Norwegian University of
Science and Technology

The Fragility of Open Source

A Case Study

Stian Haga

Master of Science in Informatics

Submission date:	June 2011
Supervisor:	Eric Monteiro, IDI
Co-supervisor:	Thomas Østerlie, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Assignment Text

Open Source development is getting increased attention, as a product of the fact that much of the most interesting technology today is developed (more or less) through open source. A central feature of open source is the striking absence of common methods and tools to support the development of the technology.

How are open source projects structured and organized with the lack of these tools? What (if anything) can common, commercially-based development learn from open source based methods?

The assignment will be based on a case study of selected open source project(s), typically through the study of mailing lists, electronic archives and IRC.

The study is based on an interpretive research method in contrast to eg. questionnaire-based surveys.

Abstract

Open source software with its distinctive communities and unique history, filled with ideology and politics, has for the past decade been of high interest to many academic fields. It is no wonder, when the communities consisting of volunteers cooperating and thereby creating these massive success histories, such as Apache, Linux and Mozilla.

In this thesis open source software is viewed as fragile entities, as opposed to the heroic perspective that is dominating current research in the field of open source. Methods for creating a sustainable community are investigated through a case study of two open source software projects. Data is collected from the publicly available e-mail archives, bug trackers and observation through internet relay chat.

Through analyzing the data gathered I have identified some fragile aspects of open source software, as well as a few means of reducing fragility. I conclude that open source developers can benefit from acknowledging these points of fragility and any company looking into open source for should carefully assess the software with this fragility in consideration.

Keywords: Open Source, Open Source Licensing, Software Maintenance

Preface

This thesis concludes my master's degree in informatics at the Norwegian University of Science and Technology (NTNU). The assignment was given by Eric Monteiro, Professor at the Department of Computer and Information Science (IDI), and I spent the last two semesters working on the results presented in this report.

The past year of research has been filled with many challenges and the process of writing a master thesis is a very rewarding experience, both academically and on a personal level. I had very little knowledge about open source software before starting my thesis, but the phenomena is a very intriguing one and it kept feeding my curiosity. Through the research that I have done in the past year, I feel my insights into the open source development process have reached a much deeper level.

First of all I would like to thank my family for their continuous support throughout my studies, and also extra points to my dad who stayed up late at night just in order to finish proofreading my thesis. I would also like to thank my girlfriend Heidi for her support and motivation, and also for supplying me with an excellent illustration (figure 2.3.1). Last, but not least, I would like to thank my supervisor, Professor Eric Monteiro, for his much appreciated guidance and invaluable feedback during the past year.

Trondheim, June 1, 2011

Stian Haga

Contents

1 Introduction **xiii**
1.1 Problem Definition xiv
1.2 Project outline xvi

I Litterature Review **1**

2 Open Source Development **3**
2.1 History And Evolution of OSS 4
2.1.1 Open Source 4
2.1.2 Open Source Software 2.0 6
2.2 Open Source Licensing 10
2.2.1 Emergence of Copyleft 10
2.2.2 Permissive and Academic Licenses 12
2.2.3 Innovation In Open Source And Within Companies 14
2.3 Mechanics of Open Source Development 19
2.3.1 Motivation For Joining And Staying In Projects 19
2.3.2 Forking 22
2.3.3 CSCW And Group Awareness 26
2.3.4 Project Stakeholders and Requirement Engineering 29

3 Traditional Software Development **33**
3.1 Agile Methodology 35
3.1.1 Scrum 37
3.2 Development Vs. Maintenance 40

II Case **43**

4 Research Methodology **45**
4.1 Getting Access 46

4.1.1 Anonymity 48
4.1.2 Collecting Data 49
4.1.3 Documents 50
4.1.4 Interview 51
4.2 Analyzing The Data 53

5 Case **55**
5.1 VideoLAN Media Player 57
5.1.1 Organization And Model Of Development 58
5.1.2 History Of The VLC Project 60
5.2 jQuery 63
5.2.1 Overview 63
5.2.2 Organization And Model Of Development 66
5.2.3 History Of The jQuery Project 67

III Analysis **73**

6 Discussion **75**
6.1 Licenses As Insentives 77
6.1.1 GPL vs. MIT - A Holy War 77
6.1.2 The Impact of Licenses 82
6.2 Sustainability Through Alliances 85
6.2.1 Forging Alliances 85
6.2.2 A Downward Spiral 87
6.3 Balancing Innovation And Maintenance 91

7 Conclusion **97**
7.1 Further Research 99
References 100

A **107**

B **108**

C **109**

D **112**

E **129**

F **131**

List of Figures

2.1.1 This graph shows the market share for the most popular browsers, over the past 12 months. As seen, the Mozilla Firefox has a substantial market share at about 30%. Only browser more popular is the Internet Explorer developed by Microsoft. It is bundled with all installations of the Microsoft Windows operating system. However, it's popularity is rapidly declining as the open source competitors are gaining momentum, as depicted by recent the growth of Google Chrome's market share. 8

2.3.1 Sharing of gifts. Exchanging source code and ideas. ©Heidi Suul Næss 20

2.3.2 Forks of the GNU/Linux project. Full page image can be seen in Appendix C 23

2.3.3 24

2.3.4 mIRC is the most widespread and popular IRC client for Windows, with millions of users. 28

2.3.5 This figure is an overview over the different kind of communication tools and what situation they serve. 29

3.0.1 The Waterfall Model as described by Royce (1987) 34

3.1.1 One iteration within a Scrum project. 38

3.1.2 Illustration of a burndown chart. 39

4.1.1 Activity on the jQuery mailing list. 49

4.1.2 Distribution of contributions in the VLC project. 52

5.1.1 VLC commit history, based on author. The top ten committers are shown individually. More than 390 different authors have contributed to the VLC project in total. The authors are based in 20 different countries world wide. 60

5.2.1 One of the most popular jQuery Plugins is called Lightbox. It has the ability to display images as you click their thumbnails in a gallery. Lightbox will create an overlay over the website and display the picture, and even resize the image to fit the browser window. 70
5.2.2 An interactive date picker. One of many widgets available in the jQuery UI Package. 71
5.2.3 Some of the improvements in the new version compared to the old release. 72

List of Tables

2.2.1 List of popular OSS projects and their license. 13

5.1.1 Various statistics for the VLC project. 57
5.2.1 Various statistics for the jQuery project. 63

Chapter 1

Introduction

Open source software (OSS) is flourishing on the internet today, with close to 300,000 projects hosted on SourceForge (`www.SourceForge.net`) alone. These community driven projects are now competing with corporate giants, such as Microsoft and Apple. Most of the projects are traditionally spawned from the need of a person or a small group, developed in a private enviroment then released to the public, setting the stage for a distributed collaboration of volunteers. During the last decade, OSS have opened up to the general public and the stakeholders are not only the internal developers, but also the current and potential users of the software.

The history of OSS entails powerful political and ideological ideas. Not only have open source communities been pictured as rebels, moving against the corporate wave of billion dollar industries, but they are also known for creating internal disagreements, known as ”holy wars”, where differences in political, ideological or purely technological views have gone so far as to split communities. This is easily recognizable in the various licenses available for OSS.

Even though open source has had its differences with the commercial software industry, they are now often seen working hand in hand. Companies intrigued by open source development, are tapping into the open source communities looking for ideas and contributing to the work. There are now few modern day companies who does not draw any benefit from open source software, either through using it as middleware within the company, frameworks and ideas for future commercial solutions or even through the brand ”Open Source”, which has become more or less a buzzword.

There is no doubt that open source software has had a huge impact on the software industry, with successful competitive examples such as Mozilla Firefox now domi-

nating the web browser market. In recent years, governments in several countries have started to see the benefits of open source. The ability to heighten security and save millions of dollars on buying proprietary licenses has lead to governments in countries, e.g. the Netherlands and the United Kingdom, encouraging or forcing public institutions to prioritize the use of open source software whenever possible.

With the ever increasing complexity of software development, it seems too good to be true that a community based on volunteers with seemingly lack of formalized project management, few means of face-to-face communication and a scarcity of resources in general, can be highly competitive, innovative and create an impact on the software market.

The open source research field is filled with interesting and astonishing facts based on these successful projects. However, with the lack of management and formalization, there is an insentive to shift the perspective over to the fragility of open source projects, as suggested by Monteiro et al. (2004), instead of focusing purely on the successful projects. There ought to be a few weaknesses in the open source development, and these fragile attributes might prove to be limitations as to what the applications of open source can be; for both developers and third parties such as commercial companies. By investigating through a case study I hope to identify some of these fragile aspects of open source.

1.1 Problem Definition

Open source software with its distinctive communities and unique history, filled with ideology and politics, has for the past decade been of high interest to many academic fields. It is no wonder, when the communities consisting of volunteers cooperating and thereby creating these massive success histories, such as Apache, Linux and Mozilla.

The development model and its features have been studied by fields ranging from understanding their motivation from an economical perspective (Lerner and Tirole, 2002), looking at open source licenses (Lerner and Tirole, 2005, Kaminski and Perry, 2007), organizational theory (Gutwin et al., 2004, Ciborra, 1996, Hippel and Krogh, 2003) and even psychology (Lakhani and Wolf, 2003). The common denominator for all these fields is collecting information and knowledge from successful open source projects.

There seem to be a picture of open source software in the academic field that they are a melting pot created by recipes of success, constantly enabling innovation by having altruistic individuals sharing their knowledge and time freely. This view

of altruistism has been proven to be wrong by several studies on the motivation behind open source software (Bonaccorsi and Rossi, 2003, Ghosh, 1998, Lakhani and Wolf, 2003, Roberts et al., 2006). The motivations behind joining projects and contributing is not dominated by one single need, but more often than not a set of complex motivations.

There are less to be found on the areas of how fragile these projects can be. The basis of open source development rests on the pillars of cooperation between volunteers working on a complex piece of software. Crashing ideologies, restrictive licenses, lack of formal communication and the complexity of software engineering sounds more like a recipe for disaster than success. Software maintenance is described as a black hole in traditional software engineering (Bennett and Rajlich, 2000), yet in open source development is characterized by continuous maintenance as a result from what is known as Linus' Law: *"Given enough eyeballs, all bugs are shallow."*(Raymond, 2001, p.30)

Based on the preceeding issues and research, the following research definition has been devised:

> *What limits and challenges are posed by the fragile development method used in open source software?*

The problem definition is divided into these three research questions:

> RQ1: What measures are taken in order to create a sustainable growth of contributors?
> RQ2: What insentives and assessments are in place to select the appropriate open source license?
> RQ3: How is the balance between innovation and maintenance handled in open source development?

These questions will be investigated through an interpretive case study. As a lot of the previous material has been focused around doing surveys(Lerner and Tirole, 2005, Hars and Ou, 2002, Lakhani and Wolf, 2003, Ghosh et al., 2002) and following the famous and successful projects (Mockus et al., 2000), this thesis will focus on some projects that might not be as prominent and commonly known. The perspective will be similar to the view of Monteiro et al. (2004), emphasizing the fragility of an open source project. By looking at the ways of communication in these cases, through observation and data mining, I hope to answer some of the proposed research questions and hopefully add to the body of knowledge within the development open source software.

1.2 Project outline

To serve as a guiding line for my readers, I will provide a short summary of the chapters in this thesis below. It is divided into three parts: theory through litterature review, the case study and lastly the analysis. In addition to these parts, there is an appendix at the end which includes some of the licenses discussed in the thesis, for reference. The different chapters provided in this thesis are arranged as follows:

Chapter 2 explores the many topics of open source development; the history, licensing, motivations in open source, forks, group awareness and requirement engineering.

Chapter 3 gives an insight in how development is done traditionally in software engineering; emphasizing the agile methodology, such as Scrum, and the divide between development and maintenance of software.

Chapter 4 will explain the research methodology used in the project from the beginning to the end.

Chapter 5 provides in-depth information about the two cases studied, namely jQuery and the VideoLAN Client (VLC), in during the time of research.

Chapter 6 discussed the cases investigated in the previous chapter in the light of theory discussed in the littature review, and tries to analyse the different fragilities in Open source.

Chapter 7 concludes the research and some topics for further studies are suggested.

Part I

Litterature Review

Chapter 2

Open Source Development

Open source software is a term that is often misunderstood and/or misused. Throughout the history various terms has emerged, and eventually lead to confusing discussions and articles. Open source is not just a technical term, but also one that entails philosophical and political meaning. As an attempt to prevent any confusion that these various terms might cause, they will be listed and explained in this section.

Free Software was coined by Richard Stallman during the 1980s as he created the Free Software Foundation (FSF). This term is ambiguous and as most people think of free as a matter of cost and not freedom/liberty, it lead to many misunderstandings. Trying to clear this Richard Stallman is famously quoted for saying *"When we call software "free", we mean that it respects the users' essential freedoms: the freedom to run it, to study and change it, and to redistribute copies with or without changes. This is a matter of freedom, not price, so think of "free speech," not "free beer.""* (Stallman, 2010)

Libre Software was later suggested to try avoiding the confusion that free software caused, but never really got a solid footing as free software did, despite its effort to relieve confusion.

Open Source Software as a term, was created in 1998 by the Open Source Initiative (OSI). With the abundance of available licenses for open source software, the Open Source Definition was created by OSI to serve as a guideline to what was considered open source licenses. The difference between the term *Open Source* and *Free Software* was mainly ideological. Open source was originally created as an umbrella term for the two previous ones, defined by the unique distributed nature of the development model, whereas the Free Software definition was defined by the freedom or liberty that the development model offered. The community

Free/Libre Open Source Software (FLOSS) was then suggested in 2001. This term was developed in order to relieve any political or philosophical commitment that follows the use of ”Open Source”, ”Libre Software” or ”Free Software”.

In this thesis the term open source software will be used as a general term for the whole FLOSS definition and will not be used to denote any political or philosophical commitment, unless stated otherwise. While the term FLOSS is designed to abolish this sentiment, it is however not a very fortunate abbreviation and might hinder the ease of reading. The term hacker will also refer to a programmers ability to devise clever ways solve an issue, and not the criminal sense that is used in todays media.

2.1 History And Evolution of OSS

The history of open source software is filled with political and ideological differences and is essential in order to understand why the open source community is where it is today and how it got there. From the conception of the term Free Software in the 1980s, spawned by the ”hacker culture”, to the commercialization and innovation that it is characterized by today. In the following sections the emergence of Open Source software will be viewed in a historical setting, portraying iconic figures within the movement such as Richard Stallman and Eric S. Raymond. The recent evolution from ”Open Source” software to ”Open Source Software 2.0” over the past decade will be discussed.

2.1.1 Open Source

The origin of open source software dates from way back in the 1960s and 1970s. The sharing of source code was at that time taken for granted. In the late 1970s, a computer network named Usenet served as a communication base for the Unix programming community. With the advent of Usenet, researchers and organizations could swiftly share and obtain source code and the network quickly grew to a large size** due to this. It is still widely used as of todays date. Usenet was a great way to distribute knowledge in an informal way. In the early 1980s some of the organizations started claiming rights to source code related to Unix.

Richard Stallman started the Free Software Foundation (FSF) in 1983 as a response to this recent development. The FSF introduced a General Public License which aimed to make source code freely

available by not forcing restrictions on others. Any modifcations to the source code also had to be licensed under the General Public License. The FSF did not necessarily believe that software should be distributed at little or no charge, but the source code should be willingly shared to the users of the system.

It was later accompanied by the Open Source Definition developed by the organization Debian in 1995. The definition said that *"License Must Not Restrict Other Software"* (Initiative, 1999). With this the developers were enabled to use proprietary software along with their software and the flexibility of open source development was greatly enhanced (Lerner and Tirole, 2002).

Open source projects are usually started based on the need for a custom piece of software, by one person or a group. This person or group starts a project by loosely defining how this software will work and creates a foundation for this to be built upon. To develop open source software, Raymond suggested adapting one of two models. These two models are called the Bazaar and Cathedral. By adapting a Cathedral model, the source that is under development will only be open to a select few that actively works on the software. The source code is available for each general release of the software. By adapting the Bazaar model however, the source code will always be available through the internet to anyone who wishes to use it, while it is under development.

The Bazaar model enables the software to be continuously reviewed, tested and debugged, by increasing the number of testers. This is one of the important characteristics and strengths of an open source development method. It heightens the rate of bug catching, which in turn makes the program more stable. A principle resembling a "the more the merrier"-ideology. This idea was formalized in the form of a law by Eric Raymond in his book "The Cathedral and the Bazaar", called Linus' Law. The law was named after the creator of Linux, Linus Torvalds:

> *"Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone."* (Raymond, 2001, p.30)

One example of a Bazaar model development is the operating system Linux, developed by Linus Torvalds in 1991. After creating a fully functional base for an operating system based on Minix, he posted the source code on the internet and encouraged people to work on it. This spawned a great swarm of developers seeking a customizable operating system that fulfilled their needs as time went by.

As of today Linux holds a very strong position on the market, leading the server market but also has a good share of the desktop computers, competing with corporate giants as Microsoft and Apple. The sheer number of open source projects spawned from the release of Linux can be seen in Appendix C.** With the nature of the distributed OSD model, the software architecture needs to be composed of modules to enable development to be done in parallel. In this way the program can also easily be rewritten and enhanced by other users (Raymond, 2001).

The mentality illustrated through Linus' Law is at the core of open source development. Large distributed teams, delivering increasingly complex software by openly sharing their knowledge among their "co-hackers", greatly enabled by the advent of the world wide web.

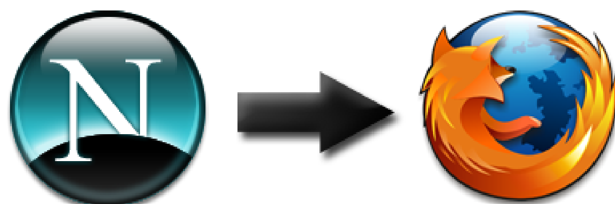
2.1.2 Open Source Software 2.0

Open source software has long been viewed as perhaps a community consisting of very talented hackers sitting in their basement, cooperating through mailing lists only, while creating groundbreaking high-quality software. This is of course somewhat of a myth, and has been questioned in more than one article. Companies deemed open source as inviable for commercial use, and blamed the viral licensing style of open source licenses. In the last decade however, developers of proprietary software has become more involved in open source software, embracing it in both use as tools for development and contributing back to the open source communities. Companies are commonly using open source as a buzzword in order to create an image of being in the front of todays technology. Brian Fitzgerald wrote an article in 2006 called *"The Transformation of Open Source Software"*, in which he uses the term OSS 2.0:

> *"I contend that the open source software phenomenon has metamorphosed into a more mainstream and commercially viable form, which I label as OSS 2.0."*-(Fitzgerald, 2006, p.587)

So what changed the game?

First off, an important mark is the creation of Mozilla Public License (MPL) when the source code for Netscape Communicator - an internet browser suite created by Netscape Communications - was released in 1998. The event that a large software company was about



to open up the source code of a pro-
prietary product to the public, was not common at the time. In fact, it had never been done before. It was not the first time Netscape Communications had done something controversial with their products in order to gain growth. Four years earlier the company decided to distribute early versions of the internet browser Netscape Navigator binary files freely over the world wide web. Its employees were familiar in working with open source software, the initial surprise quickly faded and they focused on what had to be done to make this work. Now, there is an obvious technical difficulty in converting a browser suite with more than 75 third-party modules woven into it (Hamerly and with Susan Walton, 1999), but they also had to find or create a license suiting their project. The license had to fulfill two major points:

- Enable their source code to work with the proprietary modules.
- Keeping the contributions done within the community, making sure the work is not just exploited by individuals or companies making proprietary software, and maximizing the potential growth of the project.

These requirements ruled out the permissive academic licenses such as the popular BSD License and MIT License. The GNU General Public License with strong copyleft was also out of reach, since it would not work with proprietary software. They had to get a license that is somewhere inbetween the permissive academic licenses and the ones with strong copyleft. Realizing there was no current license scheme suited for the listed requirements, Netscape Communications decided to formulate a new license specially tailored to their needs. The results of this was the Mozilla Public License (MPL). MPL was the first open source license viable for the commercial world, enabling the conversion of previously proprietary software to open source. Companies soon started following Netscape's example, and saw the release of IBM's Eclipse IDE in 2000, and Sun Microsystem's Open Office in 2004.

Now that open source software began to be viable to produce as a software company, they began getting involved in various ways but mostly for the same reason: companies are always competing against each other in the free market, so getting leverage on other competing firms is something they must constantly be on the searching for. One of the obvious advantages of using open source software as tools or middleware, is lowering cost. Companies can also lower cost on research and development (R&D), as noted by (Bonaccorsi and Rossi, 2003, p.1), by letting the open source community be their beta testers and also help them resolve bugs. Creating or supporting an open source option to the market can also lower the position of opponents. As an example, IBM decided to support Linux *"[...], be-*

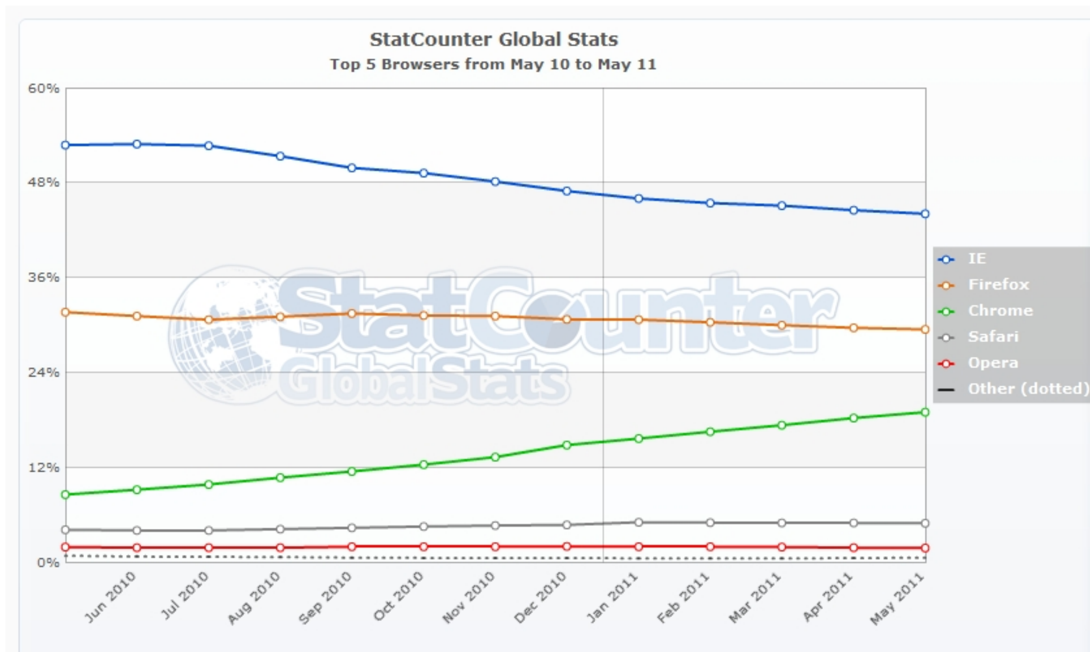


Figure 2.1.1: *This graph shows the market share for the most popular browsers, over the past 12 months. As seen, the Mozilla Firefox has a substantial market share at about 30%. Only browser more popular is the Internet Explorer developed by Microsoft. It is bundled with all installations of the Microsoft Windows operating system. However, it's popularity is rapidly declining as the open source competitors are gaining momentum, as depicted by recent the growth of Google Chrome's market share.*

cause it erodes the profitability of the operating system market and adversely affects competitors like Sun and Microsoft.“. Governments are also starting to promote the use of open source software, and in the Netherlands the government agencies and government-owned companies were restricted to move to open source software wherever it is possible within April 2008.

Another point is the fact that companies can also gain advantage by acquiring knowledge from the open source community. Be it either learning about technical issues and creating a leverage by increasing the company’s expertise, or grasping ideas from the community, then commercializing complementary products based on the current projects in the open source community (Lerner and Tirole, 2005, p.27). This burst in commercial involvement in the open source community, shifted the way in which open source software is developed. (Fitzgerald, 2006, p.589) identified a few common characteristics of OSS 2.0 development cycle in contrast to the ”old” way of doing it:

- Planning-purposive strategies by major players trying to gain competitive advantage
- Analysis and design-more complex in spread to vertical domains where business requirements not universally understood
- Implementation subphases as with OSS, but the overall development process becomes *less* Bazaar-like
- Increasingly, developers being paid to work on open source

OSS projects often had no prominent hierarchy in their organizational structure. There was the creator(s) of the project, and he decided how things were to be done. The creator or group of creators also were the ones doing the planning and design of the project before it got released as open source. In OSS 2.0 projects are often run in a more vertical domain with a clear hierarchy, understanding of business requirements and thorough planning phase, more related to the way commercially driven software development is done. The line between open source software and commercial proprietary software is not as clearly separated as it used to be; it is beginning to blur.

