

The Integration and Collaboration of Open Source Communities into Software Development Process of Commercial Companies: A Case Study on Sailfish Operating System

Nazanin Hajimirarab

Master's Thesis



ITÄ-SUOMEN YLIOPISTO

Faculty of Science and Forestry

School of Computer Science

November 2015

UNIVERSITY OF EASTERN FINLAND, Faculty of Science and Forestry, Joensuu
School of Computing
Computer Science

Student, Nazanin Hajimirarab: The Integration and Collaboration of Open Source
Communities into Software Development Process of Commercial Companies: A Case
Study on Sailfish Operating System

Master's Thesis , 80p.

Supervisors of the Master's Thesis : Professor Markku Tukiainen

November 2015

Abstract:

Open Source (OS) development is a new way of thinking, developing, maintaining, promoting, and distributing software. Every OS project has a complex community, consisting of many individuals from different areas of expertise and in different parts of the world, who are willing to volunteer to spend their time creating and improving software. Communication is the essential key for the success of every OS community.

As interesting as being involved in Open Source Software (OSS) development can be, having an OS community that delivers a high quality software requires coordination, true transparency, and excellent communication.

This thesis project reviews the scientific researches done in OSS development. We go through the importance of communication in OS projects, and how proper infrastructure needs to be prepared so that members have transparency over what they are contributing to. We will talk about Agile methodologies, and its benefits of use in OS projects. Then we will bring a real-life study of a software company developing a partially OS operating system. We will go through the mode of operation in the company, and the tools it uses to develop and maintain software. We will further discuss how the company struggled to keep the relationship with its OS community; how these struggles affected the quality of software, and what the company did to solve those problems. We will conclude by doing an evaluation of the project, earlier studies in this fields, and suggesting topics for future works.

Keywords: agile software development, collaboration in software development, operating systems, software performance, open source model

ACM Computing Classification (1998) Categories and Subject Descriptors:

K.6.3 [Software Management]: Software development, Software process. D.2.9 [Management]: Life cycle, Software process models. D.2.9 [Management]: Life cycle, Software process models. D.2 [Software Engineering]: Methodologies, Tools.

Foreword

This thesis was done at the School of Computing, University of Eastern Finland from summer 2014 to autumn 2015.

This project was a case study of a software development company through which I took efforts as a project manager, responsible to manage teams and tasks to deliver the planned infrastructure. However, it would not have been possible without the kind support of some of the individuals at Jolla. I would like to extend my thanks to all of them who supervised me in this project.

I am highly thankful to Jolla's Research and Development team members Bernd Wachter, Eric Le Roux, Carsten Munk, and David Greaves for baring with me in learning the process and guided me through it step by step; To Jolla's community chiefs Carol Chen and Iekku Pylkkä for providing necessary information regarding collaboration and community, and to Jolla's open-source community who supported this project after it was announced.

My thanks and appreciation also goes to my university supervisor professor Markku Tukianen for trusting me, and giving me the freedom to choose my thesis project, and contribute to a challenging Open Source project while working at Jolla Ltd.

Disclaimer

All the information gathered in this document are prepared by the author, from company confidential sources as well as non-confidential and scientific sources. All references are mentioned at the end of this document unless company confidentiality prevented it.

The content written on Mer wiki page about Jolla as a vendor of Mer¹ and the FAQ page for OS contribution² are all edited by the author of the thesis, therefore, used freely in this document.

Figures related to different operations in the company and all the information about how each function works are reviewed by the supervisors in this project and are free to be used in this document.

¹https://wiki.merproject.org/wiki/Vendor/Release_Structure

²https://wiki.merproject.org/wiki/FAQ#Frequently_Asked_Questions_for_Vendors

List of Abbreviations

OS	Open Source
OSS	Open Source Software
FAQ	Frequently Asked Questions
IRC	Internet Relay Chat
OBS	Open Build System
UI	User Interface
RPM	RedHat Package Manager

Contents

1	Introduction	1
1.1	Overview	1
1.2	Thesis statement and objectives	2
1.3	Thesis document structure	3
2	Open source software development	4
2.1	Communication	4
2.2	Development	7
3	Agile methodologies in software development process	9
3.1	Iterative and incremental development	9
3.2	Iterative and incremental development for a software release	12
3.3	Use of continuous integration in iterative and incremental development	12
4	Case study: the software company ‘Jolla’ and its operating system ‘Sailfish’	14
4.1	Why this project started	14
4.2	Mode of operation and software development at Jolla	16
4.3	Iterative and incremental planning at Jolla	17
4.3.1	Pre-planning	18
4.3.2	Planning	18
4.3.3	Sprint planning	19
4.3.4	Demo	19
4.3.5	Retrospective	19
4.4	Sailfish operating system	20
4.5	Software development process	21
4.5.1	An example of a Sailfish package on Git	23
4.5.2	Webhooks	28
4.5.3	Open build systems (OBS)	30
4.6	Tracking system for planning and task tracking: Bugzilla	31
4.6.1	Quick entry	32
4.6.2	File a report using a simple view	33
4.6.3	File a report using an advanced view	36
4.6.4	Description of a report in Jolla Bugzilla	38
4.6.5	What happens after a report is created on Jolla Bugzilla	38

4.7	The release process	41
4.7.1	CI-bot	42
4.7.2	Changelogs	42
4.7.3	Bugzilla statuses and their meanings	42
4.7.4	Development level	44
4.7.5	Testing level	45
4.7.6	Release level	45
4.7.7	Release snapshot	45
4.7.8	Life cycle of reports in Jolla bugzilla during the integration process	46
4.7.9	Earlier tracking of the release process before this project	47
4.7.10	Problem with the release tracking procedure before this project	48
4.7.11	What did Jolla do to solve the problem and ease the collaboration?	49
4.7.12	Upstream Bugzilla sync	49
4.7.13	How automated cloning and tracking is done	50
4.7.14	How CI-Bot works during upstream sync	53
4.8	The proposal presented to Jolla's open-source community	53
4.8.1	The request to Mer package maintainers	53
4.9	A example to show the results of the project in practice	57
4.9.1	Iteration Planning 12/2014 before the Upstream Bugzilla Sync	58
4.9.2	A Bugzilla report in the final release changelogs before the upstream Bugzilla sync	59
4.9.3	Planning 05/2015 after the upstream Bugzilla sync	60
4.9.4	Upstream Bugzilla sync and its affect on tracking the open source components	63
5	Analysis of the project	69
5.1	Earlier Works	70
6	Conclusions	71
6.1	Lessons learned	71
6.2	Open questions and future works	72
7	Glossary	73
	References	77

List of Figures

1	Iterative planning	10
2	Planning cycle at Jolla	17
3	Steps in delivering a goal	19
4	A tree-view of a goal broken down to smaller reports	20
5	Sailfish operating system architecture including open oource and prop- erty packages	22
6	Branching during software releases	23
7	Overview of Sailfish Browser development page on GitHub	24
8	An example of a commit message with a Jolla Bugzilla number	24
9	A part of the list of commits to Sailfish Browser application	25
10	An overview of the .spec file page for Sailfish Browser	26
11	Webhook settings on Git	28
12	An overview of sailfish-browser webhook mappings	29
13	Git links which point to related repositories	29
14	Other information about the webhook mappings	30
15	An overview of some open packages for Sailfish operating system on OBS	31
16	An overview of the first page of Jolla Bugzilla	32
17	Quick entry page in Jolla Bugzilla	33
18	Report entry with a simple view on Jolla Bugzilla	34
19	The advanced view	36
20	A report that is a release blocker for update 10	38
21	Bug description outline	40
22	An overview of a report filed on Bugzilla	40
23	Release cycle levels	41
24	A bug status starts from 'New' and ends in a 'Released'	42
25	Where the code for a software release comes from	43
26	Bug life cycle during software integration	46
27	Changelog for a closed-source package called 'ambiened'	47
28	Changelog for an OS package called 'ofono' on Mer Side	48
29	Upstream Bugzilla sync in detail	51
30	Report created on Jolla Bugzilla	59
31	Pull request on Git for lipstick-jolla-home-qt5 component	60
32	Webhook shows a hook for the Git commit named upgrade-1.1.6	61

33	Lipstick-jolla-home-qt5 is on Jolla's OBS	61
34	The commit in Git has landed in OBS on 3 December 2014	61
35	Lipstick-jolla-home-qt5 fix in the release changelog	62
36	Commit messages related to 'connman' component in the release changelog	62
37	A report created on Mer Bugzilla	64
38	A commit for a Fix for Connman in Mer packages repository on Git .	64
39	A commit for connman lands in Jolla webhook	64
40	Connman is on Jolla's OBS	65
41	The commit in Git has landed in OBS on 13 May 2015	65
42	Connman fix in the release changelog	66
43	An example of a cloned report from Mer Bugzilla on Jolla Bugzilla .	67
44	Clone of a Mer report shown on Jolla Bugzilla	67
45	How CI-bot updates the cloned report with Jolla's integration cycle changes	68

List of Tables

1	File specifications for sailfish-browser	28
2	Different statuses in a Bugzilla report	35
3	Different severities in a Bugzilla report	35
4	Different priorities in a report on Jolla Bugzilla	37
5	An example of a report description on Jolla Bugzilla	39

1 Introduction

In recent years, there have been a large number of researches and documents concerning OS communities and their practices in software development. Several surveys, questionnaires and experimental studies have been done on the impact of different factors and values of these communities. [10]

Open Source Software (OSS) has made a significant impact on the software development industry. Looking at projects such as Linux or Mozilla Firefox, shows the power of such development methodologies in today's world. OSS is unique in terms that it is developed by people who are geographically disperse and unpaid programmers, thus, nearly all the time having remote communication with each other. The power of communication in such communities is another topic that attracts different studies towards the quality of software delivered through OSS development.

1.1 Overview

During traditional software development end users are not actively involved in the development process [17]. On the other hand, in OSS development, end users are one of the main sources of user feedback and bug reporting for that software. End users of OSS projects do not only report bugs on the software, but some of them can provide fixes for the issues reported.

OSS development creates the core software in the beginning, with minimum required functionality, and let users start using it. As they go forward, they continue maturing the quality of the software by the requirements they receive. In traditional software development processes, requirements should be defined and analyzed before the actual development of the software begins. This also means that making changes to the software after it was released to its end users, will also require a set of procedures where changes need to be accepted by the higher management before they are applied to the software [5]. OSS development is free from all the traditional requirement and change analysis. OSS projects follow the evolutionary software development, where the newer versions of software are increasingly released by the developers.

1.2 Thesis statement and objectives

The focus of this thesis study is first to discuss scientific researches that have been done on OSS development, Agile Methodologies and use of such methodologies in developing Open Source Software. The thesis will be continued by bringing a case study of a software development company in how OSS development plays an important role in the delivery of the final software product.

There have been efforts to study the foundation of OS communities and the importance of communication among the community members. Also, several literature reviews have been done on most used software development models in OS communities. The framework of the studies in this section is to introduce – from scientific point of view – and go a little into depth of OSS development concept, and the values that members in such communities share among each other. Afterwards, we have brought a case study, at Jolla Ltd.³, to analyze what happens when OS communities become part of a commercial company. To be involved in the core of the case study, I have been selected as the Project Manager of a project Jolla Ltd. was planning to do to ease the collaboration for its OS community. Before starting to manage the actual project, I had several training sessions held by the Release Manager, Technical Chiefs, and Co-founders of Jolla to learn the software development and release process in the company.

Throughout the project the focus is more on aspects of agile software development in start-up companies, as well as, paying more attention to the relationship of Jolla's OS community with its in-house developers and management. Another focus in this project, as the Project Manager, was to first deliver the planned infrastructure to the community. The findings of the project are used afterwards to give an analysis on the whole process.

The findings of this project can be used to identify opportunities for improvement of the relationship between commercial companies and their OS communities. It can also be useful to help product managers in OS projects to pay more attention to the value of transparency in delivery of product. The outcome of this project gives clue to companies to bring in experienced leaders to coordinate the dynamic collaboration between their in-house and OS developers.

³An introduction about the company, its software development process, and mode of operation is given in chapter 4

1.3 Thesis document structure

This thesis consists of six chapters. After introduction, in the second chapter we study the fundamentals of OSS and its communities. We go into more details about how such communities keep contact with each other and how OSS development has helped the industry in software development in the recent years. In chapter three we talk about agile methodologies in software development and continue by describing why OS communities lean more towards agile software development models. Chapter four is where the case study comes into this document. In this chapter we will break down the process of software development at Jolla Ltd. into details. We first study the mode of operation in the company and the reasons behind that, what tools the company use to develop software; and using those tools, how the software release process takes place in the company. At this point, where we have an understanding of the release process, we start discussing the main problem the company were struggling with since its establishment: We have a partially OS operating system, and we have an OS community who would want to collaborate in the making of that operating system, but we do not have the proper infrastructure to make this collaboration happen.

In chapter four, we also explain the outcome of the project done in the company and how it was represented to the community, and will see real examples from the software development process of the company before and after the project was done.

In chapter five, we go through earlier studies around this topic as well as the findings of this project. Finally, in chapter six, we conclude the outcomes of the project, and talk about the lessons learned. We will also suggest further studies that can be done in the field of OSS development.

2 Open source software development

"If you want to build an open source project, you can't let your ego stand in the way. You can't rewrite everybody's patches, you can't second-guess everybody, and you have to give people equal control."(Rasmus Lerdorf)

Over the last decades, Free/Libre/Open Source Software (FLOSS) has proven to be capable of working successfully in software development processes [2]. OS projects talk about giving the liberty to all users of the software to contribute in making it better so that it meets their needs. Building an active community of developers requires thoughtful planning, analysis, patience, and support [3]. Simply because an OS project exists, does not mean that its community will be continuously contributing to it. Every product needs to deliver a clear and meaningful value to its developers. This means that any OSS project should be explicit about its software design, business model, roadmap, release schedule, and openness about bugs and when they will be addressed.

OSS allows users to improve the quality of the software by giving them access to the source code [5]. OS development has introduced a new way in developing and maintaining software which has several benefits such as lower costs in development and maintenance of the software, higher quality as well as higher reliability of the software [16].

To look at the definition of OSS, let's go back to where it was rooted. Ezeala et al. [5] explains the Free Software Foundation (FSF), founded by Richard Stallman in 1985, focuses on the liberty of the software. This means freedom from control, not price. In another case, Open Source Software, developed by Open Source Initiative (OSI) focuses on the distribution terms of software license.

2.1 Communication

In OS communities, where a diverse group of people with different educational, cultural and professional backgrounds gather around to volunteer and spend their time on creating and improving software, communication plays one of the most significant roles. Community members come from different parts of the world, with different time zones, and usually busy with their own daytime jobs. What they do in OSS develop-

ment projects is more as a hobby than something to make money out of. There are, on the other hand, people who make a living out of contributing to OSS projects.

"One of the questions I've always hated answering is how do people make money in open source. And I think that Caldera and Red Hat – and there are a number of other Linux companies going public – basically show that yes, you can actually make money in the open source area." (Linus Torvalds)

For OS communities, the nature of most of the tasks is to work in a team [22]. Therefore, a proper way of communication where responsibilities are shared and transparency exists during the whole development process, makes the project achievable and brings more satisfaction to the team members to collaborate in future projects. Communication in OS communities deals with communicating around code sharing, helping the team to release a higher quality software [22]. Since the community developers can come from anywhere in the world, the language of communication is in English, and everyone needs to obey that while e.g. sending e-mails to the *mailing lists*, writing posts on *wikis*, or *chatting* in public groups.

There are certain group of people participating in Open Source communities [22]:

- **Advanced** users who stick around more often than the others, have more knowledge of the source code, and are usually the ones whom other developers of the group come to ask questions from.
- **Normal** users who do not necessarily participate in maintaining the code. They might be end users of the software who appear in chat forums to familiarize themselves more with the community behind the software, give user feedback, or report bugs.
- **Developers** who write the code and have the authority to change the source code. They are people who have the main knowledge over the code.
- **Bug fixers** who detect bugs and report them in bug tracking systems.
- **Managers or project founders**, who might early developers in other projects. They help with the organizational aspects of development, release management and the communication.

Since the initiation of OSS, a number of researches have been dedicated to analyze the communication patterns in OS communities [15]. Why communication in such communities is important lies mainly on the fact that people who contribute to the development of the software are not in-house developers and testers working together in an office, or under the same roof. People are usually more comfortable to describe an issue when they know the person on the other side of the conversation, or they have a device they can run the software on and show the bugs they have found in the software. However, when it comes to remote communication, via services such as chats or e-mails, communities need to try to communicate with clarity. For non-experienced users or developers, adopting to communicating only via text can be difficult at the beginning, but with time, they get to know the community people and also their communication skills improve.

Hayashi et al. [7] puts the importance of communication in OSS development as a means to improve software quality. In particular, to fix a bug in the software, sometimes it is needed that multiple developers commit changes to the code. This is because every developer has limited knowledge about a large software system.

The main mode of connection in OS communities is kept via online tools. With these tools, such as chats, mailing lists, and wikis, everyone involved in the project, from the founders to developers and end users, can communicate with each other [22]. The talks involve topics around the upcoming changes to the software, bugs or defects that were found, questions regarding using the software, questions regarding how to maintain the software, and how to solve the difficulties while using the particular software. Examples of some of the communication tools are [15]:

- **E-mail** via mailing list is one of the most common ways for communication among the community people. Sending e-mails can make one thread and all the replies to that thread be reachable via one email.
- **Forums** are like mailing lists with the difference that questions and communications are posted on a website. There are different categories in a forum, and one needs to find the right group/category, and ask questions there.
- **Instant Messaging** use has increased in the recent years. Depending on the community, services such as *IRC*, *Skype*, or *Hangout* are popularly used.
- **Wiki** is a website where users who have made accounts on that website can

edit the content there, and everyone else has access rights to read the content posted on a wiki. Wikis are usually used for a more formal way of documenting stuff. For example, how to install the software, or tutorials concerning working with the software. These sort of information can be posted on wikis and be made accessible for everyone. Later if false information is found in the text, authorized users with accounts can edit the text.

2.2 Development

“Open source software communities do not have strict hierarchies as in traditional software teams; However, the community structure is not completely flat. The organizational structure of an open-source community is determined by the roles that the participants play.” (Eric Raymond)

Saini & Kaur [18] put the structure of OS community projects into these categories:

- Vendors: They are OS projects that are backed by a vendor under the open license. The full copyright of this software project goes to the vendor. For instance, in our case study, Jolla is a vendor of *Mer*⁴.
- Development community: These communities represent OS projects where it belongs to stakeholders who share the copyright of the project. Debian community is an example of such a project.
- Open Source competence centers: These are usually the conference and workshop organizers, providing services such as user training, testing, or research and development. An example of such centers is the OSS Watch⁵ in the UK.

Questions towards OSS development usually deal with 'why do people do something for free?' [18]. There can be different reasons for different people why they like to contribute to OS projects. Eric Raymond in his book '*The Cathedral & the Bazaar*' [16] says maybe it is a "personal itch", or the desire to help people. Others might find it beneficial for themselves and use OS projects to learn new skills which can later land them on better job opportunities. In GNU's philosophy the overall concept of 'doing

⁴www.merproject.org . You will read about Mer OS community in Chapter 4

⁵www.osswatch.co.uk

it for free' should be looked at like freedom of speech, not free beer. Chawner [3] also emphasizes on the motivation behind collaborating in OS projects, and how keeping that motivation is the main factor in keeping people contributing to the community.

Since OS communities consists of people distributed in different places, there also needs to be a distributed platform via which community members can develop on. *Git distributed version control system* is a platform widely used for OSS development [13]. *Github* , a code hosting repository based on Git version control system, is one of the famous platforms used for Open Source collaboration. Vasilescu et al. [23] explains that Github allows users to set up repositories with custom access, where those with access rights can take an instance copy of the repository on their local machines, develop on it, and then commit their changes to the main repository so that it can be included in the final software release. There is much more to Github than a repository with accessible data which we will discuss further in the chapter four.

