NTNU
Innovation and Creativity

Software Quality in the Trenches
Two Case Studies of Quality Assurance Practices in Free/Libre and Open Source Software (FLOSS)

Tor Arne Vestbø

Master of Science in Informatics

Submission date:	November 2007
Supervisor:	Eric Monteiro, IDI
Co-supervisor:	Thomas Østerlie, IDI

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

Tor Arne Vestbø

Software Quality in the Trenches

Two Case Studies of Quality Assurance Practices
in Free/Libre and Open Source Software (FLOSS)

Trondheim, November 2007

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics
and Electrical Engineering
Deparment of Computer and Information Science

Master of Science in Informatics
Supervisor: Eric Monteiro, IDI
Co-supervisor: Thomas Østerlie, IDI

NTNU
Innovation and Creativity

This thesis was typeset in Linotype Galliard and FontFont DIN by the author, using the $\LaTeX\,2_\varepsilon$ macro language on a Dell D420 running openSUSE 10.2.

Printed by Tapir Trykk. Available online from `http://daim.idi.ntnu.no/`

Cover photo shows trees planted in a 1st World War memorial in northern France. Each tree represents a life lost in the trenches, which are now covered in grass.

Abstract

When proponents of open source software are asked to explain the success of their movement they typically point to the quality of the software produced, which is in turn attributed to the rather unconventional development model of releasing unfinished versions of the software and having users look over the code and report and fix bugs.

This thesis investigates the open source quality assurance model from a knowledge management perspective – based on the assumption that debugging involves a high degree of knowledge work. By doing interpretive case studies of two open source projects – using direct observation, e-mail archives, and bug-trackers as data sources – I present descriptive accounts of the day to day quality practices in open source development.

The analysis shows that conceptualizing and classifying bugs is a complex process involving sensemaking and subjective considerations; that the peer-review process in open source projects has a lot in common with traditional field-testing; and that communication tools and mediums are used interchangeably, but with certain preferences depending on subject matter. I conclude that perhaps the success of the open source development model is not due to its novelty compared to traditional software engineering, but because open source developers have recognized that debugging is a knowledge-intensive process.

Keywords: Open Source, Software Quality, Knowledge Management

Preface

This thesis is the result of one year of work, and concludes the requirements for a Master's degree in informatics at the Norwegian University of Science and Technology (NTNU). The assignment was given by the Department of Computer and Information Science (IDI), with Professor Eric Monteiro as acting supervisor.

Despite the illusion that I already knew a fair bit about open source development before starting this thesis, I now feel that I have a pretty good grasp of the phenomena – and I am certain that the knowledge I have obtained will help me in my future endeavors.

I would like to thank my supervisor, Professor Eric Monteiro, for his invaluable feedback and support during these past two years, and my co-supervisor, Thomas Østerlie, for his motivating pep-talks and sound insights. I would also like to thank the developers of the two projects Amarok and Gallery for their time and hospitality.

Last, but not least, I'd like to thank my parents for finally giving in and buying me that lousy Remedy `i486` computer, so I could hack new levels for Nibbles while the other kids were out playing soccer.

Trondheim, November 2007

Tor Arne Vestbø

Table of Contents

Chapter 1

Introduction

Open source software has gone trough several shifts in the past two decades. From starting out as a lone hacker's dream of an ecosystem of free software – via charismatic evangelists and an almost religious grass root 'movement' – it has turned into a multi billion dollar industry, with companies such as Red Hat and MySQL based entirely around selling open source software.

Giants like Sun Microsystems and IBM have also picked up on the trend – and are in fact the two largest business contributors to open source software today (Ghosh, 2007). Sun has released both their office suite OpenOffice and the Solaris operating system as open source, and recently open sourced the Java programming language. IBM has open sourced their development environment Eclipse, and are investing large sums in projects such as Apache and Linux – some sources say $100 million annually (Ghosh, 2007). At the same time these companies are being criticized for exploiting open source developers as mere third party contractors, trying to rub off some magic 'open-source-dust', while not really living up to the true spirit of open source. The recent case of Sun imposing severe restrictions on projects who want to certify their open source Java implementations is an example of this clash.

Meanwhile, success stories like Firefox and Linux (the poster boys of open source software) are gaining the movement mainstream acceptance, even to the point that the average Joe wants to install Linux without knowing what it is. As I'm writing this, one of Norway's largest tabloid newspapers VG has an article on their website's front page reporting that Wal-Mart will start selling cheap computers with Linux pre-installed. The author notes that "this is a clear sign that the operating system is ready for the masses" – which is an ironic prediction coming from a newspaper that normally cannot be accused for reporting from the forefront of technology.

Governments are also starting to realize the power of open source. On the 15th of December 2006 the Norwegian minister of government administration and reform Heidi Grande Røys, presented her vision for the future of Norwegian ICT industry, in which she encouraged public administration to adopt open source and open standards when possible. Critics were quick to point out that the vision – although a step in the right direction – lacked any formal commitment. Even so, this shows that open source plays an increasing part of our lives, both for the general consumer and for large organizations.

1.1 Problem Definition

When proponents of open source are asked to explain the success of their movement they often point to the superior quality of the software produced (Raymond, 1999a; Vixie, 1999). With horror-stories of traditional software engineering projects gone awry looming in the background this is a tempting argument, but not everyone is that easily convinced. A closer inspection of this claim reveals a firm belief that open source projects have fewer defects, better performance, and improved security over closed source software (Wheeler, 2007).

These advantages are attributed to the somewhat untraditional approach to release-management. Instead of waiting months or even years between each release, open source projects strive to provide users with fresh builds of the software – often daily/nightly, or at least weekly. This is meant to foster the effect first described by Eric Raymond in his essay *The Cathedral & The Bazaar*, that "Given enough eyeballs, all bugs are shallow" (Raymond, 1999a, p. 30).

Later coined as Linus's Law, this quote refers to the idea that if enough people can study the source code you are bound to find and fix all the lurking bugs – because each person will inadvertently test a slightly different code-path than the next. This massively distributed debugging-process has been argued to be one of the key reasons for the quality of open source software, because bugs that would otherwise linger deep inside the code are found and fixed by helpful users of the software (Raymond, 1999a; Dibona et al., 1999; Feller and Fitzgerald, 2000).

While it is hard to dispute the success of projects like Linux and Apache (Miller et al., 1995; Halloran and Scherlis, 2002; Reasoning, 2003), and studies *have* found user-participation in open source projects to be very high (Zhao and Elbaum, 2003), research into the *nature* of this participation – providing empirical evidence for Linus's Law and how it affects code quality – is limited at best (Michlmayr et al., 2005).

At first glance the eyeballs-analogy may seem beautifully ingenious: you just add thousands of bug-catching users to the assembly-line, and they will proceed to fix and check off all the bugs as they pass by – leaving a bug free product at the end of the line. But, from a knowledge perspective this description seems rather simplistic and naive – almost like an echo from the early days of knowledge management when the solution to every organizational problem was to centralize the knowledge of all the workers in huge data centers and databases. In both cases the subject matter is assumed to be factual and explicit – something that can be transfered and stored easily. It is my convictions that this is not the case, and that there is a lot more to open source quality assurance than just throwing thousands of eyeballs at the code. Unfortunately, without knowing more about the underlying mechanisms, and how they play out in practice, it is very difficult to draw any conclusive lessons for improvement of the open source process, or for adoption in the wider software community.

Based on this dilemma I propose the following problem definition:

What are the day-to-day quality assurance practices of open source projects?

This definition is further broken down into three main research questions:

RQ1: How do developers conceptualize and classify the nature of bugs?

RQ2: How is peer-reviewing employed in the quality assurance process?

RQ3: How are communication-tools affecting quality assurance practices?

The existing literature on quality aspects of open source development has so far been largely focused on surveys (Zhao and Elbaum, 2000, 2003; Halloran and Scherlis, 2002; Michlmayr, 2005; Porter et al., 2006) – with one notable exception, namely the case study by Mockus et al. (2002) of development practices in Apache and Mozilla (Shaikh and Cerone, 2007).

My goal is to add to that body of knowledge by doing an interpretive case study of two open source projects – Amarok and Gallery – over the course of 12 months. The choice of lesser known projects like these two follows in the footsteps of similar studies (Monteiro et al., 2004) – deemphasizing the heroic success-stories and instead bringing out the day-to-day activities that keeps the wheels turning. Using direct observation – combined with data mining of online sources such as mailing lists and bug trackers – I aim to provide rich descriptive accounts of the day-to-day quality assurance practices in these two projects. Hopefully I will be able to answer some of the questions above, and perhaps challenge some of the slogans and mantras of blind Raymondism (Bezroukov, 1999) in the process.

1.2 Project Scope

OPEN SOURCE has been studied from a wide range of fields, including psychology (Hertel et al., 2003), business management (von Krogh et al., 2003), information systems (Bergquist and Ljungberg, 2001), and software engineering (Mockus et al., 2002). Due to the time constraints of the Master's programme, I have limited the scope of the literature review to include software engineering and information systems. This excludes computer-supported cooperative work, a field adjacent to software engineering, which could also have been explored.

