjoy coding [15], and often, for a developer, it can be more fun to work with an open source project than with the assignments at his day job [22], since in an OSS project he would usually have more freedom to choose the tasks that are interesting to him.

In this section, we have discussed online communities, the roles in them, and the different motivations people may have to contribute in them. In the following section, which

concludes the literature review, the requirements engineering practices in OSS projects will be discussed.

2.4 Requirements Engineering for Open Source Software

In this section, we will discuss *requirements engineering*, and how it is different for open source software projects as compared to traditional software development projects.

Traditionally the recommended way to do requirements engineering has been to use a multi-phase process where requirements are identified, modeled, analyzed, validated and verified ahead of starting system development (e.g. [40, 41]). In this thesis, we will refer to this approach as traditional requirements engineering to distinguish it from other, more lightweight approaches sometimes used in practice. There are projects where the traditional requirements engineering approach is not appropriate [41], and situations in which the traditional approach is seen as valuable, but is still not fully used [42]. In this section, we will first describe the traditionally recommended requirements engineering approach, and then talk about some alternative approaches that are used especially in some OSS projects.

Software is normally produced to fulfill a need, in other words to solve a problem. The purpose of requirements engineering is to define the problem that a software needs to solve [43]. To satisfy the need, the software is *required* to do something and to have specific quality attributes. Requirements can thus specify actions or properties of the software, which are called functional and non-functional requirements, respectively [44]. The non-functional requirements can specify, for example, performance, platform dependencies, maintainability, and so on [44].

The term requirements engineering is used to describe "…a systematic approach through which the software engineer collects requirements from different sources and implements them into the software development processes" [44]. The idea behind traditional requirements engineering is also that the requirements are well defined before starting to develop the system to avoid expensive rework [40].

Requirements engineering activities can be categorized in different phases. The way in which this division is done differs a bit from one research paper to another. For example, in [40], modeling of requirements is not presented as its own phase, but is included in the analysis phase. Also, in [40] documentation is presented as its own phase, while it is

missing from [43]. We will here discuss the requirements engineering phases presented in [43], which are summarized in Table 4.

Phase	Description
Elicitation	Refers to activities that aim to achieve understanding of objectives, goals, and motives of creating the suggested software [43]. In the elicitation phase the requirements that the resulting software system needs to have to achieve the identified goals and objectives, are also identified [43]. The requirements can be elicited by consulting stakeholders and utilizing for example interviews, observations, focus groups, brainstorming, prototyping or use cases [40].
Modeling	In this phase, the requirements are expressed with help of models. While models may be used already in the elicitation phase, the models in this phase are usually more precise and complete and can be used to communicate the requirements to the developers [43]. Multiple models can be used, each of them recording specific details of the requirements, for example the properties or behavior of data that the software is supposed to maintain [43].
Requirements Analysis	At the requirements analysis phase, the requirements quality is assessed [43]. The necessity, consistency, completeness, and feasibility of requirements are checked [40]. Also anomalies which can include for example missing assumptions or obstacles for satisfying the requirement can be revealed by the analysis [43]. Besides the quality of a requirement, the risks related to the requirement can be analyzed at this phase of the requirements engineering process [43].
Validation & Verification	The goal of the requirements validation is to make sure that the documents and models describing the requirements are accurate in expressing the needs of a stakeholder [43]. For example, organizational knowledge and requirements document can be used for validation purposes [40]. If there is a formal description of a requirement, it is possible to verify that the stakeholder's requirement would be fulfilled by the specified software [43].
Requirements Management	The requirements management phase consists of various tasks for managing the requirements such as managing the evolution of requirements over time [43]. According to [40], requirements management includes "…all activities concerned with change & version control, requirements tracing, and requirements status tracking."

Table 4 Phases of (Traditional) Requirements Engineering According to [43]

In traditional software projects requirements' properties such as traceability, consistency, completeness, and internal correctness are valued high, and there may be even contractual obligations to produce a requirements specification [20]. In these situations, it may be

necessary to thoroughly analyze the requirements, build models and validate and verify the requirements. A requirements engineering process where the requirements are modeled before development can also help to provide better estimates of the development time [41].

Some organizations have a more lightweight approach to requirements engineering. For example, agile software development and requirements engineering are often seen as incompatible [40]. Agile development highlights the importance of face-to-face collaboration among developers and customers to share knowledge, while requirements engineering often uses documentation to achieve knowledge sharing [40]. Also, in OSS projects the requirements are often very unstructured [45], and the focus is more in community participation and development and other sociotechnical concerns [20]. Traditionally in FOSS communities, the requirements are not formally analyzed [24].

Alspaugh and Scacchi [46] make a distinction between the requirements in traditional projects versus requirements in open source projects by calling the former classical requirements. Classical requirements are defined in a central requirements document or in a requirements repository [46]. The requirements repository or document is inspected for completeness, internal consistency and external consistency (with the domain and stakeholder needs) [46].

In OSS projects, on the other hand, requirements are decentralized [20, 45], i.e. instead of there being a central repository of requirements, they exist in places like online conversations and repositories, and in the interactions between the OSS project's members [20]. The more lightweight requirements engineering approach used in some software projects can be called *just-in-time requirements* [41]. With this approach, lightweight representations of requirements are used, and they are refined as they evolve [41].

In [41] Ernst and Murphy studied three open source software projects (Mozilla, Lucene, and CONNECT) where just-in-time requirements were used. They found that instead of having a detailed plan before starting the development, the requirements were first described in natural language as simple statements, and they got a more detailed form only when the software satisfying the requirement was being developed. Also, traceability, traditionally valued high in software projects, was used only when it was needed, with the help of an issue tracking tool (e.g. Jira) [41]. While this helped to see

where each task (a concrete task that a developer can work on) had originated, it was more difficult to find the source of a requirement (a requirement can include several tasks). In the studied projects, there also didn't seem to be clear prioritization phase. In two of the three projects prioritization of requirements was dependent on developers' interest, and in the third one complex negotiations were needed to do prioritization [41].

Even though many requirements engineering practices used in traditional software development are not used as much in open source software, it doesn't necessarily mean that they would not be useful. In a recent survey [42] Kuriakose and Parsons investigated the open source software developers' perceptions of the usefulness and level of current usage of the requirements engineering practices used in closed source software development (CSSD). The survey's 84 respondents indicated that while they regarded the CSSD requirements engineering practices as useful, according to their experience, they were not used much in the OSS development.

According to [24], traditionally FOSS projects used to be started by a single developer or a small group who had a need for the software, which also meant that they understood the requirements well. Nowadays, however, companies are increasingly looking to OSS, and trying to find ways to use it to gain competitive advantage and are paying developers to work on OSS projects, and moving to projects where the developers may not be familiar with the requirements [24].

While the literature has notions about the misunderstandings being typically minimized in OSS context (as in [20]), and thus a lightweight requirements engineering approach being possible (as in [19]), these notions may be more applicable to smaller OSS projects, and to projects where the developers are also users of the software they are developing. According to Fitzgerald, increasing attention is being paid into the design and planning phases of OSS, and he calls this new way of developing open source software *OSS 2.0* as opposed to the traditional FOSS [24]. Also, while we stated before that agile software development and requirements engineering are often seen as incompatible, software projects following an agile method can also use requirements engineering techniques closely resembling those recommended by traditional requirements engineering [40]. For example, agile development method called Extreme Programming (XP) uses elicitation techniques such as interviews and brainstorming [40].

To summarize what we have described in this subsection, there are some requirements

engineering practices that have been recommended in the literature. There are indications that at least in OSS and agile software projects the extent to which the recommended practices are used varies from project to project. In some projects, a lighter requirements engineering process is used where the requirements evolve more also during the development of the system instead of being fully understood before starting development.

2.5 Summary of Findings from the Literature

In this section, we have discussed open innovation and requirements management in the context of open source software, which was also discussed in its own subsection. The main findings are summarized in Table 5. The left-most column of the table states the overall topic under which the finding falls, the center column describes the finding, and the right-most column states the most important literature references related to the finding.

Topic	Main Findings	References
Open innovation (sections 2.1.2 and 2.2)	Traditionally big corporations had internal R&D departments for innovation, nowadays companies are opening up the innovation process by including external stakeholders.	[5, 7, 10, 18]
	Three core innovation processes have been identified. They are inbound and outbound innovation and coupled innovation process.	[14, 5, 13, 18, 2, 8]
	Inbound innovation: external resources like ideas and knowledge are sourced e.g. from suppliers, customers or competitors, or acquired by purchasing them from the market place.	
	Outbound innovation: Organization's internal resources (e.g. a proprietary technology) is sold or revealed. Open sourcing a previously closed source software is an example of revealing.	
	Coupled process: Combining both inbound and outbound innovation. Companies join strategic networks where they develop innovations together with their partners.	
Open source software (section 2.2.4 and 2.3)	Software that has been published under an open source license [15, 7], i.e. a license that allows anyone to view, use, and modify the software's source code.	[15, 7]
	Some members of an open source community can be much more active than others, and it may not even always be clear who is a member of the community and who is not. The community may be hierarchical so that some members have more permissions	[34, 19, 37]

	than others, or everyone can have the same permissions.	
	Some OSS projects are developed only by individual developers, but many are somehow supported by a company or even by multiple companies.	[16, 34, 35, 22, 39]
	Some OSS strategies used by organizations have been identified: pooled R&D, spinouts, selling complements, donated complements.	[21, 24]
Requirements engineering (section 2.4)	Traditionally, a multi-phase approach to requirements engineering has been recommended in the literature. Depending on the research paper, the phases can be for example elicitation, modeling, requirements analysis, validation & verification and requirements management. The goal is to understand and analyze the requirements well before system development is started.	[43, 40, 41]
	In traditional software development, the requirements are also managed in a central place. The requirements document or repository is inspected for completeness, internal and external consistency with the domain and stakeholder needs. Requirements traceability, consistency, completeness, and internal correctness are valued high.	[46, 20]
	In OSS, the requirements engineering approach is often more lightweight and less formal as compared to the requirements engineering approach in more traditional software projects. The requirements are often spread around in online forums, email discussions, and so on.	[41, 20, 24]
	Companies seem to be looking increasingly into OSS. They also increasingly paying developers to contribute in OSS. It may be that in some of the projects developers do not know the requirements as well as used to be the case in OSS, and thus increasing attention is being paid in the analysis and design phases of OSS.	[24]

Table 5 Main Findings of the Literature Review

We noticed that there were several different ways to practice open innovation, and also how some organizations are using open source software in their approach towards innovation. With regards to the requirements engineering, the literature seems to suggest that some open source projects have a more light-weight approach than what has been traditionally recommended. In the following sections, we will see how the findings relate to The Qt Company. We are going to pay special attention to the way the organization is managing requirements in its open source project and describe the process in section 4.

3 Case Study: Qt Software's Requirements Management Process

The subject of the case study is Qt Company, which is a for profit company, leading an open source project also called Qt, which we call here Qt software. The Qt software was originally developed already in 1994. The relationship between Qt company and the community around the OSS project can be categorized as symbiotic. With this case study, we aim to answer the research questions RQ2 and RQ3.

- RQ2: What open innovation approaches are used in the Qt OSS project?
- RQ3: What is the requirements management process in the Qt OSS project like?

In this section, we will first discuss the case study design in section 3.1. After that, we will provide background information about both The Qt Company, and the the open source project including the community around it. This will help us to understand the context around the requirements management process. Thus, the goal of the following subsections is to introduce the case study design, the case company, and some characteristics of the open source project under study and the community around it.

In section 3.2 we will briefly describe The Qt company. Section 3.3 will discuss the context of the requirements management process in high level, i.e. the systems and groups of people who are related to the process. In section 4 we will then zoom into the research questions RQ3a, RQ3b and RQ3c, presenting the results about the actors of the requirements management process and the different actions they carry out using the issue tracking system Jira and code review system Gerrit.

3.1 Case Study Design

The method for the empirical part of this study is a descriptive case study. A case study is "…an empirical method aimed at investigating contemporary phenomena in their context", and it can be used to describe phenomenon or situation [47]. As mentioned above, The Qt Company's Jira has more than 32 000 user accounts, and more than 250 commits are added to the Qt software project each week [48]. The large and active community makes the project an interesting example of open innovation and open source software. In the developer community, big portion of the contributions come from The Qt Company, its partner companies, or other commercial sources, but individual developers

contribute as well, especially by authoring code [49].