Using Git version control system benefits the software development processes with regards to easing collaboration in projects where there are number of developers working on many components. Everybody has access to the code and can propose changes. Git also works as a time capsule. Overtime changes are made to the code and checked into the source code, it keeps a time log. So there will be a complete history of all the files that have changed, and why they changed. This ensures graceful rollbacks whenever code needs to be reverted. Moreover, Git does not have a centralized repository. This means there can be multiple people making changes to the code and commit. It all goes to one control repository (aka. one server).

There are commercial companies which benefit from having Open Source communities supporting their software [4]. Sailfish operating system - developed by Jolla Ltd - which we will talk about in chapter four, is one example of such companies. What OS communities can bring for such companies are people motivated to improve the software quality voluntarily. Community participants, in most cases, are users of the software developed by the commercial company, and will report bugs or commit fixes for the software.

3 Agile methodologies in software development process

Agility talks about change and willingness to respond to change. It depends on a company to what level it wants to be adjustable to changes. The level of agility in companies differs them from other competitors. Highsmith [8] describes agility - as a noun in the concept of software development - the ability to both create and respond to change in order to profit in a turbulent business environment.

The evolution of agile techniques and the current methods of using them have been supporting management, defined processes, and concrete practices throughout the phases of the software life cycle in software development process. Agile principles promote the delivery of software in small increments and iterations, which enables frequent progress checks and provides the opportunity to refine the goals of a project in line with customer's desires [21]. Self-adaptation is also promoted at both the project and the process level.

From multiple definitions of agile software development processes, this quote explains it all.

“Agility is dynamic, context-specific, aggressively change-embracing, and growth-oriented. It is not about improving efficiency, cutting costs, or batten- ing down the business batches to ride out fearsome competitive “storms”. It is about succeeding and about winning: about succeeding in emerg- ing competitive arenas, and about winning profits, market share, and cus- tomers in the very center of the competitive storms many companies now fear.” (Goldman, Nagel, and Preiss, 1995)

Agile approaches best adopt to businesses where innovation and creativity are their key elements [8].

3.1 Iterative and incremental development

Iterative and Incremental software development is a methodology in agile practices. Jinzenji et al. [11] describes an Iterative and Incremental Development agile method- ology as development of software in cycles (i.e. iterations) and in smaller portions at a

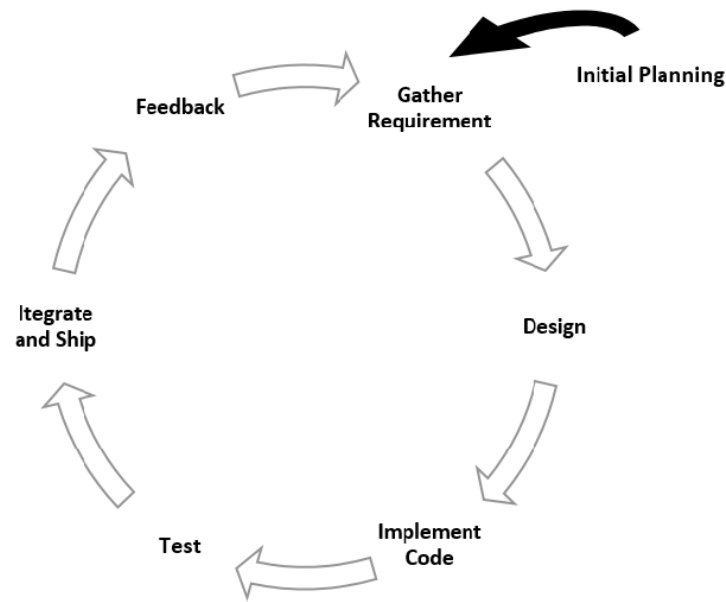


Figure 1: Iterative planning

time (i.e. incremental) (see Figure 1). This allows developers to see how the software behaved on the device, hence making it more efficient to apply improvements as they go further into development. Having shorter cycles makes it possible to develop faster and in smaller portions, and making a higher quality software. Agile methodology is about continuous feedback and change. It is characterized by short cycles where problems are inspected and changes are adopted through frequent feedback loops [20].

Uras et al. [22] explains that agile methodologies are adaptive and oriented towards people, not the process. During agile development verbal communication identifies requirements; failing, results in learning faster; the mode of operation is incremental and iterative; developers have interchangeable roles; documents are kept to minimal, and team members share important values such as respect, transparency and passion. Although agile methodologies bring a more informal way of operation in teams, it does not mean that they are easy to be coordinated. Applying agile methodologies in the right way needs strong leaders who can coordinate different teams to work towards the same goal.

Matthews [12] in *The Manifesto for Agile Software Development*⁶ talks about these values:

⁶<http://agilemanifesto.org/>

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Sindhgatta et al. [20] explains that scrum agile development method is characterized by Iterative and Incremental Development where the progress of making software takes place in a series of short iterations or sprints. The iterations or sprints are typically from one to four weeks during which team members discuss about new changes and commit to deliver a number of them. At the end of the iteration or sprint, the needed changes are committed according to the previous discussions. They are implemented, tested and integrated into the system. There are Scrum Masters who handle meetings for planning, reviewing and retrospective.

OS communities often use agile methodologies during their software development processes [22]. Via Agile methodologies they can have the flexibility to adopt to changes, learn from their mistakes, and focus more on releasing a bug-free software quickly, rather than going through time-consuming requirement analysis and documentation. Using Agile methodologies, OS communities can respond to changes while improving the overall quality of the released software.

Ruparelia [17] in the history column article regarding software development life cycle models - life cycle covers the initiation of the project to its requirement analysis, development and maintenance - one of the main strengths of incremental model are to make the delivery of the product available with early, incremental releases that evolve with each iteration. At the same time feedback is taken in constantly and the feedback from previous iterations can be used in current iterations. Incremental model also enables incremental implementation of the software, which helps to monitor changes as they are constantly coming in to the software; thus, make it easier to mitigate risks. Furthermore, everyone responsible in the making of the software, from project managers to developers and stakeholders, are involved in iterations and their opinion can directly affect the software under development.

Applying agile methodologies are mostly suggested for smaller projects [17]. Having agile mode of operation where people need to have everyday contact (usually face-to-

face), changes happen fast, and documentation is limited, makes it harder to manage bigger projects with agile methodologies.

3.2 Iterative and incremental development for a software release

Software development with incremental release procedure requires operating functions, which will then enable companies to provide system users with more stable versions of a system at regular intervals [19]. This life cycle model enables iterative enhancement of system development organization. It also supports to periodically distribute software updates to end users or user communities. Having incremental development during release cycles assures faster update intervals, where several bug fixes or feature improvements are released with each update.

3.3 Use of continuous integration in iterative and incremental development

Continuous Integration (CI) is about all the work within a team that goes through integration and testing to find defects and eliminate them as the software is going through its software development life cycle [1]. CI plays an essential role in iterative and incremental software development and its success. CI helps the team to keep track of the current stage of the development as well as reassuring that a quality software based on software planning is delivered at the end of an iteration [9]. When defects are found, it is best to fix them while the software is still in the early phase of its development. Fixing a bug during the development level is much less expensive, both in terms of financial aspects and time, than when the software is delivered to users as the final product. CI facilitates the early detection of bug through continuously integrating the code and testing it [1].

In distributed software development projects, developers commit code several times a day. Imagining one or more of these commits are carrying bugs, and the code does not go through checks before it lands in the software, the final software that comes out of the release cycle will contain those bugs. Later during system testing which happens after a complete release cycle is done, those bugs are detected. With bugs being found only after the release has been made, the code needs to go through full build again.

Developers responsible for those bug need to submit their fix, commit the new code to the system, and a new release build should be made again. Considering the probability that there is almost always more than one bug when several developers are working on a software, if their codes are not checked some time before it lands into the final software, it will be expensive to fix them. Vasilescu et al. [23] explains that what CI does is to mitigate the risk of breaking the build, or releasing a buggy software. CI is an essential tool for distributed systems, where geographical and time differences are in the context of the work; exactly like OSS development projects. The practices of CI can make improvements in the quality of software projects [12].

4 Case study: the software company ‘Jolla’ and its operating system ‘Sailfish’

Jolla ⁷ was founded in 2011, aiming to continue the *Nokia* project, *MeeGo*⁸ - a Linux-based operating system for mobile phones discontinued by Nokia - as they believed it could succeed in the open innovative mobile space. During 2011 some of the MeeGo team left Nokia to join Jolla. In November 2013, Jolla launched its first smartphone, which uses a gesture-oriented user interface. The operating system running on the phone is called *Sailfish*, which is a combination of Mer⁹ - an OSS project which acts as a middleware on Linux-based systems - in its core; with *Qt* and *QML* and C++ in its user interface.

Jolla’s aim is to be open and transparent towards their OS community in what they do. Jolla’s motto has been ‘*Doing It Together (DIT)*’ which refers to having a software that everyone who likes and uses, can contribute to make better.

Jolla’s OS developers community – the heart of any OSS project – plays an important role in the growth and improvement of the software the company releases. Through the years of development, Jolla has always tried to be transparent with its community, and also asked for their contribution; however, due to lack of sufficient infrastructure, the community could not really influence the quality of Jolla’s software development.

4.1 Why this project started

This project started to enhance the OS community collaboration at Jolla. One of the reasons the company could not prepare the needed infrastructure for community collaboration was because of its partly Open Source and partly Closed-Source operating system, Sailfish.

Sailfish operating system is built like a Linux distribution. The core of Sailfish is from

⁷www.jolla.com

⁸MeeGo was a Linux distribution using source code from Intel and Nokia. This project was canceled by Linux Foundation in September 2011.

⁹Mer is a free and open-source software distribution targeted as a middleware for Linux distribution services. After the MeeGo project was canceled, Jolla picked Mer, as a fork of MeeGo, and used it to develop Sailfish operating system. Mer is now used in Sailfish OS core development.

*Mer Project*¹⁰ – an Open Source, Linux-based, mobile-optimized software that works as a middleware [14]. The User Interface (UI) part of the operating system is based on QML – a user experience design language provided by Qt framework. The features of the QML language enables Sailfish to provide a rich set of UI elements, which creates animated UIs and lightweight applications. Sailfish operating system also includes the capability to run *Android* applications. It is based on a set of Android libraries, integrated into the software and brings performance comparable to native applications.

Jolla used to develop the closed part of Sailfish operating system with its in-house developers, and pick the changes on the open from Mer side and integrate them into Sailfish software releases. Not being able to make the open part of the code accessible for Jolla's OS community, the developer community collaboration could not happen during software release process. The reason for that was simply because the OS community could not see what was changing, or did not have any ways to commit fixes for existing bugs and push them to Jolla's repositories. What they did was to fix defects that they knew existed, push them to the Open repositories, and wait for Jolla to pick them once they needed those fixes for their software.

This was not the way the company wanted to continue its business. There was a motivated OS community motivated to help to make the software they were using better. They wanted to have visibility over what was changing or needed to be changed in the future. The OS community needed transparency, clear plans, and a platform from Jolla, where they could contribute to make a higher quality software. This situation was not desirable from Jolla's point-of-view either. They knew they had been delayed with opening up their platform, and they wanted to start making the required changes. And so this project initiated to reach this goal.

Our goal in this project was to make the first proper steps in opening up the platform and making the collaboration easier. After the project was done, We invited our community to contribute to the open-source part of the Sailfish operating system, Mer, and we requested that they could adopt a policy to support vendor tracking for Jolla on Mer.

The creators of Mer, as well as community chiefs, release manager, backend developers, and the project manager at Jolla gathered up to discuss about the planning of the project. I, as the project manager, was responsible to understand the whole process behind software planning and development, and to make sure that all parties were in

¹⁰<http://merproject.org/>

sync to deliver the product within the target schedule. I was also in charge of organizing meetings and getting updated on the process. The main start of the project was in June 2014 and the final proposal was presented to the community and in-house Jolla developers in February 2015.

This thesis project describes the process of preparing the needed infrastructure for Jolla, so that they are able to collaborate with their OS community in the making of their partly OS operating system Sailfish.

We are going to read in this document, how the company created and maintained the upstream sync they needed to open up their platform for OS contribution. Firstly, we describe of the company's mode of operation. Then, we study Sailfish operating system and its components. Then disadvantages of lack of OS community collaboration will be discussed. We follow this by looking more into the software development and release process at Jolla. We study the planning needed to deliver the goal. The case study is concluded by bringing an example of how the implemented technology is now being used in the company during software releases, and the advantages it has brought to the overall product lifecycle.

4.2 Mode of operation and software development at Jolla

The main mode of operation and software development in Jolla is based on Iterative and Incremental agile methodology, and it is applied at different scales in the company. From the common methodologies in agile development - prototyping, iterative and incremental development, spiral development, rapid application development - Jolla has chosen the Iterative and Incremental Development for its software development process as it suits the main goals of the company. Applying an iterative methodology in all parts of the company has given them the opportunity to get requirements in constantly, be close to customers, and iterate to bring quality. Jolla has chosen iterative planning to fail fast and learn faster, get fast feedback, have the possibility to rollback if needed, and have a high quality code at the end of each iteration.

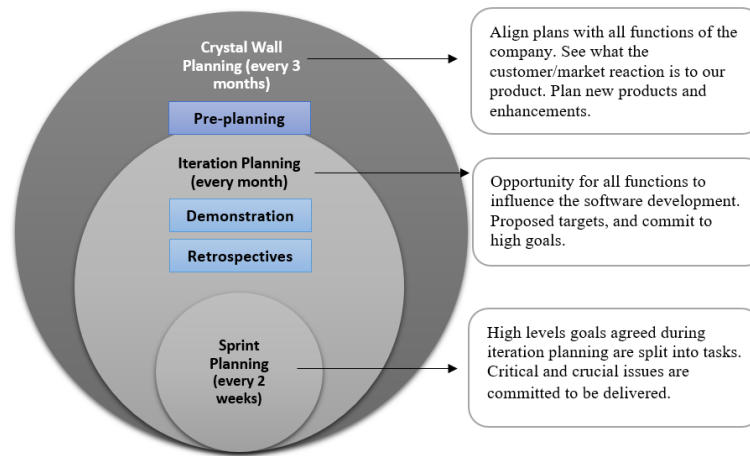


Figure 2: Planning cycle at Jolla

4.3 Iterative and incremental planning at Jolla

An iteration, at Jolla, is a planned development period lasting one month. At the end of each month, a new software update for Sailfish operating system is released. The one-month period can change to be longer in case the company hits *release exceptions*, *change requests*, or *release blockers* that need more time to be fixed before the software release is publicly out.

A typical iterative planning at Jolla consists of:

- Crystal Wall planning (every 3 months)
- Pre-planning (before iteration planning day starts)
- Iteration planning (every 1 month)
- Demos (during iteration planning day)
- Retrospective (during iteration planning day)
- Sprint planning (every 2 weeks)

Sprint, iteration, and crystal wall planning days are each targeted to achieve specific goals. To see the big picture, Figure 2 shows how each planning day is designed for the company to make the best coverage of the requirements, design, and development.

4.3.1 Pre-planning

Software pre-planning happens one week before the actual iteration planning. During pre-planning, each project and operational team including the team members and chiefs will have their own meetings, discussing what they need to deliver by the end of the next iteration. All project and operational teams including development, testing, marketing, customer care, and human resources take part in this.

In pre-planning the product management team also need to discuss with sales and marketing about the new requirements they want to be delivered. The result of all discussions should then be forwarded to the Research and Development (R & D) team, and only accepted if they can commit to deliver those requirements. Therefore a part of R & D's pre-planning schedule is to review the requests from product management and check the possibilities to assign time and resources to deliver those requirements.

4.3.2 Planning

A few days to approximately a week after software pre-planning, the iteration planning will start. On the first day of iteration planning, which is held in Jolla's local offices or remotely, each team goes through the plans they have agreed on in the pre-planning session. Therefore, it is important that all employees are present¹¹ on this day to hear the ideas and give instant feedback if required.

During the planning day, all teams go through their schedule for the upcoming iteration with the rest of the company. Every team is supposed to describe the main tasks they are going to deliver by the end of the iteration planning day (i.e. the 1-month period). All high level goals need to be documented with possible estimated target dates. While each team is presenting its plans, other teams have the right to comment or disagree with something. This will be followed by an open discussion in the company until the problem is resolved and the majority reach an agreement. If a target for delivering a task cannot be met, there is still a chance during the planning day to change the target date.

¹¹There are cases when an employee(s) cannot be participate to planning day. (S)He needs to inform his/her team chief beforehand.

4.3.3 Sprint planning

The dates for sprint meeting sessions are agreed during iteration planning days. There are two sprints for each iteration in which the high level tasks documented are broken into smaller reports¹², and assigned to related people to be done during the iteration as shown in Figure 3. Available capacity of the developers, and the availability of experts/skills needed for the sprint are estimated, too. In Sprint planning a matching set of skill and capacity is allocated to tackle and complete the agreed tasks.

In Figure 4 you see a tree-view of some tasks in *Bugzilla* - a tool for planning and tracking tasks - and how high level tasks are broken into smaller ones.

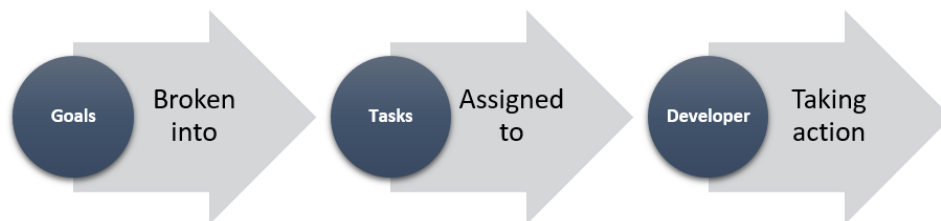


Figure 3: Steps in delivering a goal

4.3.4 Demo

A working software is the primary measure of progress in the development teams. Iteration targets are abstract concepts until we see or touch the tasks/features/bug fixes implemented. Therefore, demonstrating a working, tested software is a concrete. Currently since all Jolla employees have their goal devices at hand, they can check the look and feel of a new feature or a bug fix on their own devices, so demo sessions are mostly skipped unless something which others cannot access on their own devices need to be presented.

4.3.5 Retrospective

At the following iteration planning day, retrospectives for the past iteration will be held as well. During retrospective sessions things that could have been done better

¹²We still read later what tools the company uses to document and track process.



Figure 4: A tree-view of a goal broken down to smaller reports

during previous iteration planning will be discussed. The iteration retrospective is an important mechanism that allows the company to continuously evolve and improve. Everyone gets a chance to air their opinion in an open, honest, yet constructive atmosphere.

4.4 Sailfish operating system

Sailfish is a Linux-based operating system, with a Linux-kernel base for the hardware platform us, the Mer¹³ core middleware, the UI which is designed and maintained by in-house Jolla designers and developers and is closed-source that are maintained separately, and integrated into the operating system¹⁴ (see Figure 5).

¹³To name a few open source packages on Mer: mer-core, mer-tools, hybris-hal common stuff, essentially Hardware Adaptation Development Kit. More about Mer: https://wiki.merproject.org/wiki/Main_Page

¹⁴The property of the content in Figure 5 goes to www.sailfishos.org

Sailfish operating system consists of several components; some are developed on the open (i.e. Mer) and some are property components of the company (i.e. the UI, Android, etc.). The core of the operating system is Open Source¹⁵ while the UI-aspects of it is closed. Therefore, the complete source code cannot be distributed to the public. This means that Jolla needs to specifically define what parts of the source code can be seen by everyone, while other parts should be only viewed by the in-house developers. With this complexity in the operating system, there needs to be more advanced configurations in the way the authorization to the code is defined. When Jolla started, as a start-up company concentrating on only delivering a workable software for its devices, the goal to implement the configurations for custom access to the source code has been put to a delay. It was a time-consuming task which needed the constant work of several developers to be implemented. After implementation, it needed testing to make sure that those who are not authorized to view the property components, will not.