2.2 Open Source Licensing

Licenses in the open source software community have been a widely discussed topic since the first licenses appeared in the 1980's. It is an important part of the history and heritage of open source software. As you will see later, quite a few of the major licenses contain important parts of different ideologies in the community. Open source licenses are known for being very liberal and enabling, in the sense that they invite and accept contributions from whomever wants to modify the work. Some are on the other hand also known for being viral and obtruding as well, if not by design then by consequence, such as GNU General Public License (GPL). Open source licenses are designed to allow open distribution, open modification and preserve the integrity of the author's source code. By allowing open modification and derived works, open source software has gained its perhaps most important defining trait, which is an ideology resembling "the more, the merrier". The major benefit from this, as noted by (Lauren, 2004, p.6), is innovation. By knowing that their work wont be exploited, programmers can safely contribute to the project. Innovation is without a doubt one of the biggest characteristics associated with open source software development.

> *"The more programmers that can contribute to a given work, the more value that work is likely to have."* (Lauren, 2004, p.6)

In the proceeding sections the different types of licenses will be explained, and a few of the most popular ones will be used as concrete examples.

2.2.1 Emergence of Copyleft

Copyright is a term that most people are aware of, but the mechanics behind it can often be complex, especially without any background in law studies. In almost all countries[1] today, the effects of copyright automatically affects any work automatically, as soon as it is created. Be it in the form of written text, an image or as described in our case: source code. This law of copyright prohibits anyone but the creator from creating derivative works from the original, and even the act of displaying or copying the work is not allowed.

On the other hand most open source licenses are what is called copyleft. Due to the ambiguity of the word, it is easily misunderstood. As explained on the website of the Free Software Foundation:

[1] Either through the laws of the United States, the Bern Convention or the World Trade Organization Agreement on Trade-Related Aspects of Intellectual Property Rights.

> *"Copyleft is a way of using of the copyright on the program. It doesn't mean abandoning the copyright; in fact, doing so would make copyleft impossible. The "left" in "copyleft" is not a reference to the verb "to leave"-only to the direction which is the inverse of "right"."*-(Free Software Foundation Inc., 2011, Last accessed 4th of May, 2011)

Copyleft is what spawned one of the first open source licenses, the GNU General Public License (GPL). The GPL is by far the most used license in OSS. It is a highly restrictive and viral license, created by Richard Stallman in the late 1980s. Stallman used the open source project from Gosling Emacs, created by James Gosling, when he wrote his own version called GNU Emacs. Gosling later sold the rights to his code to UniPress. UniPress would not allow Stallman to distribute the source code for GNU Emacs anymore, as it contained work from the Gosling Emacs. Stallman created the GPL as a reaction to this, and to make sure any of his later work would not be proprietarized in the same way (Li-Cheng Tai, 2001). The license makes sure that any derived work is as open and free as the source is. All distribution of the work requires the source to be made available, should it be requested, both if it is given away for free or by charge. This attribute of a license is one of the effects of a Share-Alike license. Any modifications to the work should also carry the GNU General Public License or an equal license. With these properties any work under the GPL prevents being solicited by a firm or individual, seeking to use the work for either their own profit or to claim intellectual property rights. As a part of their article, Lerner and Tirole (2005) gathered data from SourceForge's ˜10,000 active projects. The results showed that, without any weighting of projects, 72% of the total projects use the GPL[2]. By enforcing all derived works to be under the same license, it has gained some notoriety. Anyone seeking to use or modify this code for projects that has other licenses, can not do so without being forced to change the current license of all their work. This makes it difficult to cooperate with any other proprietary work or work that has less restrictions. Many have voiced their concern for this incompatability, one of them is the CEO of Microsoft, Steve Ballmer. In an interview in 2001 he claimed:

> *"Open source is not available to commercial companies."* (Dave Newbart, 2001)

Although his generalisation of open source software might be incorrect, it is an important and fitting point when it comes to the GPL. The second and third edition of the license does allow other software to be distributed alongside the GPL licensed work, though, as long as the linking of the two does not form a single work. This act of copylefting is one of the core aspects of enabling the open source

[2]By weighting the projects by the number of bugs, the total percentage of GNU GPL licenses dropped to about 63%.

development that we see today. It is a drastical move from the normal procedure of copyrighting; instead of securing all the knowledge of the work, you openly share it with the public and make sure that everyone else that wants to use your work also does. This perplexing action is what makes open source development so unique and exciting to study. In the next section we will move on to see licenses that are even more permissive and ”open” than the GPL license.

2.2.2 Permissive and Academic Licenses

To address the issue of not being able to link with proprietary software, a different version of the GPL was released in the early 1990s. Originally it was named GNU Library General Public License, but it was later renamed and is now known as the GNU Lesser General Public License (LGPL). The target software for this license was libraries, that linked with proprietary software. As libraries are considered a derivative work of the proprietary software, publishing it with libraries under GPL would breach the terms of the license. Other than this point, the LGPL is similar to the GPL. It is somewhat of a compromise between the strong-copyleft license that is the GPL and the more liberal licenses such as the BSD license and the MIT license (also known as the X11-license).

The BSD and MIT licenses are permissive licenses, especially the MIT license. This is probably one of the shortest and most permissive open source licenses, which more or less gives the licensee the right to do whatever he or she wants to do with the work, as long as the copyright notice is included. The copyright notice contains the year, name of copyright holder and warranty disclaimer. Separating the two licenses is one clause that has been added to the BSD license:

> *“Neither the name of the <ORGANIZATION> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.”*[3]

This clause is mostly there, as noted by Lauren (2004, p.16), to protect the reputation of the creator. The derivative works can not use the original creator’s name in any product promotion without consent.

Another popular license worth mentioning is The Apache License v2.0. The Apache License v2.0, is similar to the aforementioned BSD and MIT Licenses. It is however more explicit and elaborate in its terms, and even goes the length as to explain them such as ”Work”, ”Derivative Works” and ”Contribution” in its first clause. The licensee is, as with the BSD and MIT Licenses, allowed to

[3] Full license in Appendix D

Type of License	**Example Licenses**	**Examples of Projects With the License**
Permissive Academic	MIT/X11 License, BSD License, Academic Free License, Apache License	jQuery, Ruby on Rails, PuTTY, Lua
Commercially Focused	Mozilla Public License, Eclipse Public License, IBM Public License, Sun Public License	NetBeans, Mozilla Firefox, Adobe Flex, Eclipse IDE
Viral/Reciprocal	GPL, LGPL	WordPress, FileZilla, Pidgin, MiKTeX

Table 2.2.1: *List of popular OSS projects and their license.*

license the derivative work under any other license as they see fit. It is a bit more extensive and gives the creator more legal rights compared to the other permissive licenses.

There are now aids to support the developer in choosing the right license for their work. A paper on Licensing Patterns was released by Kaminski and Perry (2007). In this paper they identified different patterns for open source software, that they added to their previously developed software licensing pattern language (Kaminski and Perry, 2005). These kind of tools are very valuable for complex open source projects as OSS 2.0 is evolving. A license could severely impair or improve your ability to cooperate with commercial forces and gain their knowledge and manpower.

In table 2.2.1 the three main types of open source licenses with examples of projects under these licenses can be viewed. All licenses are approved by the OSI and FSF:

2.2.3 Innovation In Open Source And Within Companies

One of the most popular and common connotations to the term ”Open Source” is the holy grail of many companies and individuals seeking fame and success: innovation. Without the backdrop implying innovation, Open Source probably would not be the buzzword it is today. Companies are often looking to the OSS communities for ideas to develop into business platforms, as discussed in section 2.1.2, as well as new methods to create and promote innovation within the company. Innovation is not just simply an implicit product of any open source software. There are thousands and thousands of projects hosted on SourceForge alone, and only a handful of those are considered innovative and successful pieces of software. In the following section I will try to explain how innovation has been done in open source development - or rather how the open source development has lead to innovation - and how it in contrast has been done in large software firms.

Innovation In Open Source Development

> *“To me, innovation means invention implemented and taken to market. And beyond innovation lies disruptive innovation, which actually changes social practice - the way we live, work and learn.”*(Chesbrough, 2003, p.IX, John Seely Brown)

This is the definition of the word innovation that will be used in this thesis. Some examples of innovative OSS projects are the well studied Apache project, Mozilla Firefox and Alchemy. A few might argue that many OSS projects are just copies of existing products, and therefore are not innovative (Sawyer, 2007). However, while the initial idea and framework of the project might not be innovative, the incremental development of the software has undoubtedly lead to solutions within the project that are hugely innovative. Such as the Mozilla Firefox web browser. While a web browser is not a new and innovative idea in itself, the features it carries repeatedly releases always seem to be one step ahead of the commercial competitors such as Microsoft’s Internet Explorer and Apple’s Safari web browser. Todays essentail features such as tabbed browsing, appeared in open source browsers in 2001 (Opera) and 2003 (Firefox) while Internet Explorer did not offer this until its version 7 released in 2007. Firefox is now the leading browser in Europe, with 37% market share (StatCounter, 2011), eventhough Internet Explore is preinstalled in the most popular operating system Microsoft Windows.

One of the key reasons for OSS being a creative and innovative force on the software market, lies in the fact that there has been a scarcity of software solutions available to the computer user. The demand for specialized software solutions has

been greater than the supply, due to computer science being a rather new field of research. Before the advent of the internet and commercialization of computers, research groups and indivial researchers would create their own software in lack of anything being available through software vendors and other researchers. They would cooperate and share their source code on newsgroups which ran on limited networks between universities. As internet became publicly available in the beginning of the 1990s, the arena for a much larger scale collaboration method was set.

As previously researched by Mockus et al. (2000), there is the fact that a huge part of the contributions on the projects, are done by a central core group of developers. This could lead us to believe that OSS may not benefit by its large distributed developer base, as much as we would like to think. By looking at OSS with this perspective, it could be argued that the innovation model in this regard, is not as different as it is in the commercial software development industry. The main idea of the project, emerging from a single person or small group, might be a deciding factor on the innovative result of the project. On the other hand, the importance of a diverse and massive community - despite its small core of dedicated developers - might be even bigger than expected.

Hippel and Krogh (2003) explore the two innovation models commonly used in organization sciences, the ”private investment” model and the ”collective action” model. The first model being the standard within commercially oriented companies, where innovation is expected to be returned by closely protecting intellectual property by copyright, and the use of private investment. On the other hand, the ”collective action” model entails collective collaboration of innovators to produce something that is of a public good. This is more commonly related to the work done by academic communities. Hippel and Krogh (2003) tries to merge these two models and use it to explain the innovation model used in OSS, namely a ”private-collective” innovation model. This is explained by the creator of the project who privately invests in the software with their own resources, then instead of protecting their intellectual work with copyright, it is distributed openly as a public good. This opens up the possibility for anyone who is interested to join in and share their ideas on how this project is best developed further. The motivations for doing so are explained in section 2.3.1.

Innovation in Commercial Companies

The new era of computer science emerged rapidly and with tremendous steps forward in research, beginning the late 1950s / early 1960s. The researcher Gordon E. Moore famously predicted in 1965 that the number of components on an inte-

grated circuit would double every two years for the next ten years (Moore, 1965), but he said that there was no reason to believe that this growth would continue on any longer. It appears now more than 45 years later that he was astoshingly accurate with this prediction, as it still can be applied to the progression of number of transistors per processor. He also noted that the reduced costs was also a very important part of this technological progression. Accurately so, computers are today available to nearly everyone in society. Some organizations (Child, 2011) are now even trying to release $100 laptops for children in developing countries, to aid education.

This massive and swift progression in computer science, has lead to commercial forces seizing the opportunity to create huge ventures. Some results of these early market opportunities are the very successful corporate giants such as Microsoft, Apple, IBM and Sun Microsystems. Today the software development business has grown huge, and million and billion dollar enterprises are a common sight. These type of businesses are always looking to increase their innovative process, to increase their revenue and position in the largely competitive market. In the recent decade, marking the emergence of OSS 2.0 as discussed in section 2.1.2, companies are starting to draw ideas from the OSS communities in order to try to improve their model of innovation.

The idea of companies using the ”private investment” model, securing patents and thus also perhaps creating a situation of monopoly on the market, has been noted as possibly prohibiting the further increase of the publics knowledge domain by Hippel and Krogh (2003). But why should companies try to share their knowledge and ideas with the rest of their possible competitors? After all, sharing their ”secrets” with other companies can make them loose their position on the market, as other companies will attain and use this knowledge. The move to investigate the model of development within OSS by commercial companies, might not be such a strange thing after all when you look at the facts: OSS communities are known to create software that is hugely competitive and on the edge of technology, despite their seemingly huge disadvantage in organization, funding and product management. There must be something to learn from their way of creating software.

Chesbrough (2003) saw a shift in the way companies were handling innovation, or a ”paradigm shift”. He called the two paradigms Closed and Open Innovation, the old one being Closed Innovation closely related to the Private Investment innovation model and dating back to the beginning of the twentieth century. The paradigm emerged as there was a split between universities, government and commercial industry, forcing the commercial industry to create their own research & development divisions and do the research for themselves. Companies hired the best of the best, increasing their leverage in market. It was incredibly difficult

for small companies to enter the business, as all the recent developments in the field were closed off in private R&D departments. However, in the last couple of decades, this paradigm saw a loss of efficiency, due to various factors noted by Chesbrough (2003). Firstly, there has over the past decades been a huge increase in number of highly educated people within the computer science industry, making it impossible for firms to ”capture” all these individuals and securing their knowledge. Most of the knowledge gained by private R&D departments became publicly available through education. Secondly, companies had a difficult time keeping up with the ever decreasing time-to-market. As many ideas and breakthroughs made by the R&D departments, could not always be followed up and commercialized, employees could use these ideas and start up a new company exploiting this business opportunity.

2.3 Mechanics of Open Source Development

We know, as discussed earlier, that projects are usually started as an effort to ”scratch an itch” of an individual or group, or as an elaborate, and often complex, business idea in the new era of OSS 2.0. However, to develop these projects, they are dependent on contributions from the community in order to progress and release new versions of the software. A way of promoting the software and creating interest around the project is almost a necessity for any new open source software seeking success.

2.3.1 Motivation For Joining And Staying In Projects

Motivation and reasons for joining OSS projects are areas that have been studied and researched upon quite a lot (Bergquist and Ljungberg, 2001, Bonaccorsi and Rossi, 2003, Markus et al., 2000, Ghosh et al., 2002, Rossi, 2004). It is an important part of OSS, as any project will wither without the continuous support from developers, both leaders and workers. In order to make people stay active and contribute, it is essential for anyone leading an OSS project to understand peoples motivation for joining and why they stick around. Ever since the emergence of distributed OSS development there has been a somewhat mystified view of an altruistic group of individuals who join projects, contributing time, energy, knowledge and expertise for what seemed like nothing in return. This is rarely ever the case, though. As research has shown, the reasons behind joining and contributing to OSS can be explained by a range of factors; everything from economical (Lerner and Tirole, 2002) principles to the simple idea that participants think it is fun to program (Ghosh, 1998).

Bergquist and Ljungberg (2001) suggested that the open source communities are similar to academic societies, where knowledge is given away in order to progress your carreer and not necessarily because the person is altruistic. Shared knowledge equals gained status and reputation. The idea of giving gifts and the link to the academic way of working, is also endorsed by Raymond (2001), and he suggests that the social dynamics of OSS is better explained by gift culture and not exchange-economics. In the gift cultures, status and reputation, is gained by giving away items, and in the case of OSS; knowledge in the form of source code and technical expertise. A gift culture only works if everyone participates in the exchange of gifts (Bergquist and Ljungberg, 2001), and so the mutually reinforced motiviations in the gift culture assists in keeping the community glued together (Markus et al., 2000).

Figure 2.3.1: *Sharing of gifts. Exchanging source code and ideas. ©Heidi Suul Næss*

Rossi (2004) explored the idea that motivation can be either *intrinsic* or *extrinsic*. This division of motivational factors was proposed by Aronson et al. (2004). An *intrinsic* motivation is when work is done because doing the work itself has value, or *"inherent satisfactions"* as defined by Deci et al. (1999). Such as the feeling of joy as Linus Torvalds (Ghosh, 1998) and Raymond (2001) describes as a result of hacking. A feeling of accomplishment and being a part of a group (the OSS scene) can also be examples of intrinsic motivations for joining a OSS project. *Extrinsic* motivation on the other hand is when the work is done due to some external reward, such as monetary gain, getting a good position within the project or opening up opportunities for new jobs.

There are quite a few surveys that has been done on motivational factors for joining OSS projects. Hars and Ou (2002) conducted an empirical study on the participants of a number of studies, including 389 people. Their survey showed that extrinsic factors have greater weight than the intrinsic factors, but they both played a major part in motivation. The survey also showed that students and hobby programmers are more focused on the intrinsic motivation, while programmers that are salaried and contracted are interested in the monetary gain. The people that are paid to develop OSS, about 16% of the participants, seem to focus on

fulfilling software needs and self-marketing.

Lakhani and Wolf (2003) later did a survey questioning 684 participants of 287 different OSS projects. A large part of their survey base was experienced professionals working in IT-related jobs. They also recognized the fact that 13% of the participants are being directly paid to develop OSS, something that matches the results of the survey done by Hars and Ou (2002). However, they also found that 55% of the participants worked on OSS projects during their time at work, with or without their supervisors awareness. The results of the survey concluded that a majority of the people in OSS projects think that the community is a highly creative arena, 61% of the participants claimed that their main OSS project was at least as creative as anything else they had done in their lives. They also found that determinants of how many hours each programmer spend on their project are *"Enjoyment-related intrinsic motivations in the sense of creativity"*, *"Extrinsic motivations in form of payment"* and *"Obligation/community-related intrinsic motivations"*. These findings support the survey of Hars and Ou (2002), and also notes that intrinsic and extrinsic motivations very much co-exist in OSS and that the existence of one does not extort the existence of the other.

While there is a lot of litterature available on the various types of motivation in OSS, there is less research done on how a project should proceed in order to motivate people to join and keep them interested. However, Roberts et al. (2006) conducted a research where they found that a persons contribution levels are affected by different types of motivations. Their performance rankings in the projects are also related to contribution levels.[4] As the era of OSS 2.0 emerged, people are now also getting paid by commercial companies to work full-time on open source projects. Huge investments are being placed in projects and money is a viable motivational factor for developing open source software. Roberts et al. (2006) found that developers that are paid by corporations to work on OSS projects, have above average contribution levels and therefore also attain better performance ratings. Although the corporations involvement are purely selfish and extrinsic, it is also of great help to the project, both by increasing amount of contributions and promoting the project for further growth. The involvement of commercial forces are encouraged and seen as a great opportunity to recruit new members and increase sustainability to the project, a crucial attribute for OSS projects. Roberts et al. (2006) also notes that developers with higher status motivations are often more active contributors, recognizing the achievements of developers, through e.g. a website. This type of public recognition would probably also serve to increase developers ownership within the project and increase the likelyhood of further

[4]Performance rankings are a way for OSS communities to assign contributors rank, by periodically evaluating their actual contributions.

participation.

As an example of some commercial forces in OSS communities, Google has an annual event each summer, where students within computer science can participate in development of OSS projects. The students are mentored by experienced contributors in the OSS scene, approved by Google. Google recieves applications from OSS projects and selects which ones to cooperate with from a list of criterias. This has turned out to be a profitable arena for all parties involved: Google, students and OSS organizations. Google gains reputation and it is a great way of recruiting potentially new employees, as they get to see how they cooperate with people and evaluate their technical skills. Students gain valuable credentials to add to their résumé, increasing their chance of getting jobs, and also increased knowledge and technical skills. The project organizations benefit by possibly recruiting long term contributors through the Summer of Code project, as well as increased publicity.

2.3.2 Forking

> *"Imagine a king whose subjects could copy his entire kingdom at any time and move to the copy to rule as they see fit."*(Fogel, 2005, p.88)

Forking is a term that is well known to the open source community, but not so much to the people in the proprietary software. In this section the phenomena of forking will be explained, as well as some of the consequences and why projects invest in measures to avoid it. Forking is an event that is mostly unique to OSS, and it is the process of creating a derivative work of the OSS project. The reason for forking being unique to OSS is in the nature of open source licensing, discussed in section 2.2. An open source license does not carry protection from other people using your project to create their own proprietary or open source works (depending on the license that the original work has). Due to this characteristic of OSS, a fork could be made at any time, as opposed to proprietary software, where the source code is protected by copyright. A fork is not just a derivative of the original work, though, like any modification of the software would be. It is created with the *intent* of either replacing or competing with the original work (Wheeler, 2007), which is why it is potentially a harmful event for the original project. To illustrate the event that is called forking, take a look at the timeline in figure 2.3.2, showing each fork of the original GNU/Linux operating system as a separate timeline for each fork expanding from the work it is based on:

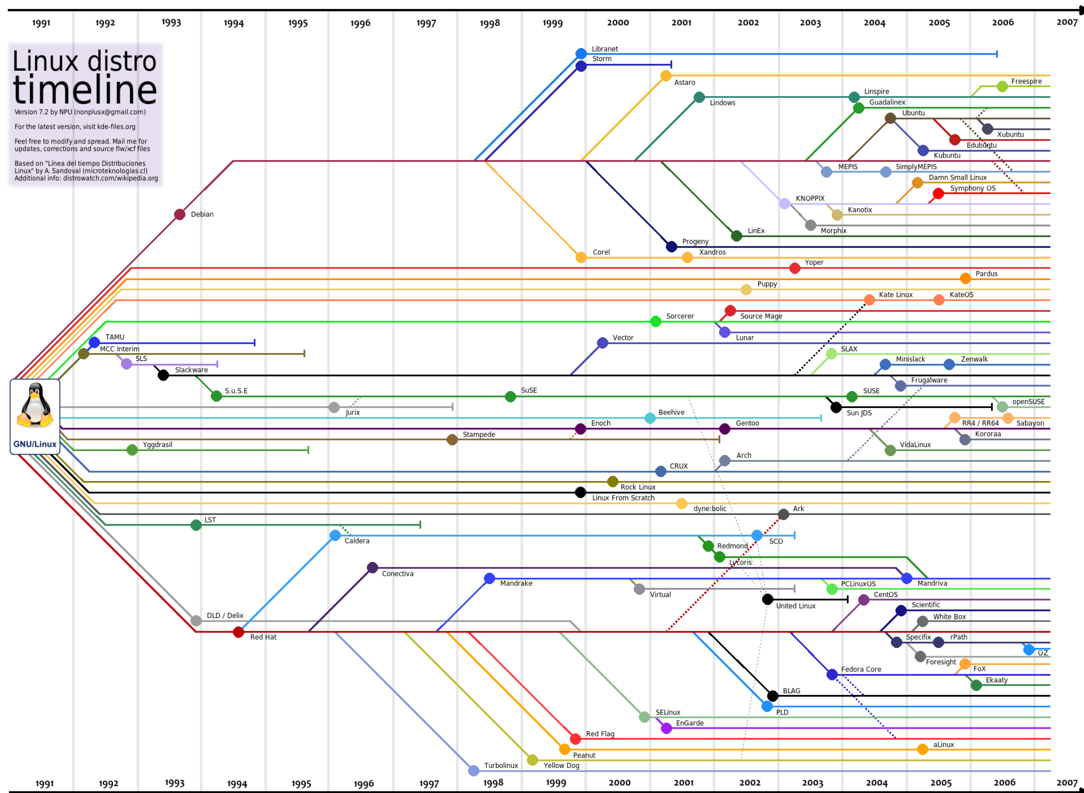


Figure 2.3.2: *Forks of the GNU/Linux project. Full page image can be seen in Appendix C*

As you can see, there has been a massive number of forks from various Linux distributions ever since its inception in 1991. These forks have different motives and background for being created. One reason behind creating a fork might be lack of one or more features that is craved by users. This feature might feel so crucial to one part of the community that a disagreement seperates them. If they are not able to make a compromise or collectivily choose a solution, one of the parts can create a fork to replace or compete with the original work. As an example, this happened about three years ago in an open source project named Pidgin. This is an instant messaging client which incorporates many different communication protocols into one application, relieving any user the burden of having to install many different clients for each messaging protocol. In one of the updates (Version 2.4) from the application, the developers removed the users ability to resize the text-input form, causing what the users felt was a step back in usability (swbrown, 2008). A large discussion was caused on Pidgin's bug tracker with a lot of members rooting for reverting the change, but developers refused to listen. Eventually this resulted in the creation of a fork, which was named FunPidgin. The new fork had a tagline that procaimed that they **worked for the user**.

Figure 2.3.3

Other reasons for creating a fork could be seizing an opportunity in the market, where there is an obvious need for a competing project. A fork could also serve the function of re-igniting an old project. For whatever reason created the fork, the outcome can be hugely different and relies on a number of factors. As identified by Wheeler (2007), there are four different outcomes from creating a fork:

1. The death of a fork (example: libc/glibc). This is by far the most common outcome; indeed, many forks never receive enough support to "die".

2. A re-merging of the fork (example: gcc/egcs); this is where the projects rejoin each other (though one or the other may be the dominant source of the combined effort).

3. The death of the original (example: XFree86/X.org).

4. Successful branching – both succeed, typically catering to different communities (examples: GNU emacs / xemacs, OpenBSD).