The main goal of the case study, is to describe the requirements management process of the Qt open source software project. By using the case study methodology, we are able to report the process in a more detailed way, than if multiple software projects were studied.

There are 5 steps in conducting a case study [47]. These steps are paraphrased below.

1. Designing the study. Involves planning and defining objectives
2. Planning data collection: Defining the procedures and protocols for gathering data
3. Collecting data: Using the planned procedures and protocols to actually collect the data
4. Analysis: Conducting analysis on the gathered data
5. Reporting: Reporting the results

We have already discussed the objectives of this study. To build a thorough understanding of the requirements management process of the Qt software, data were collected from several sources. The main data source were face-to-face interviews conducted with the case company's employees in spring 2017 in a research project called OpenReq. The interviews were done by an international research group including researchers from University of Helsinki, who kindly shared the data with the author of this thesis. The interviews did not follow a strict set of predefined questions, but instead the interviewers explored topics such as different aspects of requirements management and the software systems involved.

The interviews were recorded, and an external company transcribed the data into text. To extract the relevant data from the interviews, the author of this thesis read them, highlighting and writing notes in separate files about the parts relevant to our research questions. The audio was also listened alongside with the transcribed data file to ensure the accuracy of the transcription.

The interview data was complemented with publicly available information from the Qt web page (https://www.qt.io), and from their wiki web page (http://wiki.qt.io/). These three sources of information were used to build a conceptual understanding of the requirement management process of the Qt software. Finally, we picked two sample requirements from The Qt Company's Jira, both to verify the understanding we gained

from the descriptive sources, and to find out possible gaps in the other information sources. The results were also verified by doing a follow up interview with an employee (community manager) from The Qt Company.

This *data triangulation* achieved by using different data sources helps to provide a broader picture of the phenomenon [47]. In the analysis phase, we took all the gathered data and analyzed it from the point of view of our research questions creating conceptual diagrams and textual descriptions, which were also used to report the results in this thesis.

3.2 The Qt Company

The Qt Company is a growing company having currently about 200 employees [50]. The company is highly R&D driven having about half of its employees in the R&D department. The company's product, on which we will focus in this study, is called Qt for Application Development, or Qt for short. In this thesis, it is referred to as Qt software, because it is shorter than Qt for Application Development, and because we want to make a clear distinction between the software and The Qt Company.

The Qt software is a cross platform user interface and application framework [51], used in more than 70 industries [48]. By using the framework, developers only need to write one code base, and they can run their application on many different platforms including desktop, mobile and embedded systems [51]. Some of the supported platforms include different Windows versions, some Linux versions, Android, iOS, and so on [51].

Qt software is following a dual licensing strategy. Qt software is available under GPL and LGPLv3, and under a commercial license [52]. The benefit in choosing the commercial license is that it allows the customer to have "…official Qt support, and close strategic relationship with The Qt Company…" [52]. The Qt Company also has a product called Qt for Device Creation, which allows its users to "…create embedded devices with modern UIs with maximum performance" [52]. However, this product is only available through a commercial license [52], so it will be excluded from our analysis.

In the open source community around Qt, there are a lot of individual developers, and also developers employed by other companies. The Qt software has more than 60 weekly contributors adding more than 250 commits to the project each week [48], and there are around 32 000 user accounts in Jira (including some duplicate accounts). Jira is a central system for managing the requirements and is used both by the community and the

company employees to track the requirements. Displaying the activity of the community, one of the interview respondents commented regarding requirement gathering from the OSS community:

"*I think maybe what makes us a bit special is that we actually don't have a problem that people don't talk to us, I mean there might be cases, but in general, I can imagine for a more traditional company, they have the problem that people just, you know, curse at the product, but just don't buy it anymore, or something like that. While… I have to say that I don't have the feeling that this is our problem. So we get a lot of people wanna talk to us, and file bug reports and so on, and we have more struggling with keeping up to date, and working with that data, instead of that we missed the data."*

As we noticed in the literature review, often communication in open source projects is done through online channels, which is also true for the Qt software. Mailing lists, an issue tracking tool and a code review tool are all important communication channels for the project. While The Qt Company is leading the project, the community also plays an important part in the development of Qt software.

3.3 Context of the Requirements Management Process

In this section, we will discuss the Qt software's requirements management process on a high level. This provides background and context for the later discussion on how individual requirements are managed by different stakeholders.

Figure 4 shows the high-level context of Qt software's requirements management based on the interview material of this study. In the figure, the big dotted line rectangle separates the entities and systems inside The Qt Company from the external ones. The other rectangle shapes represent systems, and the diamonds represent groups of people or companies. The diamond shapes inside The Qt Company are different departments, and outside The Qt Company there is the community consisting of both individuals and companies.

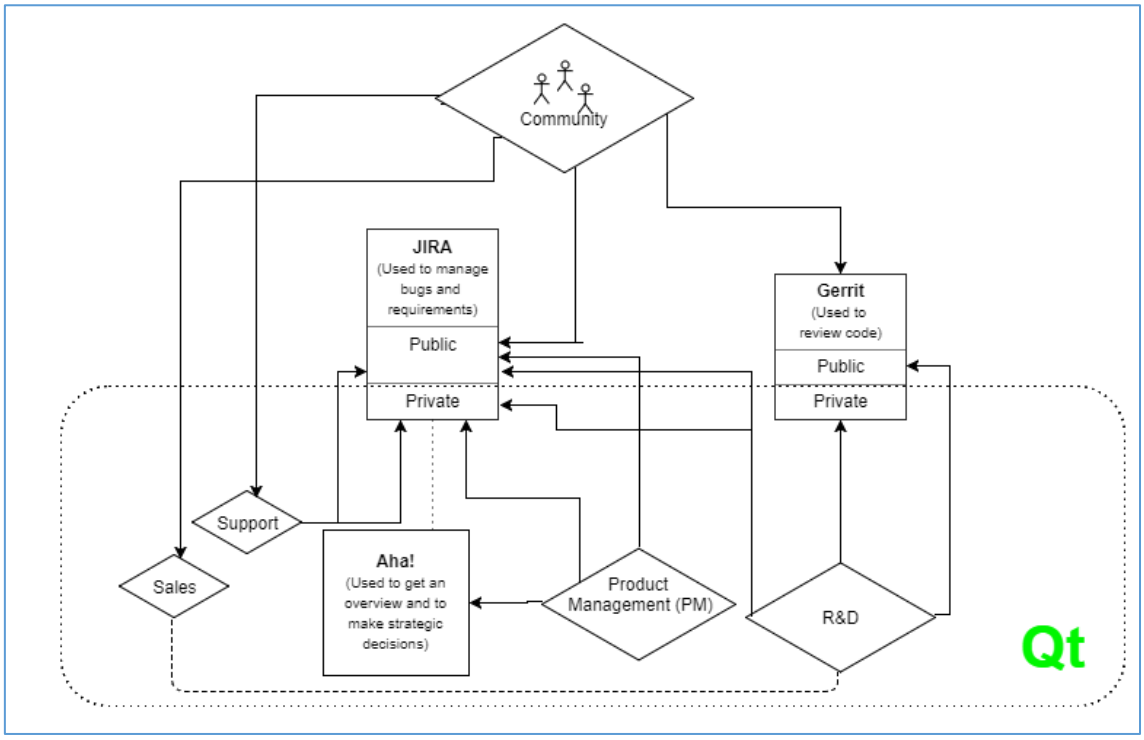

Figure 4 Context of Qt software's Requirements Management

Inside The Qt Company, the product management team uses a system called Aha! to manage the high-level requirements. With Aha! the product managers are able to get an overview, and make strategic decisions. They can view for example the company's goals and timelines. Another system called Jira is used to manage the individual tasks, which can represent for example bugs or requirements.

The dotted line between Aha! and Jira represents an integration, which was being implemented in summer 2017. When the integration is done, the product management team should move into using only Aha!, but currently they use Jira as well. The third system in the figure is Gerrit, which is a code review tool. When someone has finished a task in Jira, they need to submit the code for a review to Gerrit, and the change is not added into the product before it has been approved by someone who has a permission to do so.

Outside of The Qt Company, there is a community of individual developers, and companies. The community members benefit from the Qt software, and also benefit The Qt Company by making the product better and by providing various services around the product. Some companies use the Qt software to create their own products or applications, some provide consulting or training services, and some develop the Qt software, but don't use the product themselves. The Qt Company lists 6 types of partner organizations on

their website. The types are service, technology, automotive, training, local distributor, and community [53]. For example, the service partners provide training and consultancy services, while the community partners participate in developing and governing the Qt software [53].

The community provides feature suggestions, bug reports, reviews submitted code, fixes bugs and implements features. Therefore, the community members developing the product use both Jira and Gerrit. The community also has commercial customers who may communicate with The Qt Company more through its sales and support teams. The sales team can then communicate information about customers' needs directly to the R&D team. This communication is represented by a dashed line in Figure 4.

Both Jira and Gerrit have public and private projects. The public ones can be viewed by anyone from the community, but some of the customer projects can be confidential, and therefore cannot be managed publicly. They are still managed in the same systems, but permissions are used to restrict access.

In this subsection, we have described the environment and context in which the requirements for the Qt software are managed. In the following subsection, we will discuss the roles in the community around Qt software. In section 4 we will take Figure 4, and zoom into the area showing the R&D team, community, Jira and Gerrit looking into how the community and the R&D team manage the requirements in these systems. We will also describe how the roles described in the following subsection, and other actors are involved in the requirements management process of the Qt software.

3.4 Roles of Individuals in Qt Software's Open Source Community

The open source community of the Qt software is a hierarchical one having 5 different roles. In Figure 4 the OSS community is shown as one big entity, but as mentioned before, it consists of various companies and individuals. The roles individual members can have are depicted in Figure 5. The figure also shows the usual order of the roles for an individual. Every new member joining the community starts from the *user* or *contributor* roles. For these roles, there is no selection process. Users are "…community members who have a need for the Project" [54]. On the next level, a contributor is anyone who makes significant contributions to the project. A contribution does not have to be a code change, but can be for example a bug report or a fix or addition in documentation.

Figure 5 Community Roles in the community around the Qt software

Documentation is often the easiest way for a new contributor to help in the project. However, in the Qt software documentation is found as comments alongside the code, which means that to change documentation, the community member needs to check out the code from version control, i.e. download the code from an online repository. When the community member has done the additions or fixes in the comments, the code needs to go through the review process, even if only documentation was changed.

The *approver* role is the first one with a selection process. A person becomes an approver when someone who is already an approver nominates him or her and the nomination is supported by another approver or by a *maintainer.* If no one objects, the person gets an approver status after 15 days [54]. Compared to contributors, approvers have more rights over the project, and can approve code changes. The company employees follow the same rules as the community members, so they only become approvers after they have made sufficient contributions, so that they are nominated to become approvers in a similar way as any other member of the community.

The process to become a maintainer is similar as the process of becoming an approver. A maintainer has usually been an approver before, and may be nominated by another maintainer for the maintainer role. Yet another maintainer has to support the nomination, and then if no one objects, the nominee becomes a maintainer in 15 days [54].

The Qt software is divided into components, and a maintainer is responsible for a component. The maintainer has the last word in issues regarding the component he is responsible for in case there is a conflict. If, for some reason, the issue still cannot be resolved, the chief maintainer decides. The Qt Company's Jira lists more than 150 components for the Qt software including for example *qmake (build tool), networking* and *openGL.*

Many of the components are maintained by The Qt Company's employees, but some have maintainers from other companies participating in Qt software's development. For example, several maintainers are from company called KDAB [55]. Also, while the Qt Company is leading the open source project, for example decisions about when a new

version of the Qt software is released, are done by community consensus [56].

Finally, at the top of the hierarchy there is the *chief maintainer,* who is, at the moment, The Qt Company's CTO. He has the final say basically in all matters. If the chief maintainer decides to step down, a new one is chosen by a majority vote among the other maintainers [54].

In this section, we have gained understanding about The Qt Company, and the community around the OSS project under study. In the following section, the goal is to identify the actors and their actions in the Qt software's requirements management process as well as to describe the flow of the whole process.