Sailfish applications are packaged with an RPM file extension such as `foo-1.0.4.12.armv7hl.rpm`. RPM is a format of package management system used on Linux-based operating systems for packing applications. Packaging applications also require developers to mention what type of architecture the application works on. In the example mentioned above, `armv7hl`, is the type of architecture that the `foo` application can work on. The type of architecture that an application can run on depends on the processor used on the device that runs the software.

One of the main goals for Jolla, from the early days of development, has been to open up their platform for more OS contribution; however, the un-readiness of proper infrastructure prevented contributors from being able to collaborate since the beginning of the software development process. The purpose of this project was to help reach this goal.

4.5 Software development process

Jolla's software development process for developing Sailfish operating system is done using *Git distributed version control systems*, *Webhook*, *Open Build System (OBS)*, and *Bugzilla*. In simple words, Git is used to share the code; webhooks are used to hook the code to related packages; OBS is used to update the packages according to what

¹⁵Sailfish open source code: <https://github.com/sailfishos>

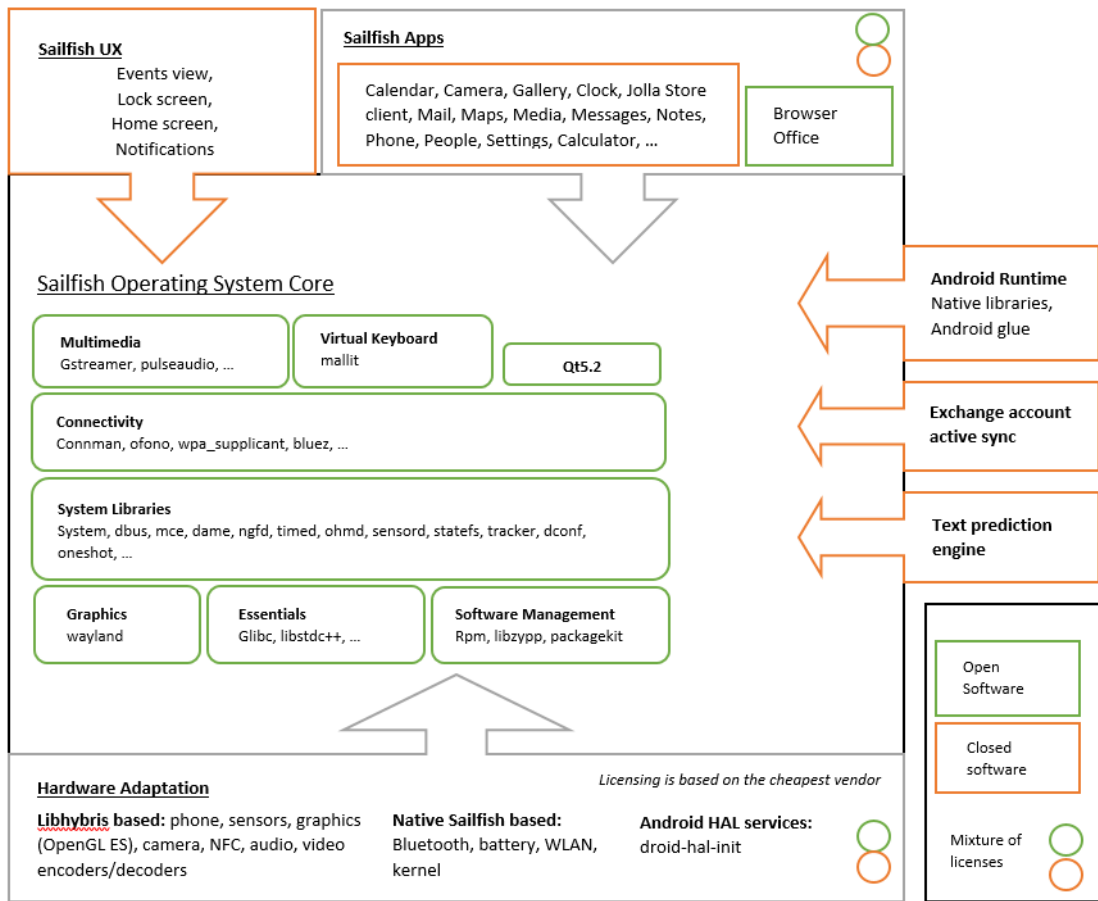


Figure 5: Sailfish operating system architecture including open source and property packages

has changed in the code, and Bugzilla is used to track the whole process. All Jolla employees access these services with specific credentials provided by the company.

The software development team uses Continuous Integration (CI) as the practice for software development process. The CI process is about continuous development of software assets which merges the developers' copies of the working code several times a day to the code in the related branch. The software asset is developed in the master branch, and from the master branch we branch off for making software updates. Therefore, the software releases are created in upgrade¹⁶ branches. This makes it possible to continue feature development in the master while finalizing a release in the upgrade branch. The reason for that is to be able to keep having stable software upgrades while

¹⁶Jolla started changing software 'updates' to software 'upgrade' some time during this project. The document is going to use the word 'update'; however, in some screenshots you might see 'upgrade'. Both words carry the same meaning in this document.

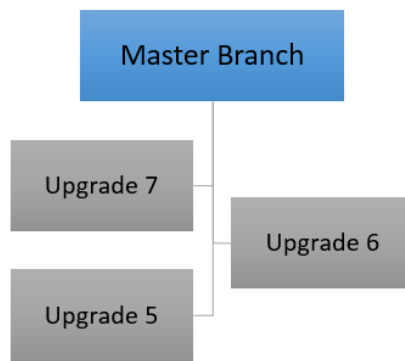


Figure 6: Branching during software releases

non-mature features are being developed in the master branch (see Figure 6).

4.5.1 An example of a Sailfish package on Git

Git distributed version control system containing the OS packages of Sailfish is available for public access. Figure 7 shows an example of a repository *sailfish-browser* on GitHub¹⁷. On the right side there are options to view the source code, branches, commits, pull requests, and downloads. In the middle section, you can find all files that are associated with making the *Sailfish Browser application*. There are 1,464 commits made on this repository. Clicking on that will navigate you to the list of the commits and all its related information (see Figure 9). In commit messages for Jolla repositories, there is also information about what the commit fixes or contributes to. The link to the full report on Bugzilla is written like JB#xxxx or MER#xxxx¹⁸. In Figure 8, JB#27682 is an example link for the commit that was pushed to Git. Such Bugzilla report numbers help the release manager, and the testing team to be able to track in what stage the fix is, and also make proper comments/actions in the report whenever needed.

Navigating to the main page of the repository, we will go through the *.spec* file from the RPM folder (see Figure 10).

Each *.spec* file consists of Tags, Scripts, Macros, File-related directives and condition-

¹⁷The GitHub repository can be found at this address: <https://github.com/sailfishos/sailfish-browser>

¹⁸JB#xxxx: JB corresponds to Jolla Bugzilla, and the xxxx corresponds to the bug number. Explained in detail in Bugzilla section. MER#xxxx: MER corresponds to Mer Bugzilla, and the xxxx corresponds to the bug number More explanation about this in Upstream Bugzilla Sync section

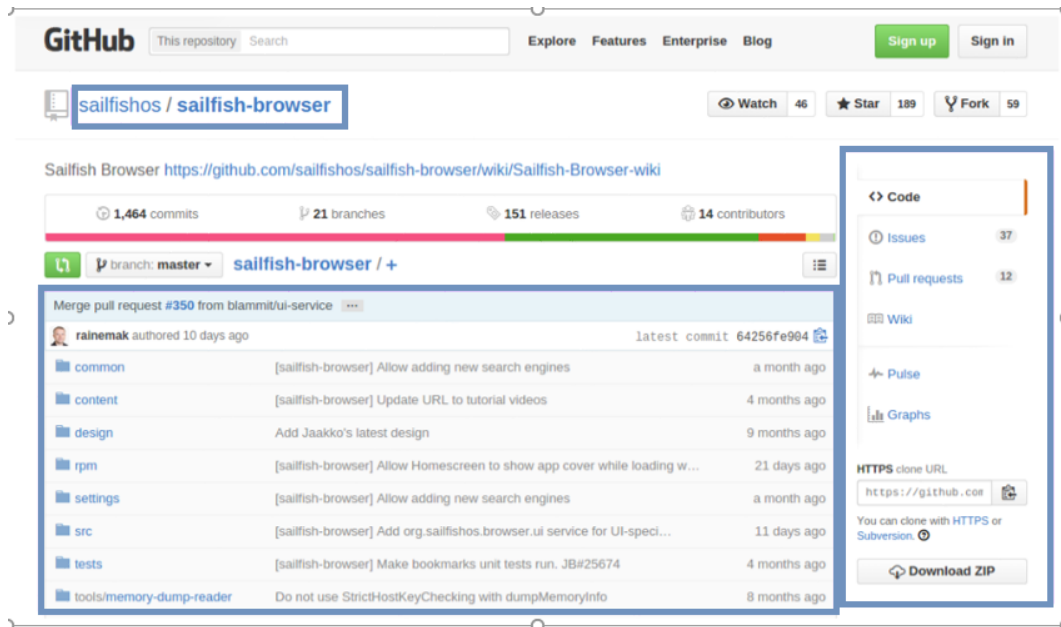


Figure 7: Overview of Sailfish Browser development page on GitHub

als.

- Tags (Not all these are necessary to be included for each RPM)
 - Name: To define the name of the software being packaged
 - Version: To define the version of the software being packaged
 - Release: This shows the number of times the software has been packaged. It follows the version number with '-'. e.g. 1.1.0-2.3 (1.1.0 is the version number and 2.3 is the release number)
 - %description: Describes the package and what it does. The description here can be more than 1 line.
 - Source: Shows the name of the source file that exists in the 'source' directory
 - BuildRequires: Any other packages that are required to be present for the successful build of the main package (i.e. sailfish-browser in our example)

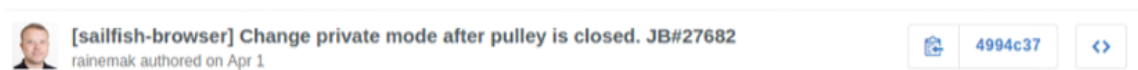


Figure 8: An example of a commit message with a Jolla Bugzilla number

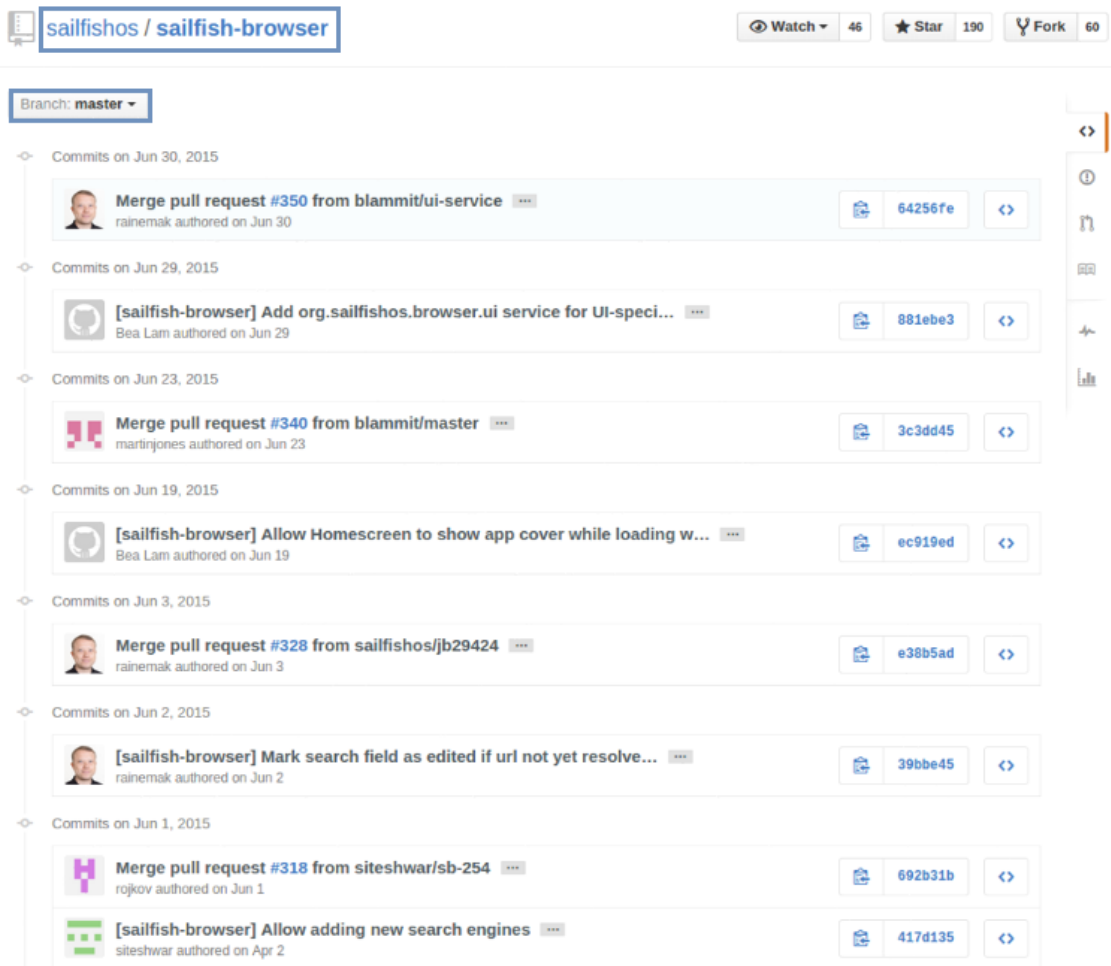
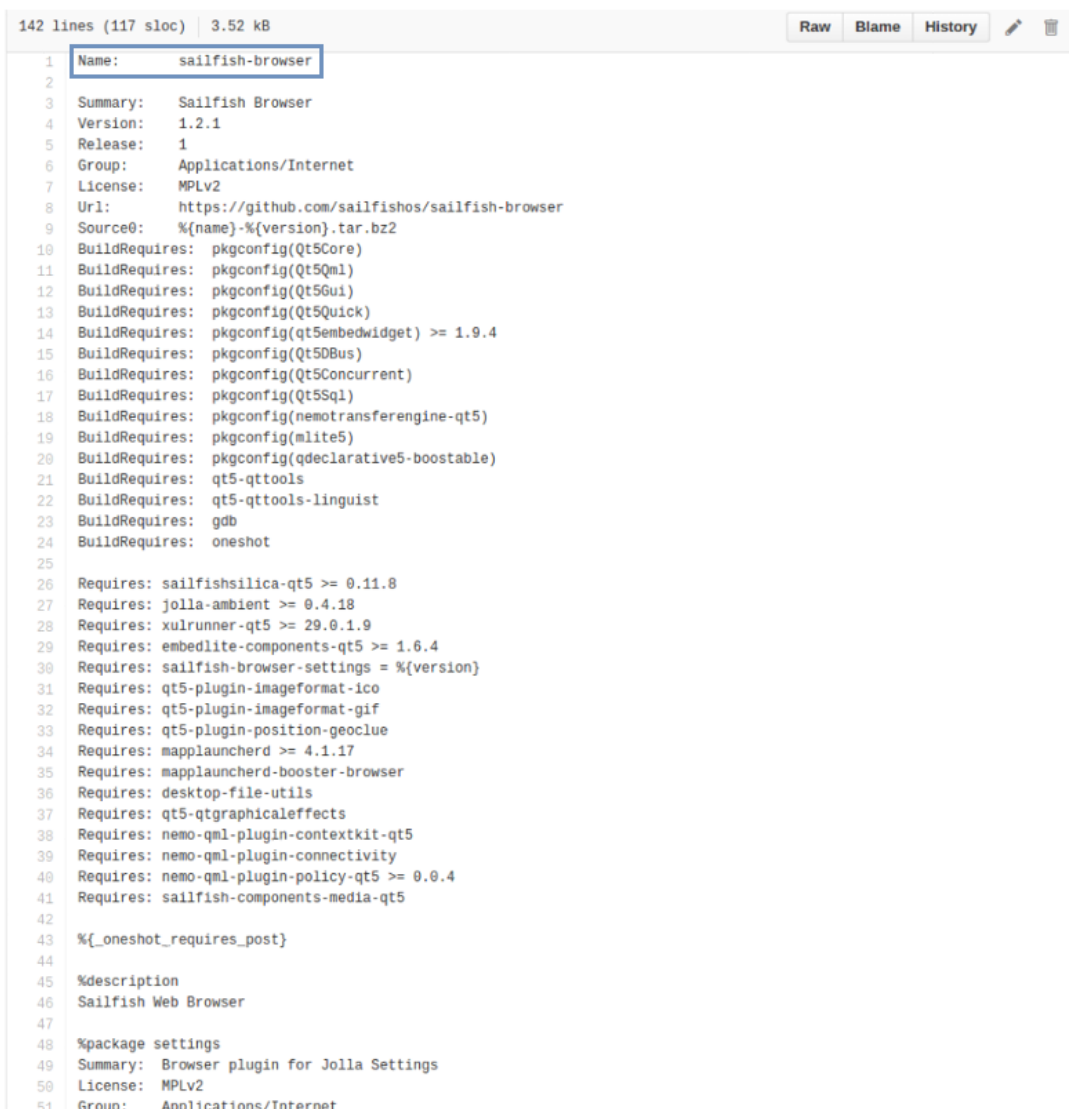


Figure 9: A part of the list of commits to Sailfish Browser application

- Requires(post): Is used to specify the required tools for the post installation section.
- Copyright: To define the copyright terms apply to the software being packaged.
- Distribution: To define a group of packages, of which this package is also a part of
- Icon: For naming the file containing an icon representing the packaged software (seen in RPMs related to applications)
- Vendor: For defining the name of the vendor who is responsible for packaging the software
- Packager: Includes the name of the people who built the package
- Requires: It is used to say that the package needs to have certain capabilities

in order to be installed successfully and operate accordingly

- Conflicts: It is used to mention if anything is not compatible with the package
 - %files: It contains a list of files that are part of the package. If the name is not in the file list, it won't be put into the package. For example for store-client, you can see in which directories it installs files, you can see where the .desktop file is located, etc.
- Scripts
 - %prep: Is executed first while the RPM is being built through the process of



```
142 lines (117 sloc) | 3.52 kB
Raw Blame History
1 Name: sailfish-browser
2
3 Summary: Sailfish Browser
4 Version: 1.2.1
5 Release: 1
6 Group: Applications/Internet
7 License: MPLv2
8 Url: https://github.com/sailfishos/sailfish-browser
9 Source0: %(name)-%(version).tar.bz2
10 BuildRequires: pkgconfig(Qt5Core)
11 BuildRequires: pkgconfig(Qt5Qml)
12 BuildRequires: pkgconfig(Qt5Gui)
13 BuildRequires: pkgconfig(Qt5Quick)
14 BuildRequires: pkgconfig(qt5embedwidget) >= 1.9.4
15 BuildRequires: pkgconfig(Qt5DBus)
16 BuildRequires: pkgconfig(Qt5Concurrent)
17 BuildRequires: pkgconfig(Qt5Sql)
18 BuildRequires: pkgconfig(nemotransferengine-qt5)
19 BuildRequires: pkgconfig(mlite5)
20 BuildRequires: pkgconfig(qdeclarative5-boostable)
21 BuildRequires: qt5-qttools
22 BuildRequires: qt5-qttools-linguist
23 BuildRequires: gdb
24 BuildRequires: oneshot
25
26 Requires: sailfishsilica-qt5 >= 0.11.8
27 Requires: jolla-ambient >= 0.4.18
28 Requires: xulrunner-qt5 >= 29.0.1.9
29 Requires: embedlite-components-qt5 >= 1.6.4
30 Requires: sailfish-browser-settings = %(version)
31 Requires: qt5-plugin-imageformat-ico
32 Requires: qt5-plugin-imageformat-gif
33 Requires: qt5-plugin-position-geoclue
34 Requires: mapplauncherd >= 4.1.17
35 Requires: mapplauncherd-booster-browser
36 Requires: desktop-file-utils
37 Requires: qt5-qtgraphicaleffects
38 Requires: nemo-qml-plugin-contextkit-qt5
39 Requires: nemo-qml-plugin-connectivity
40 Requires: nemo-qml-plugin-policy-qt5 >= 0.0.4
41 Requires: sailfish-components-media-qt5
42
43 %{_oneshot_requires_post}
44
45 %description
46 Sailfish Web Browser
47
48 %package settings
49 Summary: Browser plugin for Jolla Settings
50 License: MPLv2
51 Group: Applications/Internet
```

Figure 10: An overview of the .spec file page for Sailfish Browser

preparing the software for building. In this section the build environment for the software is created. `'%setup -q -n %name-%version'`. In `%prep`, we create the build directory, unpack the sources into that directory, and perform any other actions that are necessary to get the source code into the ready status.