Empirically I have limit the number of cases to two, and my data collection scope to the period from January 2006 to August 2007 – to not get overwhelmed by the amount of data. I also decided against participating heavily in the projects under study (despite it being a tempting challenge), due to the risk of 'going native' and ending up with badly prepared action research instead of a solid case study.

Access to the various communication-mediums and tools for the two projects was for the most part open – except for two cases where I was not granted access. The first was the developer-only IRC channel for the Amarok project (described further in Chapter 5), and the other was the security-related mailing list for Gallery project (which the developers preferred to keep private). Restricted access is a common problem when adopting the role of an outside observer instead of participating actively in the project (Walsham, 1995) – but all things considered I believe I observed the majority of the picture.

1.3 Report Outline

THE BODY of this thesis is divided into three main parts: theory, case study, and analysis. Each part is in turn divided into a small number of chapters by topic. Following the advice of Cornford and Smithson (1996) each chapter starts out with a brief introduction to what the chapter will cover, before moving on to the actual content. A short summary has also been added to some of the chapters where appropriate, to recapitulate key points.

The chapters are as follows:

Chapter 2 presents a systematic overview of open source, ranging from history and well known projects to licensing, motivation, and development practices.

Chapter 3 considers software quality from both the traditional software-engineering approach, and as peer-reviewing in open source projects.

Chapter 4 shows how knowledge management has evolved in the past two decades, and introduces important concepts such as tacit knowledge and communities of practice.

Chapter 5 describes the research process, from start to finish, and evaluates the end result using the principles of Klein and Myers (1999).

Chapter 6 gives an introduction to the two case study projects, Amarok and Gallery, focusing on history, organization, and development practices.

Chapter 7 presents excerpts from the data material, illustrating interesting findings and situations from the two cases.

Chapter 8 analyses observations from the two cases based on the problem definition and research questions presented in this chapter.

Chapter 9 concludes the thesis and suggests topics for further study.

In addition, Appendix A contains full verbatim copies of the open source licenses discussed in Chapter 2, for quick reference.

Part I

Theory

Chapter 2

An Overview of Open Source

From a distance open source may seem both mysterious and contradicting. Why on earth are people dedicating their precious time to produce software that has no apparent marked value, openly cooperating with developers around the world, and still being able to produce quality software? This chapter aims to lift the veil of open source and answer these questions.

We will start off with the history of open source, before moving on to core issues such as licensing and motivation. Along the way we will introduce some well known examples of open source projects, to get some initial context. Then we will discuss some of the notable characteristics of open source development, such as the approach to debugging and release management, before finally closing the chapter with a look at how open source can be leveraged from a business perspective.

2.1 Historical Background

THE HISTORY of open source can be traced back to the 60s and early 70s, with the early mainframe hackers[1]. These people were not criminals – like the word *hacker* is used today – but employees of respected scientific organizations like MIT, CMU, Standford and Xerox PARC, working on everything from AI to the early version of the Internet. Back then there was no such thing as commercial off-the-shelf software, so if you needed a job done you usually had to write the software yourself, or ask some wizard to do it for you. The source code for these applications was happily shared between friends and colleagues – basically because there was no intensive not to do it.

That all changed drastically in the early 80s due to the commercialization of the IT industry. Suddenly doing AI research and selling Unix distributions was hot business, and people found themselves increasingly left with binary only version of

[1] A hacker in this context refers to a programmer who uses clever tricks to solve an issue based on a solid understanding of the problem (Hannemyr, 1998).

the software they purchased. This was bad news for all the computer savvy hackers around the world, because now they had to go through the official vendor each time a piece of software broke down or had an annoying bug – instead of just fixing it themselves. The situation was complicated by software vendors often spending considerable time fixing a bug (just like today), or ignoring the request because the user base it affected was too small. Some people even argued, on ideological grounds, that the lack of source code was a violation to the freedom of the people using the software.

Figure 2.1: Richard Stallman

One of these people was Richard Stallman. As a former researcher of MIT's AI lab he had seen co-worker after co-worker being hijacked from the lab to well paid jobs in the growing IT industry. His fear of a world without sharing of source code was so strong that he decided to take it upon himself to provide an alternative. In his mind software should be *free*, as in freedom to run the software for any purpose, study it and adapt it (which requires source code), and redistribute it to anyone you want, even with changes applied.

These principles were formulated into the now well known *GNU General Public License*, which uses copyright law to secure these freedoms. The crucial part of the license that makes this possible is the term that states that any changes to the software also have to be distributed under the GPL. This effectively removes the risk of someone taking a GPL'ed software package and closing it for the rest of the world, because once you license something under GPL it will never loose that license. While this may seem a bit strict it is worth noting that the GPL says nothing about whether or not the software can be distributed for a fee – so charging 100$ for a download, or 1000$ for a physical CD, is perfectly legal (although perhaps frowned upon) as long as you provide the source code too.

The first step in Stallman's plan to create an ecosystem consisting entirely of free software was building a platform for all the other software – ie. an operating system and its system utilities and compiler. But he was soon faced with the paradox that if he was to develop a new operating system he had to do it on an existing platform. In a rare moment of pragmatic clairvoyance he realized that for the operating system to gain any momentum it had to offer something to its users right from the start. So, Stallman decided to make the OS compatible with Unix – the hacker-OS of choice at the time. That way people could work on Unix while developing software for the new free operating system. His plan was to build the support tools first, and then finish off with the OS kernel once all the support applications were in place. The new OS was playfully named *GNU*, which is a recursive acronym for “GNU's Not Unix” (Stallman, 1985).

Of the more notable tools Stallman created back then is the text editor *Emacs*, and the C-compiler *gcc*. Both are in widespread use today, and gcc is the standard compiler on almost any platform – apart from Windows, where it's only second to Microsoft's own compiler.

2.1.1 The Free Software Foundation

As Stallman's ideas about free software spread, other people joined his efforts, and in 1985 they decided to form a non-profit corporation called the Free Software Foundation (FSF). This was supposed to be an umbrella-organization for future GNU development, and they also hired internal programmers – financed though the sale of cassettes and CDs containing free software. This is a good example of how Stallman doesn't oppose charging for distribution, as long as the software distributed is free.

Following the founding of the FSF the GNU development really took off, and by the late 80s most of the essential libraries, system utilities and user applications were done. But one important part was still far from being complete: the operating system kernel, where the real voodoo happens. Development of the HERD-kernel, as it was named, was delayed over and over again, and it actually didn't boot properly until the year 1994! Even today there has still not been any release of a complete, working, version of the kernel, and most of the development has stalled.

2.1.2 The Rise of Linux

So, if an operating system needs a kernel to even boot, and the development of HERD never resulted in anything usable, how can I be typing this thesis on a computer running a free software kernel? The answer lies in a quirky Finish computer science student named Linus, who in 1991, upset about the lack of a working free kernel, set out to write his own, inspired by Andrew Tannenbaum's Minix and SunOS (Torvalds and Diamond, 2002).

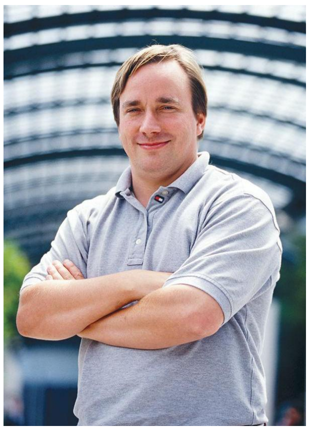


Figure 2.2: Linus Torvalds

For Linus this was just as much an exercise in doing software architecture as it was the need for a free operating system, but luckily he decided to release the kernel under a GPL-license, which meant that other people could benefit from his efforts. His newsgroup post about the release is now a classic quote, and quite remarkable seen in light of todays situation:

"I'm doing a (free) operating system (just a hobby, won't be big and professional) [...] it probably never will support anything other than AT-harddisks"

Linus' release sparked an immediate interest from the many GNU developers around the world, who hungry for a complete free operating system quickly combined the existing GNU applications with Linus' kernel into what is now known as GNU/Linux – or Linux for short. This adoption was probably one of the reasons for the stalled HERD development, since the Linux kernel was just as free, and actually worked. The fact that GNU ended up with "someone else's kernel" has always been a soft spot for Stallman, who consequently refers to Linux as GNU/Linux – to flag his contribution.

2.1.3 The Open Source Rebels

Despite the success of Linux and other free software projects not everyone approved of Stallman's ideological approach. Especially the term *free software* was seen as problematic, because the average Joe, and the software industry in particular, read this as a demand for software at no cost, or *gratis*. This made it difficult to "sell" the idea of free software to companies writing commerical software, and free software was by large seen as a direct threat to the established software industry.

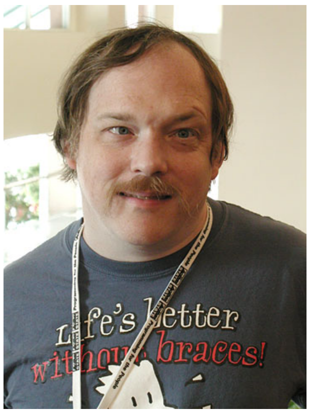


Figure 2.3: Eric Raymond

One of the people worried about the direction of the free software movement was a guy named Eric Raymond, who in 1997 wrote a classic essay titled *The Cathedral and the Bazaar*. In the essay he described some of the mechanisms and practical implications of developing software in the spirit of free software. He argued that distributed and open development of software was superior to the old model of "building cathedrals", and used Linux (which had gained a lot of momentum by then) and his own project *fetchmail* as examples.