4 Empirical Results

The community described in the previous section, and The Qt Company, use mainly Jira to manage Qt software's requirements, and a code review tool called Gerrit to verify the quality of the contributions. In this section, we describe how The Qt Company and the community use those systems to manage requirements as well as who are the actors in the different stages of the process.

New requirements, as well as problems with existing functionality (i.e. bugs), are represented as *issues* in Jira. A Jira issue is an item that a user can create, and which can then be worked on. An issue has many attributes including description, issue type, component, priority, status, reporter, assignee, creation date, update date, and so on. Some of these can be set by the person who creates the issue, and they can also be changed afterwards. Figure 6 shows the user interface of Jira for creating an issue of type suggestion. When creating an issue of type bug, the user interface is otherwise the same, but the field *Affects Version/s* is required indicated with a red star like the one next to field *Summary* in Figure 6.

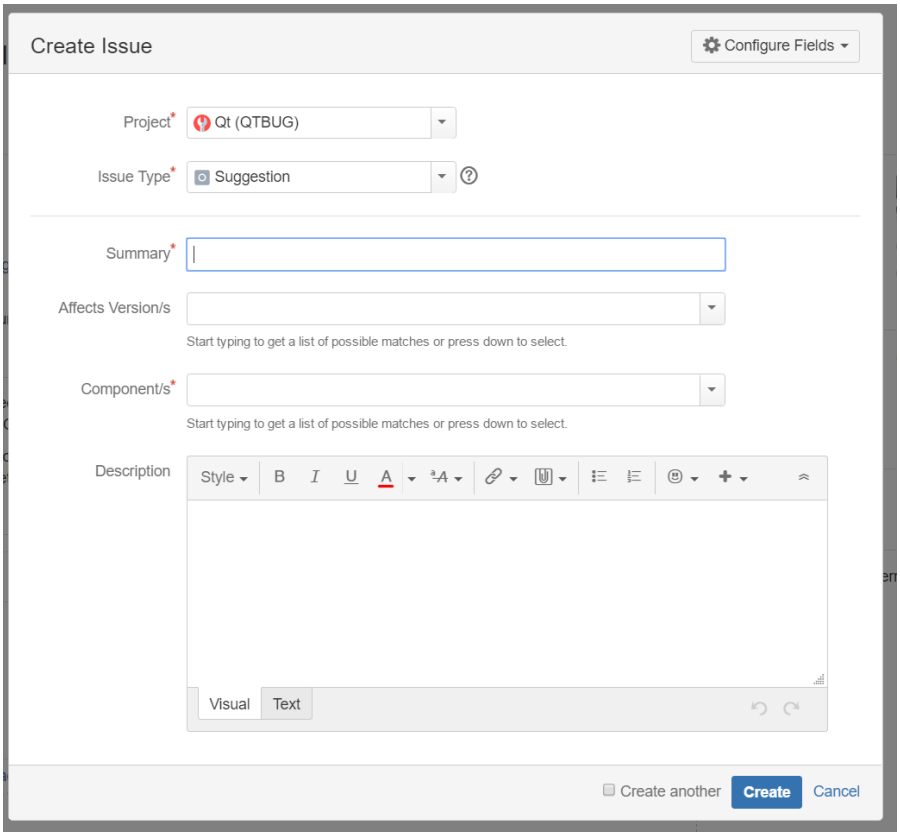

Figure 6 Jira's user interface for creating a suggestion for Qt software [57]

The different attributes of issues will be described as they are discussed later in this section. The issue types in The Qt Company's Jira are *suggestion, bug, user story* and *epic.* The Qt Company uses the types user story and epic mostly internally, not much in the public projects, so in Qt software the types suggestion and bug are mostly used. Bugs are for reporting problems with the product, and suggestions are for public feature requests. The suggestions are handled the same way as bugs, as one respondent stated in the interviews:

"*Bug and suggestion are pretty much the same set up, same work flow, same screens. The only difference is basically so that we can distinguish, this is a bug, this is something that does yield a feature. Something new*."

Since bugs and suggestions are the main issue types used in the public Jira, we are focusing our analysis on these issue types.

To describe the requirements management process, we will first study the two first parts of research question 3, which were:

- RQ3a) Who are the stakeholders/actors (in Qt software's requirements management process)?
- RQ3b) What actions do they perform?

When we have identified the actors and their actions, we can investigate the process as a whole, i.e. our findings related to the last part of research question RQ3, which was:

- RQ3c) What is the flow of the requirements management process, i.e. in what order do the actions occur?

The following subsection will present our findings related to RQ3a and RQ3b, while RQ3c is discussed in the second subsection. The results in these sections are based on interviews with The Qt Company's employees. In the last subsection, we will investigate a few sample Jira issues of the Qt software.

4.1 Stakeholders and their actions

A stakeholder, in terms of the requirements management process, could be someone who works with a requirement, for example by describing a request, writing program code, prioritizing the request etc. On the other hand, a stakeholder could mean someone who only has an interest towards the requirement because they want to use the resulting

feature, but they might not actively influence the requirement before it has been published. In this study, the scope is the requirements management process from the moment when an issue is created in the requirements management system (Jira) to the point when its quality has been verified so that it is ready to be added in a software release. Thus, our analysis focuses on stakeholders who somehow influence an issue during the time it is open in Jira/Gerrit. Those stakeholders are called *actors* in the following analysis.

One obvious actor is the *requester* or reporter, i.e. the person who originally creates the issue in Jira describing a bug to fix or a new feature the they think should be added in the software. The Jira used for Qt software's requirements management is publicly available at https://bugreports.qt.io/ where anyone can create an account for it. The Qt software is managed in the Jira as a project named Qt (QTBUG). A project in Jira is a collection of issues, and the Qt (QTBUG) project includes all the issues of the Qt software. Anyone with a Jira account can create an issue for this project, so the reporter can be anyone from the community or from The Qt Company.

As can be seen in the user interface in Figure 6, when an issue is created, one piece of information that must be provided, is the component the issue belongs to. Each component has a *default assignee*, who gets a notification (by email) when the issue has been created. In some cases, default assignee is a team, but usually it is a single person. According to the interviews, there seems to be some debate over whether the default assignee should be a single person or not. The following comment came up in the interviews when the default was discussed related to a question of whether the requirements management system could offer some kind of recommendations for example about who should be the issue's assignee.

"*…the problem with the team is there is still not someone ultimately responsible. The advantage is, I mean people you know go on vacation and I don't know, switch jobs and have other things to do, so we have mixed experiences with both, let's put this way, but that's, I don't think that there is the holy grail there, so.* "

When an issue is created by a requester, they can only add basic information about it. For example, priority, which determines the importance of the issue, cannot be set by the requester. The default assignee, who is often the maintainer of the component, gets a notification when a new issue is created for their component. They can then make a

decision about the issue's priority, although often the issues are prioritized by a triaging team, which will be described later in this section.

The highest priority an issue can have is P0, these issues are also called blockers. This is rarely used with suggestions, though, as one respondent said in the interviews:

"*P0 is like… We rarely have like must-must-haves, blockers, essentially P0 is a blocker for us, on the requirements-side, we usually only have those on the bug-fixing side, like a release goes out, and there's something that prevents the customer from being able to install – that's clearly a P0, the software does not go out. Whereas on the requirements-side, you have at most probably a P1, where you say okay, this is now very, very important for our next release,…*"

The other priorities defined in the system are P1 (critical), P2 (important), P3 (somewhat important), P4 (low) and P5 (not important). While the P0s and P1s should be fixed before other issues, the priorities are not very strict, there is no rule that someone could not work on a lower priority issue if there is a more important one.

The interviews revealed that prioritizing bugs is more straightforward than prioritizing suggestions. There are some helper questions that are meant to make the process of prioritizing bugs more objective [58], which are shown in Table 6. For prioritizing suggestions, these kind of helper questions do not exist, and it is thus more subjective. Both the business impact and the required implementation effort affect the priority of an issue. Something that has a big impact, but takes a very long time to do is of a lesser priority than a requirement that also has a big impact, but is relatively quick to implement.

"Is the cause of this bug or the use case(s) it affects a common scenario or a corner case? A common scenario is documented and/or at least moderately simple to understand for those familiar with the area. A corner case is often obscure and/or does not follow the prescribed usage of a feature."
"Is there a workaround?"
"How many users does this affect?"

Table 6 Helper Questions for Triaging Bugs [58]

The default assignee/component maintainer can be from outside of The Qt Company, and

therefore it is not required that they would go through the issues prioritizing them. All the issues should still be prioritized, so there is a process called *triaging*. All the issues in the public Jira goes through the triaging process. The Qt Company has a special triaging team consisting of two engineers. The team members are rotating, so that two engineers work in the team for a week, and then the week after two other engineers get the responsibility.

The triaging team's responsibility is to check issues that have not been evaluated. They should first check whether the issue has already been reported [58]. If the issue has not been reported before, and if it does not have a priority, the team should assign a priority to it. If, on the other hand, there is a Jira issue that was created earlier, and describes the same bug or suggestions, the team should link the issues, and change the *status* of the new one to closed. Status of an issue describes the state of the item, it shows whether the issue is being actively worked on, or if it has already been closed and so on. Sometimes the triaging team may also need to change the assignee of the issue, if for example the requester has chosen a wrong component when creating the issue.

It is also responsibility of the triaging team to make sure the issue has enough information, so that it can be worked on. In case the issue is missing some information, the team should ask the requester to provide the missing details. In these cases, the team changes the issue's status to one called *need more info.* When the reporter adds the required information in the issue, he or she should change the status back to *reported.* However, people often forget to change the status after adding the requested information on the issue, so the triaging team also goes through the issues in the need more info status to see if some of them can be put back to the reported status. According to the interviews, external requirements are quite often difficult to understand, for example, because the requester has not described the use case for the suggestion. It is the triaging step of the process that should sort out these problems.

Triaging can also be done by other members of the community with triaging rights, which can be acquired before becoming an approver. Also, the default assignee of the affected component, who is notified when the issue is created, can triage an issue. However, usually the triaging team is quicker to do it, because they are directly assigned for that job for one week at a time. Ideally, after triaging is done, the maintainer of the component would check that they agree with the triaging team's decisions, because as an owner of the component they may sometimes know better. However, in practice this may not happen very often.

When someone is assigned to work on an issue, that person becomes the *assignee*. Like requester, also assignee can be anyone with a Jira account. Assignee is the person who is writing code to actually implement the suggestion or a fix in case of a bug. The assignee should also update the Jira issue's status according to the work phases, e.g. change the status to *in progress* when he starts to work on the issue. The assignee has elevated rights regarding the issue they are working on, because otherwise their permissions could be too low to prioritize the issue, change its status, or basically do anything with it.

Issues can be assigned by the triaging team, by a maintainer, or by anyone in the community who has sufficient privileges. People can assign issues also to themselves, if they are interested to work on a specific issue. We call the person who assigns the issue, the *assigner.*

A Jira issue also has *watchers.* Watcher is someone who has done something with the issue, or has added themselves as a watcher in the issue. Watchers are notified by email when someone does something with the issue, so being a watcher is an easy way for people to track what is going on with an issue.

Jira users can also vote on issues they think are important. We call people, who vote for an issue, *voters*. Some of the respondents in the interview indicated that they didn't like voting very much and that it wasn't working very well. One respondent made the following comment about this:

"So we have voting, So you can vote for a bug, but arguably it's not working very well, people don't vote, or they vote for stuff that is… So if you look at what the highest votes in feature in Qt is – I haven't checked like the couple of weeks – but the last time I checked, it was an obscure support Z-files. Which is-, it's a nice feature and So on, but it got probably the votes, because it got exposed a lot, and is very old, So I wouldn't, I mean if you just go by the voting, we would probably not really serve the right purposes. So it's just a hint".

This indicates that some issues may get a lot of visibility or may have been around a long time and for these reasons have gathered many votes. There might be, however, other issues that are important for many people, but because they haven't gotten similar exposure, or they are newer, they may still have a smaller number of votes.