- `%build`: Is the second script that executes during the RPM build. It is responsible for performing the build. It is a shell script, and it does not have the macros like `%prep`.
- `%install`: Is responsible to do whatever that is needed to install the software that is built.
- `%clean`: Is used to clean up the build directory of the software
- `%pre`: Executes before the package is installed
- `%post`: Executes after the package is installed -`%preun`: executes before the package is uninstalled
- `%postun`: Executes after the package is uninstalled

- Macros

- `%setup`: (also mentioned in the `%prep` section) Is used to unpack the original sources while preparing for the build
- `%patch`: It applies patches to the unpacked sources

- File-related directives

- `%config`: Flags the file for being a configuration file
- `%defattr`: `%attr(<file mode>, <user>, <group>, <dir mode>)`. For example for `store-client` both the user and the group are `'root'` (non-numeric), but there are no file/directory mode set for it.

To compare the above fields with the specification (aka `.spec`) file for Sailfish Browser, we go through the `.spec` file of `sailfish-browser`, and check this information (see Table 1).

Tags	The .spec file for sailfish-browser
Name	sailfish-browser
Version	1.2.1
Release	1
Source	%name-%version.tar.bz2
Requires	In sailfish-browser, there are the names of the require packages and the version which should be used. For example: sailfishsilica-qt5
BuildRequires	As one example: pkgconfig/qt5embedwidget
%description	Sailfish Web Browser
%prep	%setup -q -n %name-%version
%setup	%setup -q -n %name-%version
%install	'All the code that is written below the install section'
%defattr	%defattr (-, root, root, -)

Table 1: File specifications for sailfish-browser

4.5.2 Webhooks

Webhooks are user-defined callbacks over HTTP. They are intended to make web applications become more extensible, customizable and ultimately more useful. During software development process at Jolla we use git distributed version control systems and Webhooks. On Settings page of Git, there is the option to choose to which HTTP link you want to hook your commit to (see Figure 11). One needs to add Jolla's specific webhook link in that section. What happens after adding the webhook link there, is that, whenever there is a new commit pushed to Git, the hook will be created for it automatically and will be shown on Jolla webhook mappings.

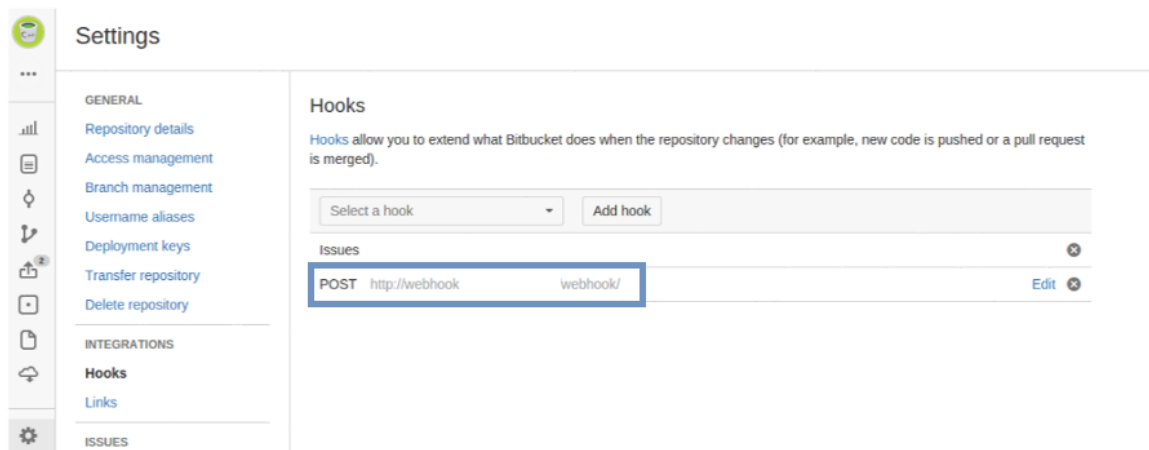


Figure 11: Webhook settings on Git

Going through webhook mappings, one can find all that have been hooked to webhook. In our example case, Sailfish Browser, if I search for sailfish-browser in search bar, there is a list of all hooks, which are the links to the repositories with the Git branch name, project name, package name, and the user name (see Figure 12).

Select web hook mapping to change

Search: sailfish-browser 28 results (11213 total)

Action: [dropdown] Go 0 of 28 selected

Report	Branch	Project	Package	Notify	Build	User
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git	upgrade-1.1.9	sj-apps-1.1.9	sailfish-browser	🟢	🟢	admin
<input type="checkbox"/> https://pootle.jollamobile.com/git/sailfish-browser.git	upgrade-1.1.9	sj-non-oss-1.1.9	sailfish-browser-112n	🟢	🟢	admin
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git	beta30389	sj-oss		🔴	🔴	admin
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git	upgrade-1.1.7	sj-apps-1.1.7	sailfish-browser	🟢	🟢	admin
<input type="checkbox"/> https://pootle.jollamobile.com/git/sailfish-browser.git	upgrade-1.1.7	sj-non-oss-1.1.7	sailfish-browser-112n	🟢	🟢	admin
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git	upgrade-1.1.6	sj-apps-1.1.6	sailfish-browser	🟢	🟢	mkakela
<input type="checkbox"/> https://pootle.jollamobile.com/git/sailfish-browser.git	upgrade-1.1.6	sj-non-oss-1.1.6	sailfish-browser-112n	🟢	🟢	admin
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git	upgrade-1.1.5	sj-apps-1.1.5	sailfish-browser	🟢	🟢	mkakela
<input type="checkbox"/> https://pootle.jollamobile.com/git/sailfish-browser.git	upgrade-1.1.5	sj-non-oss-1.1.5	sailfish-browser-112n	🟢	🟢	admin
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git	upgrade-1.1.4	sj-apps-1.1.4	sailfish-browser	🟢	🟢	admin
<input type="checkbox"/> https://pootle.jollamobile.com/git/sailfish-browser.git	upgrade-1.1.4	sj-non-oss-1.1.4	sailfish-browser-112n	🟢	🟢	admin
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git	update11	sj-apps-1.1.2	sailfish-browser	🟢	🟢	mkakela
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git	update11	sj-oss		🔴	🔴	admin
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git	update10	sj-non-oss		🔴	🔴	admin

Figure 12: An overview of sailfish-browser webhook mappings

To view the webhook page more closely, in Figure 13 and Figure 14, there is information about Git links, project name, package name, and the user who pushed the commit to Git.

Select web hook mapping to change

Search: sailfish-browser 28 results (11213 total)

Action: [dropdown] Go 0 of 28 selected

<input type="checkbox"/> Report
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git
<input type="checkbox"/> https://pootle.jollamobile.com/git/sailfish-browser.git
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git
<input type="checkbox"/> https://pootle.jollamobile.com/git/sailfish-browser.git
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git
<input type="checkbox"/> https://pootle.jollamobile.com/git/sailfish-browser.git
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git
<input type="checkbox"/> https://pootle.jollamobile.com/git/sailfish-browser.git
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git
<input type="checkbox"/> https://pootle.jollamobile.com/git/sailfish-browser.git
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git
<input type="checkbox"/> https://github.com/sailfishos/sailfish-browser.git

Figure 13: Git links which point to related repositories

4.5.3 Open build systems (OBS)

OBS offers the chance to more people to collaborate in a project or package by preparing changes in a branched project, which is a copy of the original project, and then merge them back. The merges can be approved upon submission, or sometimes in bigger projects, the changes need to go through another revision after merging to be completely checked. What one needs to do in order to take part or collaborate in a project is to make a branch of the main project and start working on it. After applying the desired changes, one can submit them by merging them into OBS. If a project needs to be accepted by another maintainer, he/she checks the changes and will accept the submission if satisfied with it.

In OBS¹⁹, packages are built locally or on a remote build server. Each package can be built multiple times. The multiple builds can be used for combination of repositories and architectures. In each build the packages are taken from the target repository. If any packages change, OBS makes sure that packages which have the same build-dependency against it are also rebuilt. If this has an effect on changing the package binaries, the rebuild chain continues. Git repositories store all sources, and OBS builds packages or projects from those sources. Webhooks are used to link OBS to Git repositories and make package building possible. How the linkage works is that each time a new commit is pushed to the Git repository, webhook will get notified about it and the change will appear as a Git link in webhook mappings. Afterwards the CI-bot user, which is a robot user in OBS packages, will automatically tell OBS to rebuild all the

¹⁹Jolla and Mer each have their own OBS system. Mer OBS can be found here: <https://build.merproject.org/>. Link to Jolla's OBS cannot be disclosed due to confidentiality.

Branch	Project	Package	Notify	Build	User
upgrade-1.1.9	pj:apps:1.1.9	sailfish-browser	✔	✔	admin
upgrade-1.1.9	pj:non-oss:1.1.9	sailfish-browser-110n	✔	✔	admin
jib30389	pj:oss		✘	✘	admin
upgrade-1.1.7	pj:apps:1.1.7	sailfish-browser	✔	✔	admin
upgrade-1.1.7	pj:non-oss:1.1.7	sailfish-browser-110n	✔	✔	admin
upgrade-1.1.6	pj:apps:1.1.6	sailfish-browser	✔	✔	rmakelai
upgrade-1.1.6	pj:non-oss:1.1.6	sailfish-browser-110n	✔	✔	admin
upgrade-1.1.5	pj:apps:1.1.5	sailfish-browser	✔	✔	rmakelai
upgrade-1.1.5	pj:non-oss:1.1.5	sailfish-browser-110n	✔	✔	admin
upgrade-1.1.4	pj:apps:1.1.4	sailfish-browser	✔	✔	admin
upgrade-1.1.4	pj:non-oss:1.1.4	sailfish-browser-110n	✔	✔	admin
update11	pj:apps:1.1.2	sailfish-browser	✔	✔	rmakelai
update11	pj:oss		✘	✘	admin
update10	pj:non-oss		✘	✘	admin

Figure 14: Other information about the webhook mappings

packages that are dependent on that Git repository. For an overview on OBS packages, Figure 15 shows a short list of some of the OS packages currently available on OBS for Sailfish operating system. Among the list of OS packages you can see *mer:core* which is on Mer.

```

All Public Projects (863)

b | BowSprit | BowSprit:1.0.0.5 | BowSprit:1.0.0.5:armv7hl | BowSprit:1.0.0.5:i486 | BowSprit:1.0.1.10 | BowSprit:1.0.1.10:armv7hl | BowSprit:1.0.1.10:i486 |
BowSprit:1.0.2.5 | BowSprit:1.0.2.5:armv7hl | BowSprit:1.0.2.5:i486 | BowSprit:1.0.3.8 | BowSprit:1.0.3.8:armv7hl | BowSprit:1.0.3.8:i486 | BowSprit:1.0.4.20 |
BowSprit:1.0.4.20:armv7hl | BowSprit:1.0.4.20:i486 | BowSprit:1.0.5.16 | BowSprit:1.0.5.16:armv7hl | BowSprit:1.0.5.16:i486 | BowSprit:1.0.6.16 |
BowSprit:1.0.6.16:armv7hl | BowSprit:1.0.6.16:i486 | BowSprit:1.0.7.16 | BowSprit:1.0.7.16:armv7hl | BowSprit:1.0.7.16:i486 | BowSprit:1.0.8.19 |
BowSprit:1.0.8.19:armv7hl | BowSprit:1.0.8.19:i486 | BowSprit:1.1.0.39 | BowSprit:1.1.0.39:armv7hl | BowSprit:1.1.0.39:i486 | BowSprit:1.1.1.27 |
BowSprit:1.1.1.27:armv7hl | BowSprit:1.1.1.27:i486

c | CentOS:6.0 | CentOS:7.0

d | Debian:6.0 | Debian:7.0

e | Epel:6.0 | Epel:7.0

f | Fedora:20

m | mer | mer:cobs | mer:core | mer:core:1.1.2 | mer:core:1.1.2:release | mer:core:1.1.2:testing | mer:core:1.1.3 | mer:core:1.1.3:release | mer:core:1.1.3:testing |
mer:core:1.1.4 | mer:core:1.1.4:release | mer:core:1.1.4:testing | mer:core:1.1.5 | mer:core:1.1.5:release | mer:core:1.1.5:testing | mer:core:1.1.6 |
mer:core:1.1.6:release | mer:core:1.1.6:testing | mer:core:1.1.7 | mer:core:1.1.7:gate:sb2-tools-i486 | mer:core:1.1.7:release | mer:core:1.1.7:testing |
mer:core:1.1.9 | mer:core:1.1.9:release | mer:core:1.1.9:testing | mer:core:feature:jb:22071 | mer:core:feature:jb:24715 | mer:core:feature:jb:29520 |
mer:core:feature:jb:29646 | mer:core:feature:jb:30351 | mer:core:feature:jb:8144 | mer:core:gate:cross-glibc-headers-inject |

```

Figure 15: An overview of some open packages for Sailfish operating system on OBS

With OBS, as well as some instances of version control systems, there are configurations to make a set of packages or repositories accessible to a specific group of people, such as employees of a company. In this case, to be able to view or access the files, one needs to sign in with company-specific e-mail address, or ask the administrator to add them to the list of authorized users.

4.6 Tracking system for planning and task tracking: Bugzilla

Bugzilla is a web-based *bugtracker* originally developed and used by the Mozilla project, and is licensed under the Mozilla Public License. It has been adopted by many organizations for use as a bug tracking system for both OS software and proprietary projects and products.

Jolla uses Bugzilla for many other tasks other than only bug reporting. It is used for documenting information regarding research and development, marketing, planning, legal issues and business management. Each report created on Jolla Bugzilla, consists of a thorough description of the matter, as well as targets or deadlines, if any. Reports filed in Jolla Bugzilla are not only about reporting glitches or defects in the software or hardware; however, in this document, the word ‘bug’ only refers to flaws that need

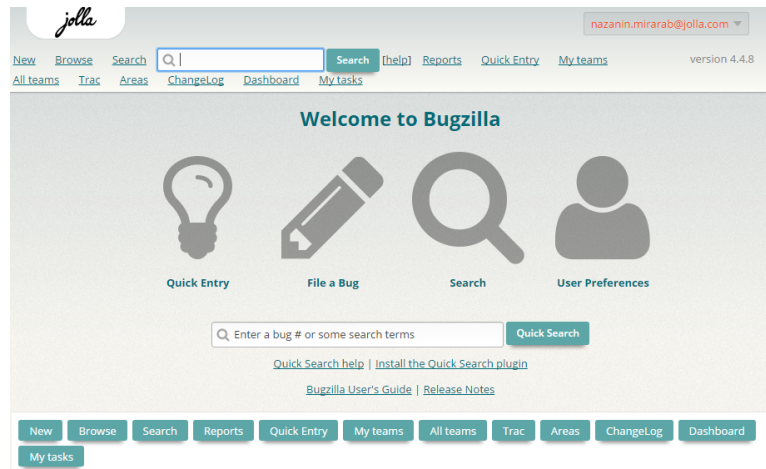


Figure 16: An overview of the first page of Jolla Bugzilla

to be fixed. Figure 16 shows a general overview of the Jolla Bugzilla page. In the middle of the page, there are four selections named Quick entry, File a bug, Search and Preferences. Other important options are search for a report, create a new report, browser Bugzilla, view different team, view different areas of operation, and see who is logged in.

4.6.1 Quick entry

For issues which need to be reported as fast and easy as possible, this entry works best (see Figure 17). All that should be done is to write a title and a short summary of the issue that you want to report. Everything else can be set to default in case otherwise stated. Fields with asterisk are mandatory.

From the drop down menu in this *Product*, you can choose the group(s) that can view the entry after it is submitted. Whether it is for in-house employees , subcontractors , or specific to any external groups working with Jolla. After choosing the product type, having the *Component* set to backlog makes sure the data will be sent to backlog and is available for later checks. *Summary* gives a title which explains in a short phrase or sentence what the reported issue is about. Any other information about the report goes in the comments section. The *Type* field specifies if the issue is a *Goal*, *Task*, *Story*, or *Bug*. The *Area* field brings on a drop-down menu with a long list of areas functioning in Jolla. Any reported issue in Bugzilla must be assigned to an area. The checkbox besides a field is t Keep the value of that field for further ideas. After checking the

The image shows a web form for creating a bug report in Jolla Bugzilla. The form is organized into several sections:

- Clone from:** A text input field at the top.
- * Product:** A dropdown menu set to "External" with a checked checkbox to its right.
- * Component:** A dropdown menu set to "Backlog" with a checked checkbox to its right.
- * Summary:** A text input field with an unchecked checkbox to its right.
- Comment:** A section with "Templates:" followed by three buttons: "[Bug Report]", "[Change Request]", and "[Release Exception]". Below this is a large text area for the comment, with an unchecked checkbox to its right.
- * Type:** A dropdown menu set to "---" with an unchecked checkbox to its right.
- * Area:** A text input field with an unchecked checkbox to its right.
- Blocks:** A text input field with a checked checkbox to its right.
- Depends on:** A text input field with an unchecked checkbox to its right.
- Orig. Est.:** A text input field with an unchecked checkbox to its right.
- Keywords:** A text input field with an unchecked checkbox to its right.

A green "Submit" button is located at the bottom left of the form.

Figure 17: Quick entry page in Jolla Bugzilla

checkbox, all settings will be preserved for the next entry you want to write. After all is done, hitting the *Submit* button will submit a new report in Jolla Bugzilla, as well as the real-time e-mail notification sent to those who subscribed to a specific area mentioned in the report.

4.6.2 File a report using a simple view

Selecting this option from the main page will open up a more detailed page for a report entry. However, this page also has a *simple* and an *advanced* settings page. The simple view fields such as Product, Component, Type, Area and Summary are the same as what was described in the Quick Entry section (see Figure 18). Although the Description field is not marked as a mandatory field, it is often filled with either a short or a thorough description of the issue. In this section we will go through the *Type* field

in more detail. We will also study more about the *Severity* and *Priority* fields, and what needs to be written in the description box.

Before reporting a bug, please read the [bug writing guidelines](#), please look at the list of [most frequently reported bugs](#), and please [search](#) for the bug.

[Show Advanced Fields](#) (* = Required Field)

* Product: External Reporter: nazanin.mirarab@jolla.com

* Component: Backlog Component Description: Product backlog

* Version: unspecified Severity: normal

Hardware: Any OS: Sailfish

* Type: ...

* Area:

* Summary:

Description: Templates: [Bug Report] [Change Request] [Release Exception]

Attachment: Add an attachment Submit Bug

Figure 18: Report entry with a simple view on Jolla Bugzilla

The type field identifies what kind of report is going to be filed on Bugzilla. According to table 2, there are four types of reports in Jolla Bugzilla.

A report in Bugzilla can be related to more than one area, and is not necessarily tied to only one. For example, if someone finds an issue with how the Browser application looks on Homescreen, both developers of the Browser area and Homescreen area should be added to the bug report. Only a developer from one area might need to fix the issue, but including all related areas makes sure that developers for both areas are aware of the bug. Areas can be selected upon filing a report then adjusted later in case more areas are needed to be added or some are needed to be removed. As soon as one starts typing the name of an area in the text field of Area, a drop down menu opens up and shows the user a list of the existing areas.