The essay made a huge impact in the software community, partly because he was the first person to put into words all the positive effects everyone was seeing. One company who took particular notice was Netscape Communications, who had played with the idea of releasing their web browser Netscape Communicator as open source. Their problem was how to do it successfully, so they hired Eric Raymond to help them make the move.

The collaboration resulted in the release of Mozilla (the browser engine) as open source in March 1998 – an event described in The Economist as the "computer-industry equivalent of revealing the recipe for Coca-Cola" (Economist.com, 1998). But Netscape didn't use Richard Stallman's GPL license for their release. Instead they chose a license created by Eric Raymond and Bruce Perens, where the "viral" clause about derived works having to use the same license was removed. This effectively meant that anyone, even Microsoft, could copy the source code, make modifications to it, and then release everything as a closed source project – much like proprietary software.

This was a huge shot to Stallman's idea of keeping software free for all eternity, but also understandable in light of Raymond's wish to promote open source as more than an ideology. Suddenly free and open source software was also a tool for improving the software development process. At the time Netscape was given a lot of heat for what they did, and most software houses thought they were mad, but time has proved them right, and Mozilla (better known as Firefox to end users) is today a highly successful browser with a large community support.

In the culmination of the Mozilla release Eric Raymond and Bruce Perens formed the *Open Source Initiative* as a way to promote the new way of thinking of free software. They formalized the term "open source" by creating a meta-license which described what a license had to include to be an "Open Source license". This meta license was based on the same principles as the Mozilla release, so not only does the GPL and other "viral" licenses conform to it, but also more relaxed licenses like the BSD license and the original Mozilla license.

The new term quickly grew popular, and is today the most used term for this phenomena (Google.com, 2007). Stallman did of course oppose of the new term, even though he admitted that the use of the word "free" was problematic, and has today completely disassociated himself from the term. It is tempting to interpret some of his frustration as coming from no longer being the guru on free software, and having his operating system baby "stolen" by Linus Torvalds and Linux, but this is something he has only partly admitted to.

Because of the obvious clash of camps and feeling surrounding these terms, the academic community has chosen to use the neutral term Free/Libre/Open Source Software (FLOSS) when describing the phenomena. This incorporates both the notion of freedom (*libre* is French for freedom), and the focus on openness of the source code. For the rest of this thesis I will mix and match these terms for the sake of language flow, but without adding any deeper "political" meaning to it than what the FLOSS term does.

I'LL DO THE WASHING UP IN MY ROOM AND BRING IT DOWN WHEN IT'S FINISHED
YOU GUYS WANT IT WASHED, COME TO ME WHEN ITS DONE AND I'LL TELL YOU IF IT'S CLEAN
YOU TWO WASH UP OR, BY THUNDER, I'LL MAKE YOU REGRET IT. WATCH YOUR STEP.

Figure 2.4: The personalities of the three FLOSS gurus are often subject to jokes in the community, as seen in this comic titled "Everybody loves Eric Raymond".

2.1.4 Open Source and Free Software Today

In recent times open source has become somewhat of a buzz word, and marketing departments around the globe have realized that associating with the term brands them as "one of the good guys". This has of course both positive and negative affects on the community – exchanging increased awareness for a possible decrease in credibility – but most people see through the FUD[2] of the big companies and recognize the ones who are truly working in the spirit of free software.

Many companies, especially hardware vendors like IBM and Sun, have realized that supporting free software helps selling their platform, because including free software is an added bonus for their customers. Others have based their business model around professional services – supporting well known FLOSS software packages. And of course you have the skeptics, usually large traditional software houses, who see free software as a threat to the whole business. Microsoft has always been the bad boy of these, much thanks to an internal memo that was leaked where Microsoft employees discuss how they best can withstand the competition from open source: "OSS poses a direct, short-term revenue and platform threat to Microsoft, [and] the intrinsic parallelism and free idea exchange in OSS has benefits that are not replicable with our current licensing model [. . .]" (Catb.org, 1998).

Despite these forces working against it, open source has become a major player in the software field, and many successful projects have been realized (Reasoning, 2003). Table 2.1 to the right lists some of the more well known projects, including their type and the licenses they employ.

Project	Type	License
Firefox	Web browser	Mozilla Public License
Linux	Operating system	GNU GPL
Eclipse	Development IDE	Eclipse Public License
Apache	Web server	Apache License
OpenOffice	Office tools	LGPL (Lesser GPL)
Gimp	Photo editing	GNU GPL
VLC	Media player	GNU GPL
OGG Vorbis	Sound codec	Public domain

Table 2.1: Examples of some open source projects

2.2 Licensing of Open Source Software

ONE OF the few obvious factors of why open source has not crumbled or eroded from the constant pressure from big software companies (many would say the opposite has happened) is the use of open source licensing. Pioneered by Richard Stallman, the GNU General Public License and similar texts have successfully ensured that the spirit of open source has been passed on through code for the past 30 years. We will now take a look at the legal mechanisms that makes this possible, and study some of the more popular licenses in detail.

[2] Fear, uncertainty, and doubt – a sales or marketing strategy of disseminating negative (and vague) information on a competitor's product.

The majority of this section is based on reading the excellent book *Understanding Open Source and Free Software Licensing* by Laurent (2004). For quick reference, the full text of the individual licenses can be found in Appendix A, but are also available online from `http://www.opensource.org/`.

2.2.1 Copyright Basics

Copyright is, as the word suggests, the right to make copies of a given artistic or intellectual creation, or *work*. More specifically it is a set of exclusive rights regulating the use of the creation – such as the right to display it, modify it, or commercially exploit it. Because copyright is protected by law it does vary from country to country and region to region, but the law has been consolidated through agreements such as the World Trade Organization and the Berne Convention, and is generally considered a universal law.

One does not have to register, or in any way mark the work for the copyright to come into effect. The copyright is automatically attached to every original expression of an idea – for example as I'm writing these words. If I were to draw a drawing on a napkin of a dog with 5 legs it would probably not be considered art, but would still be subject to copyright law. Still, it is customary to add a note such as "Copyright © 2007 Tor Arne Vestbø" to the work, to really signal that you claim copyright (but again, is is not mandatory). The copyright usually expires 50-100 years after the death of the author, and will then go into public domain, which means free for all to use as they see fit.

Copyright is often confused with patents, which is slightly different beast. The first deals with the *expression* of an idea – for example "pink dog with 5 legs, acryl on canvas, 50x50cm" – and there can be several other expressions, by other artists, as the copyright only regulates one particular expression. Patents on the other hand regulates the actual *idea*, and prevents anyone from commercially exploiting this idea, independently of the medium. If I had a patent on dogs with 5 legs, I could legally prevent other artists from painting such dogs, singing about them, or even breeding one. Because of their more general nature, patents require a strict registration process and do not last as long as copyrights. In recent years it has become more and more evident that the patent system is not suited for the digital age. It has long been accepted that you cannot patent scientific truths or mathematical expressions of it because they are the building blocks for everything else, but there is a growing trend – especially in the United States – to approve software patents that are so general that they span much more than their supposed intent. Needless to say, allowing patents such as "a method for digital transmission of data", where the "method" is actually just copying bits in memory, would be disastrous for further innovation in the software field. We will return to a more thorough discussion of software parents in Section 2.2.5.

Finally, related to the two concepts copyrights and patents, are *trademarks*, which regulate the use of logos and brand names. Since trademarks have a small relevance to open source licensing they will not be discussed further in this thesis.

Copyright	Patents	Trademark
Original works of expression	Ideas and inventions	Brand identity
Protects against commercial exploitation of original works such as paintings, novels and source code without permission.	Protects against commercial exploitation of ideas and inventions without the authors permission.	Protects against commercial exploitation of brand names, logos, or other identifiers of tradable goods.
I paint a dog. I can stop others from displaying, copying, or otherwise commercially exploiting my painting.	I invent a hammer which never misses. I can stop others from producing and selling any kind of hammer based on my idea.	I brand my hammer "NailMaster3000™". I can prevent anyone from naming their products anything similar.

Table 2.2: Comparison of intellectual property protection mechanisms

When buying a piece of proprietary software, for example the latest Microsoft Office[3], it comes with a *license*. The license states that Microsoft claims copyright for the work (the software), and then proceeds to list the rights and obligations of the user. Most of the time the only right you have as a user is to run the software – sometimes only at one of your computers at a time. Your obligations includes not copying the software, not modifying it, and not sharing it with others. Software licenses often include other terms, such as warranties, disclaimers, and prohibition on reverse engineering, but these terms are part of the *contract* between you and the software company, and is not protected by copyright law.