Another respondent was concerned that voting could decrease the human-to-human communication, stating:

"*...democracy and voting and tools having their place, but my gut feeling is many of the personal preferences of people is having natural communication with each other. So I don't want to go into my office and having votes, my vote pundits in front of me. So I want to go to natural communications and have fun with my colleagues, and not doing button presses.*

Anyone reviewing code is called *reviewer.* Code reviews are done in a system called Gerrit. The job of *reviewers* is to make sure the code contributions are of high enough quality so that they can be added in a software release. The approver role mentioned earlier also plays a significant role in reviewing the code. Code reviewer can be an approver, but they can also be a regular contributor. The difference is that while regular contributors can leave helpful comments, they cannot approve nor disapprove the change. Regular reviewers can give score +1 ("Looks good to me, but someone else must approve"), 0 ("no score") or -1 ("I would prefer that you didn't submit this"). In addition to those scores, approvers can also give a score +2 ("Looks good to me, approved") or -2 ("Do not submit") [59]. It is forbidden to give score -2 without giving a reason. With negative scores, a comment specifying the reason should always be provided.

Besides human reviewers, there is something called *early warning system*, which is a collection of automated checks for code submitted in Gerrit [60]. The warning system can give scores from +1 to -2. Score +1 means "Sanity review passed", while the other scores have same meanings as the scores given by the human reviewers [59].

As mentioned above, Gerrit is the system where the code reviews take place. The system can be accessed by anyone via a website at https://codereview.qt-project.org/. We list it here as an actor of the requirements management process, because once a code contribution has been approved, Gerrit automatically notifies Jira about it. This way anyone viewing the Jira issue directly sees that the code review step in the requirements management process has been passed.

In this section, we have been able to identify 12 actors/stakeholders in the requirement management process for the Qt software based on the interviews with The Qt Company's employees. The identified actors along with their actions are listed in Table 7.

Actor	**Description**	**Actions**
Requester	The person who created the issue (suggestion or a bug) in	• Create an issue. • Provide more information if needed, and change the

	Jira.	issue's status back to reported.
Assignee	The person who is assigned to work on the task.	• Implement the requirement. • Change the issue's status to signify the work phase. • Submit code for review.
Human reviewer	Reviews the code, leaving comments and possibly approving or disapproving the change (if they have sufficient rights).	• Leave comments. • Approve the change (only possible for approvers).
Early warning system	Collection of bots making sanity checks for the code to be reviewed.	• Run sanity checks and give a score based on the checks.
Approver	Someone who has contributed in the project enough so that they have gotten the right to approve changes.	• Review and approve changes.
Default assignee	Person, or sometimes a team, who gets a notification when an issue is created for their component.	• If triaging team has prioritized the issue, ideally check if it is correct. • If triaging team has not prioritized, do the prioritization.
Maintainer	Responsible for a component. Maintainer is often the default assignee of the component.	• Resolve conflicts regarding the component, if there are any.
Watcher	Can be any Jira user. When a user does something with an issue, they are automatically added as watchers in it. A user can also manually add themselves as a watcher in an issue. Watchers get a notification when the issue is changed.	• Add oneself as a watcher. One can also become a watcher automatically by changing an issue.
Voter	Thinks the issue is important, and shows his/her support by	• Voting.

	giving their vote for the issue	
Triaging team	Consists of two engineers of The Qt Company. This team has the main responsibility for triaging issues. Triaging can be done also by other members of the community as long as they have acquired triaging rights.	• Make sure the issue has not been reported before. Close the newer issue if it has been reported, and link the two issues together. • Prioritize the issue. • Make sure the issue has enough information to be worked on. If it does not, ask for more information in the issue's comments and change the status of the issue between "need more info" and "reported". • Close the issue if it is for a too long time in the "need more info" status.
Assigner	Assigner can be someone from the triaging team, it can be the maintainer, or it can even be the assignee himself	• Assigns the issue to someone (an assignee).
Gerrit	An online system for reviewing code	• Notify Jira when an issue has passed the code review process.

Table 7 Actors in the Qt software's Requirements Management Process

4.2 Workflow of a Requirement

With the help of the interviews we have now found the different actors in the requirements management process of the Qt OSS project. We have also described the actions and responsibilities of those actors. In this section, we aim to answer the research question RQ3c), which was:

- What is the flow of the requirements management process, i.e. in what order do the actions occur?

From Jira issue's point of view, the issue's status indicates the phase of the issue in the workflow. The statuses for the Qt software are *Reported, Need More Info, Open, In Progress,* and *Closed.* The statuses, as well as descriptions of events causing the status to change from one to another, are presented in a workflow diagram in Figure 7. The diagram can be viewed by any logged in user in The Qt Company's Jira by finding a requirement or suggestion in the project for the Qt software, i.e. project named Qt (QTBUG), and choosing View Workflow option. In Figure 7 the diagram is slightly modified, so that the information about transitions from one state to another are all visible in the same picture.

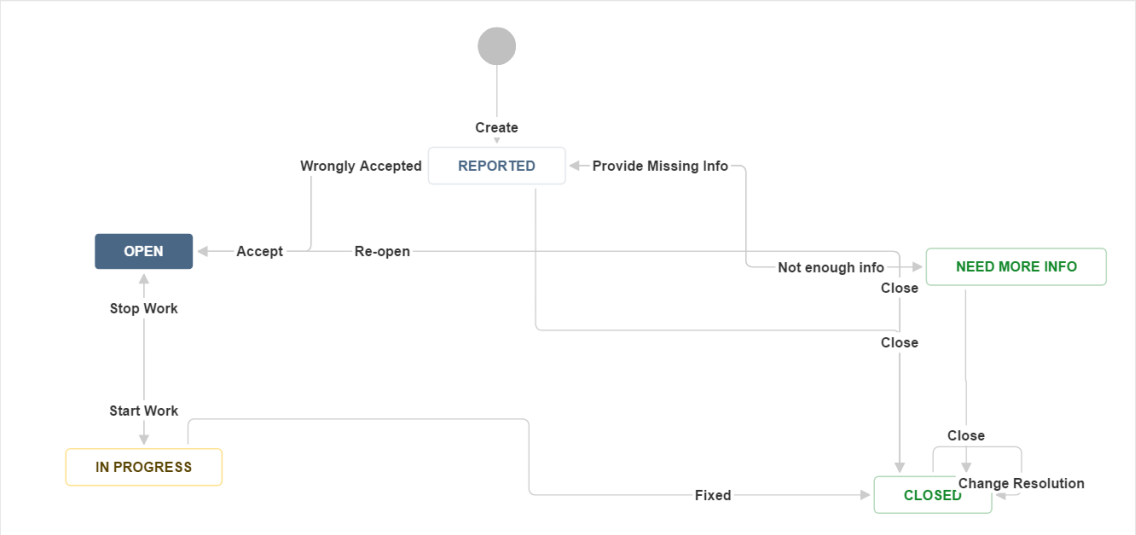

Figure 7 Issue Statuses and Transitions for Bugs and Suggestions in Qt software [61]

An issue automatically gets status Reported, when it is created. As discussed in the previous section, the status is changed to Need More Info, if some information is missing. If the person triaging the issue thinks that the issue has enough information, the status is changed either to Open or Closed. The status could be directly changed to Closed from Open or from Need More Info for example, if it is discovered that the system already had an issue about the requirement.

When a developer (assignee) starts working on an issue, they change the status to In Progress, which tells anyone looking at the issue that it is actively being worked on. If the developer stops working on the issue, for example if they are working on something else for a while, they should change the status back to Open. Also, when the developer has implemented the issue, and submitted it for review, the status is changed back to Open. In the past, the process had a separate verification status in Jira, but now the issue waits in the open state until the change has been reviewed in Gerrit, and then the issue is closed.

Jira has a concept called *resolution* to signify the reason for closing the issue. The resolution is *unresolved* as long as there is some work left with the issue, and once the work is finished, the issue's status is changed to closed and a suitable resolution is chosen. In the normal flow, the issue would first be reported, then worked with, and finally the status would be changed to closed, and resolution to *done* or *fixed* when the bug had been fixed, or the suggestion implemented. As described above, the change would also have to go through code review before the Jira issue would be closed. However, the flow does not always go like this. The issue might have been reported before, or a suggestion might

be impossible to implement, etc. For these situations, Jira has other resolutions. The resolutions an issue can have are *done, out of scope, duplicate, incomplete, cannot reproduce, moved, invalid, fixed* and *unresolved* [62].

To get an idea of how big portion of the issues get fixed or implemented, and how big portion are invalid or otherwise get some other resolution, the Qt software's bugs and suggestions with status closed were searched from The Qt Company's Jira, collecting the number of issues with given resolution. Figure 8 shows the observations of this analysis as a bar chart.

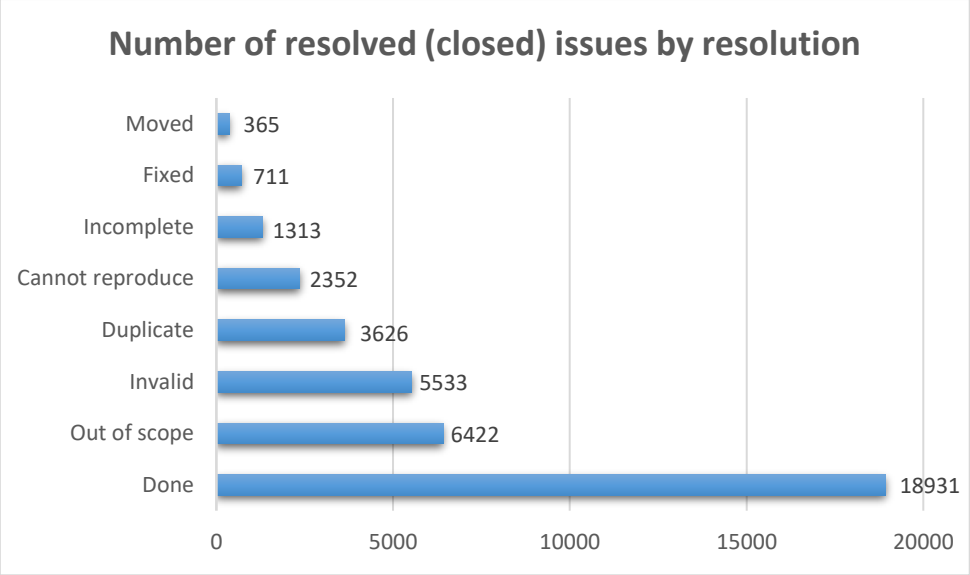

Figure 8 Number of Qt software's issues by resolution. Including bugs and suggestions closed before June 10th 2017. Data obtained from [63]

The resolution names *done* and *fixed* may be confusing. Both of them indicate that some code was added or changed in the software, but *fixed* seems to suggest that something was corrected, i.e. it seems like a suitable resolution for a bug. Resolution *done*, on the other hand, seems neutral, and seems suitable for both bugs and suggestions. A closer investigation shows that out of the 39253 closed issues, 35323 or about 90% are bugs, and only 10% are suggestions. Only 2% of the bugs, and 0.3% of suggestions have resolution fixed, whereas the most common resolution (done) was used with 50% of the bugs, and 31% of the suggestions.

Overall, 50% of all the suggestions and bugs had resolution done or fixed. In other words, half of the issues were valid, and had been fixed or implemented, while the other half had been discarded for one reason or another. In the interviews, a respondent said that they did not feel like The Qt Company would have the similar problem as a more traditional

company may sometimes have that people wouldn't inform them if the product is not working for them. The interviewee felt that people want to talk to The Qt Company, and that it was more of a problem to keep up with all the discussion. This may be also reflected in the number of created Jira issues that are not implemented. Also, as noted earlier the interviewees felt that the issues were sometimes difficult to understand for example because the use case was not explained in the issue. This may be part of the reason for the 6% of issues with resolution *cannot reproduce*, and possibly the invalid category also has some of these issues.

In Figure 9, a swim lane diagram displays the requirements management process in case where a suggestion is implemented or a bug is fixed. As seen in Figure 8, there are many reasons why the issue may not get implemented. In most of these cases the triaging team should notice that the issue cannot or should not be implemented, and the flow would go as shown in Figure 9 for a duplicate issue. It is possible that in some cases, the reason why a request should not be implemented, is only found later in the process, but these cases are highly case dependent, so they are not shown in Figure 9. We also expect these cases to be rare since the triaging process is in place, and these situations did not come up in the interviews.