Although it might seem like a common thing to do from watching the linux distribution timeline (See 2.3.2), it is really a very rare event and as (Raymond, 2001,

p.73) notes: *"There is strong social pressure against forking projects. It does not happen except under plea of dire necessity, with much public self-justification, and requires renaming."*. For a fork to succeed, it needs to have enough followers and users to create a sustainable community. This is not easily done as most OSS projects are already scarse on human resources, and it is probably the main reason for the demise of a fork. A fork will also need competitive advantage over the original, in order to prosper. This can be gained by offering the targeted audience a substantial leverage, in form of features, speed or quality, that the other applications does not have. Habits are hard to break, and people usually need very good reasons to do so, since they may have invested a lot of time adjusting to the software and it has become a part of their daily routine, either at work or home.

As noted by Fogel (2005), the *possibility* of a fork has a lot more to say on how the projects are run, than if a fork should appear. To protect the project against a possible fork, people have to make compromises when working together and the project needs some kind of governing system to deal with decision making. Fogel (2005) examplifies two common models used in running OSS projects in order to support decision making. These are rarely followed to the point in any OSS project, but the models serve as a practical example to clarify how important framework for decision making is, to avoid forks. One being the *benevolent dictator* model. In short, more often than not, the creator of the project has the final say in decisions, but only uses this power when absolutely necessary. The other model is a consensus-based democracy, which is a sign of a more mature open source project, where disagreements and other complicated matters are settled by vote. Projects tend to start out like the benevolent dictatorship model and move on to the consensus-based democracy, as more people are highly connected to the project and feel ownership they want their vote to be heard and count. These types of models are also discussed by Raymond (2001), where he notes that the consensus-based democracy model are mostly used by larger projects and with a voting system. Further he notes that some projects have a governing system with a rotating dictator system, but these complicated arrangements are difficult to maintain and a source of disagreements.

The ability to fork a project is one of the cornerstones of OSS, what makes it unique and the strings that hold project communities together. People are forced to make compromises and work together, in order to avoid forks. It might be difficult to see how much effort should go in to minimizing the risk of avoiding forks, when it is hard to predict what effects a possible fork could be. Successful forks are rare, but even so it might be one of the important parts of what drives OSS as evolutionary and innovative software.

2.3.3 CSCW And Group Awareness

In any office situation, developers are given the ability to have face-to-face communication on a daily basis. Face-to-face communication is without a doubt the most valuable communication method, with uninterrupted, focused and explicit communication. It also carries the powerful subtext of body language, a big part of human-to-human interaction. This is one of the luxuries that the distributed development of open source software does not have. People are literally scattered all over the world, but cooperating on the same projects. Not only separated by geographical distance, but also in widely different time-zones. This distributed way of developing software is one of the characteristics of open source development and it is also a major reason for people to question how and why it all works. In addition to working at separate locations, not all members of the project work on the project full time, in fact more this is a very rare case. Many, if not most members of a project, have probably just joined in to either fix a bug that they noticed while using the software, or they might want to add a feature or function. This creates a situation where swift and effecient group awareness is critical in order to keep code quality at a satisfactory level, and issues with dependency to a minimum. Great care to avoid chaos, misunderstandings and double work have to be in place.

In open source development areas of responsibility have traditionally been devised to the person of expertise. Someone who has in-depth knowledge of the field and recurringly work on this piece of code within a project, naturally gain responsibility for the area. Anyone who would like to add, remove or fix anything in regard to this field will likely want to communicate with this person. In an office situation, this is obviously just a matter of walking to the persons office, call him or talk to him while taking lunch break. This is impossible with the distributed nature of open source development. This is where computer supported cooperative work (CSCW) and group awareness comes in. Appropriate tools for communication are needed. In order to understand group awareness in open source development it is important to grasp the basic principles of how it works in co-located situations. Gutwin et al. (2004) identifies three mechanisms that provides group awareness in co-located situations:

1. Explicit Communication
2. Consequential Communication
3. Feedthrough

The first mechanism, namely explicit communication, is when people are told about what the others are working on, or about to work on. Such as meeting

colleagues in the office hallway, visiting offices and recent trends where offices are no longer separated into rooms, but workstations are spread around in a large open room. This open landscape office style further enables and lowers the bar for explicit communication. Consequential information is when people are informed of another persons current work and future intentions by looking at what they are doing. Lastly there is feedthrough, which is when any artifact from changes in the project is an indicator of who has been doing what (Gutwin et al., 2004). This kind of information can be gotten through the use of version control systems, such as Subversion or Git. By using revision control systems, developers leave trace whenever commiting an update to the software, and it can be supplemented by descriptive text for elaborate understanding of the change.

In open source development there is a need to replace the way these mechanisms works in co-located software development. If not replace, then use tools to in order to support how they function. Such tools are applications called groupware and they are defined as:

> *"[...], the term 'groupware' is applied to applications that support interactions within groups of two or more people."*(Grudin, 1989, p.246)

One of the challenges in implementing groupware, as noted by Grudin (1989), is getting people to use it. Adding extra overhead to work is often viewed as unnecessary by users, and therefore likely to get ignored or in best case a form of workaround is created. Considering the fact that most members of an open source project will not be available at the same time and same place, some means of support for asynchronous communication is needed. There are various software solutions for this, the best known and most successful, being electronic mail (email). The use of mailing lists, where people subscribe to a list and any email sent to this mail address is delivered to all subscribers, has proven to be a very successful way of communication. Gutwin et al. (2004) noted that they *"[...]were struck by the capabilities of text-based communication for supporting awareness,[...]"*(Gutwin et al., 2004, p.81) and concluded that the primary communication tools in the projects they studied (NetBSD, Apache httpd and Subversion) were mailing lists and chat tools. Mailing lists are a great way of combining both the mechanics of explicit communication and consequential communication. By explicitly requesting or giving away information, everyone in the list is made aware of the subject and thus informed consequentially whos doing what and when. There is a risk however of the project growing large too quickly and the list will get polluted by junk mail and/or questions not relating to developing the software. Other well known and frequently used tools for asynchronous communication worth mentioning, are newsgroups, bulletin board systems (BBS) and forums.

Synchronous communication is also of great support for creating awareness. Tools such as real time text communications, chat tools, are widely used and important as noted in the previous paragraph. Internet Relay Chat (IRC) emerged at the end of the 1980s as a way to enable BBS users to communicate in real time (Jarkko Oikarinen and Darren Reed, 1993). This enabled the developers to create their own chat room and ask small questions and carry out off-topic conversations without the effort of writing an email addressing everyone in the project. All in all it is an effective way of making social relationships with people, since it is not as formal as the mailing lists, and anyone in the channel can see what is going on, which promotes awareness. IRC was however shut down in the Apache project due to it excluding everyone that was not on the chat at the time of conversations. They felt that people were left out on, what could be important, conversations and thus create a bad flow of communication which could lead to misunderstandings (Gutwin et al., 2004).

Figure 2.3.4: *mIRC is the most widespread and popular IRC client for Windows, with millions of users.*

One important new way of communication which has emerged more recently with the era of OSS 2.0, is the idea of projects hosting conferences and meet-ups. This is of course mainly done by successful and large projects. Meet-ups and conferences can be beneficial in many ways. Projects create PR necessary to create both opportunities with investors and cooperation with other open source projects and commercial actors. More PR also creates increased public awareness of the project, which may lead to increased growth in developers and users. These meetings also give the project a way of planning a roadmap for the project in a more formalised and commercial way of doing it (Fitzgerald, 2006). Open source projects such as Apache and NetBSD host events more than once a year at locations all over the world. While not everyone is able to participate on these conferences, by creating online summaries and perhaps even live audio or video feeds from the event, it is a powerful way of creating group awareness.

Figure 2.3.5: *This figure is an overview over the different kind of communication tools and what situation they serve.*

2.3.4 Project Stakeholders and Requirement Engineering

Requirements engineering is generally considered a difficult but also very important part of software engineering. A clear understanding of the ambitions and plans of a stakeholder is a dominant part of the applications success. An application that does not fulfil the requirements of the stakeholder, will not be used, and further cooperation with the stakeholder might get compromised, resulting in loss of revenue. The nature of OSS development, where the stakeholder for the project has traditionally also been the developer, makes it possible to skip a step in the process of creating software. The fact that it seemingly skips this hugely important task in traditional software engineering and yet makes highly valuable, high quality

and innovative software, is something that requires further investigation.

In the early days of software engineering, OSS projects have primarily been developed due to the lack of a sufficient alternative on the market. During the 1970s and 1980s the people working with computers were predominantly researchers and hackers, and the software they created was the result of scratching a personal "itch"(See section 2.1.1). The software was not made with the intention to please a larger crowd, but mainly their own needs. Anyone else who just so happened to require the same type of tool, was more of a positive side effect. Requirement engineering was easily done, as they knew exactly what they wanted themselves, there was no need to formalize the process. There was no deadline to meet, and should the program not meet the requirements it was just generally considered not done, and work is continued. Projects were basically not developed according to a formalized requirements analysis, as they usually are in commercial software development.

As we can see, the stakeholders in the OSS projects have traditionally also been the developers. In the past few years, as the involvement of commercial companies have increased and people are seeing the benefits of OSS, we can find projects where the stakeholders in the projects are no longer limited to the developers of the software. Does this new set of external stakeholders create the way for a new formalized steps for requirement engineering? Scacchi (2002) conducted an empirical study on four different OSS projects, trying to understand how requirements for open source software were developed in contrast to traditial requirements engineering in software development. There was no evidence of any formalized requirements engineering in any of these projects resembling whats done in traditional development of software. However, Scacchi (2002) found that informal functional and non-functional requirements were analyzed, elicited, specified, validated and managed through a number of Web-based descriptions. These ways of doing alternative requirement engineering were presented in the article, identified as eight different "software informalisms":

- Discussed through community communication tools
- Made publicly visible on Web pages
- Online HowTo guides
- External publications such as technical articles and books
- Open software Web sites
- Bug tracking systems
- Software system documentation for end-users and developers

- Software extension mechanisms such as an API or Plugins

These informalisms are a way for OSS projects to dynamically deal with requirement engineering in lack of a formalized framework. They are extremely flexible, which is only natural as they need to support the volatile behaviour of OSS development.

As a practical example of this new evolvement in FLOSS development, a new open source Electronical Medical Record (EMR), is being worked on under the name of OpenMRS (`http://www.openmrs.org`). The work started in 2004, and it is aimed at being an EMR for developing countries, especially helpful for countries ravaged by the HIV/AIDS epedemic, where resources and software developers to implement these systems are scarce. This type of project is different from the traditional way of developing OSS. The stakeholders of the projects are not the developers themselves, but possible hospitals in developing countries as external entities. OpenMRS therefore has to be extremily flexible, have good scalability and most importantly it also has to be very easy to implement due to the lack of available software engineers. The last point proves a challenge as the system should be able to be implemented without any substantial programming effort. In South Africa the first efforts to implement this proved successful, and the results from this have been used as a model for further implementation in other developing countries (Seebregts et al., 2009). The OpenMRS project focused on using mailing lists, conference calls, a wiki site, code versioning systems and project tracking software to support their collaboration. These systems in addition to a data model drawing on 30 years of history from the Regenstrief Medical Record system and an API centered on ease of use, can be used as examples of how the requirements engineering for the project were informally handled.

While this is only one example of an open source project with external stakeholders, it is fair to assume that this might be a more common scenario in the future, as more projects are trying to compete with the commercial market. Competing with the commercial market implies that the program is aimed at a broader audience than just the developers itself, but despite all this, there is not yet any evidence that the requirements engineering in OSS communities has become much more formalized.

Chapter 3

Traditional Software Development

Traditionally the commercial software development process has been characterized by huge costs, focus on elaborate documentation and a very low success rate. The development process from the early days has been sequential and prone to failure, as described famously through the waterfall model by Royce (1987). These sequential development methods, were rigid and naive, in the sense that they did not account for any change during time of development. It was presumed that once you had done the requirements engineering and laid out your plans for development, you could reach the final product in one direct step of development and perhaps some debugging and testing before release.

This rigidity in the development process, lead to a process that more or less blindfolds the stakeholders in the project until the release of the final product. Surprises are not necessarily a bad thing, but since all the requirements for the project takes place early on in the project, the lack of frequent communication can lead to misunderstandings, and thus a product that does not fit the expectations of a client. With millions of dollars invested in new information and communications technology (ICT), surprises are generally not very welcome, especially when they are not the good kind.

The waterfall model was intentionally an attempt of creating a more iterative and less rigid development methodology. The idea that Dr. Royce tried to convey was that you would make the project more flexible through the ability to go back one step and re-build it, should anything not match up with the requirements. It was however misinterpreted and people saw it as direct flow to success, not taking the iterative aspect into account, it was a purely incremental development methodology. By doing the development only once it was believed that the new incremental model, separating the phases of the project, would be a great way of

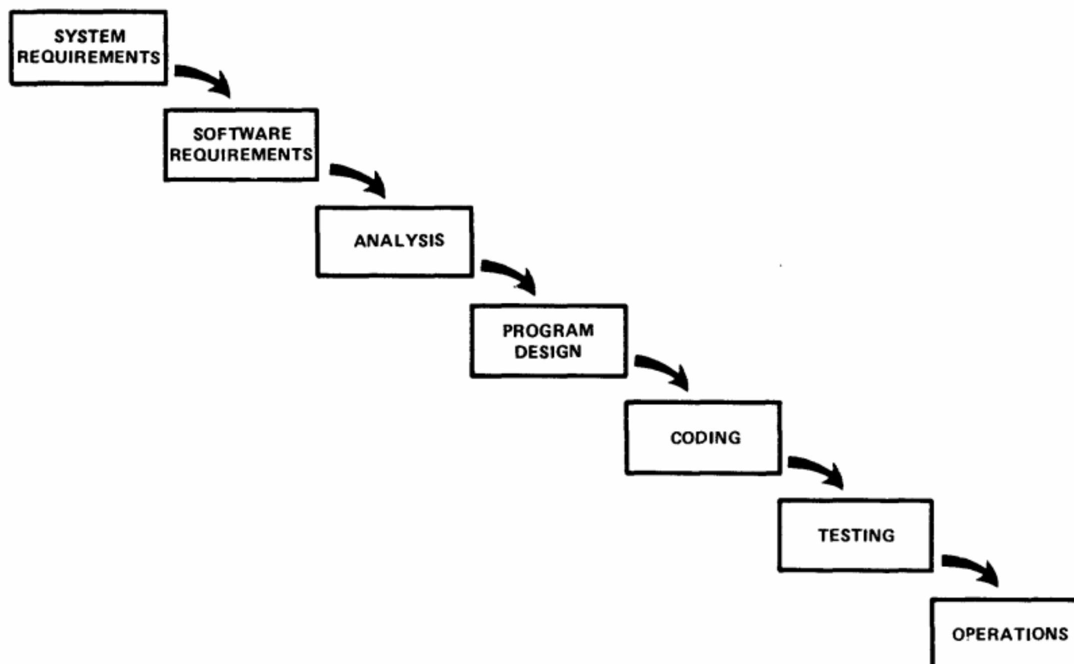


Figure 3.0.1: *The Waterfall Model as described by Royce (1987)*

reducing the cost. The problem with this type of incremental development is that requirements change and new challenges arise during the project, not just before or in the planning phase. As noted by Cockburn (2002): *"The reason that would-be-cost-optimized projects so rarely succeed in their goal is that surprises pop up at all stages in software development."*

The iterative methodology started evolving as the failure of sequential and incremental models became more apparent. Boehm (1988) presented a new iterative model, called the spiral model, with a clear focus on creating prototypes to verify the requirements more than once, to eliminate the rigidity of purely incremental models. The model was also created to better spot options of code reuse, eliminating errors and unattractive alternatives early on in the project. An important point in this model and perhaps the biggest step forward from earlier methodologies, is the fact that it recognized software development as a process with continuously changing requirements. He also notes that 25 projects that has fully incorporated this model, increased their productivity by at least 50 percent. However the metrics involved in how productivity was measured is not discussed in the article Boehm (1988).

Building further on the idea of an iterative process, spawned the methodologies

that are used in todays software industry. They are generalized under the term Agile software development and present a what can be seen as a ”paradigm shift” in development methodology. In the following section this new era will be adressed, as well as one of the most popular frameworks for Agile development in the business, named Scrum.

3.1 Agile Methodology

As a reaction to the documentation-heavy, slow and failure prone incremental process of developing software up until the 1990s, a set of various agile methodologies began to emerge. It is important to emphasize that Agile methodology is not in itself an explicit framework on how to run software development, but rather new way of thinking; a philosophy derived from the similarities in current best practices in software development. Agile methodologies are characterised by a number of features, written down in what was called the Agile Manifesto. The manifesto was a result of a conference held by a number of representatives from various agile methodologies, in 2001. It contained the essence and characteristics of the principles behind agile development. The four main values of Agile development as listed in the Agile Manifesto (Kent Beck, 2001)[1]:

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

These values are a result of many years of trial-and-error, and a much needed change in the software industry. This methodology more correctly adapted to the complex and nature of software development, requiring developers to respond to change during the project. It also focused on accelerating the products time-to-shelf, making companies more resilient to changes in the market.

In its core the Agile methodology is, as noted by Highsmith and Cockburn (2001), centered around generating early feedback from the stakeholders. Iterations should generally be somewhere around two to six weeks at maximum, not the half year or longer iterations that was common in earlier development models. The project draws several benefits from this increased flow of communication with the stakeholders. Firstly, it gains the advantage of continuously updating the requirements

[1] The full manifesto can be seen in Appendix E

in line with what the stakeholder wants. It is commonly said that the client never really realises what they want from the project when they approach the software developers. As they see how the project is developed, they can adjust and customize the requirements as they see fit. The increase in cost of making changes is often depicted as an exponentially growing curve as time moves on. Changes late on in the project timeline will also delay the final implementation of the product, something that can severely harm a clients business. A lot of software consultant companies are today placing the development teams at the offices of the stakeholders and not the company's headquarters[2]. This does of course maximize the potential flow of communication between the stakeholder and developers.

Already there are lines of similarities to be drawn between Agile and open source development. Warsta and Abrahamsson (2003) did an effort to compare Agile with OSS and found that they were actually strikingly similar in most ways. The quick initial release, frequent iterations, lack of formal communications, heavy focus on informal communication. The distributed development that entails the OSS development does serve a few limitations towards fulfilling all the primary values of Agile. Agile focuses on face-to-face communication and thereby bringing the people involved closer together. Open office enviroments are encouraged to increase the flow of knowledge within the company, by breaking down the initial effort to get in touch with people. With the geographical disparity in distributed project, these types of communication and arrangements are not feasible. It is evident from the preceeding sections, that there is a lot of insight in how the projects are run within FLOSS communities, but there seems to be a lack of litterature favoring how OSS can learn from recent Agile methodologies and business in general.

There is now an abundance of research that has been done on applying the Agile methodologies in various types of development enviroments. The field is rather new, though, and most research done on Agile is from after the year 2000. As Abrahamsson et al. (2009) stated, there are still some fields that require research and the definition of what constitutes 'agility' has to be clarified. The research until this day has been predominantly adoptions of various Agile models into specific enviroments, such as large group teams or specialized software development. It is also noted that more research on how to implement Agile methodologies on an organizational level, not only to the project teams, could be beneficial. The field of applying agile methodology also seems specialized to smaller projects, and the larger more complex projects might not be able to attain such a degree of agility, as noted by Turk et al. (2000).

[2] I have myself been visiting a few consulting companies headquarters, and even midday they can look like ghost towns with their vast arrays of empty workstations.

To sum it up, Agile methodology today consists of a number of frameworks and philosophies and is different from the older methodologies that it challenges the development teams to change the way they think about software development. The most popular Agile frameworks are:

- Extreme Programming
- Scrum
- Crystal Clear
- Dynamic Systems Development Method (DSDM)

One important feature that is emphasized in Extreme Programming and other agile methodologies is test driven development. In order to create *working* software deliveries early and often, a framework for testing is implemented. One type of testing that is often used in software companies is regression testing. By doing nightly regression testing, you can make sure that any changes to the code works as intended and does not introduce bugs or unexpected behaviour to other modules. Regression testing is then used as a safety net to ensure that software can be delivered frequently to the client.

In the next section I will go closer into the details of an Agile methodology, the Scrum framework.

3.1.1 Scrum

Scrum has become one of the most popular Agile methodologies in the recent years. It is not a set of guidelines strictly telling you what way to run a project, but like the Agile methodology, more a framework or a set of tools that the development team can use in order to increase the rate of success and speed of delivery. After years of testing this framework during several projects, making sure it really worked, Ken Schwaber released a book defining the framework that is Scrum (Schwaber and Beedle, 2001).

Figure 3.1.1 illustrates the normal life cycle of an iteration within a Scrum project. Each iteration is called a Sprint and lasts for 2-4 weeks, depending on the type of project and how Scrum is adapted. Scrum comes with a set of suggested roles to be used in the project. The ”Scrum Master”, usually a project manager with the responsibility of making sure the ”Scrum Team”[3] is not commiting to more work than they can manage within a Sprint. ”Product Owners” are the stakeholders of the project. Their role is to evaluate the work that has been done after an

[3] Generally known to work best with small teams consisting of 5-9 members.

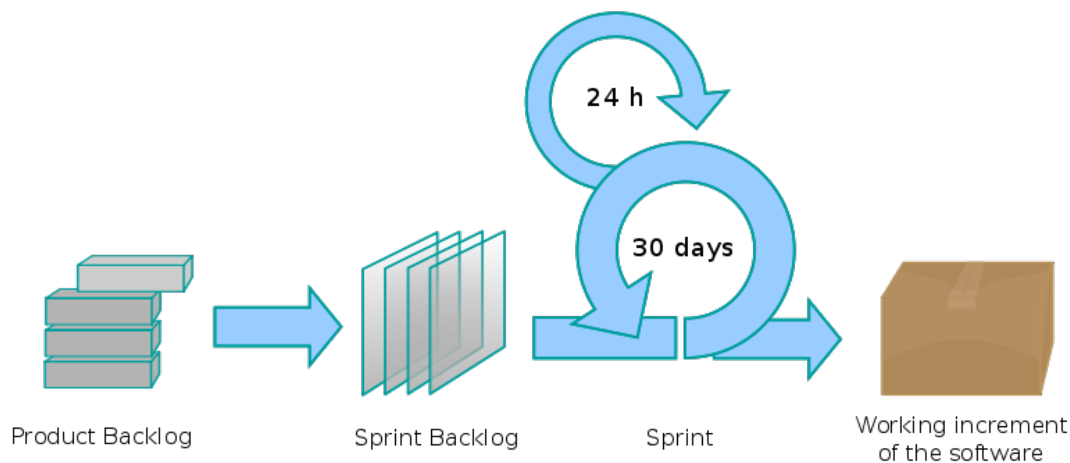


Figure 3.1.1: *One iteration within a Scrum project.*

iteration, and consequently add or remove any requirements. These requirements are contained within a ”Product Backlog”. The Product Backlog is one of the strengths of Scrum. The backlog is maintained throughout the project and tasks are often encouraged to be decomposed into as set of smaller tasks if possible. Each tasks carries a priority and an estimate of how long it will take to complete the task.

For each Sprint, there are a number of events to be done. First off is the Sprint Planning meeting which is where the team decides what tasks from the Product Backlog to include in the Sprint Backlog. They have to be sure that the time estimates for tasks that are moved to the Sprint Backlog, does not exceed the available time in the sprint.

Scrum has a great framework for raising all team members and stakeholders awareness, on the projects development status. Each morning, short (15 minutes) team meetings are held, where each member has to deliver a quick and informal status report, making everyone aware of whos doing what. In addition to this, what is called a ”Burndown Chart” is incormporated. This is often a physical draft of a graph, showing the number of tasks remaining. This graph serves two purposes. Firstly it raises awareness by showing the estimated speed compared to actual completion of task. Secondly it serves as a way of learning how fast your team works, in an actual calculated metric called ”Velocity”, and thus bettering the chance of correct estimation for the next Sprint. An example of how a burndown chart can look is seen in figure 3.1.2 below.

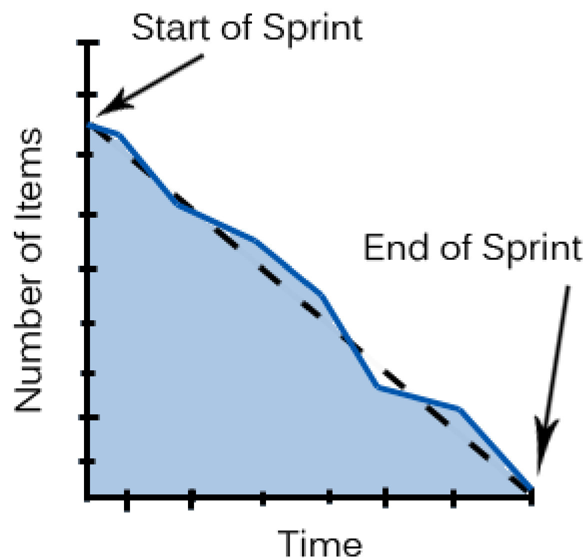


Figure 3.1.2: *Illustration of a burndown chart.*

After a few years of developing this framework and later seeing it in use throughout a number of projects whithin a few companies, Ken Schwaber reported that *"Projects are delivered on time and often exceed the expectations of both users and management."*(Schwaber, 1995, p.21). The empowering of stakeholders, making them join in on the actual development process, as well as the frequent meetings and focus on communication and presence of team members, are undoubtedly effective tools. As a testament to the success of scrum, one of the largest software projects that has been run in Norway, was done with scrum at a team level. The project was renewing the pension part of the Norwegian labour and welfare services web site. In 2011 alone the estimated expendures on consulting services and maintenance were 850 million Norwegian kroner (approximately 150 million USD), where 400 million kroner is used on the pension project alone (NAV, 2010). As noted in the preceeding section, larger projects can have a difficulty of applying Agile methodologies to the full extent. This particular project had more than 100 developers working on it, and definitely requires an elaborate framework. The process framework was solved by splitting the developers into several Scrum teams with a ScrumOfScrum team on top, that they reported to. The overall project was also planned within the framework of a waterfall model (Haugland and Rabben, 2008).