For any goals/stories/tasks/bugs, it is important that the severity matches the actual impact of the defect or the importance of having a feature. According to Table 3 the severity of a report can be selected upon filing it, and then adjusted later in case needed. Severity of a report on Bugzilla is also divided into four groups:

Type	Description
Goal	It is a structural ‘meta’ item that can refer to implementation of a bigger feature (e.g. a new application). A Goal can be achievable during one or more iteration planning depending on how big the task is and how much resources are available to implement it.
Story	A specific and distinctive feature. It can be done as one big task or can be split into several tasks. It is possible to achieve the target in a Story within one iteration period.
Task	A child of a Story or one specific feature independent of any stories. This type is used when talking about small tasks or changes that are not flaws or abnormalities, but would be good to be implemented (e.g. feature requests). The priority for when they will be fixed depends on the team and company planning.
Bug	Refers to a defect in software/hardware or process. It is about deviation from the expected or reasonable way of behaving.

Table 2: Different statuses in a Bugzilla report

Severity	Description
Minor	For small glitches or flaws one needs to polish to reach to perfection.
Normal	The normal severity is selected by default while filing a new report.
Major	For issues that are causing crashes of applications on the device. As an example if opening 5 tabs in Browser causes it to crash (i.e. Close), the report needs to have the major severity.
Critical	For issues which make the whole device completely unresponsive, cause the device to reboot, or cause data loss.

Table 3: Different severities in a Bugzilla report

4.6.3 File a report using an advanced view

When navigating to the advanced settings, some extra fields and options will appear including *Priority*, *Status*, *Assignee*, *CC list*, *Target* and *Deadline* as shown in Figure 19. All of these are optional, and are selected if necessary.

The screenshot displays a web form titled "Hide Advanced Fields" with a legend indicating that fields marked with an asterisk (*) are required. The form is organized into several sections:

- Product:** External
- Reporter:** nazanin.mirarab@jolla.com
- Component:** Backlog (with a scrollable list below)
- Component Description:** Product backlog
- Version:** unspecified (with a scrollable list below)
- Severity:** normal (dropdown)
- Hardware:** Any (dropdown)
- OS:** Sailfish (dropdown)
- Priority:** Normal (dropdown)
- Status:** NEW (dropdown)
- Assignee:** triage_needed@jollamobile.com
- CC:** (empty text input)
- Default CC:** eric.leroux@jollamobile.com
- Orig. Est.:** (empty text input)
- Deadline:** (empty text input with a calendar icon)
- Alias:** (empty text input)
- URL:** http:// (text input)
- Type:** --- (dropdown)
- Area:** (empty text input)
- Target:** --- (dropdown)
- Summary:** (empty text input)

At the bottom, there are buttons for "Description: Templates: [Bug Report] [Change Request] [Release Exception]".

Figure 19: The advanced view

Depending on the severity, type and description of a report, the level of priority it should be dealt with changes. Priority indicated the level of commitment needed to be put into taking care of a report. The priority of a report can be selected upon filing it, and changed later in case needed. According to Table 4 There are three kinds of priorities:

The Target field is used to state in which software upgrade a goal/story/task/bug is planned to be delivered. This field is used to help in planning and keeping track of things landing in a software upgrade. It is also used to track what changes need to be

Priority	Description
Optional	This means what is asked for in the description can be delivered whenever possible.
Normal	This level of priority is selected automatically upon filing a report. This means the issue will be dealt with as planned, or can change later if needed.
Mandatory	Reports with a mandatory priority should be delivered within an iteration or sprint.

Table 4: Different priorities in a report on Jolla Bugzilla

made in manual or automated test cases. This field can be set at the time of creating a report or a while after that. However, the input in the target field is not carved in stone, and if an item cannot be delivered for a specific software upgrade, with proper release exception description explaining why it cannot be delivered, the target field can change for later software upgrades.

An assignee for a report is the maintainer of the software component the report lands into. An assignee can be set upon filing a report, or after it. Usually after a report is created, area owners of the areas mentioned in it will receive a notification. After that, they assign the goal/story/task/bug to themselves and give a target for when they can deliver it.

The Status indicates the various states of a report in its lifecycle; from being filed until taken care of.

A release exception may be required when an important feature cannot be completed and tested on time for the feature integration deadline and needs to be part of the release in the making. It is also for cases when a feature needs to be reverted because it is not ready enough to be used on by end users. Release exception requests are filed in Bugzilla as well, and there is a default template for them available in the report description.

A change request may be required when an important feature, or an added component does not work as it is planned to and need to be either changed or reverted.

Release blocker reports are the ones that are blocking a release and must be fixed before the release is rolled out to end users. Jolla uses the whiteboard field (see Figure 20) to identify what release version number a specific bug is blocking.

Bug 24761 - [BUG] Device fails to list wifi network when there's only one in the range (edit) Save Changes Suppress mail

Status: **RESOLVED COMMITTED** (edit)

Product: External
Component: Backlog
Version: unspecified
Hardware: Any Sailfish

Importance: Normal normal
Assigned To: (edit) (take)

URL:
Whiteboard: update10_blocker
Keywords:
Tags:
Depends on:
Blocks:
Show dependency [tree](#) / [graph](#)

Reported: 2014-12-01 12:26 EET by
Modified: 2014-12-29 02:04 EET (History)
CC List: Add me to CC list
8 users (edit)

See Also: (add)
Type: bug
Test Case: Exists
Type of Test: Unit Test
Robot Test
Feature Test
Manual Test
Area: Connectivity Connectivity UI Design
Target: ---
Pool: N.O.W sprint 2014W48-51
[Bug list](#)
Pool ID / Order: 422 / 25
[Activity graph](#)
[To Master Plan](#)
TreeView+ [depends on](#) / [blocked](#)
Flags: None yet set (set flags)

Hide advanced fields

Figure 20: A report that is a release blocker for update 10

4.6.4 Description of a report in Jolla Bugzilla

The description of a report needs to be as precise as possible. It needs to explain the main problem and the preconditions or prerequisites needed to reproduce that problem. The report also needs to describe any type of hardware, or provide other information by attachments. Figure 21 gives an overview of a typical report description on Jolla Bugzilla. After filling out the above info, one will submit the report. Table 5 gives an example for a bug report description for the Browser application on Tablet.

4.6.5 What happens after a report is created on Jolla Bugzilla

Since Bugzilla reports have a real-time connection with reader, at the same time the report is created in the database, the email notification is generated to those who have enabled this option. The developers analyze the report and in case of validity, assign it to themselves in order to fix/implement it. Figure 22 shows an overview of a bug

Title	Description
REPRODUCIBILITY	70%
BUILD VERSION	1.1.9 RC3 (1.1.9.2)
HARDWARE	Tablet
LANGUAGE	Finnish
REPORT DESCRIPTION	When watching videos uploaded to Facebook (URL starts with video.xx.fbcdn.net), Browser gets killed to Homescreen on some occasions. Seems to be reproduced more reliably when 4-5 tabs are open.
STEPS TO REPRODUCE	<ol style="list-style-type: none"> 1. Have a few tabs open 2. Open Facebook in Browser 3. Go to a page with a lot of videos (e.g. 9GAG and LADBible have a bunch) 4. Go to 'Videos' category of the mentioned websites. 5. Try to open and watch a few videos, leaving their tabs open.
EXPECTED RESULT	Browser continues to function as expected.
ACTUAL RESULT	Browser minimizes to Homescreen and need to be restarted via the application active cover on Homescreen.
ADDITIONAL INFO	The issue seems more common with Facebook. Managed to reproduce once on Youtube.com

Table 5: An example of a report description on Jolla Bugzilla

```

DESCRIPTION
*****
...

PRECONDITIONS
*****
...

STEPS TO REPRODUCE
*****
1.
2.
3.

EXPECTED RESULT
*****
...

ACTUAL RESULT
*****
...

REPRODUCIBILITY:
BUILD VERSION & TYPE:
PLATFORM:
LANGUAGE:

(Please, ALWAYS attach relevant data to your report,
such as logs, images, etc...)

```

Figure 21: Bug description outline

report after it is filed on Jolla Bugzilla.

Bug List: (1 of 1) [First](#) [Last](#) [Prev](#) [Next](#) [Show last search results](#)

Bug 30213 - [BUG] Options in Browser after log-tapping on a link/image, are not easily click-able [\(edit\)](#)

[Start Working](#) [Save Changes](#)
 Suppress mail

<p>Status: NEW (edit)</p> <p>Product: External</p> <p>Component: Backlog</p> <p>Version: unspecified</p> <p>Hardware: Jolla 1 Sailfish</p> <p>Importance: Normal normal</p> <p>Assigned To: Not Taken, feel free to take (edit) (take)</p> <p>URL:</p> <p>Whiteboard: L3</p> <p>Keywords:</p> <p>Tags:</p> <p>Depends on:</p> <p>Blocks:</p> <p>Show dependency tree / graph</p>	<p>Reported: 2015-06-25 11:41 EEST by Nazanin Hajmirarab</p> <p>Modified: 2015-07-08 16:10 EEST (History)</p> <p>CC List: 6 users including you (edit)</p> <p>See Also: 29375 <input type="checkbox"/> Remove (add)</p> <p>Type: bug</p> <p>Test Case: ...</p> <p>Area: Browser</p> <p>Target: ...</p> <p>Pool: None</p> <p>Activity graph To Master Plan</p> <p>TreeView+ depends on / blocked</p> <p>Flags: None yet set (set flags)</p> <p>Hide advanced fields</p>
--	---

Figure 22: An overview of a report filed on Bugzilla

The See-Also field is used in Jolla for two main reasons: Either to link internal Bugzilla reports to each other, or to track items from other Bugzilla instances, such as Mer. It is also used to track items created on together.jolla.com²⁰, which is a forum for Jolla users.

²⁰together.jolla.com is a community for all Jolla users. In this community, users report issues, ask

The See-Also field plays an important role in this project. In this field one needs to put the URL from one of the distance Bugzilla instances. Once a URL is added there, the created report is linked to that URL to get information from that distant item. A thorough description of how this field exactly operates will be covered in the Upstream Bugzilla Sync section.

4.7 The release process

The release process in Jolla consists of various parts. As mentioned earlier in this chapter, there are Git, Webhooks, OBS, Jolla Bugzilla, CI-bot, and last but not least, Jolla crew with their OS community contributors to make the Sailfish release land sanely on the devices. It is explained in the following lines how the whole release process works, from different release levels and quality assurance work flow to Bugzilla integration and the CI process.

The release process in Jolla consists of three levels: *devel* (as in development), *testing*, and *release* (see Figure 23).



Figure 23: Release cycle levels

During all these levels, Jolla needs packages from two main sources:

1. Jolla in-house developers who deploy on Git, and push to webhook.
2. Mer developers who deploy on Git on the open, have an exclusive hook for Jolla webhook, and push them to Jolla's webhook.

Before this project Jolla used to cherry-pick the changes from Mer in webhook and integrate them into their software releases (see Figure 25). Before going in depth with

questions, and in general are in contact with Jolla employees such as the release manager, project manager, developers and testers.



Figure 24: A bug status starts from ‘New’ and ends in a ‘Released’

different release process cycles, the terms *CI-bot*, *Changelogs*, and Bugzilla statuses need to be described.

4.7.1 CI-bot

CI-bot is a continuous integration robot developed to replace the manual tasks by doing them automatically when given the command. During this project CI-bot is used to make a connection between a Bugzilla report and its corresponding pull requests from Git, and to link the Bugzilla reports numbers mentioned at the end of commit messages (JB#xxx and MER#xxx) in the changelogs.

4.7.2 Changelogs

Changelogs are the list of changes which inform what has been changed from one release candidate to another. For Sailfish OS software development, changelogs are the list of commit messages saying what the commit contributes to/fixes. Changelogs are available both internally for Jolla employees, and publically for the end users. It can be gained by typing the following command on the device terminal application:

```
rpm -q --changelog package_name
```

4.7.3 Bugzilla statuses and their meanings

Status indicates the various states of a Bugzilla report in its lifecycle; from being filed until getting fixed or implemented.

To explain Figure 24, a report that is just created usually has the ‘New’ status. After a developer takes the report to fix/implement it, he/she assigns it to him/herself, and the status will change from New to Assigned. The developer commits to the bug and starts hacking in Git and creating a branch from the developing repository. As soon as

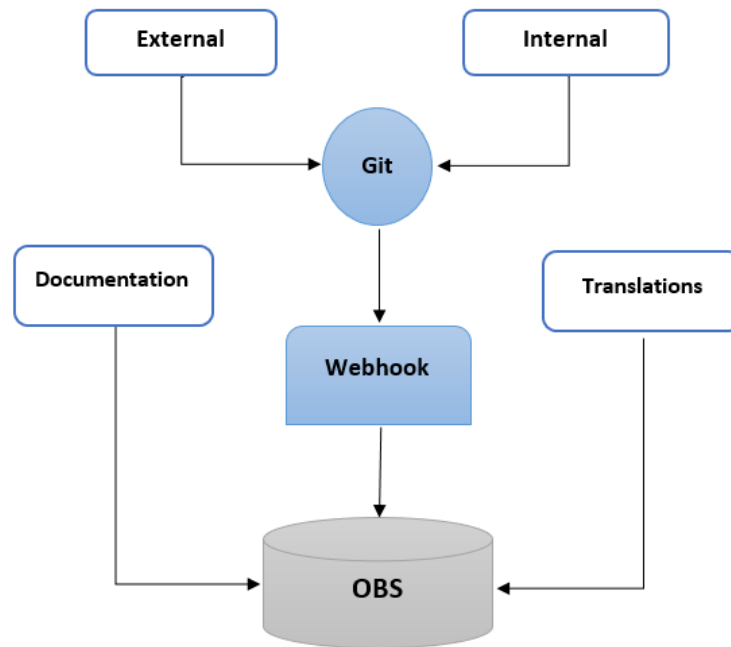


Figure 25: Where the code for a software release comes from

the commit lands in the development level in release cycle, the Bugzilla report status changes from Assigned to Committed. Next, the commit lands into the software update in testing gate. This is when the bug status will change from Committed to Fixed. At this point there is a fix, but it is not in the final software release update. When the fix lands in the release candidate, the bug status changes from Fixed to Released.

Goals, Stories and Tasks have the same situation as Bugs regarding their statuses. Status for Bugzilla reports other than bugs are set manually by either the project manager or the team chief after knowing that the report is done and implemented, or the bug reports on which a Task/Story/Goal was dependent are all resolved.

The whole process with Bugzilla status changes is done both automatically and manually, depending on how the developers want. For the automatic part, CI-bot changes the Bugzilla status and comments on the report with the changes that have been made. The reason to use CI-bot is to save developer's time to do other important stuff instead of checking a report constantly and updating its status manually. CI-bot also messages make it very clear for everyone to see what has changed, when it changed, who changed it.

4.7.4 Development level

This is the first stage in CI where quality assurance for the release cycle comes in. All changes planned for OS update releases are first pushed here.

As explained in the software development process section, the packages that build Sailfish OS are picked from two sources:

1. Internal Jolla developers deploy via Git and push the changes to webhook. The changes will then go through Jolla's OBS to be built. At this stage, for all the commits with Bugzilla numbers, there will be a comment from CI-bot automatically put to that report, so the report will be committed. Therefore, the Bugzilla status will be resolved/committed.
2. Jolla picks the needed changes from Mer developers who also push commits to Jolla webhook. These will again go through OBS. However, there are no Bugzilla report numbers for the commits made from Mer side. Also some of the Bugzilla references currently showing up in the changelog are internal reports of public things which we wanted to move to the open by doing this project.

After changes from documentation and translation servers also go through build, OBS submits a request for package testing. Packages go through automated tests, and finally if all pass, image will be built for the devel repository. The automated tests done in during the release cycle are previously written by the automation team, using mechanical and virtual robots, to be run with each build of the release software. After a release hits the development level, it automatically goes through a set of automated test cases which check the basic functionalities of the software. If any of these tests fail, the release manager or a member from the automation team needs to check the logs for the failed test cases. Sometimes failing happens due to reasons unrelated to the software, such as crash in robot systems. In these cases, the test case is run again manually on the software and the results from the re-run tests determines if there really is a defect in the software or not.

4.7.5 Testing level

Continuing from the previous level, all the accepted packages go to the testing level, along with a submit request which include fixes from both Jolla and Mer developers. On Jolla side, for the commits including a Bugzilla number, CI-bot will post a comment to the related goal/story/task/bug and change the status of it to Fixed. Everything will go through Jolla testing repository and ready for tests. The system testing here is done through automated test cases mentioned in the development level. In some cases, release manager might request additional manual testing, by the QA team members, on the image in the Testing level. This happens at times when there might be doubts by the release manager that the automated tests have not given accurate results.

4.7.6 Release level

Continuing from the previous level, all changes accepted by the release manager will go to the release level. The process is the same as two other levels: Changes from Mer and Jolla developers will go through Jolla release repository after the submit request is done, and the image will be ready for release testing. In this stage, for commits which contribute to specific Bugzilla reports, CI-bot will post a comment and change their status to Released

4.7.7 Release snapshot

With the test results being accurate from the previous level, in this level, the final snapshot of the release candidate will be made and put for manual testing by the testing team. Manual System Testing is done by going through thousands of test cases. All the test cases have explicit steps, so that anyone reading the test case can understand how to perform it. These test cases go through different applications on the software, such as Settings, Phone, Gallery, Camera, Store, etc., and test all the available features, options, and sub-pages in these applications. Manual test cases also consist of test cases related to power management, hardware configurations such as Bluetooth, SD-cards, USB, etc.

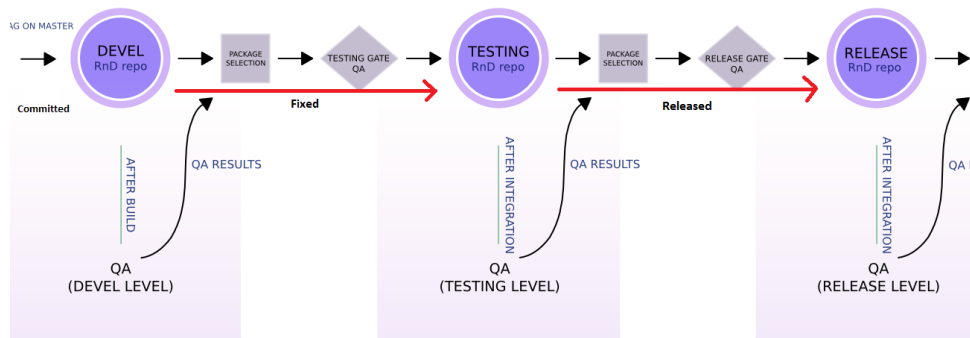


Figure 26: Bug life cycle during software integration

4.7.8 Life cycle of reports in Jolla bugzilla during the integration process

Figure 26 shows in detail the process of a goal/story/task/bug status since it is created until it is resolved completely.

After a new report is filed and assigned to a developer, he starts developing a fix for it on Git. What he does for the commits he pushes, is to add the Bugzilla number like 'JB#number' at the end of the commit message. This will help CI-bot to trigger which commit contributes to which Bugzilla report, and then comments are integrated automatically to the report during release process. After the developer deploys a fix for a report, he can hither manually put a PR²¹ link to the bug and ask fellow developers to review it, and then push the commit; or if no review is needed, the code will be committed right away.

In both conditions, the fix will be committed and CI-bot changes the status of the report to Committed. This happens in devel and the fix goes to be tested in 'devel' level of the release process. CI-bot also makes an automatic comment to the bug which includes info of the software update version, who did the change and when, and it also adds the commit message to the comment.

After going through automated tests devel and getting accepted by the release manager, the fix goes to be tested in testing level. CI-bot changes the Bugzilla report status to Resolved/Fixed. It also makes an automatic comment to the report which includes information of the software update version, who did the change and when. There is a set of automatic tests that run during the testing cycle. The results from these test indicate whether the software update is stable enough to be a release candidate or not.

²¹Pull Request on Git

A QA²² report will also be created from the results of the test, and the release manager will accept the changes in order to let it go through the release level.