When downloading or buying a piece of open source software the situation is similar, but at the same time very different. Open source software also comes with a license, and like proprietary licenses it starts off by claiming the copyright of the work, but it then proceeds to give the user almost any right. You can of course run the software, but you can also study it, modify it, and share it with others. The only obligation opposed on the user is usually that any redistribution of derived works are licensed under the same license as the original, which is a strong contrast to the proprietary license. This way of using copyright law as a tool to turn the situation around to benefit the user instead of the author is playfully called *copyleft*, marked by the symbol 🄯.

Table 2.2 above compares the three mechanisms of intellectual property protection discussed in this section. We can clearly see how copyleft borrows heavily from copyright – apart from the 'minor' detail that the protection is used to share and give away the artistic expression for free.

2.2.2 The Open Source Definition

As described earlier in Section 2.1.3 the Open Source Initiative formalized the basic ideas of opens source licensing into a meta license, which describes what a license has to include to qualify as an open source license. The definition serves as a certificate, allowing companies and others to brand their software as "Open

[3] "Microsoft Office is a trademark of Microsoft Corporation." This notice is required when naming brands under trademark, and the owner of the trademark is required to sue anyone not upholding this notice or else they will lose the trademark. That's why you see these disclaimers everywhere – even in this thesis which will never be read by a Microsoft lawyer.

Source" knowing that since the license they use is OSI approved their software is also in line with the spirit of open source. As such, the Open Source Definition is a good introduction to the principles of open source licensing, and I will now go through and comment the points of the license before moving my discussion to details of the individual licenses.

The Open Source Definition (OpenSource.org, 2006a) has the following 10 points:

1. *Free Redistribution*

   This point ensures that anyone can share the software with the rest of the world, also called distribution. It is important to note that the word "free" does not refer to price, but to liberty. In fact, you are completely free to charge people for the distribution of the software, and many companies have this as their primary business model.

2. *Source Code*

   Any distribution of the original software has to include the source code, or make the source easily available in some other way, so that changes to the software is possible. The source code has to be in a readable and practical form – not obfuscated or mangled beyond human understanding.

3. *Derived Works*

   Not only must the source code be available, as ensured by the previous point, but modifying and distributing the modified code must be allowed. The point does not say whether or not the modifications have to be released under the same license as the original software, but it does allow such a requirement too.

4. *Integrity of The Author's Source Code*

   This point permits the license to include a moderation of the previous point, requiring modifications and derived works to be clearly separated from the original author, for example by distributing them under a new name.

5. *No Discrimination Against Persons or Groups*

   By preventing discrimination against persons or groups the open source definition ensures that any open source license is in line with the wider philosophy of open source – openness and sharing. This point would for example prevent a license from limiting the use of the software to only one side of a heated debate, such as abortion clinics vs. abortion activists.

6. *No Discrimination Against Fields of Endeavor*

   This point is related to the previous, but focuses on the context of the software's usage. For example does this point ensure that the software can be used both in volunteer organizations as well as in businesses.

7. *Distribution of License*

   This point ensures that the license is of such a nature that it can be redistributed without requiring any modifications to be valid. This makes open source licenses easy to use, as the only action required of the user is to include a direct copy of the license with any new distributions.

8. *License Must Not Be Specific to a Product*

   This part of the open source definition exists to ensure that the license does not limit distribution to a specific software vendor. If the software has an open source license anyone can distribute it, so for example Company A can not stop Company B from making money selling the same software as Company A just because Company A was the first one to do so.

9. *License Must Not Restrict Other Software*

   This point prevents the license from restricting and regulating software written by other authors. This could for example happen if the software was released as part of a larger software package, where the license stated that all the software in such a package had to be open source software too. Such a restriction is not permitted by the Open Source Definition, so an open source license would have to allow distribution regardless of context.

10. *License Must Be Technology-Neutral*

    Finally, the last point relates to that of Distribution of License, ensuring that the license is transferable in any medium, and not be dependent on technologies such as digital signing, or on-line acceptance forms.

As we have seen in the points above the Open Source Definition covers most issues that could affect the free and open distribution of software. The website of the Open Source Initiative lists a whooping 58 licenses that conforms to the OSD (OpenSource.org, 2006b) – but only a handful of these are for general use. Figure 2.5 bellow lists some of the more popular licenses:

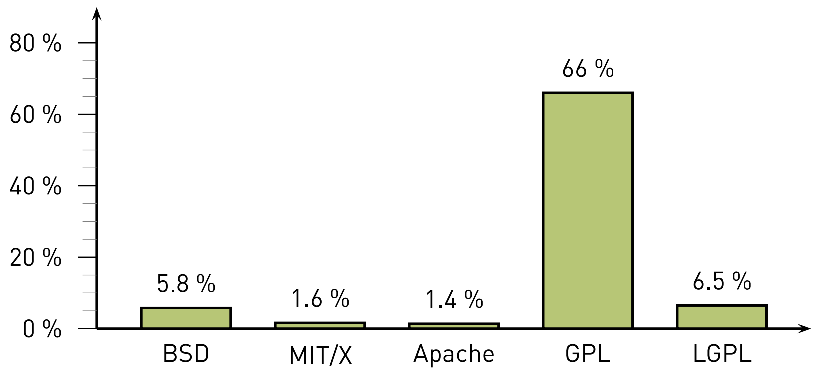


Figure 2.5: Distribution of licenses at Freshmeat (Freshmeat.net, 2007)

We will now take a closer look at these licenses, and see how they compare to the Open Source Definition, starting with the most basic ones.

2.2.3 The BSD, MIT/X and Apache Licenses

The BSD license was one of the earliest open source licenses. Although created long before the Open Source Definition, it does qualify as open source software. The license was originally used for the Berkeley Software Distribution – a UNIX distribution created by the University of California, Berkeley – but is has later been used in numerous other projects and is still widely used today.

One of the reasons for its success is ironically the relaxed attitude towards proprietary software. The license allows people to incorporate the code in proprietary projects, and does not require that derived works be distributed as open source (in accordance with point 3 of the OSD). In effect, code licensed under a BSD-license can go "closed-source" at any time.

One example of this is how the TCP/IP-stack in Microsoft Windows is based on Berkeley UNIX, but without any of Microsoft's improvements leaking back to the open source world. The laxed attitude of the BSD license may seem puzzling, as sharing is part of the spirit of open source, but must be seen in light of how Berkeley UNIX was developed: as a research project with hopes to be commercialized, and funded by U.S. Government grants.

The original BSD license had a clause that required any advertisements for software licensed under the BSD license to include the text "This product includes software developed by the University of California, Lawrence Berkeley Laboratory". This may have made sense for Berkeley UNIX, but not for other projects. On top of that people started adding their own version of this clause, causing a nightmare every time you wanted to advertise something that *maybe* included some BSD-licensed software. As a result the clause was removed in an amendment in 1999.

The MIT/X-Windows license is very similar to the revised version of the BSD license (with no advertisement clause), and is as basic as you can get while still being open source. The license doesn't even include a statement of non-attribution (point 4 in the OSD), meaning that the original author's name can be used to promote derived works.

The Apache License on the other hand does include such a non-attribution clause, and also requires the original author to be credited, but is otherwise virtually identical to the revised BSD license.

2.2.4 The GNU GPL and LGPL

If the BSD license is relaxed and business-minded then the GPL is its strict and ideological cousin, who does not see kindly on mixing proprietary and open source software. The GNU General Public License (GPL) was drafted by Richard Stallman and the Free Software Foundation to be used for the GNU project and other free software projects, and is the most widely used open source license today (see Figure 2.5 on the opposite page).

The main point separating the BSD-family and the GPL-family is that the GPL requires *all* derived works to be licensed under the GPL too, and does not permit the mixing of GPL-code and proprietary code. This is why the GPL is sometimes described as viral, because it forces anyone who modify or build upon GPL-licensed code to also license their software under the GPL if they release it. The purpose is to keep software free – forever – something that it has succeeded in doing for several decades.

The GPL also has a half brother named the GNU *Lesser* General Public License (LGPL). This license was originally named the Library GPL, but was renamed because of confusion over its purpose. It differs from the GPL in one important point, and that is that it allows static and dynamic *linking* with proprietary software. What this means is that proprietary software can build upon code released under the LGPL – but only by *referencing* it in its original form. Any changes to the LGPL'ed code will trigger the part of the license that requires redistribution under the same license.

This loop-hole might seem odd coming from the Free Software camp, but the rationale makes perfectly sense. If a free platform (such as Linux) has a library which provides functionality that is not available on proprietary platforms like Windows, then the library is a strong incentive to switch operating system, and should be licensed under the GPL. But if the library is not unique, and only replicates something that already exists on competing proprietary platforms, then there is nothing to gain from keeping it strictly GPL. In that case there is more to gain from allowing proprietary software to use the library, drawing proprietary software to the platform which will fill the gaps until open source replacements can be made.

An example is Adobe Acrobat Reader. If the Linux system libraries (like the GNU C library) were GPL, Adobe would either have to open source their Acrobat software (not very likely), or not deliver Acrobat for Linux at all (which would be a disadvantage for Linux). Instead, since the C libraries are LGPL, Adobe can keep the source closed, and the Linux community can enjoy a PDF reader while working on their own open source version (Gnu.org, 1999).