3.2 Development Vs. Maintenance

The field of code maintenance is undoubtedly a big one in the software industry. Research dated back to the late 1970s, are concerned with how maintenance of software is problematic both in relations to understaffed maintenance departments and it being a very costly and time consuming process (Lientz et al., 1978, Bennett and Rajlich, 2000). However, it seems like the area of software maintenance research has had much less attention in comparison to software development.

So what exactly is confined within the area of software maintenance? Lientz and Swanson (1980) divided maintenance of software into four different areas:

- Adaptive - changes in the software enviroment
- Perfective - new user requirements
- Corrective - fixing errors
- Preventive - prevent problems in the future

Software maintenance was for long viewed as a singular phase that occured once the initial product was done and delivered to the client. This theory was however expanded later where Bennett and Rajlich (2000) proposed a software lifecycle model where the maintenance phase after the initial product, was divided split into three separate stages.

> *"Its key contribution is to separate the "maintenance" phase into an evolution stage followed by a servicing stage and phase out stage."* (Bennett and Rajlich, 2000, p. 77)

While Bennett and Rajlich (2000) elaborated on the maintenance phase, it was also noted as a key point that the knowledge about the application domain, gained during the initial product phase, was crucial to later phases. This is due to software evolution requiring a deep understanding of the architecture, in order to make effective and non-intrusive changes to the software.

The evolution phase describes a stage that is following the success of an initial product launch. In this phase the product will be adapted according to any changes in requirements, which is very likely to happen. It could be said that this evolution phase is now a part of the development phase as well, if the developers are using Agile methodology. As discussed in the previous section, the Scrum framework incorporates this change in requirement from the client in every iteration.

As the evolution of the software comes to an end, the service phase is entered. During this phase any change to the code is likely to be minimal, such as a bug

fix, patch or a wrapper. Further down the line as the service phase endures, Bennett and Rajlich (2000) suggested that a process called *code decay* occurs. Code decay happens due to a number of reasons. First off the software can suffer what is called software erosion. Software erosion happens when the developers usually takes shortcuts in fixing code, neglecting the importance of adhering to the defined software architecture. It could be a process that is invisible in the early phases of erosion, but later on as the architecture grows less coherent the project will notice a significant rise of cost in maintenance due to an increased complexity in making changes. This loss of coherent architecture is connected to software knowledge. The erosion of software, causes less coherent architecture, as discussed, and thus any further changes will require more software knowledge. This could also happen the other way around, as the loss of software knowledge - most common due to the loss of key personel - will potentially lead to a weaker architecture.

When the software reaches the time for termination, usually marked by the emigration from the current software system to a new replacement, it reaches the final stage, the phase out stage. Users of the system are forced to work around existing faults in the system and eventually move on to the replacement.

Software maintenance has for long been viewed to be a ”necessary evil” (Banker and Slaughter, 1997) and perhaps less prioritized than software development. Even so, the maintenance of software is a huge economical drain, and of great importance to any organization. With its complex and error prone process, it might not be too difficult to understand that software maintenance has been called a necessary evil. It is generally viewed as a less ”glamorous” profession than developing new software. The essence behind that might be that people want to create something new, not fix the previous work of other developers. There is more prestige in delivering new items to clients, than maintaining old products. This new view presented by Bennett and Rajlich (2000) is an important change of mentality. It shows that maintenance of software is not just a trivial step following the development of the product, fixing a few bugs and making some changes. Maintenance should be thought of from the very beginning of the project, by planning a software architecture with both modularity and testability, then further developed with this mindset during the initial development phase.

Part II

Case

Chapter 4

Research Methodology

There are a number of research methods available, generally they are split into qualitative or quantitative research. Qualitative research is about figuring out the reasoning behind certain behaviour or phenomena through the act of interviews, interpretation of data through argumentation and is often characterized by determinating what is being looked for through the process of studying these types of material. Quantitative research, on the other hand, is good for research where you know exactly what you are looking for in your data, and can be modelled statistically or mathematically to provide a conclusion. The advantage of quantitative research is that the results will be very concrete and easy to validate, however the ability to quantify the data into statistical models is not always applicable. As I will research the cooperation of a group of individuals and their behaviour through mailing lists and other means of interaction, it will not be possible to carry out studies that rely on gathering purely quantifiable data, and quantitative research is thereby not the best choice for the thesis.

Chua (1986) managed to classify three different scientific perspectives when doing research, namely critical, positivist or interpretive. These perspectives are used in research depending on philosophical assumptions, as noted by (Klein and Myers, 1999, p.69) and can be summarized as following:

- **Positivist**: *"[...] there is evidence of formal propositions, quantifiable measures of variables, hypothesis testing, and the drawing of inferences about a phenomenon from a representative sample to a stated population."*
- **Critical**: *"[...] the main task is seen as being one of social critique, whereby the restrictive and alienating conditions of the statuos quo are brought to light."*

- **Interpretive**: *"[...] it is assumed that our knowledge of reality is gained only through social constructions such as language, consciousness, shared meanings, documents, tools, and other artifacts"*

The positivist is perhaps the most desired perspective to be used in scientific research as the results are most likely easily verifiable. However, it is not suited for the research that this thesis will be doing. By this list, the most fitting perspective is the interpretive, as I will be required to peek into the communication artifacts and thereby look at their influences and what caused this behaviour. The interpretive qualitative research method of studying will therefore be used in this thesis. Enabling me to go through with the method of doing case research will be used as a practical approach to the problem. As noted by Oates (2006), case studies are used to get in depth of a social and cultural phenomena. Doing a case research will enable me to delve into real life projects and give me the opportunity to gain a deeper understanding of the questions posed.

4.1 Getting Access

At least one or two open source projects will be needed as case data. There are a lot of open source projects out on the internet, SourceForge.net alone hosts more than 260,000 projects (Sourceforge, 2010). In order to find projects for my thesis, I have to find a set of criterias that the project will have to satisfy, in order to narrow my search. A suggestion to criterias could be:

- Size has to be more than 30 members.
- Activity level has to be high.
- It has to have publicly accessible data.
- It would be best if it is a project I get involved in.

The first two points are closely connected. In order for me to get enough data, and correct data, there has to be a large and active community. Without it, the risk of not getting any viable data to analyze is too big. Open source project usually communicate through mailing lists and Internet Relay Chat (IRC), and should it not have many active members using these channels, the risk of not getting any viable data to analyze is very high. Risk is something that one has to prioritize very highly as the time schedule for a master thesis is no more than two semesters. The mailing lists, forums and/or IRC channels need to be publicly available in order for me to draw anything at all from them.

While getting access to their documents might be trivial due to the public nature of open source software. Getting access to interviews, however, will be a more challenging task. There might be up to a hundred potential interviewees, most of them will likely have this project merely as a hobby project, making commits only once in a while. The majority of contributers will have work, friends and family to attend to in their spare time, and devoting time to an interview that is to no personal benefit for them might not be appealing. In order to make a better case when asking to interview persons I could consider getting involved in the project as suggested by Oates (2006). Getting involved in an open source project could require some technical skills, but a large project will likely have tasks divided into degrees of difficulty, requiring anything from simple proof-reading of Strings, to in-depth knowledge about protocols and algorithms. While it might be technically possible to get involved, this will require a lot of time from my side. Due to this I can not completely rely on gaining access in this matter, as (Walsham, 2006, p.322) says *"The process of gaining access, as outlined above, entails strong elements of chance, luck and serendipity."*. Getting involved in the actual case as a contributor should add to the impression of being sincere when I explain my interest in the field and for the case that is being researched.

Cornford and Smithson suggests that one should not depend on fully working with one organization, before you have gained access. While this is about gaining access to an organization, it could also be projected onto gaining access to an open source project that satisfies the criterias set.

> *"A safer strategy is to work towards developing links with a number of people or organizations who can all, potentially, contribute to your work."* (Cornford and Smithson, 2005, p.36)

To mitigate this risk, I will contact several people at the department of computer and information science, that I know might be involved in any open source projects, and ask for any guidance towards finding my case. As well as perhaps getting involved in the Software Development group at NTNU, called Programvareverkstedet. This student organization has a lot of litterature on open source projects and there should be several people that will be of potential help to my work.

As an entrance to the project, I constructed an email to be sent to the projects. The email is printed below:

> *"Hello,*
>
> *My name is Stian Haga and I am a student at the Norwegian University for Science and Technology (NTNU). I am currently on my fifth*

and last year of my master degree in information technology, and I am working on a master thesis, in which knowledge management and innovation within open source projects are studied. In light of this I need a few real world projects to use, in order to mine the data needed for analysis. I would love to use the VLC media player project for this, since it is under active development and of a team size that is corresponding to my requirements. The kind of data I am looking for, is conversations between developers, either in form of mailing lists (with archive access), IRC, forums etc. As far as I have gathered these channels are already public, but I still want to ask you for permission to use this in my master thesis. In return I will of course make my thesis available to you if you should wish. If my question has reached the wrong person, a direction on who to contact would be much appreciated.

Thanks in advance.

Best regards, Stian Haga"

This mail has several purposes. Firstly, it is a way to show my interest in the project and for the contributors to acknowledge my presence. The increased factor presence might be a minimal one, as the project might receive a big number of mails daily, and my email might drown. At least I have something to refer back to should I need to contact the project later on. Secondly, I make sure to ask permission to use their data, if there is anyone that would not like to have their data present in this thesis. Thirdly by offering them the chance to read my thesis when it is done, I offer something back to the community. While it might not be a big deal for many, it shows that I am interested in giving back and not just being a leech on their leg.

4.1.1 Anonymity

Ethics is a substantial field within research and should not be overlooked. To acknowledge the fact that the anonymity of individuals should be protected I will create a sort of mapping to allow the use of "codenames" in the thesis. This mapping will codify the names of any project members mentioned in the mails or from the internet relay chat. The coded names will simply be vDeveloper/vCore for the VLC project or jDeveloper/jCore for the jQuery project, followed by a number of no special significance other than to make the names unique. In order to recognize the hierarchy visible in the projects there is a difference in naming. Anyone who is a top contributor or member of the official hierarchical team structure, in the case of jQuery, will be denoted by the name "Core".

4.1.2 Collecting Data

The good thing about open source projects, and a huge boon to me as a researcher, is the fact that they are actually open to everyone. This will make it a lot easier for me to obtain a large set of data very swiftly, as opposed to a long process of earning trust from a company, and the various processes of gaining access to a set of documents listed by Oates (2006, p.236) can be bypassed. On the other hand, most large sized open source communities have vast amounts of e-mails archived, up to thousands of messages accumulated throughout the years of active development. As stated by Oates (2006, p.267) *"There is also a danger of researchers feeling swamped by the amount of qualitative data they have collected"*. This is one of the reasons for me aiming at one or maximum two cases for my thesis.

In order to navigate such vast amounts of data I created a graph showing the activity in the mailing lists. This activity graph could lead me to certain spikes in discussions, where there is an elevated chance of finding something relevant to any of the research questions, or serve as an initial effort to find any recurring themes in the projects. Below you will see one of the activity graphs created from jQuery's mailing list:

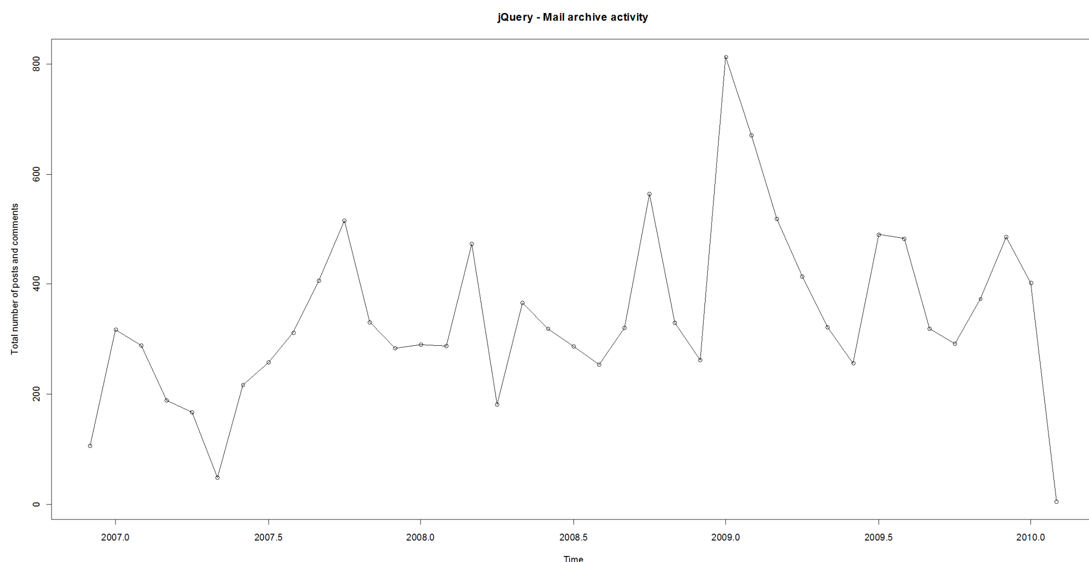


Figure 4.1.1: *Activity on the jQuery mailing list.*

As you can see from Figure 4.1.1, there are some evident spikes in the project. These spikes served as a starting point for me in an effort to increase the efficiency of the data collection. However, as you can see from the graph, there is an enormous amount of data available. With an average of approximately 300 mails

each month, and spikes up to 800 mails per month, this was no easy task. The VLC project had mailing spikes up at above 2000 mails per month, and also it is nearly double the age of jQuery. A (very) rough calculation, to illustrate the sheer amount of data, shows that the number of emails in these projects are:

jQuery: 12 months * 3 years of data * ˜309 posts per month = 11124

VLC: 12 months * 10 years of data * ˜646 posts per month = 77520

Total posts = 88644

Now while this is wildly inaccurate, it serves as an illustration to show to the amount of data that is available through the mailing lists. The graphs portraying the spikes as shown in Figure 4.1.1, did help as heuristics, serving as beacons showing places of interest. The process of searching for data was a long one, and it is further described in section 4.2.

In order to create a quick view of the software development activity, I gathered commit data from their repositories on Git[1]. This data was only obtainable as dates printed to a file, as far as I know. To format this into a list of commits per month I created a java application to parse the file and give me the desired output that can be used to create a graph. A sample from the source code can be found in appendix F.

4.1.3 Documents

The mailing list will most likely be my main source of data. I could take advantage of computer supported aid when looking for interesting topics, with text search as suggested by Oates (2006, p.276). The difficulty in this will be finding the right keywords to search for. This will obviously require a bit of work beforehand. Another technique to help me look for important and relevant data in the mailing list, is to look for bursts, that is to say where the frequence of mail per day may rise abruptly. By identifying these bursts, I can save a lot of time, as opposed to reading the entire mailing list - mail by mail.

Another method of obtaining documents, is observing the IRC-chat, should the project have one. To my experience, most active projects have got an IRC-channel for live support and discussions, and it should be a pretty safe bet. To achieve this I could code a simple parser to create logs, highlighting any words or sentences that I decide might be useful, using text search and regular expressions. This type

[1] A free and open source, distributed version control system `http://www.git-scm.com`.

of communication channel is usually more informal, and might allow me to see a different side to the conversations that I can find on the mailing list.

The IRC channels available from the projects are active sources of communication, with about 50-100 members online on the jQuery developer channel and about 250 members online regularly on the videolan channel. While this source of information can be important, it is also a strong sense of serendipity and luck involved as described in the preceeding section. There is a good number of people at the channels, but most of them are only activly involved as they are either requested or request the help of someone else.

4.1.4 Interview

In order to get in depth of my research questions, I will have to conduct interviews. This will allow me to get information regarding how the members cooperate and communicate from the community, hopefully supporting the analysis of documents, or even bring forth issues and routines that only exists as tacit knowledge within the group. It might also create a great contrast to my findings from the document analysis, making an interesting point to discuss in my thesis. Selecting interviewees will require me to do some research, as I would want to create multiple views, it would be ideal not only to interview a member who is on top of the contributer list (or even on top of the hierarchy, should there be one), but also a view from someone perhaps new to the project and an infrequent committer.

Oates (2006, p.196), sees internet-based interviews as very problematic. Most of his discussion regarding internet-based interviews are based on textual communication by chat and emails, and barely touches the aspect of VoIP[2]. This is another evidence of the quick advancements of technology, as this book was published in 2006 and only 4 years later, VoIP is free to use for everyone, and high quality video-telephony as well as of Skype version 3.0 released in 2007 (Skype FAQ, 2010). A statistic of 23 million users online simultaniously during peak hours and with a revenue of 406.2 million USD in the first half of 2010, this is a testament to the availability and accessibility of Skype today (Skype, 2010). By conducting interviews through Skypes high quality video telephony, I can negate most of the disadvantages of textual based internet interviews, namely the face-to-face communication. Although it will not be the same as an actual meeting in person, some body language will not be as obvious through a webcamera, it will still carry the advantage of being totally cost free. Without Skype, an interview face to face would probably not be considered an option, as the chance of any members being

[2]Voice over Internet Protocol

available for interview in Trondheim is very improbable. Traveling to other parts of the country or even other countries is just not a possibility, considering my limited time frame and low budget as a student.

The chance of finding someone available for interview is a tough one for a number of reasons. Firstly, as recognized in the article by Mockus et al. (2000), only a small percentage of the contributors are active and know the project well. This is verified as data from VLC's Git server is analyzed in figure 4.1.2:

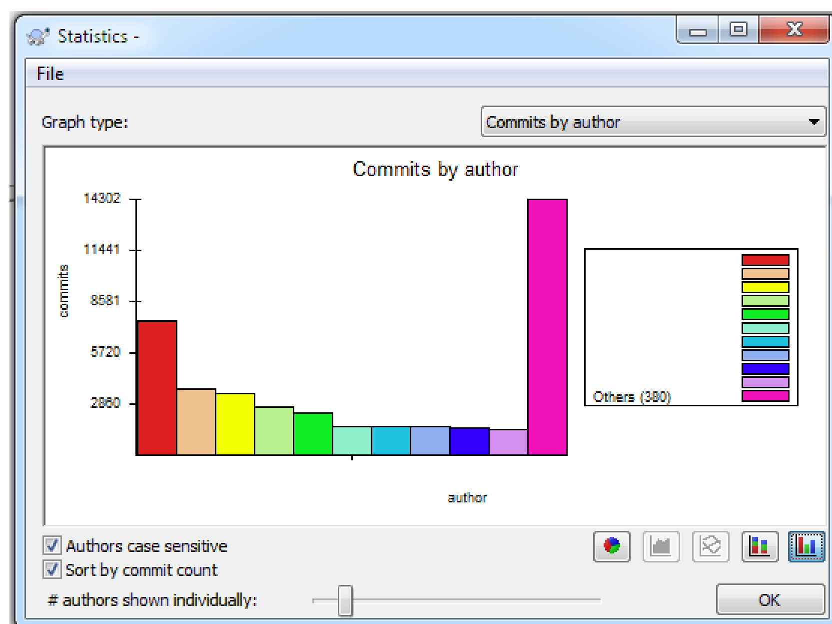


Figure 4.1.2: *Distribution of contributions in the VLC project.*

To gain valuable insight on how the project is run, ideally the interviewee has to be one of the core contributors in the project. Further making the interview more problematic, these people are often working on the project in their spare time and most likely they have other commitments filling their schedule. In order to try and arrange an interview I sent an email to the project:

> *"Hi!*
>
> *I am studying informatics and about to conclude my master thesis on open source software development, at the Norwegian University of Science and Technology. In the thesis I am investigating questions such as how projects gain growth and the issue of code maintenance vs. innovation. As cases studies for the thesis I have used the VLC Media Player project and jQuery. In this regard I would very much like to interview someone who is involved in developing these projects, and*

> *preferably have been for a few years. The interview will be informal and conducted to your liking, by the use of any real time messaging systems (IRC, MSN, Skype etc.). It should not take a lot of time, 30 minutes tops. Your identity will of course be kept anonymous, unless you wish otherwise. In return I will of course make the final edition of the thesis available to you, and I will be forever grateful. :-) Should you be interested, please contact me by mail: hagastian@gmail.com*
>
> *Thanks for your time!*
>
> *Best regards, Stian Haga"*

In this mail I have maximized the potential of receiving any reply by trying to suggest that any means of doing an interview is possible, such as quick and informal settings as instant messaging systems. An incentive for replying was also added by offering to provide them a copy of the final version of the thesis. However, a reply was never received. As an active approach in order to create an opportunity to conduct an interview, I asked the developers on the IRC channel for a chance to interview any experienced project member, but the channel remained unresponsive. However I did get the chance to ask some quick questions to jCore#1 during a public "Questions & Answer" session on the social network Reddit[3]. The questions and answers are attached in the appendix A, for reference.

4.2 Analyzing The Data

Analyzing all of the data gathered as described above, will probably be one of the most important and crucial tasks. This will create the very pillars of what I base my discussion and results on. As I will gather data from multiple sources, this will give me the advantage of being critical to whether or not the sources correspond with each other or if they differ in any way. As an example, does any interview create the same image that I will get from data mining in their mailing list archives? Does the communication via IRC differ from the way it is handled by mailing lists? Does these results comply with current research done in the same field?

As suggested by Oates (2006) I can take advantage of visual aids such as tables and diagrams to analyse data. With the repository of open source projects being publicly available, I could create some diagrams concerning the previous commit history and look for trends, as well as link it to my findings in the mail archives and

[3] `http://www.reddit.com`

IRC-chat. This will function much like a timeline as seen in research by Walsham and Sahay (1999, p.47), helping me in gaining an overview of the various phases in the case.

Concerning what will probably be a large amount of data from the projects mailing lists, Oates(2006) suggestion of initially dividing the data into three themes:

- Segments that bear no relation to your overall research purpose are not needed.
- Segments that provide general descriptive information that you will need in order to describe the research context for your readers.
- Segments that appear to be relevant to your research question(s).

As the data is divided into separate groups, I can continue to further divide the group of relevant data into appearing themes that I might discover, then later look for themes and interconnections between segments and categories Oates (2006). This has the advantage of helping me to eliminate data that is to no relevance, and will help me focus on the important part of the data, saving me much needed time.

The actual process of finding the themes was a combination of top-down and bottom-up process. That is to say after a few iterations of searching through the mailing lists and the history of the projects, any recurring topic was noted. After finding a list of posts describing the same thing I found that these topics described often were very specific problems relating to the project. During the meetings with my supervisor we discussed these findings and try to raise the level of detail to a more general level.

As an example of this, section 6.3 was originally discovered due to a large amount of posts on the jQuery mailing list concering optimalization of code. In the first iteration, this theme was just called ”Optimalization of code”. The project seemed to have conflicting views on how to balance their focus on whether to maintain and improve their current code, or develop new features for the next release. Later on the VLC project also seemed to be conflicted in this area, causing them to stop the maintenance of entire versions in order to move on to the next release. As the iterations continued, there was a gradual move towards the final theme that would be ”Innovation Vs. Maintenance”.

Chapter 5

Case

In the following two sections the two cases will be introduced and explained, through their governing system and history. First off is the VideoLAN Client (VLC), an open source project originating from a group of students in France. Secondly, the Open source project called jQuery will be discussed in the same procedure.

5.1 VideoLAN Media Player

Description:

> " VLC is a free and open source cross-platform multimedia player and framework, that plays most multimedias files as well as DVD, Audio CD, VCD, and various streaming protocols. It is simple to use, yet very powerful and extendable." `www.videolan.org`

Overview

VLC Statistics Overview

Project made public	1st of February 2001
Number of contributors	390
Commits per week AVG/MIN/MAX	71/61/8112
Average posts on the developer mailing list per month	646 since February 2001
Roughly estimated number of users	527 Million total downloads since December 2004
Primary communication channels	Mailing lists, Internet Relay Chat and Forums.

Table 5.1.1: *Various statistics for the VLC project.*

The development of VLC started in 1996 as a project by a group of students at École Centrale Paris. Their task was to enable TV watching on their computers. They also needed a reason to upgrade their network, so a bandwidth intensive

service was in order. After two years of developing the VideoLAN Server and the VideoLAN Client, they had their first successful streaming test in 1998. Later that year they decided to continue developing VLC, but from scratch with a modular and open source mindset. In 2001 after negotiating with the principal of the school, they managed to get the license changed to the GNU General Public License. The VLC project received a lot of attention and developers all over the world joined in. Only six months later a functional Windows port was released. Today it is one of the most popular media players, with hundres of millions of total downloads. One of the keys to the mass success VLC has experienced is the ability to play almost every type of multimedia files out of the box, with a very simple and accessible user interface, yet it has all the configurational options for a power user. It is now run by a non-profit organisation consisting of volunteers, called VideoLAN.