While in the process shown in Figure 9, the maintainer checks that they agree with the priority set by the triaging team, we should note that this is the optimal situation. According to the interviews, this step may often be skipped.

In Figure 9, the actors are shown in the vertical axis, and the flow goes from left to right and from up to down. Voters and watchers have been excluded from the diagram, because they don't really affect the flow of the requirement, and it is highly issue dependent at which point in the flow a watcher or a vote is added.

In the diagram, actions are represented by rectangular boxes with rounded corners, and the diamonds represent decision points in places where the flow can go to multiple directions depending on a condition. Since bugs and suggestions are handled in a similar way, Figure 9 is applicable for both issue types.

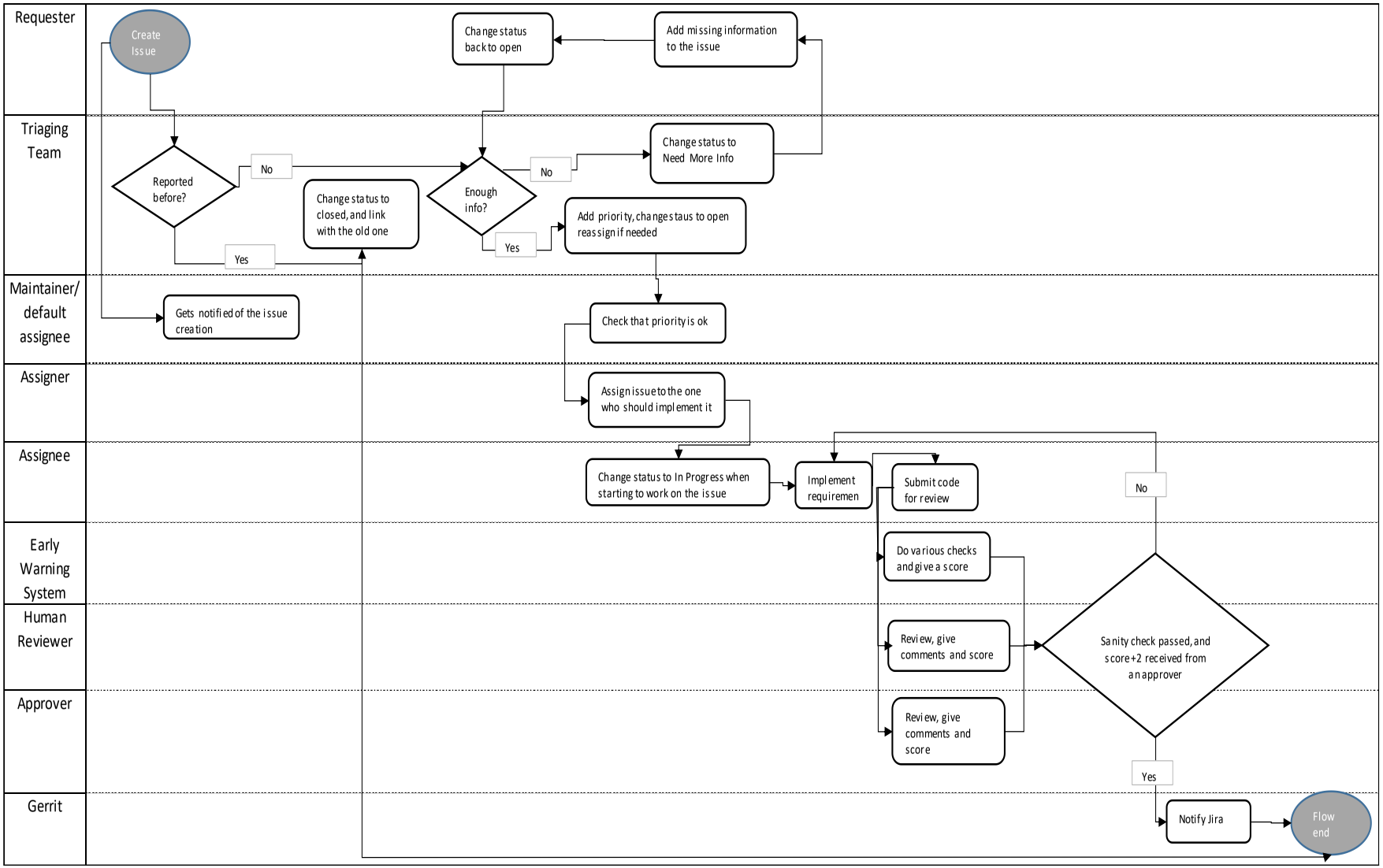

Figure 9 Roles of stakeholders in the reauirements management work flow

From the figure, we see that there are two iterative parts in the process. In the beginning, the requirement can go back and forth between the triaging team, and the requester, if the original requirement did not have enough information on it. At this point the status of the issue, which were shown in Figure 7, moves between open and need more info. After this part, the requirement should have detailed enough description so that a developer can start working on it. Next, the issue is assigned to a developer (assignee) by an assigner. When the developer starts working on an issue, they change its status accordingly.

Once the developer thinks they are ready with the implementation, they submit the code, and people can review it in Gerrit. The maintainer of the component is automatically added as a reviewer in Gerrit, but it is recommended that the developer would also add other reviewers. This can be difficult for new contributors, because they may not know who they should add as a reviewer. In practice, new contributors can in this case search similar issues in Gerrit and see who have been reviewing those. If suitable reviewers are not found they can also try to find reviewers through the mailing list.

The reviewing step is the second iterative part of the process. If either the early warning system, or the human reviewers give a bad score for submitted code, the assignee has to make corrections, submit the code again for review, and so on. Finally, when the change has passed the sanity checks, and an approver has given score +2 to the change, it can be added to an upcoming software release, and the Jira issue is closed. Gerrit also automatically notifies Jira when the code review has been passed, and this information is added to the Jira issue.

4.3 Sample Issues

In this subsection, we report observations of two issues from Jira. The author of this thesis chose one bug and one suggestion as the sample issues. The bug we investigated can be viewed in The Qt Company's Jira at https://bugreports.qt.io/browse/QTBUG-17888, and the suggestion in the same system at https://tinyurl.com/y95j26mx.

Sample 1

Our first sample issue was a bug report suggested by The Qt Company's employee (the community manager) as a good example of a basic requirement. The issue describes a situation where the memory usage increased more than it should when running a demo

application using the Qt framework. Code of the demo application was also attached to the issue. After the bug was reported, the issue's status was changed to open, and priority to P3 by a person who did not change the issue after this. The assignee changed the issue status back to reported, and right away to need more info, because he could not reproduce the issue. The assigner and reporter then exchanged some comments, and the reporter eventually added a video showing how the issue occurred. The reporter then also changed the status back to reported. User called Gatis Paeglis also commented on the issue getting a segmentation fault when running the demo application.

After those comments, no one changed the issue for a long time. Finally, the status was changed once more to need more info by a user called Giuseppe D'Angelo, who did not seem to have anything to do with the issue before this. The status change was accompanied by a comment asking the reporter to test again with a newer version of the Qt framework. Only a few hours later Giuseppe found the mistake in the code of the demo application. The mistake was a wrong ordering of two lines of code. Since the problem was in the demo application, rather than the Qt framework, no development was actually needed, and Giuseppe closed the issue with resolution invalid.

This issue was therefore one of those where no code changes were needed. In this case, it was because the test case itself had a bug in it, not the Qt software. One observation we did about this issue was that it stayed unresolved for a very long time. The original comments on this issue took place in a period of about half a year. After this, however, there was timespan of 1259 days when no one changed the issue, before Giuseppe finally solved it.

We also noticed from the issue's history that the workflow has been changed twice since 2011, when this issue was created. First, in 2013 workflow version 2.0 was taken into use, and then in 2017, version 2.1.

Sample 2

For the second sample issue, a suggestion was chosen, since sample 1 was a bug. We wanted to select a recent issue, so that it would represent the currently used workflow. The issue also had to be resolved, so that it would represent the whole flow of the process. With these goals in mind, we used Jira's web interface to search for suggestions in the Qt project having resolution done or fixed. The sample issue was then the most recently resolved issue from the resulting list. The search was conducted at 3pm 10th of June 2017,

and the resulting issue (https://tinyurl.com/y95j26mx) had been resolved the previous day.

This issue was about modifying a configure script used to build the Qt software. The requirement was that it should support new version of C++. After the issue had been reported, user called Oswald Buddenhagen, who is the maintainer of the Qt software's build system, changed the issue's status to open. He also set the priority, changed the component information, and assigned the issue to user called Thiago Macieira. After this Thiago asked in the issue's comments "*Is there any reason to allow people to select which version to use?"*, but got no response (at least not in Jira). All of the above happened in less than a day after the issue was reported. However, during the following 814 days there was no activity regarding the Jira issue. Then finally Oswald closed the issue adding also the code commit identifying hash, fix version, and resolution done to the issue.

Overall, the flow followed by the sample issues seems to be similar to the general flow described in section 4.2. The first issue was a bit different because our model mainly described an implemented issue, but the sample one was invalid. The second issue represented the whole flow, as we picked one that had a resolution *done.* This issue was prioritized by the maintainer instead of the triaging team. This possibility came up in the interviews, even though it was said that more commonly the triaging team does the prioritization.

One observation we also made, is that with both issues analyzed here, there was a long period of time with no activity in Jira before the issue was finally closed. This could be because the priorities of the issues were P2 and P3, the higher priority issues are possibly finished quicker. Also, the system had more than 14 000 unresolved bugs and suggestions in the Qt project, so that may be another reason why some of the issues take a long time to resolve.

Since the number of unresolved bugs and suggestions seemed high, we analyzed them more closely to see whether the unresolved issues were mainly low priority issues, or if the unresolved issues' distribution by priority was the same as with all the issues in the system. The data was obtained by conducting multiple searches in The Qt Company's Jira searching for issues of the Qt software. In the first search Jira's search filters were used to return all issues of type bug or suggestion in the Qt project. This resulted in 56 750 issues. After this the issues were filtered by priority, one priority at a time to see how

many issues the system had with a given priority. Similar searches were conducted for the unresolved issues. Total 14 753 unresolved issues were found in the system. The data was obtained on 20th of August 2017 in the afternoon, and the results are summarized in Figure 10.

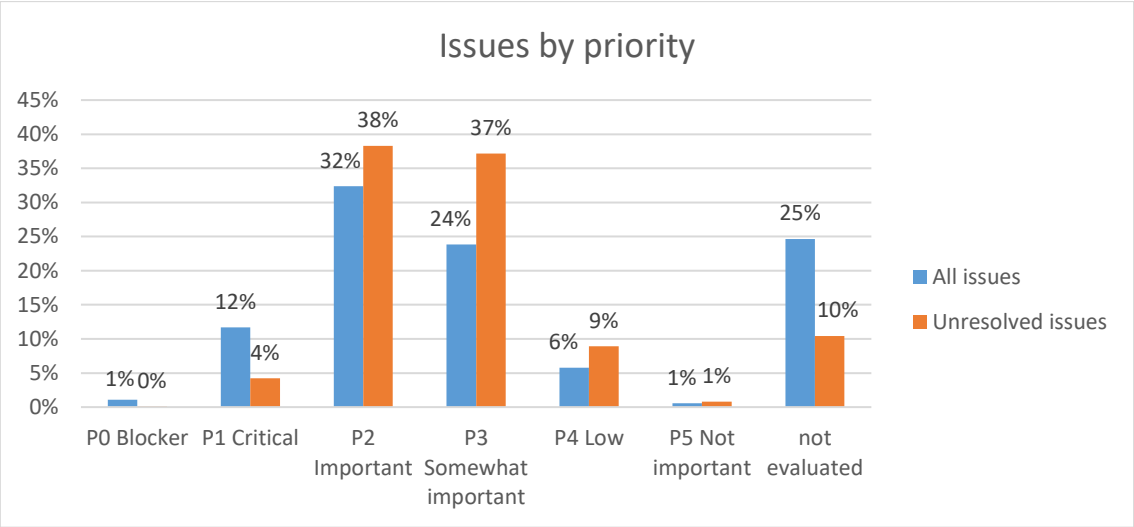

Figure 10 The Qt software's issues by priority as of 20th August 2017. Data obtained from [63].