Considering that both the GPL and the LGPL are strict on how the licensed code can be used and reused in derivative works it is interesting to note that it also limits what kind of code can be merged back into GPL-projects. If you hold the copyrights of the to-be-merged code you can of course re-license it as GPL and then include it in a GPL-project. But if the code is licensed under another open source license you might not always be able to combine the two without breaking one or both of the licenses.

That's where the notion of *GPL-compatibility* comes into play. Only a limited number of open source licenses are compatible with the GPL, and it is generally recommended to not use non-compatible licenses because it makes merging code harder. One example of a license which is not GPL-compatible is the Apache license, due to some fine print about patents. Another is the original BSD license, which conflicts on the point of the advertisement clause.

A summary of the five licenses discussed in this section can be found in Table 2.3 below. The first three – the BSD, MIT/X, and Apache licenses – are often grouper together under the name *permissive* licenses, because they all allow combining with proprietary software, and changes can be kept closed source. The latter two – the GPL and LGPL – are known as *restrictive* licenses, because they are rather strict about how the code is re-used.

Requirement/characteristic	BSD	MIT/X	Apache	GPL	LGPL
Can be combined with proprietary software	✓	✓	✓	–	✓
Changes to the source must retain same license	–	–	–	✓	✓
Must credit the original author's work	–	–	✓	–	–
Protects integrity of original author	✓	–	✓	–	–
Can be combined with GPL software	✓	✓	–	✓	✓

Table 2.3: Comparison of OSI-approved open source licenses. Note that only the modified BSD license is GPL compatible, and that the LGPL only allows combining with proprietary software through linking.

2.2.5 Software Patents

The patent system was originally created in Italy, back in the late 15th century, to accommodate mechanical inventions. It then spread to England and further on through colonization. The United States got their Patent Act in 1790, and has since been one of the most aggressive patent regimes in the world. A patent is only governed by national laws, so companies usually register patents with multiple patent offices.

Once a patent has been approved it is enforced through civil lawsuits, and these can get fairly messy. The owner of the patent usually seeks monetary compensation for the infringement, but also has to prove to the court that the patent was in fact patentable in the fist place. The infringer may point to *prior art*, showing how the invention was not novel when patented, or argue that the patent is too obvious, or not even patentable subject matter (scientific theories, mathematical methods).

Another way to fight a law suit is to bring up counter-patents that the original patent owner might infringe on themselves. Such a deadlock situation usually results in a settlement where both companies agrees to not sue each other. This practice has led to what can be best described as patent-harvesting – an arms race where companies patent anything and everything (and even buy patents from other companies) as to have a large 'portfolio' of patents to use for counter suits in case they are sued themselves (Fogel, 2005). Many companies also use these portfolios to bully smaller companies, who don't have the resources to register and keep track of possible patent infringements. Such tactics can hardly be described as good sportsmanship, and far from the spirit of the original patent system.

One area where this situation is particularly noticeable is in the software field, where software patents are a big controversy. Many would say that software inventions falls under non-patentable subject matter, along with scientific theories and mathematical methods, but the United States and many other countries allow software patents. This has resulted in patents such as the infamous Amazon ‘1-click patent’, where Amazon got the exclusive right to let their customers make purchases with a single click. This may seem too obvious a technique for patenting, but it was nevertheless granted, and 20 days later Amazon filed a lawsuit against Barnes and Noble, one of their biggest competitors. After a couple of rounds in the courts Barnes and Noble was found to be infringing on the patent, and had to remove their “Express Lane” shopping cart. The ruling spawned a massive boycott of Amazon, led by the Free Software Foundation, and it became a symbol of how damaging software patents can be. Interestingly, Paul Barton-Davis, one of Amazon’s founding programmers, has later referred to the patent as “a cynical and ungrateful use of an extremely obvious technology.” (Equalarea.com, 1996).

Today, 8 years later, software patents are as controversial as ever, especially due to the patent-debate in Europe. So far the EU has been reluctant to grant software patents, and for example rejected the Amazon 1-click patent, but they have granted other similar patents, and there are strong forces lobbying for opening up the legislation. There are also interest groups who work against the proposed changes, arguing that software patents will favor large corporations over the smaller, and pointing to the danger of granting patents that are valid for 20 years in such a fast-moving field (NoSoftwarePatents.com, 2007). The vast majority of programmers, open source or not, are also against software patents and think they should be abolished (Burton, 1996).

The many problems with software patents can also be felt on the open source community. In contrast to copyrights and trademarks, which can be solved by rewriting the implementation or changing the name, patents are so general that when faced with a lawsuit the only realistic option is to remove the feature all together. Fighting back using expensive lawyers is not really an option for the vast majority of open source projects. This has caused projects to steer away from known patented algorithms, even if these are the only way to solve a problem. The situation is also preventing hardware vendors from opening up their drivers, which would benefit the Linux community, because they are afraid that competitors will find reasons, if if ever so minor, to sue them for patent infringements.

The open source community has tried to fight back by including clauses in the licenses that discourages authors to patent their inventions. One example is the GPL, which has a clause that explicitly states that patent infringements can not invalidate the terms and obligations of the GPL. This means that any contradictions can only be solved by not distributing the software anymore. The Apache license takes this further by requiring that the distributor gives an amnesty for any patents they hold which can be covered by the code. The interesting part is that if for some reason company A decides to break that amnesty by suing company B, their own amnesty from other authors is automatically revoked, meaning that they would expose themselves from lawsuits from company C and D. Similar tricks are used in the revised GPL, version 3.0, but as the license is currently only a draft we have yet to see the effects of this change.

The future of software patents and open source seems uncertain – especially thanks to the FUD of big players like Microsoft, who recently made a statement saying that anyone using Linux was bound to be infringing one some kind of Microsoft patent. This has provoked the Linux community to launch the campaign "Show us the Code", pressing Microsoft to step up and detail the patent infringements. So far Microsoft has ignored the challenge, but unfortunately for OSS-proponents this particular incident is most likely a sign of things to come.

2.3 Examples of Open Source Projects

THERE ARE literally thousands of open source projects, and more are started every day. The open source website Freshmeat lists over 40 000 active projects, and the mother of all open source repositories SourceForge hosts a whooping 162 855 projects (SourceForge.net). This includes everything from large user applications and operating systems to small tools, utilities and libraries. Many of them are Linux centric, and make up the Linux operating system, while others are cross platform, or written for Windows or Mac specifically. To gain a better understanding of the wide range of applications available I will now present three well known examples.

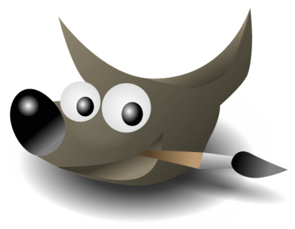
2.3.1 Linux

The Linux operating system is by far the best known open source project today, but talking about Linux as one entity is imprecise at best. The project actually consists of thousands of sub-projects and packages with varying licenses (most of them open source) – together forming what is known as a Linux *distribution*. At the heart of any Linux distribution is the Linux *kernel*, which is the part that does all the low level gritty stuff like managing process time and allocating memory. This is the part that Linus Torvalds originally created, and any reference to Linux as an operating system usually means the Linux kernel plus a variety of user applications.

As described in Section 2.1.2 Torvalds created the Linux kernel in 1991 as a hobby project, because he wanted to run something similar to the university SunOS computers at his home workstation. Version 0.01 was released in September 1991, and by December it had reached version 0.10. The kernel was far from being usable for mission-critical applications like web-servers – but it worked. As more and more developers joined his efforts the project grew fast, and in March 1994 they released version 1.0 – a complete replacement for UNIX.

Because of the increased attention following the frequent releases Linus Torvalds had to accept some criticism for his choice of architecture – among others from the creator of Minix, Andrew Tannenbaum. The argument was that a monolithic kernel (one big piece) was outdated and would not work very well. Many would say time has proven this wrong. Today millions of computers are running Linux (Li.org, 2007), and the platform has become a multi-billion dollar industry. Big businesses are investing heavily in hardware infrastructure running Linux; desktop users are catching on, and even my cell phone runs Linux. It's probably safe to say that Linux has much of the credit for the success of open source.

2.3.2 The GIMP



Another popular open source program is the photo and image editor The GIMP – short for General Image Manipulation Program. Image editors are used both by photographers for digital dark room editing and by graphic artists for creating original art or touching up existing works. The industry work-horse in this field has always been the proprietary software package Adobe Photoshop, but to this day the only supported platforms have been Mac and Windows. This is where The GIMP tries to provide an alternative.

The project was actually started as a last minute delivery in a course in compiler technique. The authors, Peter Mattis and Spencer Kimball, who at the time were attending Berkeley, wanted to do something fun instead of the boring class exercises, so they started writing an imaged editor. As the project grew they leveraged other relevant courses as an excuse to add new features, and nine months later, in February 1996, they released the first public version. The release was accompanied by a public mailing list, which helped spawn a massive user following. One of these users was Larry Ewing, who used The GIMP to draw the now famous Linux penguin 'logo' depicted in the previous section.