In recent years, the project has been stagnating, eventhough it originally had a very large user base and the huge amount of users. So despite their seemingly big success, the recent years have been riddled with calls for new developers in the news feed, and they have been forced to discontinue support of various branches of the application in order to move on.

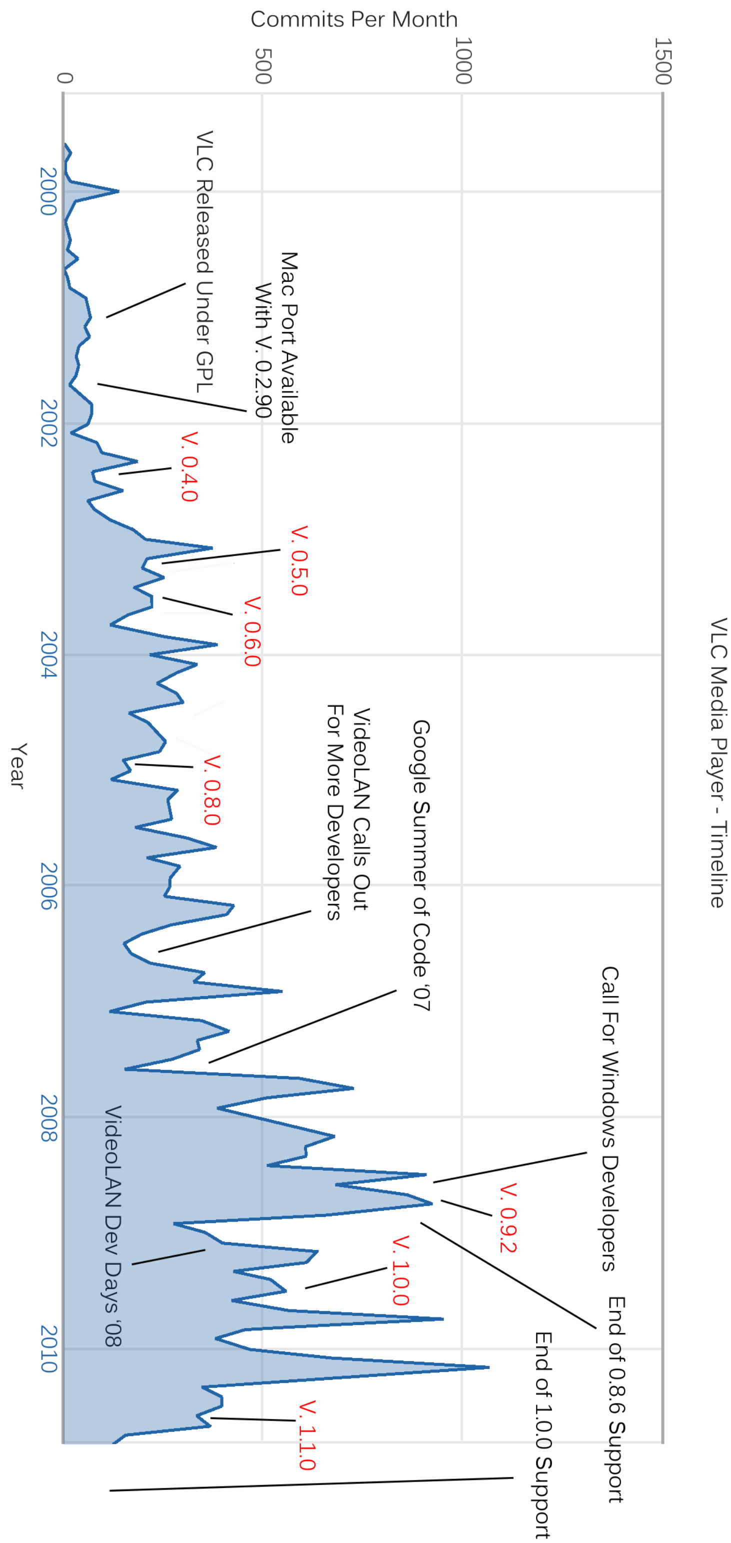
5.1.1 Organization And Model Of Development

Developers mainly communicate through a public mailing list and the forums on their web site. Quick questions and support is handled on the IRC-channel. There is no apparent hierarchy of developers, other than the three board members of the VideoLAN organization. The organization has a treasurer handling the economics of the organization, funding any events hosted by VLC (Dev Days) or attending other open source related events, such as FOSDEM[1].

The VideoLAN Client uses a few tools for project management. Trac is used for bugtracking and for version control they are using Git. These tools are well established and popular tools within the open source communities, Git being an open source project started by the famous creator of Linux; Linus Torvalds.

As seen in figure 5.1.1, the core of the project are very much responsible for a large piece of the commits done, in line with the findings done by Mockus et al. (2000). While no apparent hierarchy is evident, it could be safe to assume that these core developers are the ones that people go to when they have questions regarding the development of VLC.

[1] Free And Open Source Software Developers' European Meeting - `http://www.fosdem.org`.


VLC Media Player - Timeline
Commits Per Month
0
500
1000
1500
2000
2002
2004
2006
2008
2010
Year
VLC Released Under GPL
Mac Port Available
With V. 0.2.90
V. 0.4.0
V. 0.5.0
V. 0.6.0
V. 0.8.0
VideoLAN Calls Out
For More Developers
Google Summer of Code '07
Call For Windows Developers
V. 0.9.2
V. 1.0.0
VideoLAN Dev Days '08
End of 0.8.6 Support
End of 1.0.0 Support
V. 1.1.0

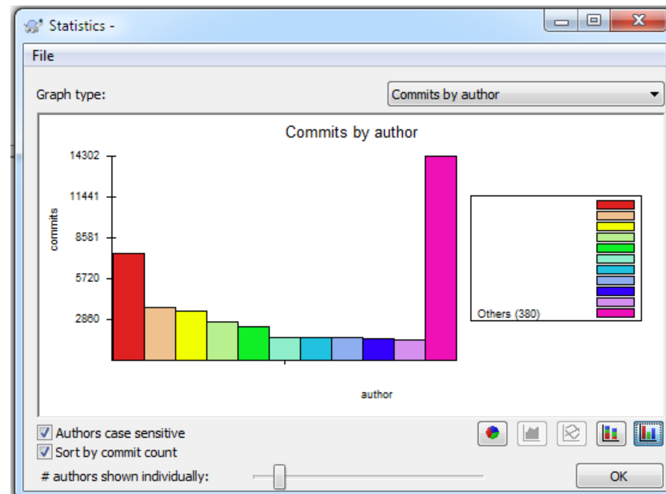


Figure 5.1.1: *VLC commit history, based on author. The top ten committers are shown individually. More than 390 different authors have contributed to the VLC project in total. The authors are based in 20 different countries world wide.*

5.1.2 History Of The VLC Project

During the 1990's and early 2000's the internet became available to everyone. Computing power and storage capacity were increasing at a near exponential rate, and still are, and the thought of storing multimedia such as audio and video on the computer were getting continually more feasible. New formats and codecs for storing these files were flourishing and steadily evolving. Users would have to download new codecs and media players, more often than not, and both installing and finding codecs could require some computer profiency, making it inaccessible for a typical user. What VLC offers today is the solution to all this. One multi-platform player that out of the box supports nearly all common file formats and codecs, and as a result it grew to become what many defines as the "de facto" media player.

After the VLC project acquired a GPL license in February 2001, the first major release was later in June the same year, with version 0.2.80. Binaries were made available for Debian x86, BeOS x86, and for the first time a Windows port was released with it. In October 2001 a new version was released with a working MacOS X port. By this time they had managed to add support for all current dominating operation systems and could focus on further developing features.

Within one year, version 0.4.1 was released in June 2002 and popular formats such as MP3, MPEG4, and DivX-encoded files were now supported. As seen in the timeline provided for the project, the project experienced a significant increase in

commit numbers in 2002 and 2003.

Audio Architecture Scrapped and Rewritten

Following the 0.3.0 release of VLC one of the core developers, vDev#1, had been having some trouble debugging a certain audio output module. After spending a lot of time on this, he was convinced the only reasonable way to solve this problem, was to rewrite the whole audio output architecture:

> "Dear friends,
>
> We're having more and more problems with our current audio output architecture, and I am fully convinced that the only option we have, is to annihilate it. Nuke it.
>
> [...]" -vDeveloper#1

Following this statement is a comprehensive and detailed proposal for a new architecture. At the end of the document he requests input from developers and also states that volunteers will be needed to help achieve this new undertaking. vDeveloper#1 is a frequent poster and well renowned member of the VLC project. Two of the developers, vDeveloper#2 and vDeveloper#3 raised a few concerns with the proposed architecture, and after a few technical discussions vDeveloper#1 revised his plan and updated it to reflect the changes that were reached during the discussion. This was then given a "go signal" by the two developers, and vDeveloper#2 wanted to help out with the development of this new architecture.

Scrapping the whole project is a drastic move. Scrapping the whole module and rebuilding it from scratch will take a lot of extra time. Ultimately though, there was no alternative. These kinds of events would be catastrophic in traditional software engineering where you need to deliver to your customer as soon as possible. Rewriting a whole module is clearly a sign of lack of formal planning, and possibly one that takes a lot of toll on the development should it happen often.

End of Life

VLC stopped supporting the 0.8.6 branch due to lack of time to update it in 2008. About three years later they had to discontinue their support for the 1.0.0 branch, shortly after the release of version 1.1.0, for the same reason. There was a lack of developers, and constant issues with the builds such as holes in security, demanded too much time. They chose to prioritize a new stable release 1.1.0 and continue developing new features for a 1.2.0 release.

These kinds of drastic changes, discontinuing support for entire branches of their product, could be a symptom of a stagnating project. The reasons for the project stagnation could be several, but most likely they are caused by the lack of developers as they have announced in years preceeding these events. It could also be the sheer size and complexity with lack of any formalized planning of the software and change of personell, causing software erosion as described in section 3.2 and in the previous section (section 5.1.2). Either way, it seems that the project is clearly suffering at this point of development.

Releasing ”The Luggage”

After discontinuing the 1.0.0 branch, they focused all their development on stabilizing the 1.1.0 release and working on new features in the 1.2.0 branch. The 1.1.0 branch was named ”The Luggage” and was released June 2010. Changes in the application are a few new features such as GPU decoding on a few platforms for High-Definition movies, but mostly the release offers improvements done by code maintenance; rewriting and removing modules for performance mainly.

This branch is the latest version as of today and they have yet to release version 1.2.0, although there have been a few maintenance releases on the 1.1.0 branch.

5.2 jQuery

Description:

> "jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is designed to change the way that you write JavaScript." `www.jquery.com`

5.2.1 Overview

jQuery Statistics Overview

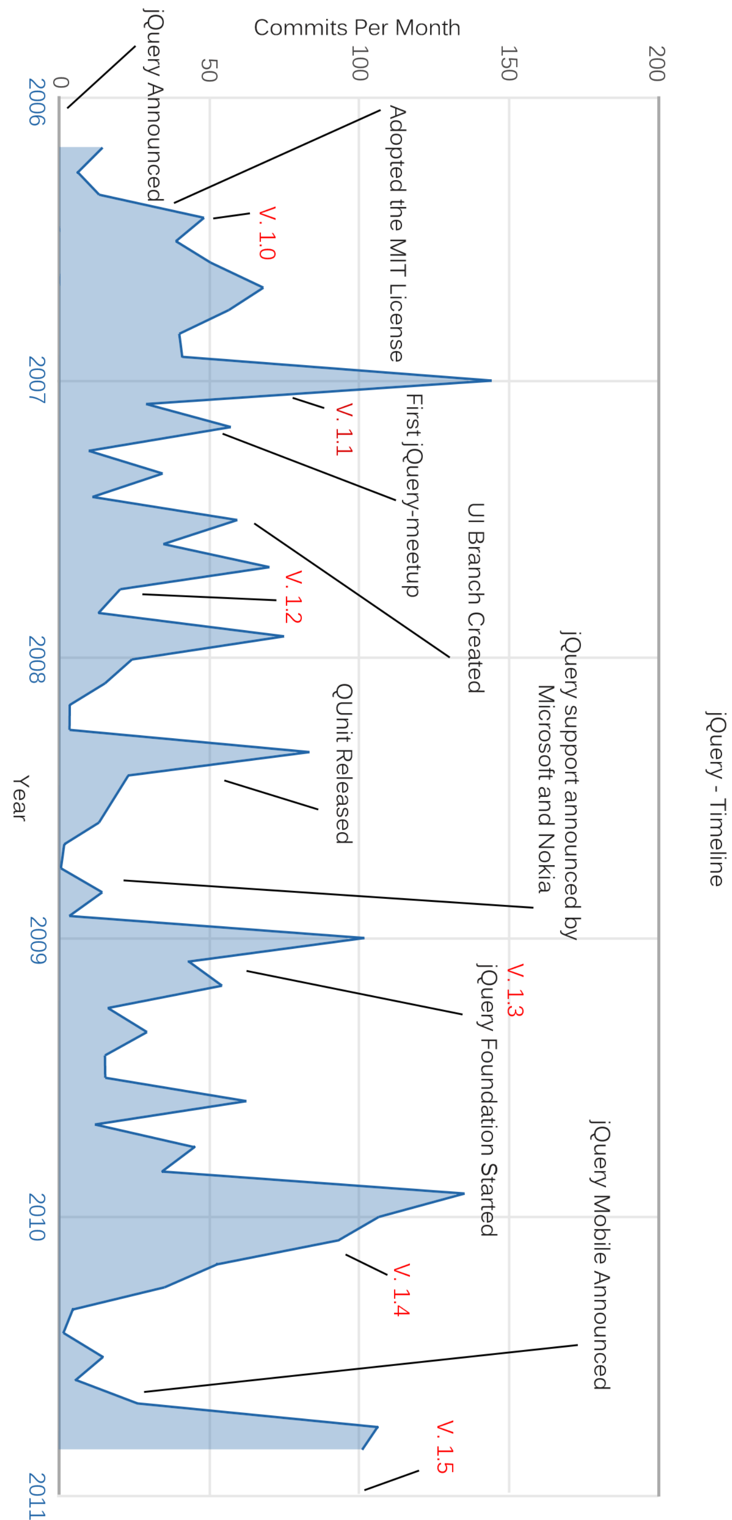
Project made public	14th of January 2006
Number of Contributors	77
Commits per week AVG/MIN/MAX	11/15/277
Average posts on the developer mailing list per month	309 since December 2006
Roughly estimated number of users	19 Million websites using jQuery
Primary communication channels	Mailing lists and Internet Relay Chat (IRC) and forums.

Table 5.2.1: *Various statistics for the jQuery project.*

jQuery has its roots in one person's need for a framework to bind Javascript functions to various HTML elements in the DOM (Domain Object Model). He tried a library called Behaviour, created by Ben Nolan(`http://bennolan.com/behaviour/`), but was not satisfied, claiming it was *"Too tedious and verbose for*

everyday use"(http://ejohn.org/blog/selectors-in-javascript/). In just a week after his jQuery demo at BarCampNYC, in January 2006, the interest in this project proved to be huge. jQuery made it to the front page of www.delicious.com and www.digg.com. Two of the largest web page popularity surveys on the net.

Roughly half a year later the first stable version of jQuery was released, jQuery 1.0.


jQuery - Timeline
Commits Per Month
0
50
100
150
200
2006
2007
2008
2009
2010
2011
Year
jQuery Announced
Adopted the MIT License
V. 1.0
V. 1.1
First jQuery-meetup
UI Branch Created
V. 1.2
jQuery support announced by Microsoft and Nokia
QUnit Released
V. 1.3
jQuery Foundation Started
V. 1.4
jQuery Mobile Announced
V. 1.5

5.2.2 Organization And Model Of Development

Developers were using mailing lists up until recently for communication. At the time being they are using a forum to keep track of discussions. Both the forum and mailing list archives are available to the public. They also have an IRC channel devoted to answer questions about JavaScript, jQuery syntax, problem solving and bugs.

While the project has a large community that contributes the project, it also has a hierarchical structure in the way that they have a core team running the project. The organizational structure of the jQuery Core Team is not as flat as many other OSD projects that have a developer group based on a distributed team of volunteers. The organization is divided into several sections. The development team consists of six members, at the time of writing. The other sections are:

- Developer Relations Team. Can be seen as the jQuery evangelists. Responsible for recruiting new jQuery users as well as making sure the wishes of the user population reaches the development teams. Has six members.
- jQuery User Interface Team. Responsible for maintaining the UI code, the UI community as well as developing new features for jQuery UI. Has four members.
- Infrastructure and Design Team. This is the team maintaining all the properties of `http://www.jquery.com` and `http://www.jqueryui.com`. Has two members.
- Operations Team. Responsible for everyday administration of the jQuery project and events. Has one member.
- Plugins Team. Responsible for maintaining all the official jQuery plugins. Has one member.

In addition to having the team divided into different sections, most of the members are formally assigned areas of responsibility and even some titles. Most team members are professional developers with years of relevant experience from relating fields.

There is a voting system is in place for deciding on a few project but not developer related issues, such as:

- Travel reimbursements.
- Accepting and removing members from the jQuery Project Team.
- Conference related costs.

- Equipment related to the jQuery project.

The votes are carried out on a public Google Discussion group. A vote is passed if a majority of the votes are in favor, after 48 hours has passed. In certain cases an absolute supermajority of two thirds is needed.

One of the things that sets jQuery apart from other open source ”version 1.0” projects, is the fact that they have got an own company dedicated to enterprise jQuery training, support and consulting services. This company was created by a few of the core members in October 2009 and it is named appendTo.

5.2.3 History Of The jQuery Project

jQuery was developed by a single person during the last half of 2005. It was only vaguely suggested that a JavaScript library was under development. This library was announced to use CSS selectors and easy to comprehend syntax, to create something different, quicker and more accessible than the current major JavaScript libraries. As jQuery was made public in the beginning of 2006, the activity levels on the newly created mailing list quickly surged from 40 posts in January to a staggering 3094 posts in July the same year. The initial interest for his project was huge, and it just kept on growing. With a growth like this, a project could quickly become a messy affair if it does not have any way to track features and bugs that are to be handled, as well as an overview of the projects progress towards the next milestone.

Release 1.0 and need to handle project growth

In the first few months leading up to the first stable release of jQuery, there were a few recurring topics on the mailing list. Quite a few about the practical use of the library, how to take use of its functions and power. They were basically general support questions. The mailing list was not yet split into different segments, so all questions regarding development, bugs and support were huddled together in the same mailing list. This was not really a problem yet, as the number of posts each day were not overwhelming. During the time that lead up to the 1.0 release of jQuery, a number of developers were asking for a roadmap. There was no official list of bugs or features that were to be fixed within the 1.0 release, and some developers requested a sort of project management system. As the number of developers rose swiftly, the need for project management grew proportionately. As an example of one of these roadmap requests, one of the developers raised

the concern as release 1.0 was getting closer and the number of daily mails were skyrocketing:

> "I'm seeing a lot of e-mails flying around about a lot of subjects on this list. But right now, I think what jQuery _desperately_ needs is some coordination and a roadmap. Don't get me wrong. I still love jquery. But Right now, it sounds like development is in a complete freefall. If jQuery is to get close to the popularity of Rails coattail-rider Prototype, it needs to flesh out it's development structure and start working on a feature freeze and a roadmap towards a 1.0 release asap. I know what that means - it sounds f-ing boring. But I think (and I'm sure that others will agree) that jQuery needs, first and foremost, stability. A core 'stable' branch. And organized bug tracking and management. What's the likelyhood of this happening any time soon?" **-jDeveloper#1**

It did not take long before a reply was sent, saying that it was already taken care of:

> " I've already beat you to it. [...] I plan on making an official announcement about this on the blog, very soon. I agree with you, though, what jQuery needs, more than anything, is a solid, usable, base to work off of for future releases. " **-jCore#1**

In this case the jCore#1 member had been quick to identify a problem and adopt a solution to the quickly growing developer base. Three days earlier the project had been transfered over to the popular project management tool known as Trac. The transition to Trac had only been mentioned on the mailing list and he later put up a post on the official jQuery blog. The Trac project management tool features a built in Wiki site, ticket system, roadmap with milestones and SVN interface. With an up to date ticket system and Wiki site, developers wanting to assist in the jQuery project could easily join in without having to explicitly ask any veterans for tasks to do.

Extensibility by Plugins

As many open source projects are often tailored to the needs of a person or a small group of people. Should the project just so happen to catch on with the public, there is a large possibility of people wanting to add functionality or even rewrite how the program works. Through the jQuery Core framework - the main project and origin of jQuery - you get your basic features for everyday use when creating website functionality with JavaScript. These include, but are not limited to:

- Handling events.
- Traversing elements.
- Ajax capabilities.
- Various visual effects.
  - Fading
  - Hide/Show item
  - Sliding
- Getting and setting CSS properties and DOM attributes of elements.
- Plugins.
- Selectors.

Karl Fogel defines forkability as *"The indispensable ingredient that binds developers together on a free software project, and makes them willing to compromise when necessary, [...].*"(Fogel, 2005, p. 88). The decision of having a basic Core project for jQuery, lies behind the fact that it sets out to be a lightweight and quick framework, as opposed to its current competitors. In theory you could implement all sorts of fancy features in one bulk package, but seeing as one of the key points of preserving usability for websites is response time, adding a huge framework like this would make the site sluggish. To increase the adaptability to each developers needs, the jQueryCore can therefore be extended by plugins. This enables developers to only include the features you require. By writing a plugin and releasing it to the community developers can also get feedback on their work, as well as helping other developers by either solving a problem or helping them to avoid "re-inventing the wheel". One of jQuerys advantages over the competitors is exactly this quality. Plugins can provide everything from a fancy visual image gallery to subtle automatic scrolling on a page.

The first third-party plugin was released only eleven days following the jQuery announcement in January 2006. This particular plugin loads a remote JSON (JavaScript Object Notation) file and makes it possible to manipulate it further. While this may not serve a lot of purpose, it is a testament to how quickly people began to develop plugins as they saw fit. Initially information about plugins were stored on a separate wiki page. In June 2007 however, almost a year after the 1.0 release, the Web Team released a plugin repository site. Greatly enabling users to browse the many plugins available.

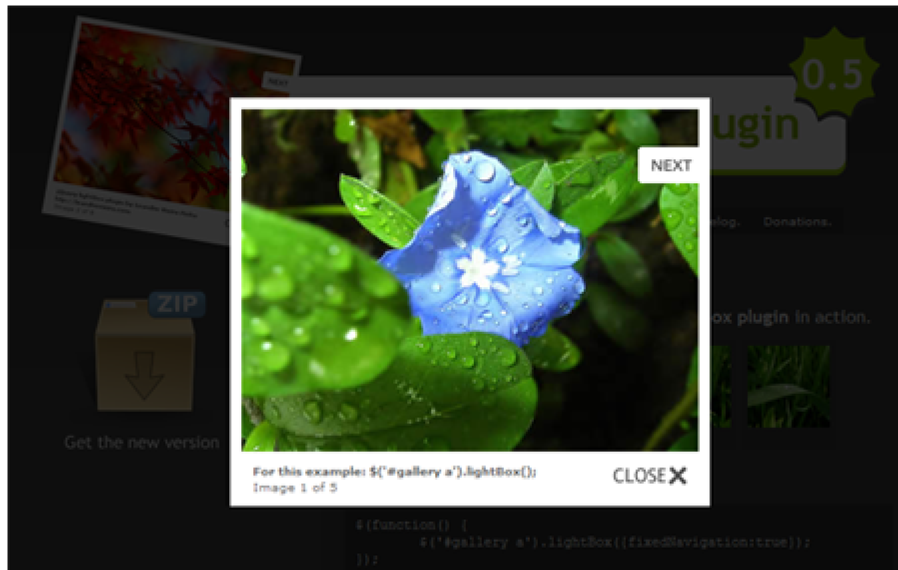


Figure 5.2.1: *One of the most popular jQuery Plugins is called Lightbox. It has the ability to display images as you click their thumbnails in a gallery. Lightbox will create an overlay over the website and display the picture, and even resize the image to fit the browser window.*

By now, plugins are of great value and importance to jQuery users and the jQuery Project Team. As the jQuery framework grew and its reliability, stability and features increased, organizations and even commercial companies are using time and money on developing and maintaining plugins, working in cooperation with the jQuery Team. Some of the most popular and largest plugins are officially supported by the jQuery Team. October 4th in 2010 it was announced that Microsoft developed plugins were to be officially supported by the jQuery Team:

> "Today, we're very happy to announce that the following Microsoft-contributed plugins - the jQuery Templates plugin, the jQuery Data Link plugin, and the jQuery Globalization plugin - have been accepted as officially supported plugins of the jQuery project. As supported plugins, the jQuery community can feel confident that the plugins will continue to be enhanced and compatible with future versions of the jQuery and jQuery UI libraries."