In Figure 10, the blue bars show the relative number of all issues and their priorities, while the relative number of unresolved issues is shown with orange bars. The percentages were calculated for the unresolved issues by searching the number of unresolved issues by each priority from P0 to P5 (a separate search was done for the *not evaluated* ones), and dividing the number with the total number of unresolved issues. The same process was applied for all the issues, the only differences being that when searching the issues, they were not restricted by them being unresolved or not, and that when calculating the relative numbers, the number of all the issues was used as a divisor.

From the figure, we can see that 32% of all the issues in the system are considered as important, and 24% as somewhat important. For both of these priorities the number of unresolved issues is higher, 38% and 37%, respectively. Also for the priorities P4 and P5, the relative number of unresolved issues is higher than the relative number of all issues. However, the unresolved issues have relatively lower number of higher priority issues, i.e. issues with priorities P0 and P1 (0% and 4 % for the unresolved issues and 1% and 12% for all issues, respectively). These numbers indicate, as could be expected that the higher priority issues are resolved quicker than the lower priority issues. It looks like the issues with priority P0 or P1 receive special attention, while other priorities are not considered as important.

Furthermore, of the blocker issues, half had been created within past 69 days. Two of the oldest blockers had been created many years ago, but the priority had only been changed to P0 in April 2017. All other issues with P0 priority had been created less than a year before conducting this analysis. It seems that while there is a high number of unresolved issues in the system, the number of unresolved issues with the highest priority is relatively low.

5 Discussion

The Qt open source software project seems to manifest a coupled open innovation process. As an inbound activity, The Qt Company brings ideas and knowledge from the community into the company and into the Qt software. On the other hand, as an outbound activity, The Qt Company's engineers also actively develop the product, and in that way, bring their ideas and skills available in the market, where both other companies and individual developers can benefit from them. It seems that The Qt Company is also participating in a strategic network consisting of partner organizations and individual developers. Several organizations in the community are selling complements like training and consulting, and many organizations are also participating in development of Qt software.

It seems that The Qt Company is using two of the open innovation strategies related to OSS and identified in section 2.2.4 in the literature review. Firstly, The Qt Company is following the selling complements strategy with its dual licensing approach. Secondly, the OSS community, The Qt Company and the partner companies seem to be participating in pooled R&D effort.

While The Qt Company has an open innovation process, it also has software projects that are not visible for the community. Even though these private projects are managed in the same systems as the open source project, the permissions have been set up so that the community cannot see these projects. These projects are therefore managed with a more closed innovation approach. Some of these projects can include confidential information, which makes the closed approach necessary in these cases.

In the open source project, most of the tasks are shared between The Qt Company and the community. Everyone in the community can report issues, and if they have sufficient permissions also approve code changes, assign and prioritize issues, and so on. However, the triaging team consists always of The Qt Company's employees, and can thus be seen as a service the company provides to the open source project. The community members can also participate in the triaging process, but there is always a two-person triaging team provided by The Qt Company. Besides The Qt Company's employees' development efforts, the work of the triaging team can also be seen as an outbound innovation activity from The Qt Company. They bring their knowledge and expertise into the market in their triaging decisions.

In the literature review part of this thesis we noted that the community around an OSS project can have a hierarchical or flat structure. The Qt OSS project is large. There are about 32000 accounts in The Qt Company's, and the project has more than 60 weekly contributors. In this big a project, it is probably beneficial to have some hierarchy in the community structure, like the Qt software does. The structure is actually fairly similar to the one presented in Figure 3. Probably the biggest differences are that in the Qt project the core developers are called maintainers, and there is the distinction of approvers, which is not shown in Figure 3.

The relationship between The Qt Company and the community seems symbiotic according to the categorization by Dahlander and Magnusson. It seems that The Qt Company is very active in developing the open source project, they have a sizable R&D team, they provide the triaging team for the project, and also most of the maintainers are employees of The Qt Company. Also, the chief maintainer is from The Qt Company, and he has the last say in development decisions. However, some of the maintainers also come from other companies, and decisions such as when the Qt Software is released, are done by the community. Software development and fresh ideas are not the only thing The Qt Company gets in return from the community. The Qt Company's various partner companies provide services that make the Qt software more attractive for customers who may require additional training, consulting or other services.

While The Qt Company is strongly participating in developing Qt software, the same rules apply to The Qt Company's employees as to any other members of the community. This means that the employees do not automatically have for example the approver status, but they have to contribute enough to gain it.

An interesting observation from the interviews was that external requirements were quite often difficult to understand, for example, because the requester had not described the use case for the suggestion. The comments about the difficulties to understand the requirements were interesting, because in the literature review we found a notion that misunderstanding requirements in OSS would typically be minimized. On the other hand, it was also said that OSS developers are also users of the software they are developing. This is also what Fitzgerald said about traditional FOSS projects. However, the Qt software perhaps more closely resembles what Fitzgerald called OSS 2.0, at least in that many of the developers are paid to work on the project, and that the requirements are not universally understood.

There can be a few other reasons for the difficulty of understanding requirements in the case of the Qt software. Firstly, since many of the people developing Qt software are employed by The Qt Company, or by some other company, their motivation may be primarily external, meaning the payment they get for their work, rather than making better a piece of software that they would need for their own purposes. If the employees are not themselves using the software they are developing, it seems natural that understanding requirements is more challenging than if they were developing a product they would use themselves. Secondly, Qt software is an application framework used in more than 70 industries. This means that even if the developers employed by The Qt Company would use the Qt framework themselves, it would be very difficult for anyone to be familiar with all the use cases the software is used for across different industries, companies, and individual developers. According to the interviews, missing use cases were often the reason for an issue being difficult to understand.

Highly valuing open communication was observed in the interviews in a few separate occasions. As mentioned in the previous section, one of the interviewees was concerned that a voting system might decrease the natural communication, which seems to suggest that open communication is seen to have value in itself. On the other hand, the requirements management process of the Qt software seems to rely quite heavily on open communication.

In this thesis, we did not find the Qt software's requirements being managed with a requirements engineering process traditionally recommended in the literature, i.e. a process having phases for elicitation, modeling, requirements analysis, validation and verification, and requirements management. The focus of this thesis was mostly in the requirements management part, though, so the other phases were not fully investigated. For example, we did not investigate whether there is some kind of requirements elicitation effort before a requester creates an issue for the Qt software. At least the validation & verification step does not seem to be very formal, since it depends on the subjective judgement of the triaging team whether an issue has enough information.

As often in OSS projects, also in the Qt project the requirements are unstructured. There are only a few pieces of information required when an issue is created in Jira, and the description does not have any structure, it is just text allowing the requester to describe the issue in any way they want. It seems that there is a tradeoff of making it easy to create new issues, by only having few pieces of required information, and on the other hand

getting well enough specified issues from the community, so that they can be worked on. It is easy to create new issues for the Qt software, and the triaging team's responsibility is then to make sure the issues have enough information. In Qt project, communication is mainly done through online channels, for example through mailing list discussions and Jira issues, which is also common in OSS projects according to the literature.

One aspect of the requirements management process of the Qt software that more closely resembles a traditional software project is that the requirements are gathered in a central repository, in Jira. We should note that this study only focused on the part of the requirements management process where the issue has already been created in Jira. One interesting research direction might be to investigate how the requirements are actually formed through discussions by email as well as discussions through other channels. Since we did not investigate this, it is possible that the issues are sometimes discussed for example by email or through other communication channels before they are created in Jira.

5.1 Implications of the Empirical Results

The Qt Company uses Jira as their main tool for requirements management. The requirements management tool and its features are tightly linked in how the requirements management process can be designed. If another organization tried to manage their requirements in the way described in this thesis by using some other tool than Jira, it would be important to make sure the tool supported some key functionalities required by the desired process.

An example of a Jira feature that may be important for the requirements management process of the Qt software, is support for communication between an issue's stakeholders. As described in Table 7, in Qt's OSS project an issue can have stakeholders such as assignee, triaging team, requester, etc. Jira supports communication between an issue's stakeholders by allowing them to add comments on the issue directly in Jira, and the watchers of the issue (i.e. Jira users who have subscribed to be notified of changes/modifications to the issue), will get an email notification about the new comment, and can of course view the comment in Jira as well.

In Qt software's requirements management process, Jira's support for communication seems to be especially important in the early phases of the process when the persons

triaging the issue (usually a triaging team consisting of two employees of The Qt Company) are trying to make sure the requirement has sufficient information. If the requirements management tool didn't support communication directly, it could be done for example via email. However, people might be less prone to communicate with each other if they could not do it from the tool they already use to manage the issues. Another downside of using only email for communication, is that there would be no central place where everyone could view the discussion. In Jira, not only people who are actively working on an issue can see the discussion, but anyone who develops an interest towards an issue can view it in Jira. For example, someone who has not been following an issue about an erroneous behavior of the software, might encounter the error themselves, find the issue in Jira, see the discussion, and contribute in it with the details on how the error occurred for him/her.

Because of the relationship between Jira's features and Qt software's requirements management process, the process is probably most suitable for projects using Jira or a tool with similar capabilities for managing requirements. Also, as the Qt software is divided in many different components, the component maintainer can be used as the default assignee. In a project with different kind of architecture and management structure, another kind of approach might be needed to find the (initial) assignee.

An organization leading an open source software project needs to decide how much control they want in the project, and how much autonomy they want to give to the OSS community. The Qt Company's approach to the Qt OSS project is democratic in the way that The Qt Company's employees have the same rules as other members of the OSS community, e.g. they have to contribute enough in the project before getting more permissions over the project such as permission to approve code changes. In some situations, another company leading an OSS project, might decide that they want to have more control over the project. They might, for example, give the company's employees more privileges than the community members, or they might develop certain parts of the software completely inhouse, i.e. not allow the OSS community to make any changes in some parts of the software. In this kind of a situation, the requirements management process described in this paper would need at least tweaking and some thought on how the changes should be implemented.

As noted earlier, the Qt software project has more than fifty thousand issues with type suggestion or bug in the Jira. Therefore, it seems that the requirements management

process described in this thesis can work for an open source software project with a big number of issues. However, we must note that since this thesis is a case study focused on a single OSS project, it is not possible to generalize the results. Therefore, we cannot, based on the results of this thesis, create an exhaustive list of characters describing an open source software that would be a good candidate for the requirements management process outlined in this thesis. In general, a downside of the case study approach is that a theory based on a case study may lack generality [64].

An example characteristic of an OSS project that can have an effect on the requirements management process is the criticality of the developed software. It may have an effect on how heavy review process is needed when the program code is changed as a result of new features or fixes to existing problems. If an OSS project has been set up to provide some critical systems for a space shuttle, a heavier review process may be needed than what is used in the case of the Qt software. On the other hand, if the software is less critical, and maybe a completely new product where it is important to get users' feedback on new features quickly, a lighter review process may be beneficial to cut down the time of introducing new features.

5.2 Comparison of the Theoretical and Empirical Findings

Table 8 summarizes the findings of this study. The first column shows the overall topic of the finding, whereas the second column describes a finding of that topic from the literature. The third column explains how the finding seems to apply in the case of The Qt Company, and the Qt software, according to the results of this study. Finally, the most important literature references related to each of the findings from the literature, are listed in the right-most column. Table 8 is otherwise the same table as Table 5, but the third column is only presented here in Table 8.