The initial release of GIMP used the commercial widget toolkit MOTIF, so the first thing Peter and Spencer focused on after the release was replacing this with a free toolkit. Since they didn't find anything suitable at the time they decided to create their own, and ended up with what is now known as GTK+ – a toolkit that forms the base of not only The GIMP but also huge software packages like the desktop environment GNOME. Version 1.0 of The GIMP was released in June 1998, utilizing the new GTK+ toolkit, and sporting advanced features such as user macros and a plug-in architecture. Although the original authors were no longer actively involved in the project it had grown enough for other developers to step in and take over.

Today The GIMP is providing a very solid alternative to commercial alternatives, and has placed itself as the *de facto* image editor on Linux. It also works on both Windows and Mac thanks to the cross-platform support in GTK+.

2.3.3 Eclipse

Eclipse is an integrated development environment (IDE) built on the Java platform. The purpose of an IDE is to integrate the various tools used when developing software, such as text editors, compilers, and debuggers – relieving the programmer from the endless switching of contexts. The software also usually provides added functionality such as a GUI-designer, code browsing, and version control. The early IDEs were mostly language specific, but recently the trend has moved towards multi-language generic environments such as Microsoft Visual Studio, KDevelop and Eclipse.

The Eclipse platform has a quite different story than the two previous examples. Unlike Linux and The GIMP, Eclipse did not start out as a pet project by bored university students. It was actually founded as a proprietary project in 1998 by a large corporation – IBM – as a way to help out customers who were frustrated by the cohesive sets of tools provided by the company. IBM set out to create a commercial IDE that could compete with the marked leader, Microsoft Visual Studio[4], and their tactic was simple; create a Java IDE better then all the others, and then use that to attract customers to the general platform.

But their plan did not work out as well as planned. The platform was immature, so customers were reluctant to switch and invest in something they did not know. As a result IBM decided in November 2001 to open source Eclipse – to increase exposure and accelerate adoption. They created a consortium of eight commercial software vendors, led by IBM, who all agreed to use, promote, and build products on top of Eclipse. This worked well for some time, but due to the dominating role of IBM the model was changed three years later to an independent foundation. The Eclipse Foundation was formed in 2004, and soon after they released Eclipse 3.0

Today Eclipse is considered one of the best Java IDEs out there. It also has solid support for languages like C^{++}, Python, PHP, and Ruby, technologies like Web services and embedded development, and thanks to the plug-in architecture and its open source license it is fairly easy to leverage the existing infrastructure to build new tools. This thesis was written using the open source LaTeX editor TeXlipse, which runs on top of the generic Eclipse platform. Eclipse also maintains its strong business focus through the Eclipse Foundation, which today consists of 18 member organizations. Each member has committed to provide at least eight full-time employers working on Ecipse, and some heavy annually funding. This focus has also affected how the Eclipse platform is developed, as most of the contributers are employees of member organizations. The project has received some flack for its bureaucratic and rigid development process (compared to more loosely coupled project), but judging from the results this seems to work well for Eclipse.

[4] The name Eclipse is said to mean "To Eclipse Visual Studio"

2.4 Open Source and Motivation

Back in the 60s and 70s the average free software hacker was usually employed in a scientific institution like a university or a research lab, and wrote and shared code because there was no other way. Today the situation is a bit different. We will now look at what kind of people participate in open source projects, and what motivates them to share both their time and the code they produce without any direct monetary rewards.

2.4.1 Who Participates

There has not been any direct large-scale studies on the demographics of open source participation, but we can use other related studies to get an understanding of who all these nice people are.

Hertel et al. (2003) did a study of contributers to the Linux kernel, and found that an overwhelming majority of the respondents were males of Western backgrounds (Europe, The United States and Australia), with an average age of 30. Similar results were found in a study by Lakhani and Wolf (2005), which also showed that most of the respondents had a solid background in IT – typically as computer science students or as professional software developers. These results coincide with an earlier study by Hars and Ou (2001), where the respondents were male (95%), in their 20s (50%), and almost half of them were working as professional programmers.

In the past few years business involvement in open source projects has grown significantly, and it is not unusual to have paid employees working full time on an open source project. This is also reflected in the studies mentioned earlier. For example did Lakhani and Wolf (2005) find that almost 40% of the respondents were either paid directly for their work or were allowed by their boss to work on a project during their regular working hours.

2.4.2 What Makes People Participate

The key question is then why people devote their precious time to contribute to open source projects, or as phrased by Lerner and Tirole "Why do top-notch programmers choose to write code that is released for free?" (Lerner and Tirole, 2001, p. 821). The classic stereotype depicts these programmers as altruistic and kind souls, who improve our software without asking for anything back, and surely many would like to fit that stereotype. Even though the reality of the situation is perhaps not that far off, it's still interesting to reveal some of the deeper motivations and mechanisms of open source development.

As noted in the previous section there is a growing interest from the IT industry to participate in open source projects. Such involvement has been studied in depth (Dahlander and Magnusson, 2005; Røsdal and Hauge, 2006), and the involvement usually stems from either heavy reliance on open source IT infrastructure, or because the company is providing/selling solutions based on free software (Lakhani and Wolf, 2005). I will not go into the details of business motivations here, but rather focus on motivation of individual developers doing volunteer work. We will however return to the business side of things in Section 2.6.

Raymond (1999b) was one of the first to reflect on the wider issue of motivation, pointing out the striking similarity to what anthropologist describe as *gift cultures*. In these cultures there is an abundance of resources, leading to an economy not based on exchange of money or commodities, but rather the giving of gifts. The act of gift-giving becomes a symbolic action used to define social structures, where part of the effect comes from the expectancy of a returned gift. This resemblance has been developed further by Bergquist and Ljungberg (2001), who argue that there is a notable difference between gift-giving in primitive cultures and that of open source development. They also downplay the dependencies created by gift-giving, and focus more on the importance of socializing new contributers to the gift culture – especially the process of peer reviewing.

When discussing human motivations in psychology it is common to distinguish between intrinsic and extrinsic motivations (Aronson et al., 2001). Acting on *intrinsic* motivations is doing something for its own sake – ie. for its inherent satisfactions – for example because it is "fun" and "makes you feel good". This type of motivation is linked to the human need to feel competent and fulfilled, and includes doing something good for a community (due to socialized norms).

The other type of motivation – *extrinsic* – is rooted in external incentives for behaviour, with the basic idea that if the benefits exceeds the costs then the behaviour will follow naturally. Here we find direct payoffs such as financial gains and increased utility value of the application, but also delayed rewards such as possible career advancements and improved skills.

Using these two distinctions as a rough basis we can summarize some empirical studies of open source development. Raymond does agree that the metaphor of a gift culture does not fully explain the mechanisms of open source development (Raymond, 1999b, pg. 82), so having empirical data to enlighten the situation is of great help.

One such study is Hars and Ou (2001), who did a web survey of participants in various open source projects – 389 persons in total. The authors found that intrinsic motivations like the joy of programming and identification with a community were outweighed by extrinsic motivations – especially that of improving a software product for own personal use. Another important factor was building human capital, such as knowledge, for future monetary rewards. In a study of the Linux kernel the participants reported a strong desire to be identified with the kernel project and the wider Linux community, as well as wanting to improve their own software (Hertel et al., 2003).

This mix of intrinsic and extrinsic motivations was also found in a recent study by Lakhani and Wolf (2005), who followed the same approach as Hars and Ou

(2001), using cross-project surveys in multiple iterations. They found that the single most important determinant of project participation (hours per week) was the sense of personal creativity and joy of hacking. As any programmer will tell you, being "in the zone" can in fact be highly creative and easily make you lose track of time[5], which should be an obvious motivation factor for project participation. Adding to this were factors such as improving programming skills, giving something back to the community, and identifying with the philosophy of FLOSS. Lakhani and Wolf also reported that monetary payment were a factor, but surprisingly they found no negative impact on intrinsic motivations from also having extrinsic motivations. This is usually the case in experimental settings (for example getting paid to do a task will make you enjoy the task less), but not in the survey results. Being paid to do FLOSS development can in fact be just as creative and joyous as volunteer work.

As we have seen there are many varying motives for participating in open source projects – some of which are summarized in Table 2.4 to the right. Lakhani and Wolf describes FLOSS development as "a creative exercise leading to useful output, where the creativity is a lead driver of individual effort". This fits well with what von Hippel and von Krogh refer to as the "private-collective" innovation model – which is a merge of the classic "private investment" model of intellectual property protection and the "collective action" model of collaboration and sharing (von Hippel and von Krogh, 2003). The volunteer participants invest their time and resources in OSS projects for personal enjoyment and to solve their own software problems, while at the same time sharing their innovations with the open source community. This is of course only one way to look at it, but illustrates how open source development can be seen for many different perspectives.

Type	Motivational Factor
Intrinsic	Joy of programming
	Community identification
	Community obligations
	Feelings of accomplishment
Extrinsic	Need for software improvements
	Monetary rewards
	Self-marketing
	Improved skills
	Professional recognition

Table 2.4: Motivational factors in FLOSS

2.5 Characteristics of Open Source Development

WHEN LOOKING at the history of open source as presented in Section 2.1 – with its rebels and free thinkers – and the seemingly complex motivations of participation in such projects – from the simple joy of programming to monetary rewards – and all this combined with a myriad of licenses and legal fine-print, it is only fitting to wonder how this can result in the argueably successful projects presented in Section 2.3. What is it with open source development that unites all these elements into developer-communities that can produce solid software packages like Linux?