After being tested through automated test cases and approved by the release manager, the fix goes to be tested in release level. CI-bot changes the Bugzilla status to Resolved/Released. It also makes an automatic comment to the report which includes information of software update version, who did the change and when along with the commit message. In this level, the complete system testing is done which is both manual and automated. The testing team will run test cases against the new release candidate. There is also a QA report made out of the result of the tests. In the end the release manager will either accept or reject the changes. If later someone realizes the fix does not work after the release, he/she can reopen the report.

There is also a case where a developer does everything manually. This way there will be no Bugzilla number in the commit and the responsible developer needs to update the bug with different statuses manually.

4.7.9 Earlier tracking of the release process before this project

Currently in OBS changelog, the commits for non-open-source packages are followed by a Bugzilla number which the commit contributes to. Take the below picture as an example of changelog with its Bugzilla numbers for the 'ambianced' package (see Figure 27).



```
10
11 * Tue Jan 27 2015 Sami Kananoja <sami.kananoja@jollamobile.com> - 0.1.5
12 - [ambianced] TOH id added to AmbienceContent and AmbienceModel. Contributes to JB#25952
13
14 * Wed Dec 31 2014 Sami Kananoja <sami.kananoja@jollamobile.com> - 0.1.4
15 - [ambianced] Make sure IOH is really attached before notifying store. Fixes JB#24887
16
17 * Tue Dec 16 2014 Antti Seppälä <antti.seppala@jollamobile.com> - 0.1.3
18 - [ambianced] Set pixel ratio parameter for wallpaper blur. Contributes to JB#25176
19
20 * Mon Nov 24 2014 Sami Kananoja <sami.kananoja@jollamobile.com> - 0.1.2
21 - [ambianced] Fixed writing alpha value of colors to conf key. Contributes to JB#24478
22
23 * Tue Nov 11 2014 Sami Kananoja <sami.kananoja@jollamobile.com> - 0.1.1
24 - [ambianced] Enable color modifications. Contributes to JB#13727
25 - [ambianced] Fix db migration path to support migration from versions /->8->9. Contributes to JB#15127
26
27 * Fri Oct 31 2014 Marko Saukko <marko.saukko@jollamobile.com> - 0.1.0
28 - [systemd] Require booster and dbus socket to be there when running. Contributes to JB#24157
29
```

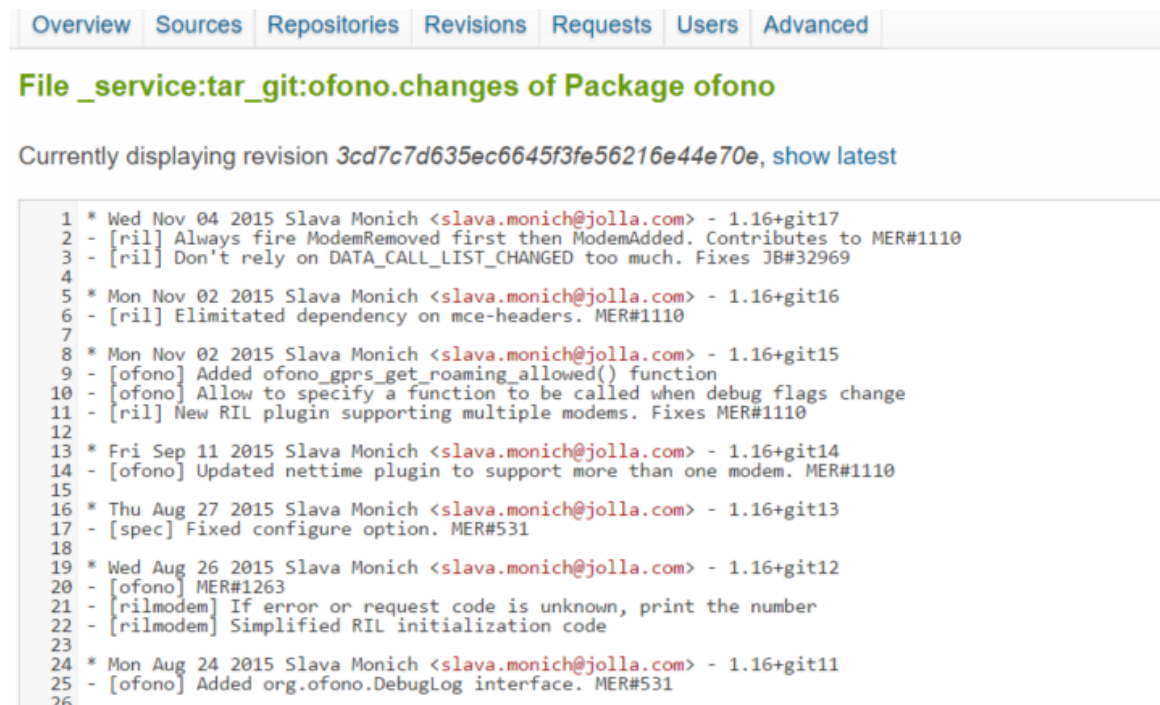
Figure 27: Changelog for a closed-source package called 'ambianced'

However, for changes coming from open-source packages, the commits do not follow any Bugzilla numbers (see Figure 28). Existing open packages include some on Mer side such as mer-core, mer-tools, hybris-hal common stuff essentially Hardware Adaptation Development Kit; some on Sailfish open side, and nemo:mw.

²²Quality Assurance report used to show the results of testing procedures.

4.7.10 Problem with the release tracking procedure before this project

In Mer projects, looking at the changelog only shows the commit messages without any Bugzilla numbers, therefore nothing is traceable from Jolla side. When releases are made, it is important to be able to track the changes from Bugzilla. Internal developers need to provide the release manager at least one Bugzilla number which addresses to the changes they have made for the new update. On Mer side, there are not Bugzilla numbers representing the commits from the open source components, in changelogs, and the release manager on Jolla side cannot track the changes via Bugzilla numbers, but only remembering them which is not sufficient enough. On the other hand, lack of proper infrastructure made it harder for Jolla open-source community to contribute in the development process. As one of the main purposes behind this project Jolla wanted to prepare needed infrastructure, and ask the Mer package maintainers and the Mer community to adopt a policy to require bug numbers in some commits to Mer packages to support vendor tracking of changes in Mer.



```
1 * Wed Nov 04 2015 Slava Monich <slava.monich@jolla.com> - 1.16+git17
2 - [ril] Always fire ModemRemoved first then ModemAdded. Contributes to MER#1110
3 - [ril] Don't rely on DATA_CALL_LIST_CHANGED too much. Fixes JB#32969
4
5 * Mon Nov 02 2015 Slava Monich <slava.monich@jolla.com> - 1.16+git16
6 - [ril] Eliminated dependency on mce-headers. MER#1110
7
8 * Mon Nov 02 2015 Slava Monich <slava.monich@jolla.com> - 1.16+git15
9 - [ofono] Added ofono_gprs_get_roaming_allowed() function
10 - [ofono] Allow to specify a function to be called when debug flags change
11 - [ril] New RIL plugin supporting multiple modems. Fixes MER#1110
12
13 * Fri Sep 11 2015 Slava Monich <slava.monich@jolla.com> - 1.16+git14
14 - [ofono] Updated nettime plugin to support more than one modem. MER#1110
15
16 * Thu Aug 27 2015 Slava Monich <slava.monich@jolla.com> - 1.16+git13
17 - [spec] Fixed configure option. MER#531
18
19 * Wed Aug 26 2015 Slava Monich <slava.monich@jolla.com> - 1.16+git12
20 - [ofono] MER#1263
21 - [rilmodem] If error or request code is unknown, print the number
22 - [rilmodem] Simplified RIL initialization code
23
24 * Mon Aug 24 2015 Slava Monich <slava.monich@jolla.com> - 1.16+git11
25 - [ofono] Added org.ofono.DebugLog interface. MER#531
26
```

Figure 28: Changelog for an OS package called 'ofono' on Mer Side

4.7.11 What did Jolla do to solve the problem and ease the collaboration?

The plan for this project was to be able to track Mer Bugzilla reports on Jolla Bugzilla so that when a report is filed on Mer, there will be an automatic Jolla report created for it on Jolla Bugzilla. Before having the Bugzilla numbers for commits from the Mer community, changes on the Mer side was tracked manually. The reason to have this Bugzilla number is to be able to track Mer report on Jolla Bugzilla and automate as much of the process as possible. Our goals here are:

- To work on open components, in the open, using the external Bugzilla (i.e. Mer Bugzilla)
- To have an internal interface for tracking the changes in both open and closed components, because:
 - There are dependencies between the open and closed components
 - There are dependencies to tasks in other areas like business development, marketing, etc., that need to be tracked internally
 - Collecting and updating information manually in several places is difficult and time consuming
- Minimize the amount of manual work required in development, testing and releasing to make this tracking possible

To achieve this we need to prepare internal and external infrastructure for the upstream Bugzilla sync.

4.7.12 Upstream Bugzilla sync

By going upstream, we want to work openly on some components and have an internal interface for the external (i.e. Mer Bugzilla) reports. Also, we want to work on open Bugzilla reports using the external Bugzilla, and be able to comment on the cloned report in internal (i.e. Jolla) Bugzilla by going to the external Bugzilla. Moreover, we want to make sure no writing of status information will be made on Bugzilla reports on Mer. Each side (Jolla and Mer) set their own Bugzilla status. With upstream Bugzilla sync as shown in Figure 29, there is going to be an automatic Jolla report created

in Jolla Bugzilla (internal) for the report on Mer Bugzilla (external). The report is, however, only readable from Jolla Bugzilla.

In order to make any comments to the report, one needs to go to Mer Bugzilla to have the write access. This is done in order to reduce the hassle of not having reports on Bugzilla with the same content and different comments, as well as making it easier for Jolla developers to be able to track external Bugzilla reports internally. They most probably won't have time to check Mer Bugzilla separately, so it is more convenient to have the report already created on Jolla Bugzilla. They read it on Jolla Bugzilla, and if they want to make any comments, then they have to go to Mer Bugzilla.

Porting the report from Mer Bugzilla to Jolla Bugzilla is done automatically; however, the Bugzilla status is something independent from Mer. Each side, Jolla and Mer, has their own Bugzilla status. The reason for independent statuses is that Jolla uses this status to track integration changes for the time new OS updates want to be delivered. The Bugzilla statuses are in aligned with Jolla's software integration system, and it is for a different purpose than used on Mer.

To have this tracking mechanism also on the open side, developers can file a goal/story/task/bug on Mer Bugzilla explaining the issue they have a fix for, and then provide the bug number (e.g. MER#12345) in the commit message they push to Git. This will benefit both sides since Mer can provide the same kind of automation and tracking; Jolla can cut down the manual effort required for tracking the changes in the open parts; and Jolla can move planning of open components to Mer Bugzilla.

4.7.13 How automated cloning and tracking is done

Basically there are three steps:

1. A report gets created on Mer side (external)
2. Either a developer creates a tracking report on Jolla side, or Jolla CI-bot creates it automatically when it encounters a Mer Bugzilla reference in the package changes (i.e. MER#xxx)
3. Jolla CI-bot reports the progress through the different integration levels to the tracking report in Jolla Bugzilla.

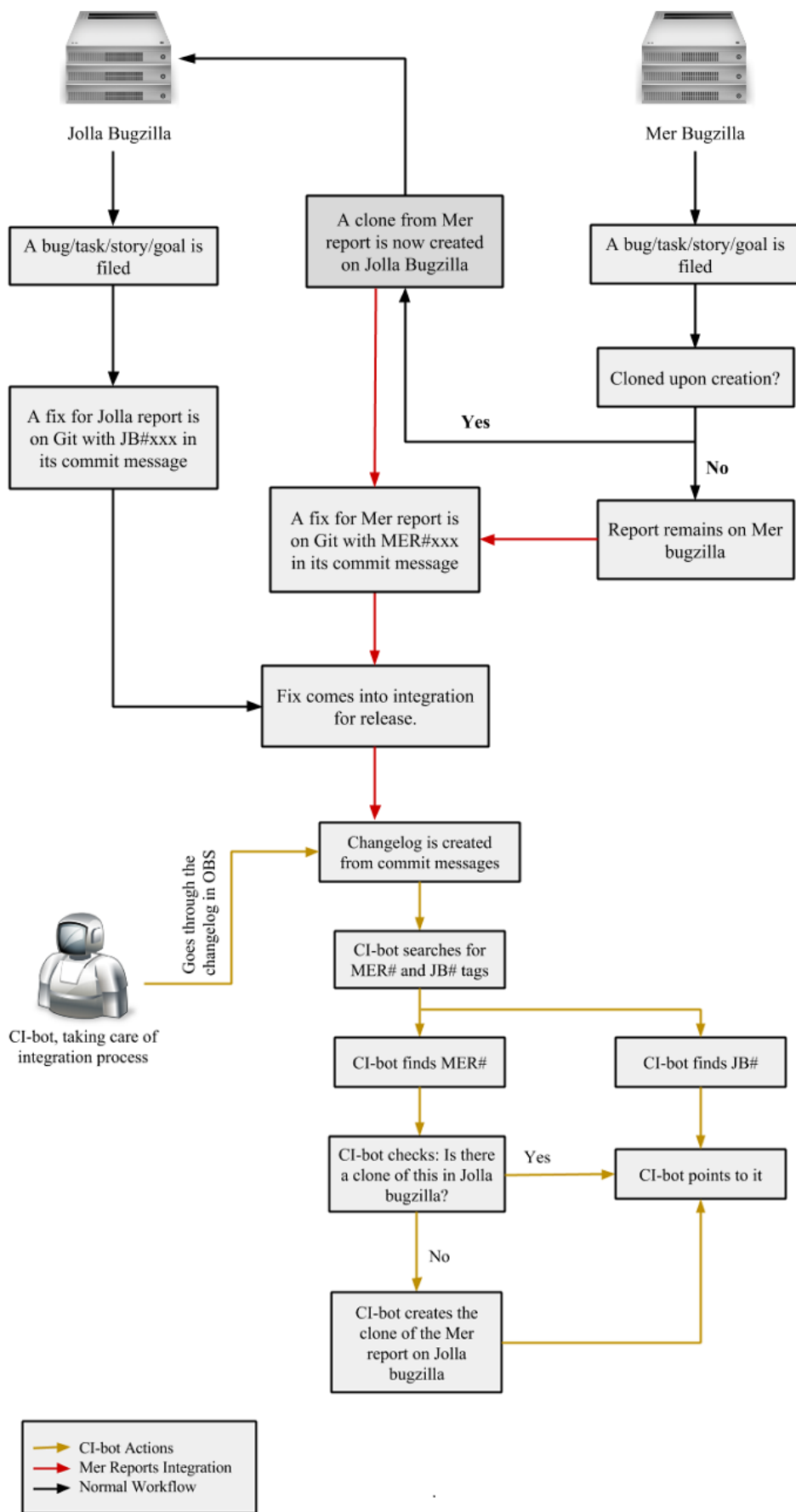


Figure 29: Upstream Bugzilla sync in detail

We need the information from reports in Mer Bugzilla to also be visible in Jolla Bugzilla for tracking and planning during the release process. We need the synchronization to be automated to some degrees. The linking works in a way that there is a report on Mer linked to a Jolla report and this will make the Jolla report be the tracking report. Creating a tracking report on Jolla Bugzilla can be done while creating the report on Mer Bugzilla, or any time later. In Mer Bugzilla, upon creating a report there is the option to clone from, which can be used to link the Mer report to be synced in Jolla Bugzilla.

There are a few terms on how the tracking bug on Jolla Bugzilla, and the open bug on Mer Bugzilla need to behave:

- There can be multiple tracking reports on Jolla Bugzilla for one report on Mer Bugzilla, but a tracking report can only track one report on Mer Bugzilla.
- Tracking reports need to make an easy access to the report on Mer Bugzilla. In Jolla, we make visibility of the Mer Bugzilla by adding the link of the Mer report to the See Also section of the cloned Jolla Bugzilla report. Whenever the See Also is removed, there needs to be a notification saying that the report on Mer Bugzilla is no longer being tracked.
- All the tracking reports are like other reports on Jolla Bugzilla, which means they can be goals/stories/tasks/bugs, and they can be used during iteration planning, and software integrations.
- When there is a new comment in the report on Mer Bugzilla, the tracking report will also have the comment added as a reflection of the Mer report followed by an email notification in Jolla Bugzilla.
- When there is a status change in the report on Mer Bugzilla , there will be an email notification triggered for the tracking report about this status change. On Jolla side, developers have the freedom to change the bug status accordingly if needed. Bugzilla statuses in Mer and Jolla Bugzilla are independent from each other. Also when the status of a tracking report changes, the user who changes the status on the closed Bugzilla (i.e. Jolla Bugzilla) will be prompted whether a comment should be sent to the related report on Mer Bugzilla or not.

4.7.14 How CI-Bot works during upstream sync

In this level, CI-bot works with OBS changelog. It goes through the OSS (open source) projects. From the packages, it scans sources and the .changes file, searching through the commits. For every commit, CI-bot checks if there is already a cloned report for it from Mer Bugzilla to Jolla Bugzilla. If the tracking report is already created, CI-bot makes a link between the two reports. If not, CI-bot will make a read-only report for it on Jolla Bugzilla and then points to it.

4.8 The proposal presented to Jolla's open-source community

With upstream Bugzilla sync being implemented and in place, it was time to represent the changes to Jolla's open source community, and ask Mer package maintainers to try to adopt to the new policy. I represented the main project, our goal and the result of the project to the community members through developer mailing list²³ on 27 January 2015 . In the representation, I first explained the all-time goal of the company regarding collaborating with its open source developer community, and the difficulties that had prevented us to reach this goal for so long. I continue by explaining how the release process for the Sailfish operating system was done at Jolla. I then describe limitations on community collaboration in the open-source code, and what we had done to make this collaboration happen. I also explain how they can do the contribution to the software development at Jolla, and I finish by asking them to adopt a policy to support vendor tracking of changes in Mer. In the following lines of this chapter, you will read the proposal exactly as it was represented to Jolla's open-source community.

Section 4.8.1 shows the content of the proposal exactly as it was sent to the OS community members.

4.8.1 The request to Mer package maintainers

This is a request to Mer package maintainers and the Mer community to please adopt a policy of requiring bug# in some commits to Mer packages for supporting vendor tracking of changes in Mer

²³The mailing list is accessible from: <https://lists.sailfishos.org/pipermail/devel/2015-January/005571.html>

Jolla started with the emphasis of having community -the heart of any open source project- involved in the long run ahead of it. We caught the ribbon with the motto of 'doing it together', and tried our best to stick to it from day one. And today we are proposing and requesting a change which will help us all to be one step closer to achieving more togetherness, by opening up our development to anyone interested in getting a closer view of the code, and going further in contributing. We would like to ask our community to take part and collaborate in Jolla's release project.

How Jolla tracks work required for a release

As you may know Jolla obtains its source code both from open-source²⁴ and closed-source codes. The closed-source part is maintained by our internal developers, and the open part by both internal and external developers. For each software release, this source code is taken from both places and shapes the release image. As expected, with each release candidate there is a changelog containing the info about what has been changed from the previous release to the current one. In this changelog there are both internal and external changes, with just one difference: For the internal changes, there is at least one bug number which shows what the git commit is contributing to²⁵.

Up until now it's been that for changes in the closed-source packages (=the internal part), developers need to include at least one bug number in their commit messages which addresses to that change. However, for the packages on the open side there has never been such a tracking procedure. Therefore, tracking changes in the open has always been difficult, whether for the release manager, testers, or anyone else who has been interested in following the changes.

Having this bug number in the changelog for the closed-source part has had several advantages:

1. From the release manager's point of view, it can make tracking the changes easier and more efficient; so that during each release cycle, there is some kind of a documentation for him about what has been changed and in what level.
2. From the testers' point of view, it is beneficial because they can test those bugs

²⁴Existing open packages include some on Mer side such as mer-core, mer-tools, hybris-hal common stuff, essentially HADK (<https://github.com/mer-hybris>); some on Sailfish open bits (<https://github.com/sailfishos>), and nemo:mw

²⁵Some of the bug references currently showing up in the changelog are internal bugs of public things which need to move to the open.

and make sure the commits have really fixed the issues.