Topic	Main Findings	Situation in Qt	References
Open innovation (sections 2.1.2 and 2.2)	Traditionally big corporations had internal R&D departments for innovation, nowadays companies are opening up the innovation process by including external stakeholders.	The Qt Company's innovation process, especially in the Qt software project studied in this study, is open including external developers and other companies. The Qt Company has, on the other hand, other projects which may, for example, have confidential client information, and thus need to be managed with a more closed approach.	[5, 7, 10, 18]

	Three core innovation processes have been identified. They are inbound and outbound innovation and coupled innovation process.	The Qt Company uses the coupled innovation process, combining inbound and outbound innovation.	[14, 5, 13, 18, 2, 8]
	Inbound innovation: external resources like ideas and knowledge are sourced e.g. from suppliers, customers or competitors, or acquired by purchasing them from the market place.	The community, i.e. individual developers as well as other companies participate in developing the Qt software, and this way the external ideas and knowledge flow both into the Qt software and through the software and the interactions between external developers and Qt Company's employees also in the Qt Company itself.	
	Outbound innovation: Organization's internal resources (e.g. a proprietary technology) is sold or revealed. Open sourcing a previously closed source software is an example of revealing.	The Qt Company's own developers participate in developing the Qt software, and through their development efforts bring their ideas and knowledge to the market, where it can benefit The Qt Company's partners and customers as well as competitors.	
	Coupled process: Combining both inbound and outbound innovation. Companies join strategic networks where they develop innovations together with their partners.	The community around the Qt software seems like a strategic network for The Qt Company. In addition to the developers, and other companies who develop the Qt software further, there are organizations who sell complements such as training and consulting.	
Open source software (sections 2.2.4 and 2.3)	Software that has been published under an open source license [15, 7], i.e. a license that allows anyone to view, use, and modify the software's source code.	The Qt Company follows a dual licensing strategy. The Qt software is available both under GPL and LGPLv3, and a commercial license.	[15, 7]
	Some members of an open source community can be much more active than others, and it may not even always be clear who is a member of the community and who is not. The community may be hierarchical so that some members have more permissions than others, or everyone can have the same	In the Qt software project the main roles regarding permissions are: contributor (the most basic level, no admission process), approver, and maintainer. At the top of the hierarchy, there is a chief maintainer who has the last word.	[34, 19, 37]

	permissions.		
	Some OSS projects are developed only by individual developers, but many are somehow supported by a company or even by multiple companies.	While The Qt Company is leading the Qt software project, many other companies and also individual developers participate in its development.	[16, 34, 35, 22, 39]
	Some OSS strategies used by organizations have been identified: pooled R&D, spinouts, selling complements, donated complements.	From these strategies, we found The Qt Company following the pooled R&D and selling complements strategies.	[21, 24]
Requirements engineering (section 2.4)	Traditionally, a multi-phase approach to requirements engineering has been recommended in the literature. Depending on the research paper, the phases can be for example elicitation, modeling, requirements analysis, validation & verification and requirements management. The goal is to understand and analyze the requirements well before system development is started.	It seems that the traditionally recommended requirements engineering process is not completely followed in the Qt software project. However, in this thesis we did not thoroughly investigate the whole requirements engineering process, but focused mostly on the requirements management phase. Nonetheless, it seems that there is no strict process for example for determining the validity and to verify the requirements. The triaging team asks more information from the requester if they think some information is lacking, but there isn't any more formal process in place.	[43, 40, 41]
	In traditional software development, the requirements are also managed in a central place. The requirements document or repository is inspected for completeness, internal and external consistency with the domain and stakeholder needs. Requirements traceability, consistency, completeness, and internal correctness are valued high.	In the Qt software project, the requirements are gathered in a central place, an online repository using an issue tracking tool Jira. Since the tool is online and open, the issues can be viewed by anyone with an Internet connection. It seems that in the Qt software project, communication is valued higher than the completeness of requirements.	[46, 20]
	In OSS, the requirements engineering approach is often more lightweight and less formal as compared to the requirements engineering approach in more traditional software projects. The requirements are often spread	In this study, we focused only in requirements that were already in Jira. It should be noted that some of the requirements might be discussed for example in email discussions or online forums, before they make their way into Jira. However, in the Qt	[41, 20, 24]

	around in online forums, email discussions, and so on.	software project, eventually at least all the requirements that are worked on should be in Jira.	
	Companies seem to be looking increasingly into OSS. They also increasingly paying developers to contribute in OSS. It may be that in some of the projects developers do not know the requirements as well as used to be the case in OSS, and thus increasing attention is being paid in the analysis and design phases of OSS.	Paid developers are working on the Qt software both in The Qt Company and other companies, but there are some individual developers as well. The interviews indicated that the developers found the requirements sometimes difficult to understand, because the use cases were not well described. A possible reason for this may be that the developers are not familiar with all the use cases of the software maybe because they are not using the software for their own purposes, or maybe because the software is used for such a diverse set of use cases that the developers cannot be familiar with all of them.	[24]

Table 8 How the Findings from Literature Apply to The Qt Company and to the Qt software

This section has provided discussion about Qt software's requirements management process, and the open innovation approaches The Qt Company is using with the help of the Qt software. The following section summarizes and concludes the thesis.

6 Conclusion

In this thesis, we have described the requirements management process of the Qt open source project. We did first a literature review where open source development, open innovation, and requirements engineering were discussed. The purpose of this discussion was both to provide context for the empirical part of the study, as well as to answer our first research question about open innovation in general.

Our research question and its sub questions for the literature review were:

- RQ1: What is open innovation with OSS based on literature?
    a) What kinds of approaches can be used for implementing open innovation?
    b) How can the OSS development model be used as an open innovation strategy?
    c) How is requirements engineering different in OSS compared to other software projects?

With regards to open innovation we found that the academic interest towards open innovation has been constantly growing since the concept was introduced in 2003, and the commercial interest seems to be growing as well. We found three core open innovation processes identified in the literature. These were the inbound, outbound and coupled open innovation process. The two first ones differ in whether the ideas, resources and knowledge are flowing from the company to the market or from the market to the company. The coupled process, on the other hand, combines the two others.

We found that open source development is used in many software projects, some of which are competing head to head with their (sometimes) closed source competitors. Commercial organizations can be involved in OSS projects in a hosting, supporting or collaborating roles. It was also found that there are several open innovation strategies an organization can use with OSS. These were pooled R&D, spinouts, selling complements, and donated complements.

From the requirements engineering literature one observation was that in the OSS projects requirements tend to be less formal than in traditional software engineering projects. On the other hand, it was found that OSS developers are often also users of the software they develop, and thus should understand the requirements well. We found, however, a notion that developers are increasingly paid to develop OSS, and that nowadays in the OSS

projects the requirements may not be as well understood as they used to be.

Our empirical case study aimed at answering the following research questions:

- RQ2: What open innovation approaches are used in the Qt OSS project?
- RQ3: What is the requirements management process in the Qt OSS project like?
    a) Who are the stakeholders/actors?
    b) What actions do they perform?
    c) What is the flow of the requirements management process, i.e. in what order do the actions occur?

To answer research questions 2 and 3, we used multiple data sources including interviews, and publicly available online resources. As a finding to RQ2, we found The Qt Company using multiple approaches to open innovation in the Qt software project. The pooled R&D strategy, selling complements strategy, and from the core open innovation process, the coupled process, seem to be present.

With regards to the coupled open innovation process, The Qt Company is giving out its know-how and resources by developing the product and providing support functions such as the triaging team. On the other hand, The Qt Company is participating in the community having several other organizations as well as individuals who are also developing The Qt software. The open source community was found to be hierarchical, and its structure to resemble quite closely the hierarchy we found in the literature, and presented in Figure 3.

As a finding to the research questions RQ3b) and RQ3c) we identified 12 actors or stakeholders and their actions summarized in Table 7. Some of the actors can work with an issue for a long time, for example, it can require long discussion between the triaging team and the requester before sufficient mutual understanding has been reached. Other actors, for example voters, do not change the flow of the requirement, and an issue may not have any voters if no one has voted for it.

To answer the last sub question of RQ3, the flow of the requirements was presented as a swimlane diagram in Figure 9. From this diagram, we were able to visually identify two iterative parts in the process, as well as to see which actors are working on the issue at a given stage of the process. The notion from the literature that in OSS the requirements

are not very formal seem to be applicable to the Qt software, as a Jira issue only has few required fields which have to be filled by the requester. Perhaps because of this, the requirements are not always clear enough for an assignee to start working on them when they are created in the system. For example, the requester may not have presented the use cases, as was observed from the interview responses. Therefore, the iterative part of the requirements management process in the beginning is designed to clarify the requirement.

At the end of the process there is another iterative part, where both human reviewers and automatic sanity checks are used to make sure the submitted code is of high enough quality, so that it can be submitted in the product. If the reviewers find some shortcomings, the developer is informed, and after fixing the problems, they can submit the code again for review.

To find additional issues we might have missed with the interviews and online resources, we took two sample issues of the Qt software, and investigated how those issues went through the process. One observation from this was that both of the sample issues had a phase in their lifecycle where no one seemed to do anything about them, and this phase lasted for more than two years for both of the issues. This raised a question of whether it was normal that it would take years to complete an issue, but we thought that the reason why the example issues took a long time to complete may also have been due to the fact that they didn't have the highest priority (their priorities were P2 and P3).

To see whether there were many unresolved high priority items in the system, a brief analysis of the issues and their priorities was done. We found that there was relatively lower number of unresolved high priority issues (with priority P0 or P1) as compared to the number of all the high priority issues in the system. Also, unresolved issues with priority P0 had been created, or the priority had been changed from a lower priority to be P0 within the past year, so it seemed that the higher priority items are worked on quicker than the lower priority items. Thus, there's a good possibility that the long periods of time where nothing happened to our sample issues were due to their relatively low priorities.

An interesting future research direction might be to study the temporal aspects of the requirements management process, i.e. the length of time the issues spend in different parts of the requirements management process, the possible bottlenecks and problems, and ways to make the process more efficient. Especially, this kind of an analysis comparing the processes in multiple open source projects could help to find aspects that

make a requirements management process efficient in an OSS project. A major challenge, though, in this kind of a study would be to control the effect of different properties of the projects, such as the size, audience, skill of developers and the governance model etc.

With these results, we have described the requirements management process of the Qt software project. The case study approach, though, has some limitations. As we only observed one project, the process that we found cannot be generalized to other projects. Other projects may have totally different ways to manage their requirements. In the future, the requirements management process of the Qt software, which we identified in this thesis could be compared with the requirements management processes in other OSS projects in search for a set of best practices that could be applicable in various OSS settings.

Another limitation is that interviews were a major data source for this study, so most of the results are from self-reported data. We tried to tackle this challenge by also using other data sources including information from The Qt Company's website, Qt software's wiki page, a follow up interview, and actual sample issues from Jira. In the future, the sample issue approach could be extended by conducting a statistical analysis on a big number of issues from Jira. This could be combined with the above mentioned temporal analysis of the requirements management process.

Regardless of the limitations, we believe the process description developed in this thesis to be accurate, and that it can be used in future research as an example for a successful way of handling requirements in OSS settings where the Jira platform or a system with similar key functionalities is used for managing the requirements.

References

[1] Scopus, “Statistics: Scopus,” 21 March 2017. [Online]. Available: https://tinyurl.com/mny3v5q. [Accessed 21 March 2017].

[2] E. K. Huizingh, “Open innovation: State of the art and future perspectives,” *Technovation,* vol. 31, no. 1, pp. 2-9, 2011.

[3] European Commission, Open Innovation Open Science Open to the World - a vision for Europe, Brussels: European Commission, 2016.

[4] High Tech Campus Eindhoven, “Who we are: High Tech Campus Eindhoven,” [Online]. Available: https://www.hightechcampus.com/who-we-are. [Accessed 18 March 2017].

[5] V. Van de Vrande, J. P. de Jong, W. Vanhaverbeke and M. de Rochemont, “Open innovation in SMEs: Trends, motives and management challenges,” *Technovation,* vol. 29, no. 6, pp. 423-437, 2009.

[6] E. Enkel, O. Gassmann and H. Chesbrough, “Open R&D and open innovation: exploring the phenomenon,” *R&D Management,* vol. 38, no. 4, pp. 311-316, 2009.

[7] E. von Hippel, Democratizing innovation, Cambridge: The MIT Press, 2005.

[8] J. West and M. Bogers, “Leveraging External Sources of Innovation: A Review of Research on Open Innovation,” *Journal of Product Innovation Management,* vol. 31, no. 4, pp. 814-831, 2014.

[9] A. Baregheh, J. Rowley and S. Sambrook, “Towards a multidisciplinary definition of innovation.,” *Management decision,* vol. 47, no. 8, pp. 1323-1339, 2009.

[10] M. Bogers and J. West, “Managing Distributed Innovation: Strategic Utilization of Open and User Innovation,” *Creativity and Innovation Management,* vol. 21, no. 1, pp. 61-75, 2012.

[11] M. Hossain, K. Z. Islam, M. A. Sayeed and I. Kauranen, “A comprehensive review of open innovation literature,” *Journal of Science and Technology Policy Management,* vol. 7, no. 1, pp. 2-25, 2016.