[5] As a programmer myself I can testify to that. Nothing beats pulling an all-nighter and fixing a particularly hairy bug or adding a much needed feature.

Eric Raymond used his landmark essay *The Cathedral and The Bazaar* to describe how open source has more to it than just great visions and ideology. It is also a viable software development model – with many advantages over traditional software engineering practices (Raymond, 1999a). Raymond playfully contrasted the two by comparing traditional proprietary in-house development to building cathedrals – "fully crafted by individual wizards or small bands of mages working in splendid isolation" – and open source development to swarming bazaars – "babbling of differing agendas and approaches" (Raymond, 1999a, p. 21). Although often cited, this analogy has received its fair share of criticisms, and many authors have argued that open source development practices are really not that different from traditional software engineering (Fuggetta, 2003, 2004; Fitzgerald, 2006). This has probably become even more true in recent years – as business involvement has increased. But then again, even the original release of Mozilla was financially motivated and commercially supported.

Still, there are some characteristics of open source development that are – if not unique – then at least peculiar enough to warrant further study. I will now present some of these characteristics, using Raymond's original arguments as a basis for the discussion.

2.5.1 Openness and Joyful Hacking

It might seem obvious, but the single most important condition for open source development is openness and transparency – starting with the source code itself. It is the foundation for experimentation and innovation, and it is not unusual to find that the source is also the primary basis for technical discussion (as opposed to requirement specifications or diagrams). The openness often extends to other areas of the development too, such as design documents and communications logs, as they are expected to be easily accessible for project members. The focus on openness has led many people to believe that open source projects are less secure (as adversaries can scrutinize the code for possible attack vectors) but this fallacy has been debunked many times (Witten et al., 2001; Di Giacomo, 2005; Hoepman and Jacobs, 2007). Making the source code openly available lets people *find and fix* more bugs – not exploit them – and the net effect is more secure software, at least in the long run.

One of the most used explanations for why OSS has been successful is that the developers not only *work* on the code but are also *users* of the application. Raymond states that "Every good software starts with a developer's personal itch" (Raymond, 1999a), and Paul Vixie describes open source projects as "a labor of love" (Dibona et al., 1999). Both of these statements point to the fact that having developers who are highly motivated (either because something is annoying them or because they are inspired), and who also can see the problem from a user's standpoint, is a good step towards a successful application. This can be traced back to the motivational factors of Section 2.4, where intrinsic motivations like the joy of programming was combined with the extrinsic motivation of improving an application you use yourself.

2.5.2 Leadership and Organization

With Raymond's portrait of the OSS bazaar development model as "babbling of differing agendas and approaches" it is only natural to wonder how all these voices can result in concise and solid systems such as the Linux kernel or the Apache web server. At some level there has to be a mechanism that unites the voices and keeps the direction stable. In most OSS projects this is typically a charismatic leader, such as Linus Torvalds, or alternatively a committee such as the Apache Foundation. In either case the founder of the project usually has a dominant role in deciding the direction and high level goals of the project.

Mockus et al. (2002) did a study of two successful open source projects, Apache and Mozilla, and found that the core developers in these projects were responsible for 80% of the code produced dealing with new features. They also had the largest influence on what made it into the code base. Adding to this core was a group (about 10 times as big as the core) of developers dealing mostly with fixing bugs, and then a group of people (100 times the size of the core group) doing bug reporting.

This symbiotic relationship between core developers and more peripheral developers is a strong determinant of a project's success, as a lone core group focusing only on adding new features will have a hard time fixing enough bugs to make the product stable and usable for others. It also makes it easier for people to join and contribute to OSS projects - starting out with reporting bugs and fixing small problems and then later progressing to more advanced tasks as they rise in ranks and experience. The implicit 'rituals' that users have to go through to become project developers are known as *joining scripts* (von Krogh et al., 2003), and they typically involve getting to know the culture of the project and the day-to-day activities.

Having a leader or committee is of course no guarantee for keeping a project on a steady course. Both developers and users must respect the leadership, and feel that their opinions and contributions are valued and taken seriously. Without that confidence the motivation for reporting bugs or making patches is diminished, as there is a good chance that the work you do will be ignored and in vain. Sometimes disagreements between strong personalities in a project with conflicting goals result in a so called *fork*, where one of the developers starts a new project with a new name, but based on the same code as the original project.

A possible consequence of a fork is that one or both of the projects die out. This may happen when they loose too many peripheral developers and users – as they get tired of choosing sides – or because the fork takes time away from development. These obvious negative effects of forking is a strong incentive to avoid forking at all costs, and it is usually frowned upon unless there was a valid reason for the fork. The Linux project, which in theory is vulnerable to fragmentation and forking (because of its size and many facets) has solved this by actively supporting sub-projects while still keeping them under the common Linux umbrella to strengthen the community feeling and branding.

2.5.3 Coordination and Cooperation

We have seen that FLOSS projects have a 'guiding star', to ensure a common direction. But with the source code readily available for anyone to hack on, what is preventing developers from stepping on each others toes and tearing down the code others have built? Coordinating a large number of developers like this would normally be subject to Brook's law (Brooks, 1995), so how do they avoid using all their time on making sure everyone is up to sync? Gutwin et al. (2004) showed that there is indeed a high degree of coordination in open source projects, but that it has a large implicit element. Coordination is maintained through group awareness – of everything from who's who and what they are working on at the moment to future plans and who is knowledgeable about a certain topic or technology.

Maintaining group awareness in distributed work-situations is usually accomplished through mechanisms such as *explicit communication*, i.e telling someone directly what you plan to do, *observation*, where you learn what others are doing by observing discussions or actual work, and *feedthrough*, where you form a picture of the current situation based on work that's already been done.

Gutwin et al. showed that even though OSS projects are geographically distributed they do not usually employ strict partitioning of the code base. This may come as a surprise, as it makes coordination even more critical, but must be seen in light of the agile nature of open source development, where anyone should be able to fix a bug, no matter which module it is located in.

They also discovered that the usage of e-mail lists, chat, and CVS-logs was central to the day to day awareness of the project participants. Not only was the e-mail lists used for explicit communication such as announcements and direct questions, but they were also an important source for learning through observation. Discussions on the list gave a good overview of who was working on what – not only for the people directly involved in the discussion but also for the people 'watching' from the side-line – and this often happened as a byproduct of the real subject of the discussion, for example a technical issue.

When messages are discussions rather than announcements, awareness information comes along as an unintended byproduct of an activity that was occurring anyways.

So the awareness was basically "free", as a consequence of another required activity, namely fixing bugs. Also, changes to the CVS controlled code base was automatically sent out to a separate mailing list, and by observing the activity on that list the developers got the feedthrough effect discussed earlier. Finally, the lists made it very easy to ask questions because you didn't have to first figure out who was the expert in the area. You just posted a message to the list, and the relevant person would answer you.

2.5.4 Frequent Releases and Parallel Debugging

This quick feedback loop gained from using CVS-logs and e-mail lists brings us over to another important characteristic of open source development, namely the focus on short time-to-market and frequent releases – casually referred to as “release early, release often” (Raymond, 1999a). The idea is that instead of working for months or even years until you have a ‘bug-free’ and perfect product you instead give the users the choice of how cutting edge they want to be. They can choose to use the latest development version directly from the source code repository, or they can use the more stable weekly/monthly releases. By doing release-management this way you increase the chance of finding bugs because more users are exposed to the unfinished source code, and it also helps building a group of peripheral developers as described by Mockus et al. (2002).

This practice of frequent releases foster what is known as *parallel debugging*, or as coined by Raymond as Linus’s Law; “Given enough eyeballs, all bugs are shallow” (Raymond, 1999a, p. 30). What he refers to is the often observed situation of a user finding a bug, but not understanding why it is happening. The user will then pass the information over to the community, where sooner or later some other user is bound to realize why the bug is happening, just because of the sheer size of the community – and the bug gets fixed. Having thousands of user looking at and experimenting with the source code basically increases the number of code paths that are tested and debugged.

When the cause of a bug has been found, or a new feature is added, a *patch* is submitted to the mailing list describing how to fix the bug. The patch is usually in the form of one or more *diff-files*, describing only the incremental changes between the original code and the patched code. An example of a diff file is provided in Listing 2.1 below.

Listing 2.1: Example of diff file. Added lines are noted by a plus sign and removed lines with a minus sign. The surrounding lines are there for context and can be limited to for example three lines above and below changes.

```
--- main-old.c  Tue May  1 12:43:34 2007
+++ main-new.c  Tue May  1 12:47:28 2007
@@ -2,6 +2,6 @@

 int main(int argc, char* argv[])
 {
-    printf("Hello, World!\n");
+    printf("So Long, and Thanks for All the Fish!\n");
     return 0;
 }
```

This notation makes it easy to see what is added and removed, and thus allows more efficient quality assurance of the code. The quality assurance happens through a process called *peer reviewing*, where the patch is evaluated and improved by fellow developers on the mailing list. When the patch has been accepted by all developers it is finally committed into the common code base permanently.