(`http://blog.jquery.com/2008/09/28/jquery-microsoft-nokia/`)

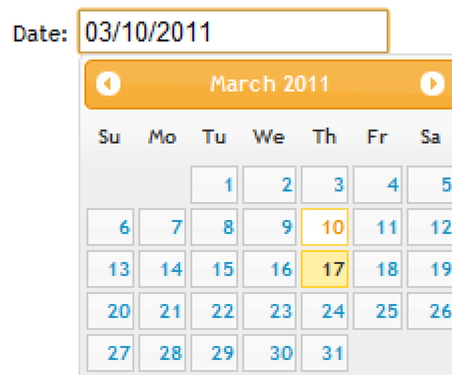


Figure 5.2.2: *An interactive date picker. One of many widgets available in the jQuery UI Package.*

New jQuery Branches

From the beginning of the jQuery release, a few large libraries concentrating on enhancing the visual appeal of the websites gained a lot of popularity. Most notably was the jQuery based library called Interface. It contained Drag-and-Drop functionality, animation and widgets. Much of the items one might associate with the web 2.0 experience. In June 2007 it was announced that one of the jQuery team members, coincidentally the creator of Interface, had been secretly working on a library called jQuery User Interface (UI). This had been built from the ground up and with extensibility and performance in mind. The jQuery UI library was released only two months later on September 17th, with the help of a dozen team members. The jQuery UI description:

> "jQuery UI provides abstractions for low-level interaction and animation, advanced effects and high-level, themeable widgets, built on top of the jQuery JavaScript Library, that you can use to build highly interactive web applications." (`http://www.jqueryui.com/`)

While most of the functionality of this library could already be gotten through user created plugins, this library was thoroughly coded with strict standards, well documented, contained themes and demo standards. As it became an official branch of the jQuery project, it also entitles regular updates and support, a huge boon for all web developers.

As projects grow, so does the need for unit testing, in order to maintain modifiability. Making sure the code quality does not wither as you create updates, is very important. In order to facilitate growth and code quality, a new unit testing

framework was developed in the jQuery project. The framework is called QUnit and it is a complete test suite for not only jQuery JavaScript, but also all other JavaScript, including server-side code. This is closely related to the test driven development discussed in section 3.1.

Optimizing The Performance And Adding New Features

As discussed earlier the performance of jQuery has always been a central motivation for the project. Optimizing code is important for jQuery and new releases continually emphasized new speed improvements. As an example, in the 1.1.3 release the selector speed is reportedly 800% faster, 1.2.6 release quickened event handling by 103%, 1.4 release had tripled the performance of a few functions (see figure 5.2.3).

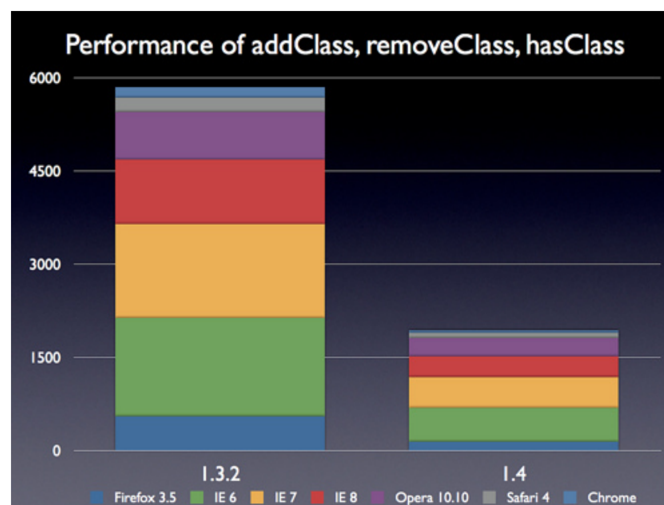


Figure 5.2.3: *Some of the improvements in the new version compared to the old release.*

Part III

Analysis

Chapter 6

Discussion

The development model of open source software is a thorougly researched affair. The innovation model of this type of community is an intriguing one, and it has been proven to be a successful one through various products such as the well investigated Apache Web Server, the many forks of Linux and Mozilla Firefox. Although there are many success stories, the sheer number of total projects compared to successful ones leads us to believe that the open source development should be viewed as a more fragile process (Monteiro et al., 2004).

Innovation and maintenance are two strong points of character within OSS, known for delivering new ideas, game changing software, being of high quality and ever evolving without a finishing state. It is strange how a network of uncoordinated are able to provide both of these characteristics when there is such a split between development and maintenance in traditional software development (Bennett and Rajlich, 2000, Lientz et al., 1978).

As the OSS projects grows mature, one of the fragilities of the process is that there might be a lack of interest and loss of contributors. Although there are many motivational factors for joining in on these projects (Roberts et al., 2006, Lakhani and Wolf, 2003), they focus on the initial effort to joining. There is less research available on how to create and maintain a sustainable growth within the projects.

There is research available on the different types of licensing within OSS (Lerner and Tirole, 2005, Lauren, 2004) and even a set of patterns for selecting license has been proposed (Kaminski and Perry, 2007). Licensing is generally viewed as a political endeavour in OSS with its rich history.

The fragility of OSS might be a sign of limitations within the process. Through a

few recurring themes in the case material we will look at some issues behind the possible reasons behind failure in OSS development.

6.1 Licenses As Insentives

As we have seen in the discussion of licensing in section 2.2, the two most important traits of open source licenses are whether they permit the cooperation with proprietary software or not. The reasons behind having these very different licenses available lies in the history of OSS itself, and there are strong ideologic views associated with them. In the following sections some events in the projects lifetime will be portrayed and then viewed in the light of the litterature review and discussed.

6.1.1 GPL vs. MIT - A Holy War

Choosing the appropriate license for the work is not a trivial task for an open source project. There are currently a lot of options available. Perhaps the most popular among them are:

- GNU General Public License (GPL)
- GNU Lesser General Public License (LGPL)
- Apache License
- Mozilla Public License (MPL)
- Creative Commons Share Alike License
- MIT License (X License)
- BSD License

The licenses here have been discussed in section 2.2, but to recap, the MIT, Apache and BSD Licenses are permissive academic style licenses, allowing cooperation with proprietary software. The MPL is designed for converting proprietary software to open source, thus it is also compatible working with proprietary software. The GPL and Creative Commons Share Alike License carry strong copyleft and are not compatible with proprietary software, as well as not being compatible with many permissive style licenses. LGPL carry a weak copyleft and does not apply the restrictions to other software coupled with the work.

The choice of which license to choose in open source software is often a trivialized one, not being considered a big importance when starting a project. Fogel (2005, p.231) notes that *”The license you select probably won't have a major impact on the adoption of your project, as long as the license is open source.“*, but he also takes care to imply that the licensing scheme should be compatible with the goals

of the project. As seen in section 2.2 where open source licensing was discussed, the impact any license carry could mean the difference between commercial success and a stagnated project.

The jQuery project was initially released under a Creative Commons Share Alike license. While the CC license is not written for use with software in general, it has the advantage of being a well constructed and thoroughly written license, upon which one can tailor a license that fits the work at hand. The Share Alike point of the license works in the way that if you were to distribute a derivative work, it has to carry the same license that the original work is governed by, or at least be compatible with it. Much in the same way that copyleft works.

In April 2006, about half a year after the project was started, a developer at a web design firm is working on an internal CMS project for his company. The developer is considering using the jQuery framework for the client side interaction, but is unsure whether or not their CMS will have to be distributed openly under the same Share Alike license. A long discussion is created on this topic, and Core#1 decides to change the license to a more flexible and less stipulating one. They finally ended up using the MIT license. He explains that his only requirement when choosing the CC Share Alike license, was that there would be a link back to the source.

> " As far as your company using jQuery, you're completely in the clear. I only picked the CC Share Alike license so that future versions of the code would have a link pointing back to jQuery.com. In retrospect, this license may have been overkill for something so simple."**-jCore#1**

The MIT license is not as viral as the Share Alike point of the Creative Commons license. In the open source community it is generally viewed as the most flexible and open license of them all. It allows for jQuery to be used in all sorts of derivative work, even proprietary commercial software. The jQuery part of the derivative work still has to be under the MIT License, and it will link back to the original source of the material.

The issue of license was later brought up a new, as the developers at Drupal - an open source CMS - was interested in using jQuery for their work. However, their policy did not allow them to use sources that were not compatible with GPL. And so they wished to see if there was any possibility for jQuery to adapt both MIT and the GPL license. The Drupal project was already a well established open source project at the time with a lot of users (350,000 as of 2008) and contributors.

> "I note in this thread http://jquery.com/discuss/2006-April/000623/ that jQuery was relicensed last month from CC to "MIT". The "MIT"

> license is very permissive and so presents few barriers. However, several developers with the Drupal content management system http://drupal.org are very impressed with jQuery and interested in the possibility of using it, and our policy doesn't allow code with non-GPL licenses in our packages (see http://drupal.org/node/66113). Copywrite owners are free to license software under as many licenses as they wish. Would you consider making jQuery available under the GPL *as well as* the "MIT" license? Thanks, Drupal#1" **-Drupal#1**

This proposal sparked a long thread of discussion among the members of the jQuery project. First off, Core#1 raised a few questions concerning this move to a dual license with GPL and MIT:

> "[...] 1) If someone submits a change to jQuery through the GPL version, I won't be able to integrate that change back into the public version, since it's under a dual license. I would have to go through the hassle of asking the author first if it's ok to make that addition.
>
> If I could get some sort of assurance that any jQuery changes would go through here first, I would feel much better about the whole ordeal.
>
> 2) Someone could fork jQuery based upon the GPL'd version, which seems a whole weird area of drama. If anyone has any input on this, let me know.
>
> If I could get your feedback on this matter, I'd really appreciate it. Licensing (and the GPL) is always a bender..." **-jCore#1**

There are a couple of areas of interest with this post. Firstly, it shows that the possibility of a fork is very much a concern for the project. Much in conformity with the litterature reviewed in section 2.3.2, this shows that the possibility of a fork is a great force in choosing the licesense. On the other hand, the GPL license itself does not increase the risk of a fork, more than any other open source license. This is also pointed out in the reply by Drupal#1.

The post illustrates that the aspect of open source licensing is a difficult one for developers, by calling it a *a bender*. This could be a possible limitation to OSS as they generally do not have an own section of lawyers or people trained in law. Many of the big software companies has got some kind of law support, as illustrated in section 2.1.2, where the Netscape Communications had the resources to make their law department craft a new open source license from the scratch, in order to best suit their kind of project.

The reply to this by Drupal#1, reassured the jQuery developers that code to the core would be returned to the main project. It is evident that the jQuery project is very interested in making this work, as it would increase the number of jQuery users by quite a bit:

> "Although, considering the number of current jQuery users who also love and use Drupal, I definitely think that something will work out between us. " **-jCore#1**

As the discussion seemed to go towards a conclusion there was a sudden involvement of more developers, pleading jCore#1 to not adopt the GPL or LGPL license instead of keeping the MIT license.

> "jCore#1.
>
> I beg of you do not go to the LGPL.
>
> It offers no benefit over MIT beyond "Drupal folks would use it" and, as stated rather bluntly by the previous posters, their decision not to use MIT-licensed code is fairly ridiculous.
>
> The GPL and LGPL are overly complex licenses and completely useless. " **-jDeveloper#3**

> "[...] non-copyleft means 'non stallman infused sandal wearing scary dude politics".
>
> MIT is the most liberal license out there, going LGPL or GPL would only ADD restrictions not take them away." **-jDeveloper#1**

These are clearly very strong opinions and while rational, there are also a clear sense of ideology behind them. Especially the comment made by jDeveloper#1, refering to the GPL/Copyleft movement that Richard Stallman started, as *"stallman infused sandal wearing scary dude politics"*. Now, this is clearly an exaggeration with humoristic stereotyping, but it shows that the split in the OSS community as discussed in section 2, values created in the late 1980s and through events in the 1990s, is still very much present in todays OSS communities.

The conversation is further expanded as more developers utter their opinion about the licenses. It turns out to be a heated discussion with a lot of personal opinion, closely relating a "holy war" as described by Fogel (2005), where the exact topic that is being discussed in the jQuery mailing list is one of the top reasons for starting a holy war. It is often recognized by the disability to see the points made by "the other side", where the dividing line in this discussion is whether

or not you support the GPL license. This is not just about a choice of license, but a question of the freedom of the source code, which is linked to an ideologic and political stance. Some people believe that the viral, or reciprocal, nature of the GPL, prohibiting the involvement of proprietary software is just the opposite of freedom. They believe that the GPL license works against it own principles. The other side of the discussion is on the other hand believes that this denial of ”hijacking” from proprietary software is the essence of free software.

These views are easily recognized in the discussion:

> “[...]Do you GPL fanatics even know anything about your own license?[...]” **-jDeveloper#4**

> “[...]As to my opinion why the GPL is not useless; it’s simply because I do not want to see mine or anyones freely contributed coding efforts subverted by anyone else who is not prepared to freely give back any improvements to the folks who contributed the code in the first place. The GPL is the best guarantee that code stays free and does not get sucked off into closed and proprietary projects.[...]” **-jDeveloper#6**

> “The best guarantee that code stays free is releasing free code yourself. GPL is not free. It’s one of the least ”free” license out there.”**- jDeveloper#3**

While this was seemingly spiraling out of control, with no conclusion to be made, jCore#1 tried to provide the thread with a summary to create a progress and conclusion:

> Just to reiterate some points: - This would be a dual license situation. The MIT (Expat) license will still be the default. In fact, I’ll probably end up tucking the GPL’d code away in some hidey-hole so that people don’t use it be default. - I will not provide the code under the GPL unless the Drupal people can give me assurances that they will not make changes to the GPL’d jQuery, and instead commit it straight back to the original code base. So far, it’s looking like this will be the case. - It is my interpretation that the jQuery code that could be included with Drupal would be quite static, just there as a reference. Any plugins that people develop would be completely separate and under their own licensing scheme(s). IMO, I don’t have a problem providing a secret, static, dual licensed copy of jQuery, if it means that

> jQuery's user base will quadruple [ed. random number] overnight. Let me know if you have any issues with these points, as I'd like to try and clear things up ASAP.**- jCore#1**

This way of clearing out things are very much similar to the benevolent dictator model described in section 2.3.2. By summarizing and concluding, the "holy war" is brought to an end. The influence that jCore#1 has over the way that the project is run and descision are made, is big, but as discussed in section 2.3.2, only interferes when required. From this discussion it was decided that the jQuery core would be available both under the MIT license as before, but also under the GPL license if it were required. Dual licensing is not very common within OSS communities, most likely due to the possibility of increased complexity in licensing, but in this case it was an effective solution which reaped the rewards from both sides. The choice to relicense to GPL was never really a choice as it would limit the involvement of commercial companies, as explained in section 2.2. Still it is evident that the potential gain of users is a big factor in this project, and opens up to the kind of opportunistic behaviour as seen in the result to implement dual licensing in order to satisfy the involvement of one potential project.

6.1.2 The Impact of Licenses

As seen from the previous section, where the process in deciding what licesense to go with in the jQuery project was portrayed, there are large forces of ideologi and political ideas that stand behind the reasoning for selecting a license. Generally divided by GPL and non-GPL followers, described as two opposites with widely different values, it is easy to see why history is a big part of open source development.

Throughout its lifetime the jQuery project has received support from various commercial companies through contributions related to improving it for their own good. One example of this:

> "Cheers jQueriers, I'm a new member here. Our company is thinking about standardizing on jQuery as our core JS platform and I've been doing some extensive analysis on the library to make sure it meets our needs. I have to say I'm really really impressed with the performance and ease of use - you guys have done really stellar work! One of our JS requirements is to encourage use of OO code as we create our dynamic portal elements. After using jQuery for a while I found that using bind to signal methods on object instances was kinda cumbersome,

> so I devised a way to deal with this a little more nicely. [...] " - **jDeveloper#7**

In addition to this, both Microsoft and Nokia now officially supports jQuery. Microsoft distributes jQuery through their developing platform Visual Studio, and Nokia includes jQuery in their WebKit for their phone web browser. These kinds of cooperative efforts with commercial companies would certainly not be available should the project only have the GPL license. The way Microsoft and Nokia includes jQuery in their proprietary software would not be supported by the GPL license as the software would be viewed as one single work, and the GPL is not compatible with proprietary licenses. Cooperations like these can benefit the project both by greatly increasing the number of contributors and as well any money payed to the open source would enable them to create a more robust community by hosting events and also hire professionals to increase profitability by assisting in i.e. planning, development or product management.

As seen from discussions on VLC's internet relay chat, there have been efforts of trying to convert the project to LGPL, a less restrictive license than the GPL that is compatible with proprietary software:

> [...]
> [22:15] <vDeveloper#7> vDeveloper#8: we did try going LGPL
> [22:15] <vDeveloper#7> vDeveloper#9: cool piece of trivia
> [22:15] <vDeveloper#8> vDeveloper#7 where are the difficulties?
> [22:15] <vDeveloper#9> you need to trace the origin of everything
> [22:15] <vDeveloper#9> and get permission from everyone
> [22:16] <vDeveloper#7> we got agreement from everyone but like 3 contributors
> [22:16] <vDeveloper#9> getting permission from 95% of devs, covering 99% of code is easy
> [22:16] <vDeveloper#9> but the rest 1%
> [22:16] <vDeveloper#9> is approximately impossible
> [...]

This discussion illustrates the difficulties of trying to relicense a project after its initial starting phase. OSS projects can potentially attract a lot of contributors and over the years the project will quickly grow complex without any measures taken to prevent it. In order to relicense an OSS under the GPL license, the consent of all contributors are needed, so it is more or less an impossible task for a project that is mature and especially with the size of VLC. Had it been one of the permissive licenses, they could have created a fork on another more restrictive license, or adopted a dual license as the jQuery project did, but it is difficult to

go the other way around. Not being able to relicense a project can be a major fragility of the project development, as seen from the VLC example.

The two cases presented in the thesis are consistent with the findings of Lerner and Tirole (2005). Projects with highly restrictive licenses results in less contributions and vice versa. Although the jQuery is dual licensed under the MIT and GNU license, as they are both available, the least restrictive license is the dominating one. It might seem that the choice of license profoundly affects the lifespan of the project, and even so the focus on attaining the correct license is trivialized in many projects. The reason behind this might be because most open source projects are motivated by ”the joy of hacking” as discussed in section 2.3.1, and not commercial success. Another reason for emphasizing the choice of license is the disability to relicense later on in the project. There seems to be a difficulty in going from a highly restrictive license to a permissive license, once the project has matured. Through these examples it is evident that the choice of license can severely affect the outcome of the project should it become a success. By choosing the GPL license you will effectively lock out any cooperation and, promotion through use, with commercial companies.

6.2 Sustainability Through Alliances

In section 2 the development and history of open source Software was discussed through litterature review. The emergence of the OSS 2.0 era, shifting open source Software from a projects driven by ”scratching an itch” to commercially involved projects with complex structure and business goals. The changes during the past decade might suggest that projects are now safe from going stale or forking, with major companies being their pillars of support and guiding light. Another possible issue is the inability for a project to adapt to the newly added complexity incorporated as companies invest both time and money, and the overhead for running the project increases. In this chapter a few issues found in the two case studies found in part two, will be analyzed and discussed.

Both of the cases studied are considered mature open source projects. They have got a large community with more than 100 people contributing on a worldwide basis. The projects are beyond the initial release of a stable version 1.0 and they are both run by non-profit organizations, namely the jQuery Foundation for jQuery and the VideoLAN Organization who run the development of the VLC Media Player. However, there is a significant difference in age of the two projects; the VLC Media Player began its development in 1999, and jQuery six years later in 2005. VLC was not released to the public until they managed to move to the GNU GPL license in 2001, but it is still nearly twice the age of jQuery.

The licenses of the VLC Media Player and jQuery projects are very different, they are under the GPL and a dual license MIT/GPL respectively. As discussed in chapter one, the GPL is a controversial license with strong copyleft, and will not allow any involvement of proprietary third-party software, which leaves out potential cooperation with commercial companies. The MIT License is on the other side of the spectrum being a permissive academic style license, allowing the involvement of third-party software. The hierarchy of jQuery’s governing system is a more complex and vertical one, than the flat hierarchy shown in the VLC Media Player. jQuery corresponds better in this regard, to the OSS 2.0 characteristics discussed in section 2.1.2.

6.2.1 Forging Alliances

jQuery did not always have the MIT/GPL dual license, though. As it started out, it was only licensed under the GPL, but migrated to a dual license in a matter of months in order to facilitate the involvement of commercial software (see section 6.1.1). This kind of opportunistic behaviour involving compromises, is

not only seen once throughout the history of these cases. In the following extract from the mailing list in 2007, Core#1 informs the community of a recent offer by WordPress.

> Hi Everyone - In case you haven't heard already, Wordpress 2.2 is going to include jQuery and Interface. This is in addition to Prototype and Scriptaculous. They're including both libraries in order to support Theme developers who are looking for one, or the other. However, the Wordpress admin area is another matter, entirely. It's currently written using Prototype, but WordPress#1 [The creator of WordPress] would much rather be using jQuery. He's personally asked me for our help, in any way that we can provide it. I'd like to form a temporary strike team that would be responsible for helping Wordpress move over to using jQuery. The majority of their code doesn't appear to be "that bad", and we could wrap it up really quickly. Reply to this message and let me know if you're interested, then we can move over to a more-appropriate venue for discussing this matter.**jCore#1**

Within the end of the day, the thread had five volunteers raising their hands in interest for the project. As the communications and agreements to this was done by other means than the mailing list, an inquiry to the status of this cooperational effort was posted later that week:

> Yea what happened with that 'strike-team' jCore#1 mentioned? I'd love to see some improvements on the WordPress Admin area... **jDeveloper#2**

> I'm in contact with Wordpress#1 - I'm trying to figure out where we should be discussing this (I figure that it'll be in some forum, or mailing list, on the Wordpress site). Once we figure that out, we'll be moving over there. **jCore#1**

Wordpress is an open source project delivering a framework for blogging as well as hosting blogs for free. They are according to a survey in 2010 by Jon Sobel (2010), the most popular blog hosting service on the net. This is not just an opportunity to gain publicity, but also an opportunity to gain leverage on their biggest competitor, namely Prototype. WordPress has a huge audience currently serving more than 25 million people, so this is no trivial event. Gaining leverage over competitors, reducing their market share and even the possibility of eliminating them in the process**, is similar to the kind of opportunities that commercial companies seek out in the open source community, as discussed in section 2.1.2.***

> "[...] IMO, I don't have a problem providing a secret, static, dual li-

> censed copy of jQuery, if it means that jQuery's user base will quadruple [ed. random number] overnight. [...]" **-jCore#1**

The opportunistic behaviour seen in the above discussion, as in making compromises in order to allow alliances to form, seem to be highly prioritized in jQuery. It has enabled the project to form numerous alliances in similar fashion to the cooperation with WordPress. Alliances has even been formed with Microsoft and Nokia (see section 6.1.2), two huge commercial software companies, and are benefitting from a huge increase in number of users.

6.2.2 A Downward Spiral

The VLC project are at a disadvantage when it comes to forming alliances. Firstly, it would have to absorb any potential project of cooperation, due to the reciprocal nature of the GPL license. Secondly, it has limited use-value for developers, as it is a media player aimed at end-users. End-users of media players can be assumed to not be consisting of a large percentage of software developers, as opposed to jQuery that is aimed directly at the software developers creating web sites. The issue of VLC having the GPL license, has been discussed in the IRC channel (section 6.1.2), and later on they did some brainstorming on how to attract more developers given their current situation:

> [...]
> [22:21] <vDeveloper#8> if you have money you can do what you wan't
> [22:22] <vDeveloper#8> old but same problem.
> [22:22] <vDeveloper#8> videolan needs money
> [22:28] <vDeveloper#9> not necessarily
> [22:29] <vDeveloper#9> vlc can continue as a project of hobbyists having fun
> [22:29] <vDeveloper#9> and then perhaps die a slow death once the key devs switch hobbies and fail to be replaced
> [22:30] <vDeveloper#9> money is needed for some stuff. Depends where you want to take it.
> [22:35] <vDeveloper#8> yeah so a good marketing would be grate, so new people come to "us" (i put me as part of the actual groupa
> [22:35] <vDeveloper#8> no? even firefox/gnome/... are making special activitys to find new devs
> [22:36] <vDeveloper#7> do you suggest we should organize a Code of Duty?
> [22:37] <vDeveloper#8> code of duty?
> [22:40] <vDeveloper#7> vDeveloper#8: some kind of marketing pro-

gramming contest
[22:41] <vDeveloper#8> vDeveloper#7: yeah and a new marketing position on fosdem and things like this ;)
[22:41] <vDeveloper#7> what should we offer to the winner?
[22:42] <vDeveloper#8> i don't know :D
[22:44] <vDeveloper#9> hmm, nobody is going to join a "needy" project
[22:45] <vDeveloper#10> nobody is going to work for free unless they get something in return
[22:46] <vDeveloper#10> for many, that's as simple as a media player that does precisely the thing they need
[22:46] <vDeveloper#8> yeah yeah we will find something at the time :)
[22:46] <vDeveloper#10> combine the specific needs of enough devs and you get a fairly capable player
[...]

Without the ability to create any alliance, due to license restrictions, other means of sustaining a working community could be events as described in the case. Attending conferences and even hosting them can be a valuable way to be seen and attract developers, unfortunately these events require money. The issue of money is clearly stated by vDeveloper#8, as well as vDeveloper#9 saying that the alternative to this is basically for the project to "die a slow death".