[12] M. Bogers, A.-K. Zobel, A. Afuah, E. Almirral, S. Brunswicker, L. Dahlander, L. Frederiksen, A. Gawer, M. Gruber, S. Haefliger, J. Hagedoorn, D. Hilgers, K. Laursen, M. G. Magnusson, A. Majchrzak, I. P. McCarthy, K. M. Moeslein, S. Nambisan, F. T. Piller, A. Radziwon, C. Rossi-Lamastra, J. Sims and A. L. Ter Wal, “The open innovation research landscape: established perspectives and emerging themes across different levels of analysis,” *Industry and Innovation,* vol. 24, no. 1, pp. 1-33, 2016.

[13] L. Dahlander and D. M. Gann, “How open is innovation?,” *Research Policy,* vol. 39, no. 6, pp. 699-709, 2010.

[14] O. Gassmann and E. Enkel, “Towards a Theory of Open Innovation: Three Core Process Archetypes,” 2004.

[15] M. Osterloh and S. Rota, “Open source software development - Just another case of collective invention?,” *Research Policy,* vol. 36, no. 2, pp. 157-171, 2007.

[16] D. Riehle, “The economic motivation of open source software: Stakeholder perspectives,” *Computer,* vol. 40, no. 4, 2007.

[17] L. Dahlander and M. W. Wallin, “A Man on the Inside: Unlocking Communities as Complementary Assets,” *Research Policy,* vol. 35, no. 8, pp. 1243-1259, 2006.

[18] O. Gassmann, E. Enkel and H. Chesbrough, “The future of open innovation.,” *R&d Management,* vol. 40, no. 3, pp. 213-221, 2010.

[19] C. Gacek and B. Arief, “The Many Meanings of Open Source,” *IEEE software,* vol. 21, no. 1, pp. 34-40, 2004.

[20] W. Scacchi, “Understanding requirements for open source software,” in *Design Requirements Engineering: A Ten-Year Perspective*, K. Lyytinen, P. Loucopoulos, J. Mylopoulos and B. Robinson, Eds., Berlin, Springer Berlin Heidelberg, 2009, pp. 467-494.

[21] J. West and S. Gallagher, “Challenges of open innovation: the paradox of firm investment in open-source software,” *R&D Management,* vol. 36, no. 3, pp. 319-331, 2006.

[22] J. Lerner and J. Tirole, “Some simple economics of open source.,” *The journal of industrial economics,* vol. 50, no. 2, 2002.

[23] MySQL, “Licensing: MySQL,” July 2010. [Online]. Available: https://www.mysql.com/about/legal/licensing/oem/. [Accessed 4 April 2017].

[24] B. Fitzgerald, “The Transformation of Open Source Software,” *Mis Quarterly,* vol. 30, no. 3, pp. 587-598, 2006.

[25] GNU, “Philosophy: GNU Operating System,” [Online]. Available: https://www.gnu.org/philosophy/free-sw.html. [Accessed 15 March 2017].

[26] E. von Hippel and G. von Krogh, “Open source software and the “private-collective” innovation model: Issues for organization science,” *Organization science ,* vol. 14, no. 2, pp. 209-223, 2003.

[27] Open Source Initiative, “About: Open Source Initiative,” [Online]. Available: https://opensource.org/about. [Accessed 5 April 2017].

[28] Open Source Initiative, “Frequently Answered Questions: Open Source Initiative,” [Online]. Available: https://opensource.org/faq#free-software. [Accessed 5 April 2017].

[29] K. Crowston, K. Wei and J. Howison, “Free/Libre Open-Source Software Development: What We Know and What We Do Not Know,” *ACM Computing Surveys (CSUR),* vol. 44, no. 2, 2012.

[30] Netcraft, “February 2017 Web Server Survey: Netcraft,” 27 February 2017. [Online]. Available: https://news.netcraft.com/archives/2017/02/27/february-2017-web-server-survey.html. [Accessed 14 March 2017].

[31] Free Software Foundation, “About: Free Software Foundation,” [Online]. Available: https://www.fsf.org/about/what-is-free-software. [Accessed 11 March 2017].

[32] R. Stallman, “Philosophy: GNU Operating System,” 12 04 2014. [Online]. Available: https://www.gnu.org/philosophy/pragmatic.html. [Accessed 11 March 2017].

[33] GNU, “GNU General Public License: GNU Operating System,” 29 June 2007. [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.en.html. [Accessed 11 March 2017].

[34] J. West and K. R. Lakhani, “Getting clear about communities in open innovation,” *Industry and Innovation,* vol. 15, no. 2, pp. 223-231, 2008.

[35] M. Zhou, A. Mockus, X. Ma, L. Zhang and H. Mei, “Inflow and Retention in OSS Communities with Commercial nvolvement: A case study of three hybrid projects,” *ACM Transactions on Software Engineering and Methodology,* vol. 25, no. 2, 2016.

[36] M. M. Wasko and S. Faraj, “Why should I share? Examining social capital and knowledge contribution in electronic networks of practice,” *MIS quarterly,* vol. 29, no. 1, pp. 35-57, 2005.

[37] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida and Y. Ye, “Evolution patterns of open-source software systems and communities,” in *Proceedings of the international workshop on Principles of software evolution.*, 2002.

[38] G. Von Krogh, S. Spaeth and K. R. Lakhani, “Community, joining, and specialization in open source software innovation: a case study,” *Research Policy,* vol. 32, no. 7, pp. 1217-1241, 2003.

[39] L. Dahlander and M. G. Magnusson, “Relationships between open source software companies and communities: Observations from Nordic firms,” *Research Policy,* vol. 34, no. 4, pp. 481-493, 2005.

[40] F. Paetsch, A. Eberlein and F. Maurer, “Requirements Engineering and Agile Software Development,” in *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings*, 2003.

[41] N. A. Ernst and G. C. Murphy, “Case Studies in Just-In-Time Requirements Analysis,” in *Empirical Requirements Engineering (EmpiRE)*, 2012.

[42] J. Kuriakose and J. Parsons, “How Do Open Source Software (OSS) Developers Practice and Perceive Requirements Engineering? An Empirical Study,” in *Empirical Requirements Engineering (EmpiRE)*, Ottawa, 2015.

[43] B. H. Cheng and J. M. Atlee, “Research directions in requirements engineering,” in *Future of Software Engineering (FOSE '07)*, 2007.

[44] D. Pandey, U. Suman and A. K. Ramani, “An effective requirement engineering process model for software development and requirements management.,” in *International Conference on Advances in Recent Technologies in Communication and Computing*, 2010.

[45] J. Linåker and K. Wnuk, “Requirements Analysis and Management for Benefiting Openness,” in *IEEE 24th International Requirements Engineering Conference Workshops*, 2016.

[46] T. A. Alspaugh and W. Scacchi, “Ongoing software development without classical requirements,” in *International Requirements Engineering Conference (RE)*, 2013.

[47] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical software engineering,* vol. 14, no. 2, 2009.

[48] “About Qt: Qt Wiki,” 14 February 2017. [Online]. Available: https://wiki.qt.io/About_Qt. [Accessed 12 July 2017].

[49] H. Mäenpää, T. Kojo, M. Muenzero, F. Fagerholm, T. Kilamo, M. Nurmela and T. Männistö, “Supporting management of hybrid OSS communities - A stakeholder analysis approach,” in *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016*, Trondheim, 2016.

[50] Qt, “About Us: Qt,” [Online]. Available: https://www.qt.io/company/. [Accessed

9 May 2017].

[51] Qt, “Supported Platforms: Qt,” [Online]. Available: http://doc.qt.io/qt-5/supported-platforms.html. [Accessed 7 May 2017].

[52] Qt, “Qt: Licensing,” [Online]. Available: https://www.qt.io/licensing/. [Accessed 7 May 2017].

[53] “Qt: Qt Partners,” [Online]. Available: https://www.qt.io/partners/. [Accessed 30 July 2017].

[54] “The Qt Governance Model: Qt Wiki,” 24 March 2017. [Online]. Available: http://wiki.qt.io/The_Qt_Governance_Model. [Accessed 11 May 2017].

[55] “Maintainers: Qt Wiki,” 5 May 2017. [Online]. Available: https://wiki.qt.io/Maintainers. [Accessed 12 July 2017].

[56] “Release Management: Qt Wiki,” 22 November 2016. [Online]. Available: https://wiki.qt.io/Release-Management. [Accessed 13 July 2017].

[57] “Jira: Qt,” [Online]. Available: https://bugreports.qt.io. [Accessed 15 August 2017].

[58] “Triaging Bugs: Qt Wiki,” 24 May 2015. [Online]. Available: http://wiki.qt.io/Triaging_Bugs. [Accessed 21 May 2017].

[59] “Gerrit Introduction: Qt Wiki,” 25 Novermber 2016. [Online]. Available: http://wiki.qt.io/Gerrit_Introduction. [Accessed 13 May 2017].

[60] “Early Warning System: Qt Wiki,” 1 February 2017. [Online]. Available: http://wiki.qt.io/Early_Warning_System. [Accessed 16 May 2017].

[61] “Workflow diagram: Qt Bug Tracker,” [Online]. Available: https://tinyurl.com/lyf6uca. [Accessed 21 May 2017].

[62] “Issue types: Qt Bug Tracker,” [Online]. Available: https://bugreports.qt.io/secure/ShowConstantsHelp.jspa?decorator=popup#IssueTypes. [Accessed 10 June 2017].

[63] “Search: Qt Bug Tracker,” [Online]. Available: https://tinyurl.com/y95j26mx. [Accessed 10 June 2017].

[64] J. West, A. Salter, W. Vanhaverbeke and H. Chesbrough, “Open innovation: The next decade,” *Research Policy,* vol. 43, no. 5, pp. 805-811, 2014.

[65] Open Source Initiative, “The Open Source Definition: Open Source Initiative,” 22 March 2007. [Online]. Available: https://opensource.org/osd. [Accessed 5 April 2017].

Appendices

The central concepts used in this thesis are defined in the below table in an alphabetical order.

Concept	**Definition**
Commit rights	The rights to add and modify software code in the version control system used to manage code contributions
Community	"…a voluntary association of actors, typically lacking in a priori common organizational affiliation (i.e. not working for the same firm) but united by a shared instrumental goal—in this case, creating, adapting, adopting or disseminating innovations." [34].
Coupled process	Combines the inside-out and outside-in approaches to open innovation. With this process, an organization is trying to obtain knowledge from outside (outside-in process) and bring ideas to market (inside-out process) [14].
External contributor	Someone who contributes in a company-led open source project, and is not employed by the company.
FLOSS	Free/Libre and Open Source Software: Similar concept as OSS, but focuses more on the moral rightness and importance of ensuring the users' freedom [26].
Free Software	Software that the user is free to study, modify, and share [31].
Innovation	"Innovation is the multi-stage process whereby organizations transform ideas into new/improved products, service or processes, in order to advance, compete and differentiate themselves successfully in their marketplace." [9].
Inside-out	An organization using this approach to open innovation tries to bring its own ideas or resources to market [14]. It can either sell or license them out, or it can freely reveal them, for example by open sourcing code that has been internally

	developed and not previously available outside the organization.
Jira issue	An item in Jira, a requirements management software. Often represents a task to be worked on.
Online community	A group of people or organizations connected through the Internet, for example by email and websites, and sharing an interest or a common goal
Open innovation	Often contrasted to closed innovation where organizations innovate with an internal R&D team. An organization with an open innovation approach either tries to obtain knowledge, ideas, resources, etc. from external sources, or tries to bring its own ideas to market.
OSS (Open Source Software)	Software that has been published under an open source license [15, 7], i.e. a license that allows anyone to view, use, and modify the software's source code.
Outside-in	Approach to open innovation. In the outside-in approach the organization is using external ideas, resources and knowledge contributed by various stakeholders — customers or suppliers for example [14]. Sometimes also called technology exploration (e.g. [5]).
Requirements engineering	Requirements engineering is "…a systematic approach through which the software engineer collects requirements from different sources and implements them into the software development processes" [44].
Requirements management	Includes "…all activities concerned with change & version control, requirements tracing, and requirements status tracking." [40].