3. And last but not least, everyone else who is interested can follow the changes from one release to another, getting a clear insight on the releases content and changelogs²⁶.

However, for the open-source parts, the commits in the changelog are not followed by any bug numbers. This means for the release manager, testers, or anyone else interested to track the changes on the open, there are not any specific documentation (i.e. bug numbers) addressing them. Therefore, in order to track the changes in the open, one needs to do it manually, which is both time consuming and not beneficial. On the other hand, not offering a proper system to our community has also made it harder for you to contribute to Jolla's software development process .

Jolla is always seeking the support and contribution from its community. We've already had 10 software releases, and had your words of support through all the ups and downs since day one. Having our community's continuous trust gave us the courage to be able to row in this stormy sea, and deliver a part of what we'd promised. Although Jolla couldn't make all the infrastructures ready for better contribution from the beginning, we continued to back each other up; it has never been Jolla's sailors alone, but Jolla's sailors with its community. These all moved us forward, and now that we've put our feet on the ground, we are ready to deliver a new part of our words of support since day one' to you. We have implemented a suitable system to ease the ways for our dearest community's contribution.

As we talked about changelog, Jolla's internal developers provide at least one bug number (the bug is created on Jolla Bugzilla) in their commit messages. This bug number helps our bot to link what issue each commit contributes to. Then the bot will update the bug automatically in each level of the release cycle; from the development level, to testing and release levels. These automatic updates show detailed information about 'who changed what' .

The reasons Jolla uses a bot to do the tracking are:

- Lots of developer's time can be saved to do other important stuff instead of checking a bug constantly and updating its status manually.

²⁶These changelogs are available on the Jolla phone via running `rpm -q --changelog package_name` in terminal.

- The CI-bot message makes it very clear for everyone to see what has changed, when it changed, who changed it and lots of other info you can read in Mer wiki.

So, as the intro mentioned, as one of Mer's vendors, Jolla would like to ask the Mer package maintainers and the Mer community to please adopt a policy of requiring bug numbers in some commits to Mer packages for supporting vendor tracking of changes in Mer.

To have this tracking mechanism also on the open side, developers can file a bug on Mer Bugzilla explaining the issue they have a fix for, and then provide the bug number (e.g. MER#12345) in the commit message they push to Git. This will benefit both sides, as:

1. Mer can provide same kind of automation and tracking.
2. Jolla can cut down the manual effort required for tracking the changes in the open parts .
3. Jolla can move planning of open components to Mer Bugzilla

To keep a community successful, unified and going, participants need to have shared goals, and respect the cultural values. In Jolla, as one of our main values, we respect communication, advancement, challenge and invention. We want to move forward by having the transparency in what we do, and showing our passion in what you do. We would like to work with you as openly as possible. With your ideas, passion and creativity combined, we believe we can take another big step in our journey. The infrastructure is alive and kicking; you contribute, and we take care of the rest. Please read our FAQ (https://wiki.merproject.org/wiki/FAQ#Vendor_.27Jolla.27_Frequently_Asked_Questions) for more detailed info, and don't hesitate to contact us in case of any questions. Best regards,

Nazanin Mirarab (on behalf of the Jolla team)

An online meeting regarding this proposal was handled a week after representing it to the community members. The purpose of the meeting was to answer any questions or uncertainties regarding these changes.

4.9 A example to show the results of the project in practice

To show all the previous sections explained in this document in practice, and make it more clear how the upstream Bugzilla sync could improve the software release process in the company, I will illustrate examples of software planning before the upstream changes, and software planning after the changes were implemented. After upstream changes have been implemented, Open Source software development changed in two places:

1. Open Source components on the open side i.e. Mer Bugzilla
2. Reports of Open Source components which were created on Jolla Bugzilla. These reports were filed on Jolla Bugzilla since that was the only way for an in-house developer to be able to track them during the release process before upstream changes take effect.

We show an example of reports in category 2, and how they have changed after the new infrastructure has been implemented.

In the before phase I will explain:

1. A report for an open-source/closed-source component is created in Jolla Bugzilla, to be tracked during iteration planning
2. A fix for that report in Bugzilla is in a Git repository
3. How the Git commit is connected to a report in Jolla Bugzilla
4. How the Git commit is connected to Jolla Webhook
5. How Jolla Webhook is connected to OBS
6. What happens after OBS build an image with the fix from Git in it
7. Exploring the software release changelog and comparing commits with and without bug numbers
8. How lack of information in changelog can cause difficulties in the release process

In the after phase I will explain:

1. A fix for an open source component is in a Git repository
2. How the Git commit is connected to a report in Mer Bugzilla
3. how Mer Bugzilla is connected to Jolla Bugzilla
4. How the Git commit is connected to Jolla Webhook
5. What happens after OBS build an image with the fix from Git in it
6. Exploring the software release changelog and pointing out bug numbers related to both internal and external commits
7. Explaining how this can help software planning in Jolla as well as building a better relationship with Jolla's open-source community

4.9.1 Iteration Planning 12/2014 before the Upstream Bugzilla Sync

During the last iteration planning period for 2014, changes regarding the upstream Bugzilla sync were not fully implemented, thus, not in use at that moment. This means that open source components could not be tracked by the release team, testing team, or Jolla's open-source community. Any changes regarding the component in the open that appeared in the changelogs could not be tested since the commit messages alone were not enough to give sufficient information for testing, and there was no Bugzilla report mentioned in the commit message via which one could obtain more information.

Therefore, during the 12th iteration planning in 2014 - and all the ones before that - only internal issues could be tracked in our internal tracking system (i.e. Jolla Bugzilla). Open source issues were known to some people in the company who were working closely with them, but not to the rest.

The disadvantages of iteration planning where only closed components are being tracked:

- More risk for product management in predicting matters related to product development.
- More risk for software project management in mistakenly exclude important open source fixes, which were coming in the next software release, in the software planning documents.

- Less visibility of the open source component in the changelogs of software releases, making it nearly impossible for the testing team to test those new features or bug fixes, before they land on the end users' devices.
- More limitations for Jolla's open-source community to be able to contribute in the making of software releases. They had access to changelogs and could see changes in open source components; however, if they wanted to know more about those changes and contribute in improving the code, they had no information regarding what the components were doing.

4.9.2 A Bugzilla report in the final release changelogs before the upstream Bugzilla sync

A report (see Figure 30) is created in Jolla Bugzilla to be tracked during iteration planning 12/2014.

Bug 24761 - [BUG] Device fails to list wifi network when there's only one in the range [\(edit\)](#) [Save Changes](#) Suppress mail

Status: **RESOLVED COMMITTED** [\(edit\)](#)

Product: External

Component: Backlog

Version: unspecified

Hardware: Any Sailfish

Importance: Normal normal

Assigned To: [\(edit\)](#) [\(take\)](#)

URL:

Whiteboard: update10_blocker

Keywords:

Tags:

Depends on:

Blocks:

Show dependency [tree](#) / [graph](#)

Reported: 2014-12-01 12:26 EET by

Modified: 2014-12-29 02:04 EET [\(History\)](#)

CC List: Add me to CC list
8 users [\(edit\)](#)

See Also: [\(add\)](#)

Type: bug

Test Case: Exists

Type of Test: Unit Test
Robot Test
Feature Test
Manual Test

Area: Connectivity Connectivity UI Design

Target: ---

Pool: N.O.W sprint 2014W48-51

[Bug list](#)

Pool ID / Order: 422 / 25

[Activity graph](#)

[To Master Plan](#)

TreeView+ [depends on](#) / [blocked](#)

Flags: None yet set [\(set flags\)](#)

[Hide advanced fields](#)

Figure 30: Report created on Jolla Bugzilla

A fix coded by Jolla's internal developers for that Bug will be in its Git repository lipstick-jolla-home-qt5 as a pull request (see Figure 31). After review, the pull request, names upgrade-1.1.6 is accepted and the commit will be pushed to Jolla Webhook (see Figure 32). From Webhook, the commit is ready to be triggered for software build. The 'trigger build' button is available after you click on the Git link . After software build is done, the commit lands in Jolla's OBS and can be fetched for release image building whenever it is called by the release manager (see Figure 33). From the .changes file in OBS, you can see when the bug fix has landed in OBS (Figure 34).



Figure 31: Pull request on Git for lipstick-jolla-home-qt5 component

After the image containing this fix is built, from the release changelog you can find the commit related to the fix (see Figure 35). JB#24761 is visible in front of the commit message and is clickable hyperlink that will take you to the main Bugzilla report shown at the beginning of this section in Figure 31.

From the same changelog, if you click on connman component, which contains the connectivity-related packages of the software, and open its commit messages, you can see that it only has a commit message without any Bugzilla reference number (see Figure 36).

The commit message alone cannot help in testing what has been fixed in connman. It cannot help release manager to track back the issue, Jolla's internal testing team cannot test this fix due to lack of information regarding the fix, and the external open-source community cannot retrieve any information about where this fix has been used.

4.9.3 Planning 05/2015 after the upstream Bugzilla sync

During the 5th iteration planning period for 2015, changes regarding the upstream Bugzilla sync were implemented, an announcement was made to the open-source com-

<input type="checkbox"/>	/ui-lipstick-jolla-home.git	u7-sneak-peek-coloring	pj:non-oss
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	unlock-arrows-no-jump	pj:non-oss
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	update10	pj:non-oss
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	update10-jb24761	pj:non-oss
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	update11	pj:non-oss:1.1.2
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	update4-sfa	pj:non-oss
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	update7	pj:non-oss
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	update8	pj:non-oss
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	update9	pj:non-oss
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	upgrade-1.1.4	pj:non-oss:1.1.4
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	upgrade-1.1.5	pj:non-oss:1.1.5
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	upgrade-1.1.6	pj:non-oss:1.1.6
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	upgrade-1.1.7	pj:non-oss:1.1.7
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	upgrade-1.1.9	pj:non-oss:1.1.9
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	upgrade-1.2.0	pj:non-oss:1.2.0
<input type="checkbox"/>	/ui-lipstick-jolla-home.git	upgrade-2.0.1	pj:non-oss:2.0.1

Figure 32: Webhook shows a hook for the Git commit named upgrade-1.1.6

Open Build Service > Projects > pj:non-oss > Packages > lipstick-jolla-home-qt5 > Sources

Overview Sources Repositories Revisions Requests Users Advanced

Source Services

Services are applied in the displayed order. You can use drag&drop to re-order them.

Create a tar ball from a git repository
Show Parameters

Source Files

Filename	Size	Changed	Actions
_service:tar_git:lipstick.service	344 Bytes	24 months ago	
_service:tar_git:lipstick-jolla-home-qt5.spec	5.8 KB	1 hour ago	
_service:tar_git:lipstick-jolla-home-qt5.changes	695.2 KB	1 hour ago	
_service:tar_git:lipstick-jolla-home-qt5-0.32.59.tar.bz2	789.8 KB	1 hour ago	
_service:tar_git:lipstick-bluetooth.service	325 Bytes	4 months ago	
_service	364 Bytes	1 hour ago	

Figure 33: Lipstick-jolla-home-qt5 is on Jolla's OBS

```
Wed Dec 03 2014 Pasi Siöholm [redacted] 0.24.10
[lipstick-jolla-home] Remove the "no networks round"-status. JB#24761
[lipstick-jolla-home] Revert flaec41 and a93ef3c. JB#24761
```

Figure 34: The commit in Git has landed in OBS on 3 December 2014

```

BowSprit 0.20141208.0.1
connman
  Updated : 1.24+git48-1.4.1 -- 1.24+git48.1-1.5.1
droid-system-sbj
  Updated : 0.4.0-10.21.1.jolla -- 0.4.0.1-10.22.1.jolla
flac
  Updated : 1.2.1-1.1.1 -- 1.2.1-1.2.1
jolla-email
  Updated : 0.1.65-10.46.2.jolla -- 0.1.67-10.47.1.jolla
kernel-adaptation-sbj
  Updated : 3.4.98.20141031.1-10.28.3.jolla -- 3.4.98.20141031.3-10.29.1.jolla
libcommhistory-qt5
  Updated : 1.7.18-1.20.1 -- 1.7.18.1-1.21.1
lipstick-jolla-home-qt5
  Updated : 0.23.20.2-10.56.1.jolla -- 0.23.20.4-10.57.2.jolla
  * Sat Dec 06 2014 Denis Zalevskiy [REDACTED] - 0.23.20.4
  - [events] convert NotificationList to list and use it as the root widget of the EventsView
  - [events] set NotificationList as view root widget. Fixes JB#22237
  - [lockscreen] remove column to fix items positioning

  * Wed Dec 03 2014 Pasi Sjöholm [REDACTED] - 0.23.20.3
  - [lipstick-jolla-home] Remove the "no networks found"-status. JB#24761
  - [lipstick-jolla-home] Revert f1aec41 and a93ef3c. JB#24761

```

Figure 35: Lipstick-jolla-home-qt5 fix in the release changelog

munity, and the service was in use. This means that open source components could be tracked by the release team, testing team, and Jolla's open-source community .

The upstream changes, however, started to gradually make open source components accessible. Since there are quite many open source components, it takes time to have Mer Bugzilla reports for all of them at once.

For those changes that a Mer Bugzilla report were created, they appeared in the changelogs as MER#number, and could be tested since the commit messages included a hyper-link report that could provide information for testing, and further collaboration from Jolla's open-source community.

The upstream Bugzilla changes started to be active from February 2015; however, the

```

BowSprit 0.20141208.0.1
connman
  Updated : 1.24+git48-1.4.1 -- 1.24+git48.1-1.5.1
  * Tue Dec 02 2014 Simo Piironen [REDACTED] - 1.24+git48.1
  - [connman] Start/stop ntp activity when TimeUpdates property is changed

```

Figure 36: Commit messages related to 'connman' component in the release changelog

example I provide is for May 2015 since by this time many improvements have been done to make the syncing process more efficient. The advantages of iteration planning where both open and closed source components can be tracked is the availability for product management to predict matters related to product development, as well as availability for software project management to include important open source fixes, which are coming in the next software release, in the software planning documents. There is also more visibility for the open source component in the changelogs of software releases, making it more efficient for the testing team to test those new features or bug fixes, before they land on the end users' devices. Moreover, it will bring less limitations for Jolla's open-source community to be able to contribute in the making of software releases. They now have access to changelogs with more visibility on open changes; and as the main goal of this project communication between Jolla's internal developers and its OS community will be improved.

4.9.4 Upstream Bugzilla sync and its affect on tracking the open source components

A report is created in Mer Bugzilla as shown in Figure 37. Reports on Mer Bugzilla are created regardless of Jolla's release cycle. Jolla will fetch and use the ones required for its releases whenever needed. Since the report shown in Figure 37 is created on the open, it is accessible via this link: https://bugs.merproject.org/show_bug.cgi?id=930

A fix for that Bug will be in its Git repository mer-packages/connmann as a pull request (see Figure 38). This Git repository is connected to Mer Webhook as well as Jolla Webhook. Therefore, after the pull request in Figure 39 is approved, the commit will be hooked to Jolla Webhook automatically. After review, the pull request is accepted and the commit will be pushed to Jolla Webhook (see Figure 40). From Webhook, the commit is ready to be triggered for software build. The 'trigger build' button is available after clicking on the Git link. After software build is done, the commit lands in OBS and can be fetched for release image building whenever it is called by the release manager (see Figure 41).

From the .changes file in OBS, you can see when the bug fix has landed in OBS (see Figure 42). Since connman is an open source packages, all the information about the contributor to this package can be shown in the screenshot.

Bug 930 - wlan deaths can leave connman in "associating"-state with some particular chipsets

Status: NEW

Reported: 2015-04-25 22:33 UTC by Pasi Sjöholm

Allas: None

Modified: 2015-04-28 11:54 UTC ([History](#))

CC List: 0 users

Product: Mer Core

Component: connman ([show other bugs](#))

Version: unspecified

Hardware: Other Mer

Importance: Undecided normal

Assignee: Pasi Sjöholm

URL:

Keywords:

Depends on:

Blocks:

See Also:

Figure 37: A report created on Mer Bugzilla

https://github.com/mer-packages/connman/pull/218

This repository Search Pull requests Issues Gist

mer-packages / connman Watch

[connman] wifi: disconnect if 4way-handshake fails while roaming. MER#930 #218

Merged tigeli merged 2 commits into mer-packages:master from tigeli:carrier on 28 Apr

Conversation 1 Commits 2 Files changed 1

tigeli commented on 27 Apr Owner

While wifi->state is G_SUPPLICANT_STATE_COMPLETED and gets changed into G_SUPPLICANT_STATE_4WAY_HANDSHAKE the connection is most probably roaming.

Figure 38: A commit for a Fix for Connman in Mer packages repository on Git

Select web hook mapping to change

https://github.com/mer-packages/connman Search 10 results (14821 total)

Action: Go 0 of 10 selected

<input type="checkbox"/>	Repo url	Branch	Project	Package
<input type="checkbox"/>	https://github.com/mer-packages/connman.git	master	pj:connman:release	connman
<input type="checkbox"/>	https://github.com/mer-packages/connman.git	mer	mer:core	
<input type="checkbox"/>	https://github.com/mer-packages/connman.git	update11	nemo:mw:1.1.2	connman
<input type="checkbox"/>	https://github.com/mer-packages/connman.git	upgrade-1.1.4	nemo:mw:1.1.4	connman
<input type="checkbox"/>	https://github.com/mer-packages/connman.git	upgrade-1.1.5	nemo:mw:1.1.5	connman
<input type="checkbox"/>	https://github.com/mer-packages/connman.git	upgrade-1.1.6	nemo:mw:1.1.6	connman
<input type="checkbox"/>	https://github.com/mer-packages/connman.git	upgrade-1.1.7	nemo:mw:1.1.7	connman
<input type="checkbox"/>	https://github.com/mer-packages/connman.git	upgrade-1.1.9	nemo:mw:1.1.9	connman
<input type="checkbox"/>	https://github.com/mer-packages/connman.git	upgrade-1.2.0	nemo:mw:1.2.0	connman
<input type="checkbox"/>	https://github.com/mer-packages/connman.git	upgrade-2.0.1	nemo:mw:2.0.1	connman

Figure 39: A commit for connman lands in Jolla webhook

Overview Sources Repositories Revisions Requests Users Advanced

Source Services

Services are applied in the displayed order. You can use drag&drop to re-order them.

Create a tar ball from a git repository
Show Parameters

Source Files

Filename	Size	Changed	Actions
_service:tar_git:main.conf	94 Bytes	26 months ago	
_service:tar_git:connman.tracing	11 Bytes	26 months ago	
_service:tar_git:connman.spec	4.8 KB	5 months ago	
_service:tar_git:connman.changes	23.4 KB	5 months ago	
_service:tar_git:connman-1.24+git90.1.tar.bz2	1.1 MB	5 months ago	
_service	360 Bytes	5 months ago	

Figure 40: Connman is on Jolla's OBS

```

1 * Wed May 13 2015 Pasi Sjöholm <pasi.sjoholm@jollamobile.com> - 1.24+git90.1
2 - [connman] wifi: disconnect if 4way-handshake fails while roaming. MER#930
3 - [connman] wifi: disconnect if wpa s state changes from completed to scanning. MER#930
4
5 * Tue Apr 21 2015 Slava Monich <monich@users.noreply.github.com> - 1.24+git90
6 - [connman] Fixed memory leak in supplicant.c
7 - [connman] MER#891
8
9 * Wed Apr 08 2015 Slava Monich <monich@users.noreply.github.com> - 1.24+git89
10 - [connman] agent: corrected usage of g list delete link
11
12 * Thu Apr 02 2015 Slava Monich <monich@users.noreply.github.com> - 1.24+git88
13 - [connman] dnssproxy: request data in request list need the request data
14 - [connman] Fixed crash in wispr portal web result
15

```

Figure 41: The commit in Git has landed in OBS on 13 May 2015

After the image containing the connman commit is built, from the release changelog you can find the commit related to the fix (see Figure 42). MER#930 is shown after the commit message and is a hyperlink which can take you to the main Mer report shown in Figure 37.