In order to create a sustainable community in a mature open source project, the intrinsic motivational factor described as the "joy of hacking" may not be enough in the long run, even though it is noted as one of the most important factor in joining projects (Ghosh, 1998). As vDeveloper#9 notes, *"nobody is going to join a "needy" project"*. A project becomes "needy" when it remains understaffed for a longer period of time and the maintenance work is not being caught up with. Understaffing has characterized maintenance in software engineering since the late 1970s, as the survey by Lientz et al. (1978) shows. By being able to forge alliances with other open source projects and/or commercial companies, manpower is increased and the maintenance part of the development is handled by "outsourcing" the source.

Several fragilities of open source development and ways of countering them, have been identified through the previous examples. Finding the right kind of people, and enough of them, for the project is a difficult task that requires acknowledgement early on. Alliances can serve as a powerful way of increasing the chances of acquiring developers and thereby reducing the fragility of the project, but it can be hampered through the use of reciprocal licensing. After initially becom-

ing a ”needy” project, there seems to be a downward spiral. An increased need of software maintenance through lack of formalized architecture and processes of development, leads to a ”needy” piece of software which does not function as an inviting enviroment for developers. It is important for developers of open source software to recognize this fragility as they start out, and then plan to cope with scenarios that could reinforce fragility. Creating a project that is able to forge alliances through the acquisition of a non-reciprocal licenses and increased formalisation of requirement engineering could be two examples of reducing fragility of the development process.

6.3 Balancing Innovation And Maintenance

Projects are limited to the developers they manage to attract, and even though there could be a massive amount of contributors to a open source project, few people are working full time developing it. As discussed in section 5.1 about VLC, June 18th 2010 they had to announce the "End of life" for the 1.0.x branch. They could no longer afford spending valuable developer time on supporting the newest release that had a few security holes and bugs. Their developers would focus on releasing a new stable 1.1 version as well as developing new features for the 1.2 version. The official statement:

> Hello, The official release of VLC media player and LibVLC version 1.1.0 is coming to a close. The badly stretched VLC development team is not currently able to maintain more than two development branches at a time. The team has been focusing on the VLC 1.2 future series and the VLC 1.1 stable series.
>
> As a consequence, source code for VLC 1.0 is not officially unmaintained[1] anymore. There will be no further security or major bug fixes. The last version was 1.0.6 and will be marked formally obsolete if/when a major issue is discovered. I would also like to remind you that: - the LibVLC API is known to be broken in all 1.0.x releases, - that the Mozilla plugin is broken on X11 platforms in release 1.0.6, and - that binary packages (Windows, MacOS) have already been discontinued.
>
> If you need any of these, please update to VLC 1.1.0-RC3 already, or 1.1.0 at the earliest.
>
> N.B.: VLC 1.0.5, 0.9.10, 0.8.6i and older versions exhibit known published security issues. Update urgently if you have not already done so.

It is evident from the statement that they are struggling with the number of developers that they need in order to offer support for their previously released version. In order to be able to push the project forward, they are forced to stop the maintenance of an older version, and even go the step as to mark it obsolete if it would be proven that there is a major issue with the release. This is a drastic move, but nevertheless seems to be a necessary choice. After a decade of development it could be possible that a decline in interest and increasing complexity in the project is one of its weak points.

[1] Double negative, probably intended to write "maintained".

As discussed in section 3.2 maintenance of code is viewed as a very costly and time consuming process described as a ”necessary evil”. As we can see from the VLC project, maintenance of code is very important to the project. Holes in security are critical elements of software and it could lead to a lack of trust between the users and developers should it not be adressed. The VLC project has for long been struggling with security issues ever since the 0.8.6 release which had multiple security fixes released over the course of six months. The 0.8.6 branch was also announced to be no longer be officially maintained, in similar fashion to version 1.0.0. As seen on the timeline in the VLC case chapter, the time between the release of version 0.8.0 and 0.9.0 is substantial and took serveral years.

The balance between maintenance and innovation looks like it could be a decisive issue for the project after a decade of development. The work of maintenance is slowing down the progress of the project, as people are forced to put their time to fix old code instead of working on innovative new features for new releases. The distributed nature of open source development is a source of inspiration for many companies seeking to increase their rate of innovation. As seen in section 2.2.3, it does seem like the dispersity of developers has a few limitations when confronted with the issue of growing maintenance efforts.

In order to make progress in creating new features in this time of decline, the VLC project gathers some information on what the users would like to see implemented in the media player. This is done by communicating to the users via the forum. The choice to look ”outside” the project for ideas and innovation is what lies at the core of Open Innovation (Chesbrough, 2003), and inspires innovation in many of todays companies.

One of the big differences between traditional software engineering and open source development is that there is no definite end or finalization state for a open source project. It will not cease to evolve until there are no developers left working on it, and the growing complexity of the VLC project seems to cause problems with maintenance. As discussed in section 3.2, the effect of software erosion and loss of key personel were two of the biggest difficulties of software maintenance. There is also lack of formalization that might make the project spiral out of control, or create extra work later on when any of the modules needs to be updated. Although they have adapted to certain informalisms to make up for the lack of formal frameworks and planning (Scacchi, 2002), the costs of not having these will perhaps be too big as the project grows complex and there is loss of personel. So instead of making progress and innovating by focusing on new features, they are forced to maintain the project instead, as seen in VLC.

jQuery project has to continually work on getting better performance, but it also has to present new features in order to give a sense of progression and attract new users. This poses a real challenge as you need to find a balance between doing maintenance, such as tweaking the existing code, or work on new features to be implemented. In the beginning of the project, jQuery were characterized by the lack of a tool for project management. There were quite a few posts signaling the need for a roadmap, and there was uncertainty in whether to invest time in tweaking the performance, or implement new features. Below, some of the discussions mentioning this issue of fine balancing between spending time on optimizing and implementing new features are highlighted.

> It seems to me that with the tweaking you guys have done already that selectors are no longer a big performance issue. [...]
>
> The difficulty I have with jQuery right now is that we still don't have many of the high-level components that people expect out of a framework, and what we have doesn't always fit together well. [...] **-jCore#4**
>
> That's fine - if you want to work on it, you can go ahead. However, optimization of the jQuery core should never stop - and a treeview or a splitter is never going into core. Interface has an example of a folder view, and there's already the splitter that you wrote. If you'd like to improve the help improve the quality of jCore#3's API view, then that's fine - but it's fairly unrelated to this discussion. In fact, I was going to propose a bunch of mini-projects that developers could undertake, if they so desire (such as a live demo on the home page, and an interactive download area). But we can discuss that in another thread.**-jCore#1**
>
> I didn't mean to imply they were mutually exclusive, just that things other than selector performance likely to be the stumbling blocks for jQuery users/developers. You might want to resurrect jCore#3's roadmap post as a starting point for the new thread.**-jCore#4**

This discussion emphasizes the importance of maintenance in the jQuery project. *Optimization of the jQuery core should never stop.* While optimization might seem like a task concerning maintenance of the code, it could be argued that the optimization in the case of jQuery is closely linked to innovation as it is one of the core features of jQuery; a lightweight and fast JavaScript framework. Achieving the high performance that jQuery has, as seen in figure 5.2.3, is a substantial feat and one that requires acknowledgement. However, they still distinguish work on optimization in jQuery from implementing new features. Unlike what has been

described in the section about maintenance (section 3.2, the work done on maintaining code through optimization is recognized as a very high achievement and contributors get much praise for any increase in performance. Increasing performance of a project requires deep knowledge of both algorithms, programming language and the interactions between elements. This kind of recognition, by praising skill, could be one of the reasons behind code maintenance being a more glamorous field than what it has generally been in traditional software development. One example of such praise and the attention given to any potential optimization of code:

> I'm going to play and read through your code. This is a massive change and one that deserves a lot of thought. But as it stands, we're showing nominal improvements in speed in Firefox and /massive/ speed improvements in IE. That alone is deserved of much praise. (IE is such a hard nut to crack) **-jCore#1**

Encouraging developers that are doing valuable contributions to the maintenance of the project by praise on the mailing list, can be closely related to the motivational factor noted by Roberts et al. (2006). In the article, acknowledging achievements done by contributors in a formal way was proven to be linked with increased performance rating in projects.

The very mechanisms of innovation in open source development, informal project tracking and distributed development, can also seem to serve as a limit to the amount of innovation that gets to be done. By reducing the amount of innovation in the project, the fragility of stagnation is introduced. Projects are, as they evolve, forced to focus more time on maintenance than innovating by adding new functionality. They also seemed more focused on getting the maintenance done, than what has characterized traditional software development. The idea that maintenance is a less glamorous activity seems to fade in these projects, as they recognize that the lifespan of open source projects is indefinite, as opposed to software projects made by a corporate firm who are enveloped in strict framework with deadlines and a need to deliver as quickly as possible to satisfy the customer. However, maintenance is a big drain of hours spent on the project and can be a huge toll on the project, such as the events seen in the VLC project. Open source developers could benefit from realizing this fragility of keeping a balance between innovation and maintenance early on, and create methods and tools for maximizing innovation and minimizing the effort needed to be spent on maintenance. As an example, maintenance can be kept to a minimum by introducing a solid framework for doing test driven development introduced in agile methodologies such as Extreme Programming (see section3.1). Test driven development was introduced to jQuery as they created their own framework known as QUnit. The

innovation process could be increased by clearly formalizing a roadmap with the use of project management tools early on, and increasing rate of contributions by formally recognizing good contributions to maintenance as noted by Roberts et al. (2006).

Chapter 7

Conclusion

In the previous discussions we have seen how crucial it is to maintain growth within the FLOSS community. While motivation is a thoroughly and well researched area of open source, the ability to create a sustainable increase in contributors can be further investigated. Eventhough a project might have years of success behind them, the time needed to maintain code will also increase as the software matures and grows more complex. Result of this being that more and more time is required to keep the project going forward and not growing stale. Perhaps especially important in applications that are connected, or somehow affiliatated, to the internet where there is a significant need for security management.

This thesis provides some implications for researchers; the idea of FLOSS development focusing on constantly creating innovations, seems to be a bit skewed. As noted by Monteiro et al. (2004), the research of open source software has been dominated by success stories such as Apache and Mozilla Firefox. The reality is that perhaps most of the effort in a mature FLOSS project is maintaining code and not necessarily creating new and innovative features. While looking at the sheer number of FLOSS out on the internet, only a tiny percentage of these are the successful, innovative and groundbreaking types of software that the research tend to address as the norm. Further research might benefit from looking at the open source communities as more fragile entities, as opposed to studying the ”heroes”, knowing the limitations and what makes an open source project sustainable will further increase the base of knowledge that is already in place.

There are some implications concerning third parties wanting to take use of this type of technology, such as any framework or software for middle management. The open source software in question, should be carefully assessed with this knowledge of OSS being a fragile entity. As seen in the cases, open source project focused

towards developers with a premissive license, such as jQuery, attract a lot of efforts from many sources and have a supporting framework through these contributions, and might be less fragile than projects aimed at end users only. This is also supported by the survey presented by Lerner and Tirole (2005).

From the analysis, some implications for open source developers can be presented. Firstly, one important part of creating a flow of immigrants to the projects is by publicity and involvement of third-party software. Adopting any license that is not reciprocal will severely increase the possibility of involvement and cooperation both from commercial companies and other third-party software. These cooperation effort can greatly increase the number of contributors in the project and be a valuable cornerstone for creating a sustainable FLOSS enviroment. It can on the other hand also create the possibility of a fork or the commercial exploitation of the project. Another issue that a sudden increase of members might lead to is the increased organizational complexity, requiring further effort to create a more rigid way of project management and communication. The process of selecting an appropriate license is a delicate one, and the ideological complexion behind each license might cause a ”holy war” to emerge as each project are bound to have contributors with different opinions.

Ideologies within communities still present some issues when it comes to handling decisions such as selecting the right license for the project, as stated above. The choice of license does have a possibility of creating a huge impact on the project, by either blocking out a large part of the commercial software sector, or on the other hand creating a situation where your work could be exploited and commercialized. Some practical methodological framework for selecting the correct license for the project would probably benefit any new and aspiring communities.

As a OSS project becomes mature enough it could take advantage from hosting both informal meet-ups and formal conferences. These events can create additional awareness by communicating the projects current standing and a roadmap for further progression. It can also be a way for members to create social bonds increasing the likelyhood of future contributions. The feeling of belonging to a community and being a part of something that has a clear purpose is after all a big motivational factor. Hosting such conferences and meet-ups, however, is most likely a trait that is only feasible by large and already successful projects and should be looked at as a way of sustaining growth and not beginning it.

From the above suggestions it is evident that the developers of open source should be very much aware of the fragility of these projects to begin with. The process of developing open source software is a complex one, for those wanting to succeed and create a sustainable community. This perspective of fragility should be employed

as they plan for licensing, project management and an inviting community.

7.1 Further Research

As the thesis only consist of an investigative study of two cases - in a field of thousands and thousands of projects - it is impossible to draw any real final conclusions on how to best sustain growth and manage code maintenance. However, the basis for further research and validation of the results, should be in place.

As an example, one could start out with a different perspective and approach, such as the mechanics behind maintenance of code within open source software, with the intent of comparing it towards the traditional way of doing it in software engineering, but still bearing in mind this view of fragility in open source.

Bibliography

Abrahamsson, P., Conboy, K., and Wang, X. (2009). ”lots done, more to do”: the current state of agile systems development research. *European Journal of Information Systems*, 19.

Amarok (2010). Amarok. (available online at `http://amarok.kde.org/`).

Aronson, E., Wilson, T. D., and Akert, R. M. (2004). *Social Psychology*. Prentice Hall, fifth edition.

Banker, R. D. and Slaughter, S. A. (1997). A field study of scale economies in software maintenance. *Management Science*, 43(12).

Bennett, K. H. and Rajlich, V. T. (2000). Software maintenance and evolution: a roadmap. In *International Conference on Software Engineering*, pages 73–87.

Bergquist, M. and Ljungberg, J. (2001). The Power Of Gifts: Organizing Social Relationships In Open Source Communities. *Info systems Journal*, (11).

Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer*, 21(5).

Bonaccorsi, A. and Rossi, C. (2003). Altruistic individuals, selfish firms? The structure of motivation in Open Source software. *First Monday*.

Chesbrough, H. (2003). *Open Innovation: The new imperative for creating and profiting from technology*. Harvard Business School Press.

Child, O. L. P. (2011). One laptop per child. available online at `http://one.laptop.org/`.

Chua, W. F. (1986). Radical developments in accounting thought. *The Accounting Review*, 61(4).

Ciborra, C. (1996). *Mission Critical: Challenges for Groupware in a Pharmaceutical Company*, pages 91–120. John Wiley and Sons, Inc.

Cockburn, A. (2002). Agile software development joins the"would-be crowd. *International Journal of Information Technology and Management*.

Cornford, T. and Smithson, S. (2005). *Choosing a Project*, chapter 3, pages 29–52. Palgrave Macmillan.

Dave Newbart (2001). Microsoft CEO take launch break with the Sun-Times. Chicago Sun Times, June 1, 2001, p.57.

David A. Wheeler (2007). Forking. (available online at `http://www.dwheeler.com/oss_fs_why.html#forking`).

Deci, E. L., Koestner, R., and Ryan, R. M. (1999). A meta-analytic review of experiments examining the effects of extrinsic rewards on intrinsic motivation. *Psychological Bulletin*, 125(6).

Fitzgerald, B. (2006). The Transformation of Open Source Software. *MIS Quarterly*, 30(3):587–598.

Fogel, K. (2005). *Producing Open Source Software*. O'Reilly, 1. edition.

Free Software Foundation Inc. (2011). What is Copyleft? (available online at `http://www.gnu.org/copyleft/`).

Ghosh, R. A. (1998). Interview with linus torvalds: What motivates free software developers? *First Monday*, 3(3).

Ghosh, R. A., Krieger, B., Glott, R., and Robles, G. (2002). Free/libre and open source software: Survey and study. available online at `http://www.flossproject.org/report/index.htm`.

Ghosh, R. A. and Prakash, V. V. (2000). Orbiten Free Software Survey. *First Monday*, 5(7).

Grudin, J. (1989). Why Groupware Applications Fail: Problems in Design and Evaluation. *Office: Technology and People*, (4).

Gutwin, C., Penner, R., and Schneider, K. (2004). Group awareness in distributed software development. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*.

Hamerly, J. and with Susan Walton, T. P. (1999). *Freeing the Source: The Story of Mozilla*, chapter 14. O'REILLY.

Hammersley, M. and Atkinson, P. (1983). *Researching Information Systems And Computing*. Tavistock, 1. edition.

Hars, A. and Ou, S. (2002). Working for free? motivations for participating in open-source projects. *International Journal of Electronic Commerce*, 6:25–39.

Haugland, C. and Rabben, B. (2008). Er det mulig aa kombinere fossefall og scrum i samme prosjekt? available online at http://www.nsp.ntnu.no/agilemetoder/files/articles/dataforeningen-20081022-v6%5B1%5D.pdf.

Highsmith, J. and Cockburn, A. (2001). Agile software development: the business of innovation. *Computer*, 34(9):120–127.

Hippel, E. v. and Krogh, G. v. (2003). Open source software and the "private-collective" innovation model: Issues for organization science. *Organization Science*, 14(2):pp. 209–223.

Initiative, T. O. S. (1999). The open source definition. available online at http://www.opensource.org/osd.html.

Jarkko Oikarinen and Darren Reed (1993). Internet Relay Chat Protocol. (available online at http://www.faqs.org/rfcs/rfc1459.html).

John Horgan (2010). Margaret Mead's bashers owe her an apology. (available online at http://www.scientificamerican.com/blog/post.cfm?id=margaret-meads-bashers-owe-her-an-a-2010-10-18).

Jon Sobel (2010). HOW: Technology, Traffic and Revenue - Day 3 SOTB 2010. (available online at http://technorati.com/blogging/article/how-technology-traffic-and-revenue-day/).

Kaminski, H. and Perry, M. (2005). The pattern language of software licensing. In *Proceedings of EUROPloP*, pages 177–219.

Kaminski, H. and Perry, M. (2007). Open Source Software Licensing Patterns. *Computer Science Publications*, (10).

Kent Beck, e. a. (2001). Agile manifest. available online at http://agilemanifesto.org/iso/en/.

Klein, H. K. and Myers, M. D. (1999). A set of principles for conducting and evaluating interpretive field studies in information systems. *MIS Quarterly*, 23(1).

Lakhani, K. R. and Wolf, R. G. (2003). Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects. *SSRN eLibrary*.

Lauren, A. M. S. (2004). *Understanding Open Source & Free Software Licensing*. O'Reilly, first edition edition.

Lerner, J. and Tirole, J. (2002). The simple economics of open source. *The Journal of Industrial Economy*, 50(2).

Lerner, J. and Tirole, J. (2005). The Scope of Open Source Licensing. *The Journal of Law, Economics, and Organization*, 21(1).

Li-Cheng Tai (2001). The History of the GPL. (available online at `http://www.free-soft.org/gpl_history/`).

Lientz, B. P. and Swanson, E. B. (1980). *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Lientz, B. P., Swanson, E. B., and Tompkins, G. E. (1978). Characteristics of Application Software Maintenance. *Communications of the ACM*, 21(6).

Markus, M. L., Manville, B., and Agnes, C. E. (2000). What makes a virtual organization work? *MIT Sloan Management Review*, 42(1).

Martin Gollowitzer (2009). What is the GNU project? (available online at `http://fsfe.org/freesoftware/basics/gnuproject.en.html`).

Massey, B. (2003). Why oss folks think se folks are clue-impaired. In *Proceedings of the 3rd Workshop on Open Source Software Engineering, International Conference on Software Engineering. 2003*, pages 91–97. ICSE.

Mockus, A., Fielding, R. T., and Herbsleb, J. (2000). A case study of open source software development: the Apache server. In *Proceedings of the 22nd international conference on Software engineering*.

Monteiro, E., Oesterlie, T., Rolland, K. H., and Roeyrvik, E. (2004). Keeping it going: The everyday practices of open source software. *Unpublished*.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

Myers, M. D. (1999). Investigating Information Systems With Ethnographic Research. *Communications of the Association for Information Systems*, 2(23).

Myers, M. D. and Newman, M. (2007). The Qualitative Interview in IS Research: Examining the craft. *Information and Organization*, (17).

NAV (2010). Trapper ned konsulentbruken. available online at `http://www.nav.no/Om+NAV/Om+NAV/Trapper+ned+konsulentbruken.239187.cms`.

Nonaka, I. (1991). The Knowledge Creating Company. *Communications of the Association for Information Systems*, 6(79).

Oates, B. J. (2006). *Researching Information Systems And Computing*. SAGE Publications, London, UK, 1. edition.

Open Source Initiative (2011). The BSD License. (available online at http://www.opensource.org/licenses/bsd-license.php).

Orlikowski, W. J. and Iacono, C. S. (2001). Research Commentary: Desperately Seeking the "IT" in IT Research - A Call to Theorizing the IT Artifact. *Information Systems Research*, 2(12).

Raymond, E. S. (2001). *The Cathedral And The Bazaaar: Musings On Linux And Open Source By An Accidental Revolutionary*. O'Reilly, revised edition edition.

Roberts, J., Hann, I.-H., and Slaughter, S. (2006). Understanding the Motivations, Participation and Performance of Open Source Software Developers: A Longitudinal Study of the Apache Projects. *Marshall School of Business Working Paper No. IOM 01-06*.

Rossi, M. A. (2004). Decoding the "free/open source(f/oss) software puzzle" a survey of theoretical and emirical contributions. *Siena, Universita degli Studi di Siena. DIPARTIMENTO DI ECONOMIA POLITICA*.

Royce, W. W. (1987). Managing the development of large software systems: concepts and techniques. In *International Conference on Software Engineering*, pages 328–339.

Sawyer, R. K. (2007). Open source is not innovative. available online at http://www.huffingtonpost.com/dr-r-keith-sawyer/open-source-is-not-innova_b_53256.html.

Scacchi, W. (2002). Understanding the requirements for developing open source software systems. *Software, IEE Proceedings -*, 149(1):24 –39.

Schwaber, K. (1995). Scrum development process. In *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA*, pages 117–134.

Schwaber, K. and Beedle, M. (2001). *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

Seebregts, C. J., Mamlin, B. W., Biondich, P. G., Fraser, H. S., Wolfe, B. A., Jazayeri, D., Allen, C., Miranda, J., Baker, E., Musinguzi, N., Kayiwa, D., Fourie, C., Lesh, N., Kanter, A., Yiannoutsos, C. T., and Bailey, C. (2009). The openmrs implementers network. *International Journal of Medical Informatics*, 78(11):711 – 720.

Silverman, D. (2005). *Writing Research Proposal*, chapter 10. SAGE Publications.

Skype (2010). Skype. (available online at `http://about.skype.com/`).

Skype FAQ (2010). Skype FAQ. (available online at `http://www.hl7.com.au/Skype-Video-Conferencing.htm`).

Sourceforge (2010). Sourceforge. (available online at `http://sourceforge.net/about`).

Stallman, R. (2010). The free software definition. available online at `http://www.gnu.org/philosophy/open-source-misses-the-point.html`.

StatCounter (2011). Top 5 browsers in europe from jul 08 to mar 11. available online at `http://gs.statcounter.com/#browser-eu-monthly-200807-201103`.

swbrown (2008). Automatic chat input field resizing should be optional, regression from 2.3. available online at `http://developer.pidgin.im/ticket/4986`.

Turk, D., France, R., and Rumpe, B. (2000). Limitations of agile software processes. In *IN PROCEEDINGS OF THE THIRD INTERNATIONAL CONFERENCE ON EXTREME PROGRAMMING AND FLEXIBLE PROCESSES IN SOFTWARE ENGINEERING (XP2002*, pages 43–46. Springer-Verlag.

Videolan (2010). Videolan. (available online at `http://www.videolan.org/`).

Walsham, G. (2006). Doing Interpretive Research. *European Journal of Information Systems*, 3(15).

Walsham, G. and Sahay, S. (1999). GIS for District-Level Administration in India: Problems and Opportunities. *MIS Quarterly*, 23(1):39–65.

Warsta, J. and Abrahamsson, P. (2003). Is Open Source Software Development Essentially an Agile Method? In *Proceedings of the 3rd Workshop on Open Source Software Engineering, International Conference on Software Engineering. 2003*, pages 143–147.

Wheeler, D. (2007). Why open source software / free software (oss/fs, floss, or foss)? look at the numbers! available online at `http://www.dwheeler.com/oss_fs_why.html`.

Appendix A

Question: What do you feel has been the most challenging task in establishing and maintaining jQuery as a stable and creative open source community?

Answer: It's absolutely a challenge to run a project like jQuery - although, thankfully, it's gotten far easier over the years as more people step up to help. There are many more people on the jQuery team now that are helping to run things and tackle important tasks (such as making sure that the servers are running, conferences are doing well, etc.). Thus I would say that the hardest task was finding good people to help me run the project. It took a while but the team that we have now is really solid and is running the project well. - Core#1

Question: Would you have done anything differently if you were to start over?