The open component connman, which contains the connectivity-related packages of the software, now has two MER# in front of both of its commit messages. In the previous example, there were no bug numbers representing the commit messages for connman; however, upstream Bugzilla sync made it possible.

Figure 44 also shows the detailed view of fixes in the closed source component lipstick-jolla-home-qt5. Before changes regarding the upstream Bugzilla sync, only closed components such as lipstick had hyperlinks to the Bugzilla reports representing them.

Packages modified (19) (toggle all)

BowSprit 0.20150520.0.1

PackageKit

Updated : 0.8.9+54-1.21.1 -- 0.8.9+54.1-1.23.1

as-daemon

Updated : 0.8.12-10.31.5.jolla -- 0.8.12.1-10.32.1.jolla

connman

Updated : 1.24+git90-1.16.1 -- 1.24+git90.1-1.18.1

* Wed May 13 2015 Pasi Sjöholm <pasi.sjoholm@jollamobile.com> - 1.24+git90.1

- [connman] wifi: disconnect if 4way-handshake fails while roaming. MER#930

- [connman] wifi: disconnect if wpa_s state changes from completed to scanning. MER#930

csd

Updated : 0.2.48-10.8.1.jolla -- 0.2.48.1-10.10.1.jolla

hybris-libsensorfw-qt5

Updated : 0.8.12-10.17.1.jolla -- 0.8.12.1-10.18.1.jolla

jolla-settings-accounts

Updated : 0.2.56.1-10.71.1.jolla -- 0.2.56.3-10.72.1.jolla

jolla-settings-sailfishos

Updated : 0.0.29-10.15.1.jolla -- 0.0.30-10.17.1.jolla

jolla-signon-ui

Updated : 0.0.27-10.15.1.jolla -- 0.0.27.1-10.17.1.jolla

libjollasignonuiservice-qt5

Updated : 0.1.3.1-10.18.2.jolla -- 0.1.3.2-10.19.1.jolla

lipstick-jolla-home-qt5

Updated : 0.24.60.3-10.79.2.jolla -- 0.24.60.6-10.81.2.jolla

* Mon May 18 2015 Matt Vogt <matthew.vogt@jollamobile.com> - 0.24.60.6

- [lipstick-jolla-home] Critical urgency level for connectivity notifications. Contributes to JB#28071

* Fri May 15 2015 Matt Vogt <matthew.vogt@jollamobile.com> - 0.24.60.5

- [lipstick-jolla-home] Do not emit change signals during model move operation. Contributes to JB#28331

* Tue May 12 2015 Matt Vogt <matthew.vogt@jollamobile.com> - 0.24.60.4

- [lipstick-jolla-home] Ensure correct notification sorting. Contributes to JB#28331

Figure 42: Connman fix in the release changelog

After opening the cloned report, the report will be shown as Figure 44 on Jolla Bugzilla, with the See Also section containing the original report from Mer Bugzilla. On the SeeAlso section, there are two options to choose from: Track and Remove.

By selecting Track, CI-bot keeps an eye on the original report on Mer Bugzilla, as well as showing on the Mer report that it is being tracked by a report on Jolla Bugzilla. Any changes in the original Mer report will be notified to the area owners mentioned in the Jolla Bugzilla report (i.e. Connectivity, Connectivity UI, Mer Project, and WLAN).

By selecting remove, the Mer report will be removed from the cloned report on Jolla Bugzilla. Any changes regarding the Mer report will not trigger any notifications on the Jolla report.

At this stage, CI-bot goes through the release changelog and searches for any MER#number. CI-bot will then take this Mer report and searches through Jolla Bugzilla to see if it can find a clone of the Mer bug there or not.

- If CI-bot finds a clone for the Mer bug, it points to the cloned report.

Status	Resolution	Summary
NEW	---	[AUTOCLONE][BUG] wlan deauths can leave connman in "associating"-state with some particular chipsets

Figure 43: An example of a cloned report from Mer Bugzilla on Jolla Bugzilla

Figure 44: Clone of a Mer report shown on Jolla Bugzilla

- If CI-bot cannot find a clone of the Mer bug in Jolla Bugzilla, it will create one.

A clone of a report on Mer Bugzilla looks like Figure 43 on Jolla Bugzilla

Figure 45 shows how CI-bot uses the cloned Mer report on Jolla Bugzilla to update it according to Jolla's integration changes. For instance, in the second comment made by CI-bot, it is mentioned that the commit is in Sailfish OS, upgrade 1.1.7.1, and in the testing level. Such information can help anyone exploring the report to find out where the commit is recently being used in. The third comment by CI-bot on the report shows the commit is now in upgrade 1.1.7.2 and in the development level.

CI Bot 2015-04-27 00:40:50 EEST [Description](#) Private [\[reply\]](#) [-]

From: https://bugs.merproject.org/show_bug.cgi?id=930

CI Bot 2015-04-27 00:41:07 EEST [Comment 1](#) Private [\[reply\]](#) [-]

Referenced in or above SailfishOS 1.1.7.1 (Björnräsket) (TARGET_CPU,testing)
 connman :
 * Sun Apr 26 2015 Fasi Sjöholm <pasi.sjoholm@siirappi.com> - 1.24+git95
 - [connman] wifi: disconnect if connections seems to be stuck. MER#930

CI Bot 2015-04-27 01:45:14 EEST [Comment 2](#) Private [\[reply\]](#) [-]

Referenced in or above SailfishOS 1.1.7.2 (Björnräsket) (TARGET_CPU,devel)
 connman :
 * Sun Apr 26 2015 Fasi Sjöholm <pasi.sjoholm@siirappi.com> - 1.24+git95
 - [connman] wifi: disconnect if connections seems to be stuck. MER#930

CI Bot 2015-04-28 14:30:53 EEST [Comment 3](#) Private [\[reply\]](#) [-]

Referenced in or above SailfishOS 1.1.7.1 (Björnräsket) (TARGET_CPU,testing)
 connman :
 * Tue Apr 28 2015 Fasi Sjöholm <pasi.sjoholm@siirappi.com> - 1.24+git96
 - [connman] wifi: disconnect if 4way-handshake fails while roaming. MER#930

Remote Tracker 2015-04-28 14:54:42 EEST [Comment 4](#) Private [\[reply\]](#) [-]

Remote tracking item https://bugs.merproject.org/show_bug.cgi?id=930 changed

 Comment by pasi.sjoholm@siirappi.com
https://bugs.merproject.org/show_bug.cgi?id=930#c2
 proper fix <https://github.com/mer-packages/connman/pull/218>

Figure 45: How CI-bot updates the cloned report with Jolla's integration cycle changes

5 Analysis of the project

Referring to one of the essays on Open Source by Eric Raymond ‘The cathedral and the bazaar’ [16], we can explain, in other words, how Jolla developed its Sailfish operating system. Raymond uses cathedral as a metaphor for corporations, where everything needs to be planned beforehand, and without setting the exact requirements following the hierarchy, one cannot proceed to the process of making the product. The bazaar, however, is used as a metaphor for companies where everyone can decide when, how and to what extent to contribute to a project. Jolla, as a software company, is a little bit of both: A bazaar inside a cathedral.

Let us go deeper into the definitions of a bazaar and a cathedral at Jolla. Although Jolla is a start-up company, many operations in it still follow the traditional way of decision making and software planning. One of the reasons for that can be that the founders of the company, as well as some of the team managers come from Nokia – a big corporation with a lot of hierarchy. Therefore, decisions tend to be made more with a corporate way of thinking, rather than a start-up one. On the other hand, they are trying to adopt to agile decision making modes. This transition of course, takes time, and the response rate the company can have towards some of the planning will be done by delay. This is exactly what happened with the Upstream Bugzilla Sync project.

Another factor influencing the operations in agile environments might be due to lack of resources in the company. Implementing this project required skilled developers to be able to develop the whole framework and test it before delivery. They also needed to automate as many manual actions in the framework as possible. The reason for automating the actions was to make the job of filing a report and committing a fix for it less time-consuming for the developers, so that they did not have to spend most of their time on manually updating statuses or comments on Bugzilla, Webhook, and OBS. The developers in charge of preparing the platform for this project were also busy with their other day-to-day tasks in the company that they needed to take care of.

One other aspect I observed as a participant in this project, during the meetings with the chiefs and responsible people for the project, was the lack of understanding the goal. For some people in the project, the main goal and reasons for its delivery were clear, while some thought that there was no necessity to do the project when the company did not have the needed resources.

Further in this chapter, we will read about earlier experiments and studies that were done in similar environments.

5.1 Earlier Works

Based on a study that was conducted to analyze the factors that influence satisfaction with free and open source application software project, several factors were investigated [3]. The research question on this study was to examine what factors influence the satisfaction of participants in OS application software projects. Their conceptual model showed that perceived system complexity has been the main factor in staying motivated in the community. On the other hand, perceived product openness was the least important value among the participants in the study. Comparing this study to the one done at Jolla, we can notice that not having product openness was a reason to decrease the satisfaction in Jolla's OS community.

In another study done on the Eclipse project – a widely recognized open source project which provides platforms for developing integrated tools - the development of Eclipse has been one of the successful OSS development processes, with a developers' community who are committed to deliver quality software [6]. Eclipse achieved this success by always releasing beta versions that would get into the feedback loop of end users, until the feedback was taken in and applied to the software. Comparing this study to the one done at Jolla, releasing more frequently in beta versions to get quick user feedback is one of the main approaches that Jolla has been doing with a higher quality after the Upstream Bugzilla Sync got in place. Implementing this project helped Jolla to give its OS community the ability to contribute to the software development process, while the in-house developers could spend more time on maturing the Closed Source components for a release. This also resulted in faster release cycles since changes from the Open Source side were easily traceable, and could be added to the software release build.

6 Conclusions

The experience of working on an Open Source project in a commercial company has resulted in number of lessons learned, which we will go through in this chapter. There are also several ongoing problems that show opportunities for future research.

6.1 Lessons learned

One of the most important findings – and probably an essential topic for future studies – that was discovered during this project was how badly OSS projects need coordination. They need leaders who are determined on what the community should achieve, and can help the participants reach a common understanding of the goals. Having a common understanding during software planning is another factor that can significantly speed up the process. Not having that understanding on the other hand, can cause friction among the team members and deviate them from their main goals.

Working in an OS project, means getting involved with people, their time, their goals, and their motivation. Unfortunately, commercial companies who work with OS communities tend to forget that the community participants are not their employees. Companies should promise something only if they know they can deliver it. Company leaders should involve community members in the software planning through online meetings, e-mails, etc. Members of a community need to know that they are part of a project they are supporting.

In terms of agile practices in start-up companies, team leaders as well as the rest of the employees need to remember although requirement analysis is not necessary before starting a project, it influences the outcome of that project. If exact requirements are not set at the beginning of a project, it does not mean that they never come into the planning process. Agile gives more flexibility; being careful not to get into a chaos in project planning from this flexibility is the duty of the leaders.

6.2 Open questions and future works

As more commercial companies are getting OS communities involved in their software development processes, it is important to investigate on the psychological and behavioral aspects of the relationship between the companies and their communities. Studies can find out what factors strengthen or weaken this communication.

Another question concerning working with OS communities is how much transparency affect the motivation of the participants in practice. We need to look more into the correlation between communicating clear information and the motivation to continue contributing in a community.

In terms of agile studies, we can observe how agile methodologies are being implemented in companies, specifically start-ups. The experience we have achieved from the case study in this thesis project shows that operating in an agile way usually leads to not taking the requirements into account. Since documentation and proper requirement analysis in smaller and newly established companies are not done as much as in corporations, future researches can conduct how lack of proper analysis and documentation can lead into failure of a project during agile practices.

7 Glossary

Automated Testing: Act of testing a software/hardware using special software (separate from the software being tested) to execute test cases and compare the results of the executions with the predicted outcomes that were already given to the software.

Backend Developers: Developers who create the logical back-ends and core computational logic of a software.

Branch (in Git): A branch in Git is a movable pointer to one of the commits. The default branch is called the master branch.

Bugzilla : A system used for task tracking as well as bug reporting

Build Dependency (in RPM): Defines what the RPM depends on in case one wants to build that RPM. To be able to build an RPM, all of its dependencies should be met/built beforehand.

Build Number: The build number in the version of a software shows the number of times that software has been built. It is the 4th number in version number (e.g. 1.6.0.12). 12 is the build number.

Closed Source Software: A type of software for which the codes (binaries) are only accessible to certain group of people assigned to develop the software.

Commit (in Git): A commit in Git is used to record changes in connection with the local

Repository (in Git): A place where the source code of a software is stored.

Commit Message: A text used to describe what a certain commit does.

Community (in Software Development): A group of developers and backers who maintain and support the software they are using.

Community Contribution: Open source software have communities who use the software and at the same time collaborate with its main developers in making the software better and bug-free. Community contributors are people who are not employed by the software company, but who are supporting the company by supporting their software and maintaining it for free.

Component (in Software Development): A reusable program that can be combined with other components in a distributed network to form a software or an application.

Continuous Integration: In development, continuous integration is the act of integrating code into a shared repository several times a day, and each time it is verified by automated tests to detect problems.

Distributed Version Control System: A version control that record changes to a software, is distributed among different developers, and is being maintained by them.

Embedded System : A computer system with a specific function within a larger system. It is part of the larger system and is needed for the bigger system to function properly.

End User : Users who will get and use the final product that came out of a software development cycle.

External Developers : Used in open source projects, external developers are the ones who are not directly responsible to maintain code. (Aka. open source community developers).

Feature Integration Deadline : In software development process, feature integration deadline is the deadline used to let developers know up to when they can push changes to the code before the code is ready for a software release.

Integration : In software release process, act of collecting, merging and integrating the code into the software release.

Interface : A point where a software can communicate with its end users (e.g. graphical user interface).

Internal developers : Developers who are directly responsible to maintain code and fix bugs related to the code.

Linux : A Unix-based operating system assembled using free and open source software development.

Manual testing: Act of testing a software manually (using people) without using any other specific software.

Merge (in Git) : Act of integrating a commit into the current branch.

Middleware : Software that acts as a bridge between the operating system and applications.

Nemo Middleware : Is a vendor of Mer, using Mer core for its architecture and has its own UI

Open Build System (in Software Release) : A system to build and distribute packages from sources in an automatic way.

Open Source Software : A type of software for which the codes (binaries) are accessible to everyone.

Open-Source Community : A group of backers of a software helping to maintain the software.

Package (in Linux): A compressed file archive containing all of the files that come with a particular application.

Package Binary : The code from which an application is made.

Package Maintainer : Developers who are responsible for maintaining the code in packages of an application.

Product Management : A product is anything that can be offered to the market, and it has a life cycle. The life cycle of a product consists of different projects that need to be managed for the delivery of the product.

Project Management : A project is a temporary endeavor to create a service, result or a product.

Proposal: A plan or suggestion, written formally, that is put forward for consideration by others.

Pull Request (in Git) : A pull request tells others about the changes one has made to a repository.

Push A Commit : When a commit is pushed to a repository, the changes that were done on the local repository will be pushed to the remote repository.

QA Report : A report used in quality assurance to show the results that were conducted out of testing a software.

QML (Qt Modeling Language) : Is a user interface markup language.

Quality Assurance : Process of assuring the quality of a product.

Release Candidate : Using during software release process, a release candidate, is a possible candidate of a specific release which might be suitable to be used by the customers. A release candidate is tested by the testing team, and in case no issues are found, it can be released to the main customers. If the testing team find bugs or regressions, they should be fixed and a new release candidate will be created.

Release Number : The number in the version number of a software which shows the release after bug fixes. (e.g. in red hat package manager).

RPM : A package manager used with application on Linux operating system. It can be used to build, install, update, or remove individual software packages.

Scripts (in Programming) : A list of commands that can be executed without user interaction. Scripts are widely used in automation.

Snapshot (in release) : Snapshot build are for users to download and review while the platform is still being developed. It indicated a view of the source code taken at a specific time.

Software Upgrade : A replacement of software with a newer version, and a better quality.

Vendor : A company offering something for sale. In this document, that something refers to software.

Version number : Numbers assigned to give a software a unique characteristic. As the version number increase, the quality of the software increase as well.

References

- [1] Arka Bhattacharya. Impact of continuous integration on software quality and productivity. Master's thesis, The Ohio State University, 2014.
- [2] Kevin Daniel Andre Carillo, Sid Huff, and Brenda Chawner. It's not only about writing code: An investigation of the notion of citizenship behaviors in the context of free/libre/open source software communities. *2014 47th Hawaii International Conference on System Sciences*, 0:3276–3285, 2014.
- [3] Brenda Chawner. Community matters most: Factors that affect participant satisfaction with free/libre and open source software projects. In *Proceedings of the 2012 iConference*, iConference '12, pages 231–239, New York, NY, USA, 2012. ACM.
- [4] Nigel Edwards and Liqun Chen. An historical examination of open source releases and their vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 183–194, New York, NY, USA, 2012. ACM.
- [5] Adanna Ezeala, Hyunju Kim, and Loretta A. Moore. Open source software development: Expectations and experience from a small development project. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, pages 243–246, New York, NY, USA, 2008. ACM.
- [6] Erich Gamma. Agile, open source, distributed, and on-time: Inside the eclipse development process. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 4–4, New York, NY, USA, 2005. ACM.
- [7] Hironori Hayashi, Akinori Ihara, Akito Monden, and Ken-ichi Matsumoto. Why is collaboration needed in oss projects? a case study of eclipse project. In *Proceedings of the 2013 International Workshop on Social Software Engineering*, SSE 2013, pages 17–20, New York, NY, USA, 2013. ACM.
- [8] Jim Highsmith. *Agile Software Development Ecosystems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [9] Chin-Yun Hsieh, Yu Chin Cheng, and Chien-Tsun Chen. Emerging patterns of continuous integration for cross-platform software development. In *Proceedings*

- of the 2Nd Asian Conference on Pattern Languages of Programs, AsianPLoP '11, pages 9:1–9:9, New York, NY, USA, 2011. ACM.
- [10] Jordan Hubbard. Open source to the core. *Queue*, 2(3):24–31, May 2004.
- [11] Kumi Jinzenji, Takashi Hoshino, Laurie Williams, and Kenji Takahashi. *An experience report for software quality evaluation in highly iterative development methodology using traditional metrics*, pages 310–319. 2013.
- [12] Charles E. Matthews. Agile practices in embedded systems. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPEs'11, NEAT'11, & VMIL'11, SPLASH '11 Workshops*, pages 249–250, New York, NY, USA, 2011. ACM.
- [13] Nora McDonald and Sean Goggins. Performance and participation in open source software on github. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems, CHI EA '13*, pages 139–144, New York, NY, USA, 2013. ACM.
- [14] Carsten Munk and David Greaves.
- [15] Guido Porruvecchio, Giulio Conrcas, and Michele Marchesi. *Open Source Communities Structure and Communication Patterns: A Social Network Approach*. PhD thesis, 3 2010.
- [16] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.
- [17] Nayan B. Ruparelia. Software development lifecycle models. *SIGSOFT Softw. Eng. Notes*, 35(3).
- [18] Munish Saini and Kuljit Kaur. Understanding the community culture in open-source software projects. *SIGSOFT Softw. Eng. Notes*, 39(5):1–4, September 2014.
- [19] Walt Scacchi. *Process Models in Software Engineering*. John Wiley and Sons, Inc., 2002.
- [20] Renuka Sindhgatta, Nanjangud C. Narendra, and Bikram Sengupta. Software evolution in agile development: A case study. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems*

Languages and Applications Companion, OOPSLA '10, pages 105–114, New York, NY, USA, 2010. ACM.

- [21] Thomas Tan, Qi Li, Barry Boehm, Ye Yang, Mei He, and Ramin Moazeni. Productivity trends in incremental and iterative software development. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.
- [22] Selena Uras, Giulio Conrcas, and Alessandro Giua. *The Communicational Side of Open Source Communities*. PhD thesis, 2 2008.
- [23] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 805–816, New York, NY, USA, 2015. ACM.