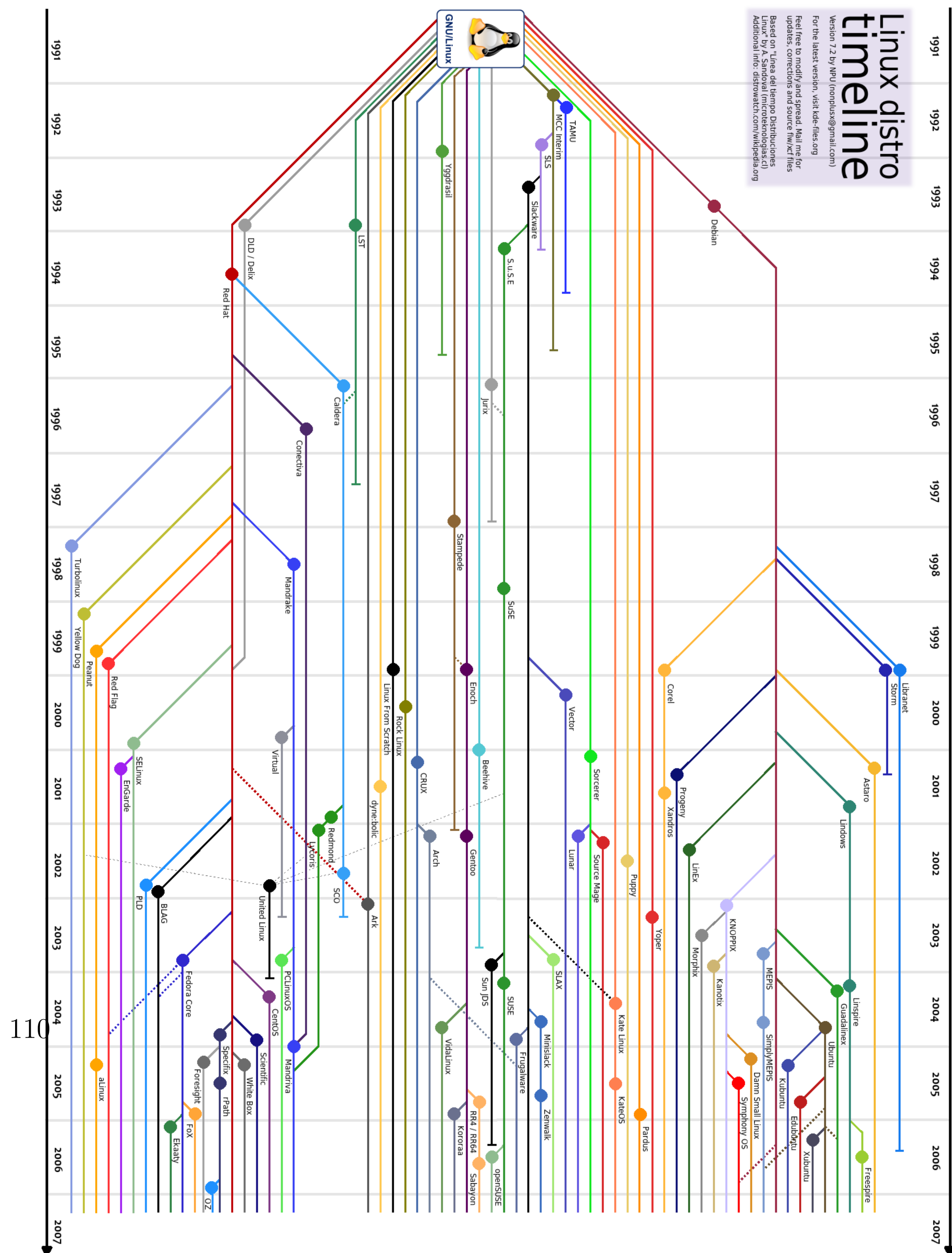
Answer: I think most of the changes I would've made, in the beginning, would've been code changes, not necessarily structural/project changes. - Core#1

Appendix B

E-mail written to the VLC team, to make myself seen and to ask permission to use the mailing lists for my work.

> Hello,
>
> My name is Stian Haga and I am a student at the Norwegian University for Science and Technology (NTNU). I am currently on my fifth and last year of my master degree in information technology, and I am working on a master thesis, in which knowledge management and innovation within open source projects are studied. In light of this I need a few real world projects to use, in order to mine the data needed for analysis. I would love to use the VLC media player project for this, since it is under active development and of a team size that is corresponding to my requirements. The kind of data I am looking for, is conversations between developers, either in form of mailing lists (with archive access), IRC, forums etc. As far as I have gathered these channels are already public, but I still want to ask you for permission to use this in my master thesis. In return I will of course make my thesis available to you if you should wish. If my question has reached the wrong person, a direction on who to contact would be much appreciated.
>
> Thanks in advance.
>
> Best regards, Stian Haga

Appendix C



The linux distributions timeline. Retrieved from http://kde-files.org/content/show.php/latest+Linux+distro+timeline7.2%28Updated%29?content=57722 on May 20th, 2011.

Appendix D

In this appendix some of the Open Source licenses, used in the thesis will be listed in no spesific order. The licenses are unmodified versions from `http://www.opensource.org/licenses`.

The MIT License (MIT)

Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ”Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Mozilla Public License 1.1 (MPL-1.1)

1. Definitions.

1.0.1. ”Commercial Use” means distribution or otherwise making the Covered Code available to a third party. 1.1. ”Contributor” means each entity that creates or contributes to the creation of Modifications.

1.2. ”Contributor Version” means the combination of the Original Code, prior Modifications used by a Contributor, and the Modifications made by that particular Contributor.

1.3. ”Covered Code” means the Original Code or Modifications or the combination of the Original Code and Modifications, in each case including portions thereof.

1.4. ”Electronic Distribution Mechanism” means a mechanism generally accepted in the software development community for the electronic transfer of data.

1.5. ”Executable” means Covered Code in any form other than Source Code.

1.6. ”Initial Developer” means the individual or entity identified as the Initial Developer in the Source Code notice required by Exhibit A.

1.7. ”Larger Work” means a work which combines Covered Code or portions thereof with code not governed by the terms of this License.

1.8. ”License” means this document.

1.8.1. ”Licensable” means having the right to grant, to the maximum extent possible, whether at the time of the initial grant or subsequently acquired, any and all of the rights conveyed herein.

1.9. ”Modifications” means any addition to or deletion from the substance or structure of either the Original Code or any previous Modifications. When Covered Code is released as a series of files, a Modification is:

A. Any addition to or deletion from the contents of a file containing Original Code or previous Modifications. B. Any new file that contains any part of the Original Code or previous Modifications.

1.10. ”Original Code” means Source Code of computer software code which is described in the Source Code notice required by Exhibit A as Original Code, and which, at the time of its release under this License is not already Covered Code governed by this License. 1.10.1. ”Patent Claims” means any patent claim(s), now owned or hereafter acquired, including without limitation, method, process, and apparatus claims, in any patent Licensable by grantor.

1.11. ”Source Code” means the preferred form of the Covered Code for making modifications to it, including all modules it contains, plus any associated interface definition files, scripts used to control compilation and installation of an Executable, or source code differential comparisons against either the Original Code or another well known, available Covered Code of the Contributor’s choice. The Source Code can be in a compressed or archival form, provided the appropriate decompression or de-archiving software is widely available for no charge.

1.12. ”You” (or ”Your”) means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License or a future version of this License issued under Section 6.1. For legal entities, ”You” includes any entity which controls, is controlled by, or is under common control with You. For purposes of this definition, ”control” means (a) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (b) ownership of more than fifty percent (50

2. Source Code License. 2.1. The Initial Developer Grant. The Initial Developer hereby grants You a world-wide, royalty-free, non-exclusive license, subject to third party intellectual property claims: (a) under intellectual property rights (other than patent or trademark) Licensable by Initial Developer to use, reproduce, modify, display, perform, sublicense and distribute the Original Code (or portions thereof) with or without Modifications, and/or as part of a Larger Work; and (b) under Patents Claims infringed by the making, using or selling of Original Code, to make, have made, use, practice, sell, and offer for sale, and/or otherwise dispose of the Original Code (or portions thereof).

(c) the licenses granted in this Section 2.1(a) and (b) are effective on the date Initial Developer first distributes Original Code under the terms of this License. (d) Notwithstanding Section 2.1(b) above, no patent license is granted: 1) for code that You delete from the Original Code; 2) separate from the Original Code; or 3) for infringements caused by: i) the modification of the Original Code or ii) the combination of the Original Code with other software or devices.

2.2. Contributor Grant. Subject to third party intellectual property claims, each Contributor hereby grants You a world-wide, royalty-free, non-exclusive license

(a) under intellectual property rights (other than patent or trademark) Licensable by Contributor, to use, reproduce, modify, display, perform, sublicense and distribute the Modifications created by such Contributor (or portions thereof) either on an unmodified basis, with other Modifications, as Covered Code and/or as part of a Larger Work; and (b) under Patent Claims infringed by the making, using, or selling of Modifications made by that Contributor either alone and/or in combination with its Contributor Version (or portions of such combination), to make,

use, sell, offer for sale, have made, and/or otherwise dispose of: 1) Modifications made by that Contributor (or portions thereof); and 2) the combination of Modifications made by that Contributor with its Contributor Version (or portions of such combination).

(c) the licenses granted in Sections 2.2(a) and 2.2(b) are effective on the date Contributor first makes Commercial Use of the Covered Code.

(d) Notwithstanding Section 2.2(b) above, no patent license is granted: 1) for any code that Contributor has deleted from the Contributor Version; 2) separate from the Contributor Version; 3) for infringements caused by: i) third party modifications of Contributor Version or ii) the combination of Modifications made by that Contributor with other software (except as part of the Contributor Version) or other devices; or 4) under Patent Claims infringed by Covered Code in the absence of Modifications made by that Contributor.

3. Distribution Obligations.

3.1. Application of License. The Modifications which You create or to which You contribute are governed by the terms of this License, including without limitation Section 2.2. The Source Code version of Covered Code may be distributed only under the terms of this License or a future version of this License released under Section 6.1, and You must include a copy of this License with every copy of the Source Code You distribute. You may not offer or impose any terms on any Source Code version that alters or restricts the applicable version of this License or the recipients' rights hereunder. However, You may include an additional document offering the additional rights described in Section 3.5. 3.2. Availability of Source Code. Any Modification which You create or to which You contribute must be made available in Source Code form under the terms of this License either on the same media as an Executable version or via an accepted Electronic Distribution Mechanism to anyone to whom you made an Executable version available; and if made available via Electronic Distribution Mechanism, must remain available for at least twelve (12) months after the date it initially became available, or at least six (6) months after a subsequent version of that particular Modification has been made available to such recipients. You are responsible for ensuring that the Source Code version remains available even if the Electronic Distribution Mechanism is maintained by a third party.

3.3. Description of Modifications. You must cause all Covered Code to which You contribute to contain a file documenting the changes You made to create that Covered Code and the date of any change. You must include a prominent statement that the Modification is derived, directly or indirectly, from Original Code provided by the Initial Developer and including the name of the Initial

Developer in (a) the Source Code, and (b) in any notice in an Executable version or related documentation in which You describe the origin or ownership of the Covered Code.

3.4. Intellectual Property Matters

(a) Third Party Claims. If Contributor has knowledge that a license under a third party's intellectual property rights is required to exercise the rights granted by such Contributor under Sections 2.1 or 2.2, Contributor must include a text file with the Source Code distribution titled "LEGAL" which describes the claim and the party making the claim in sufficient detail that a recipient will know whom to contact. If Contributor obtains such knowledge after the Modification is made available as described in Section 3.2, Contributor shall promptly modify the LEGAL file in all copies Contributor makes available thereafter and shall take other steps (such as notifying appropriate mailing lists or newsgroups) reasonably calculated to inform those who received the Covered Code that new knowledge has been obtained. (b) Contributor APIs. If Contributor's Modifications include an application programming interface and Contributor has knowledge of patent licenses which are reasonably necessary to implement that API, Contributor must also include this information in the LEGAL file.

(c) Representations. Contributor represents that, except as disclosed pursuant to Section 3.4(a) above, Contributor believes that Contributor's Modifications are Contributor's original creation(s) and/or Contributor has sufficient rights to grant the rights conveyed by this License.

3.5. Required Notices. You must duplicate the notice in Exhibit A in each file of the Source Code. If it is not possible to put such notice in a particular Source Code file due to its structure, then You must include such notice in a location (such as a relevant directory) where a user would be likely to look for such a notice. If You created one or more Modification(s) You may add your name as a Contributor to the notice described in Exhibit A. You must also duplicate this License in any documentation for the Source Code where You describe recipients' rights or ownership rights relating to Covered Code. You may choose to offer, and to charge a fee for, warranty, support, indemnity or liability obligations to one or more recipients of Covered Code. However, You may do so only on Your own behalf, and not on behalf of the Initial Developer or any Contributor. You must make it absolutely clear than any such warranty, support, indemnity or liability obligation is offered by You alone, and You hereby agree to indemnify the Initial Developer and every Contributor for any liability incurred by the Initial Developer or such Contributor as a result of warranty, support, indemnity or liability terms You offer.

3.6. Distribution of Executable Versions. You may distribute Covered Code in Executable form only if the requirements of Section 3.1-3.5 have been met for that Covered Code, and if You include a notice stating that the Source Code version of the Covered Code is available under the terms of this License, including a description of how and where You have fulfilled the obligations of Section 3.2. The notice must be conspicuously included in any notice in an Executable version, related documentation or collateral in which You describe recipients' rights relating to the Covered Code. You may distribute the Executable version of Covered Code or ownership rights under a license of Your choice, which may contain terms different from this License, provided that You are in compliance with the terms of this License and that the license for the Executable version does not attempt to limit or alter the recipient's rights in the Source Code version from the rights set forth in this License. If You distribute the Executable version under a different license You must make it absolutely clear that any terms which differ from this License are offered by You alone, not by the Initial Developer or any Contributor. You hereby agree to indemnify the Initial Developer and every Contributor for any liability incurred by the Initial Developer or such Contributor as a result of any such terms You offer.

3.7. Larger Works. You may create a Larger Work by combining Covered Code with other code not governed by the terms of this License and distribute the Larger Work as a single product. In such a case, You must make sure the requirements of this License are fulfilled for the Covered Code.

4. Inability to Comply Due to Statute or Regulation. If it is impossible for You to comply with any of the terms of this License with respect to some or all of the Covered Code due to statute, judicial order, or regulation then You must: (a) comply with the terms of this License to the maximum extent possible; and (b) describe the limitations and the code they affect. Such description must be included in the LEGAL file described in Section 3.4 and must be included with all distributions of the Source Code. Except to the extent prohibited by statute or regulation, such description must be sufficiently detailed for a recipient of ordinary skill to be able to understand it. 5. Application of this License. This License applies to code to which the Initial Developer has attached the notice in Exhibit A and to related Covered Code. 6. Versions of the License. 6.1. New Versions. Netscape Communications Corporation ("Netscape") may publish revised and/or new versions of the License from time to time. Each version will be given a distinguishing version number. 6.2. Effect of New Versions. Once Covered Code has been published under a particular version of the License, You may always continue to use it under the terms of that version. You may also choose to use such Covered Code under the terms of any subsequent version of the License

published by Netscape. No one other than Netscape has the right to modify the terms applicable to Covered Code created under this License.

6.3. Derivative Works. If You create or use a modified version of this License (which you may only do in order to apply it to code which is not already Covered Code governed by this License), You must (a) rename Your license so that the phrases ”Mozilla”, ”MOZILLAPL”, ”MOZPL”, ”Netscape”, ”MPL”, ”NPL” or any confusingly similar phrase do not appear in your license (except to note that your license differs from this License) and (b) otherwise make it clear that Your version of the license contains terms which differ from the Mozilla Public License and Netscape Public License. (Filling in the name of the Initial Developer, Original Code or Contributor in the notice described in Exhibit A shall not of themselves be deemed to be modifications of this License.)

7. DISCLAIMER OF WARRANTY. COVERED CODE IS PROVIDED UNDER THIS LICENSE ON AN ”AS IS” BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES THAT THE COVERED CODE IS FREE OF DEFECTS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE COVERED CODE IS WITH YOU. SHOULD ANY COVERED CODE PROVE DEFECTIVE IN ANY RESPECT, YOU (NOT THE INITIAL DEVELOPER OR ANY OTHER CONTRIBUTOR) ASSUME THE COST OF ANY NECESSARY SERVICING, REPAIR OR CORRECTION. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY COVERED CODE IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER. 8. TERMINATION. 8.1. This License and the rights granted hereunder will terminate automatically if You fail to comply with terms herein and fail to cure such breach within 30 days of becoming aware of the breach. All sublicenses to the Covered Code which are properly granted shall survive any termination of this License. Provisions which, by their nature, must remain in effect beyond the termination of this License shall survive. 8.2. If You initiate litigation by asserting a patent infringement claim (excluding declatory judgment actions) against Initial Developer or a Contributor (the Initial Developer or Contributor against whom You file such action is referred to as ”Participant”) alleging that:

(a) such Participant’s Contributor Version directly or indirectly infringes any patent, then any and all rights granted by such Participant to You under Sections 2.1 and/or 2.2 of this License shall, upon 60 days notice from Participant terminate prospectively, unless if within 60 days after receipt of notice You either: (i) agree in writing to pay Participant a mutually agreeable reasonable royalty

for Your past and future use of Modifications made by such Participant, or (ii) withdraw Your litigation claim with respect to the Contributor Version against such Participant. If within 60 days of notice, a reasonable royalty and payment arrangement are not mutually agreed upon in writing by the parties or the litigation claim is not withdrawn, the rights granted by Participant to You under Sections 2.1 and/or 2.2 automatically terminate at the expiration of the 60 day notice period specified above.

(b) any software, hardware, or device, other than such Participant's Contributor Version, directly or indirectly infringes any patent, then any rights granted to You by such Participant under Sections 2.1(b) and 2.2(b) are revoked effective as of the date You first made, used, sold, distributed, or had made, Modifications made by that Participant.

8.3. If You assert a patent infringement claim against Participant alleging that such Participant's Contributor Version directly or indirectly infringes any patent where such claim is resolved (such as by license or settlement) prior to the initiation of patent infringement litigation, then the reasonable value of the licenses granted by such Participant under Sections 2.1 or 2.2 shall be taken into account in determining the amount or value of any payment or license.

8.4. In the event of termination under Sections 8.1 or 8.2 above, all end user license agreements (excluding distributors and resellers) which have been validly granted by You or any distributor hereunder prior to termination shall survive termination.

9. LIMITATION OF LIABILITY. UNDER NO CIRCUMSTANCES AND UNDER NO LEGAL THEORY, WHETHER TORT (INCLUDING NEGLIGENCE), CONTRACT, OR OTHERWISE, SHALL YOU, THE INITIAL DEVELOPER, ANY OTHER CONTRIBUTOR, OR ANY DISTRIBUTOR OF COVERED CODE, OR ANY SUPPLIER OF ANY OF SUCH PARTIES, BE LIABLE TO ANY PERSON FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES, EVEN IF SUCH PARTY SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO LIABILITY FOR DEATH OR PERSONAL INJURY RESULTING FROM SUCH PARTY'S NEGLIGENCE TO THE EXTENT APPLICABLE LAW PROHIBITS SUCH LIMITATION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THIS EXCLUSION AND

LIMITATION MAY NOT APPLY TO YOU. 10. U.S. GOVERNMENT END USERS. The Covered Code is a ”commercial item,” as that term is defined in 48 C.F.R. 2.101 (Oct. 1995), consisting of ”commercial computer software” and ”commercial computer software documentation,” as such terms are used in 48 C.F.R. 12.212 (Sept. 1995). Consistent with 48 C.F.R. 12.212 and 48 C.F.R. 227.7202-1 through 227.7202-4 (June 1995), all U.S. Government End Users acquire Covered Code with only those rights set forth herein. 11. MISCELLANEOUS. This License represents the complete agreement concerning subject matter hereof. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. This License shall be governed by California law provisions (except to the extent applicable law, if any, provides otherwise), excluding its conflict-of-law provisions. With respect to disputes in which at least one party is a citizen of, or an entity chartered or registered to do business in the United States of America, any litigation relating to this License shall be subject to the jurisdiction of the Federal Courts of the Northern District of California, with venue lying in Santa Clara County, California, with the losing party responsible for costs, including without limitation, court costs and reasonable attorneys' fees and expenses. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any law or regulation which provides that the language of a contract shall be construed against the drafter shall not apply to this License. 12. RESPONSIBILITY FOR CLAIMS. As between Initial Developer and the Contributors, each party is responsible for claims and damages arising, directly or indirectly, out of its utilization of rights under this License and You agree to work with Initial Developer and Contributors to distribute such responsibility on an equitable basis. Nothing herein is intended or shall be deemed to constitute any admission of liability. 13. MULTIPLE-LICENSED CODE. Initial Developer may designate portions of the Covered Code as Multiple-Licensed. Multiple-Licensed means that the Initial Developer permits you to utilize portions of the Covered Code under Your choice of the MPL or the alternative licenses, if any, specified by the Initial Developer in the file described in Exhibit A.

EXHIBIT A -Mozilla Public License.

“The contents of this file are subject to the Mozilla Public License Version 1.1 (the ”License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at `http://www.mozilla.org/MPL/` Software distributed under the License is distributed on an ”AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

The Original Code is ______________________.

The Initial Developer of the Original Code is ________________________. Portions created by ____________________ are Copyright (C) _____ _______________________. All Rights Reserved.

Contributor(s): ______________________________________.

Alternatively, the contents of this file may be used under the terms of the _____ license (the [___] License), in which case the provisions of [______] License are applicable instead of those above. If you wish to allow use of your version of this file only under the terms of the [____] License and not to allow others to use your version of this file under the MPL, indicate your decision by deleting the provisions above and replace them with the notice and other provisions required by the [___] License. If you do not delete the provisions above, a recipient may use your version of this file under either the MPL or the [___] License."

[NOTE: The text of this Exhibit A may differ slightly from the text of the notices in the Source Code files of the Original Code. You should use the text of this Exhibit A rather than the text found in the Original Code Source Code for Your Modifications.]

The BSD 3-Clause License

Copyright (c) <YEAR>, <OWNER> All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of the <ORGANIZATION> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ”AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The GNU Lesser General Public License, version 3.0 (LGPL-3.0)

GNU LESSER GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. ¡http://fsf.org/¿

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions. As used herein, ”this License” refers to version 3 of the GNU Lesser General Public License, and the ”GNU GPL” refers to version 3 of the GNU General Public License.

”The Library” refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An ”Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A ”Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the ”Linked Version”.

The ”Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The ”Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL. You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions. If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility

still operates, and performs whatever part of its purpose remains meaningful, or b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy. 3. Object Code Incorporating Material from Library Header Files. The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. b) Accompany the object code with a copy of the GNU GPL and this license document. 4. Combined Works. You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. b) Accompany the Combined Work with a copy of the GNU GPL and this license document. c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. d) Do one of the following: 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of

the GNU GPL for conveying Corresponding Source.) 5. Combined Libraries. You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work. 6. Revised Versions of the GNU Lesser General Public License. The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License ”or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy’s public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

Appendix 5

This appendix will contain The Open Source Definition as listed on http://www.opensource.org/docs/osd.

The Open Source Definition

Introduction Open source doesn't just mean access to the source code. The distribution terms of open-source software must comply with the following criteria:

1. Free Redistribution The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.

2. Source Code The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.

3. Derived Works The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

4. Integrity of The Author's Source Code The license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.

5. No Discrimination Against Persons or Groups The license must not

discriminate against any person or group of persons.

6. No Discrimination Against Fields of Endeavor The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

7. Distribution of License The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

8. License Must Not Be Specific to a Product The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.

9. License Must Not Restrict Other Software The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.

10. License Must Be Technology-Neutral No provision of the license may be predicated on any individual technology or style of interface.

Appendix E

Manifesto is accessed from the homepage for the Agile Manifesto at `http://www.agilemanifesto.org`.

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

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

That is, while there is value in the items on the right, we value the items on the left more.

Appendix F

```
try
{
  scan = new Scanner( new BufferedReader+(new FileReader("
      outputjquery.txt")));

  while(scan.hasNextLine())
  {
    dateMatcher = datePattern.matcher(scan.nextLine());
    if(dateMatcher.find())
    {
      totalCommits++;
      commitDate = dateMatcher.group(1)
     +dateMatcher.group(2) +dateMatcher.group(3);
      currentCommitDate = sdf.parse(commitDate);
      calendarDate = cal.getTime();
      currentWeek = cal.get(Calendar.WEEK_OF_YEAR);
      cal.setTime(currentCommitDate);
      commitWeek = cal.get(Calendar.WEEK_OF_YEAR);
      cal.setTime(calendarDate);
      if(currentWeek != commitWeek)
      {
        commitsPerUke.add(""+counter);
        checkCommits += counter;
        counter = 0;
        missCounter+=1;
        while(!rightWeek)
        {
          cal.add(Calendar.DATE, 7);
          currentWeek = cal.get(Calendar.WEEK_OF_YEAR);
          if(currentWeek!=commitWeek)
          {
          commitsPerUke.add("0");
          }
          else
          {
```

```
35                  rightWeek = true;
36              }
37          }
38          rightWeek = false;
39      }
40      else
41      {
42      counter += missCounter + 1;
43      missCounter = 0;
44      }
45    }
46  }
47 }
```