The peer reviewing process is well known from science, where most journals and conferences are peer reviewed, but it also has a place in software-engeneering. Weinberg argued for looking over each others code as early as 1998, and in recent years this technique has been adopted by agile methods such as extreme programming (XP), where the technique *pair programming* involves co-locating two developers physically with only one computer and keyboard. This forces the developer who's not typing at the moment to inspect the other's code. The practice of quality assurance through peer reviewing is an important part of open source development, and we will get back to this in detail in Chapter 3.

2.6 Commercial Use of Open Source Software

THE gut reaction for any company to the principles of open source is usually that it has to be bad for business – at least this was the case in the early years. We will now look at how open source can be leveraged from a business-perspective – both for shrink-wrap-vendors and in-house development.

2.6.1 Business Models

Before considering business models for open source software it can be helpful to take a step back and look at how the traditional IT economy works. What most people associate with the software industry is run-of-the-mill over-the-counter shrink wrapped software packages like Microsoft Office or computer games. This model is based on selling *licenses* – not the software itself – giving the user basic rights such as running the software and receiving limited support. The customer will normally think that they own the software, but this is not the case. This model can of course be scaled up, resulting in what is known as enterprise software, where licensing fees can easily reach millions of dollars. In both cases the software is produced for its *sale value*, meaning that the vendor will receive direct, or indirect (through service agreements), monetary rewards for producing it.

Surprisingly the sale-value based revenue model only accounts for a small portion of the software written each year; conservative estimates say about 10% (Raymond, 1999c). The majority of software it actually written for its *use value*, for example to support a company's primary business domain. This kind of software is traditionally developed and maintained by in-house programmers or hired consultants, and usually only has one real user – the company that initiated the development. In this group we find everything from banking systems to web portals – plus the numerous enterprise resource management (ERM) and customer relations management (CRM) systems around the world.

At first it might seem counter-intuitive to give away intellectual property – especially for sell-value software where the revenue stream is largely dependent on making sales. Huge resources are put into developing software, so why sacrifice all that by sharing it? Proponents of free software may sometimes point to the philosophical issues, but there are also pragmatic reasons, and several business models for open source development has been suggested (Hecker, 1999; Raymond, 1999c; Karels, 2003).

I will now describe some possible business models, based on the models first described by Hecker (1999) in his essay *Setting up shop: The business of open-source software*. Hecker worked at Netscape Communications at the time, and played an important role in the release of Mozilla as open source. I will use examples from the open source industry to highlight success-cases, and relate the models to current trends in the open source business world.

Support Sellers This is the original free software business model, as advocated by Richard Stallman in the GNU Manifesto. It is based on selling support and training services for free software. It also includes selling distributions of free software on media such as CDs or DVDs. Stallman himself sold his editor Emacs on cassettes in the early Free Software Foundation days, and today Linux distributions like Red Hat and Suse are a huge marked – both for media distribution and for training and support. The first pure support services was Cygnus Solutions (now owned by Red Hat Inc.), who provided support for the popular GNU tools back in the 90s. They paved the way for consultant companies like Linpro and countless other Linux shops around the world. Interestingly this service-and-support model is very similar to how enterprise software is sold, where the licensing fee is often small compared to the costs of product support and services.

Loss Leader This business model is based on the classic supermarket trick of 'price-dumping', where a product is sold below cost to stimulate the customers to buy further items. In the software world this can be accomplished by releasing one or more of the company's products as open source (at no cost for the customer), hoping for increased sales of the products still sold using the traditional software model. This was how Netscape could release Mozilla as open source; because it helped them sell more copies of their other products. Stopping Microsoft's dominance in the web browser marked was of course an added and welcomed bonus.

A more recent example is IBM, who open-sourced Eclipse to increase interest for their development IDE WebSphere Studio (which is based on Eclipse). A variant of this model is what is known as *dual-licensing*, where a company produces software using the traditional in-house cathedral model, but then distributes the software under both a restrictive open-source license (usually the GPL) and a commercial license. The result is that companies who wish to use the software commercially without returning changes to the community will have to pay licensing fees, but those who want to develop in the spirit of open source can use the GPL version for free (Dibona et al., 2005). This quid-pro-quo approach to licensing has been successfully employed by companies such as Trolltech and eZ Systems, who both use the GPL for their open source version.

Widget Frosting The hardware parallel to the loss leader model is *widget frosting*, where hardware vendors use open source software to make their platform more interesting for customers. Vendors like IBM use this model to sell servers running Linux (pure frosting for the customers), and in recent years there has been a growing marked for Linux on embedded devices such as routers and cell phones. This allows companies to cut development costs – since using Linux frees them from coding custom proprietary drivers for their hardware – while at the same time being able to market their device as "coming with all the possibilities of Linux".

Accessorizing In this model the company does not directly participate in the development of open source software, but rather bases its revenue on selling accessories like books and stuffed Linux penguins. There is a huge marked for training material and manuals, and companies such as O'Reilly Media has made good money by educating the world in everything from Python programming to Linux administration and maintenance.

Service Enabler A business using the service enabler approach is leveraging open source tools and technologies to build an online service, where the revenue comes from advertising banners or though subscription fees. These companies build on the traditional LAMP stack (Linux, Apache, MySQL, PHP) and then extend it with domain-specific projects (both open and closed source) for their particular service. Examples range from blogging and online publishing software to the recently massively popular social-networking site Facebook, which has open sourced parts of its underlying technology.

When considering the preceding business models it is worth noting that most companies combine these models into their own hybrid model, for example by open-sourcing parts of their business as a loss-leader approach while still providing support and services for the open sourced parts. Krishnamurthy (2005) takes this into account by dividing businesses into distributors, software producers, and third-party service providers (but chooses to ignore hardware vendors), and this grouping may better reflect the current software landscape A study by Ghosh (2002) – based on a survey of over 2700 developers from open source projects – identified similar groupings, with distributors/retailers and open-source related service providers as the two main categories. A hierarical list of the findings is presented in Figure 2.6 below.

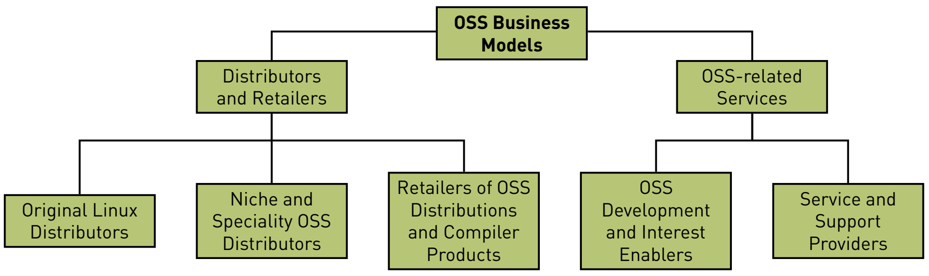

OSS Business Models
Distributors and Retailers
OSS-related Services
Original Linux Distributors
Niche and Speciality OSS Distributors
Retailers of OSS Distributions and Compiler Products
OSS Development and Interest Enablers
Service and Support Providers

Figure 2.6: Open source business models. Adapted from Ghosh (2002)

2.6.2 Utilizing Open Source

The business models presented in the previous section are all in one way or the other designed to compensate or replace sale-value revenue-streams – for example by trading shrink-wrap or enterprise software sales for services and support agreements for that software. But what about the huge chunk of in-house software that is developed each year? How does open source affect the business dynamics of software developed for its use-value rather than its sale-value? Eric Raymond answers this by stating that "only sale value is threatened by the shift from closed to open source; use value is not". He argues that the benefits of cost-sharing and risk-spreading that follows from developing use-value software as open source largely outweighs the possible risk of giving up some competitive advantage to other players in the field (Raymond, 1999c, pp. 128-129).

A more nuanced approach is taken by the on-going COSI project, which aims to understand how the recent trends towards heterogeneous and distributed development affects the software business (Røsdal and Hauge, 2006). They group in-house software into three basic categories based on the technological uniqueness of the software. The first category is *commodity* software, which is shared by companies across business domains. Classic examples are word processing software, salary systems, and accounting software. Then there's *basic-for-business* software, which is specific to a given business domain, but still shared by many companies within that domain. Finally there is *differentiating* software, which captures the uniqueness of a company and gives it a business-edge over its competitors. The three categories are illustrated by the vertical axis of Figure 2.7 below.

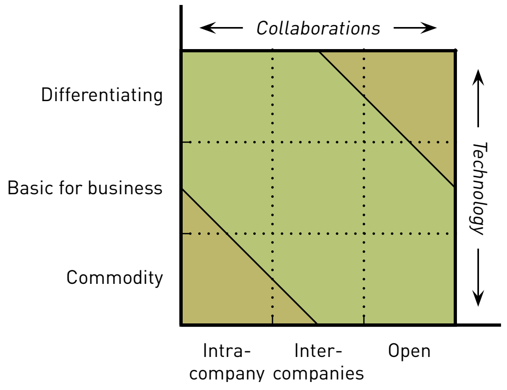


Figure 2.7: Classification of in-house software according to the COSI project (Itea-Cosi.org, 2007).

The rightmost column in the figure suggests that both commodity and basic-for-business software can be successfully developed as open source – because there is little or no risk of giving away competitive advantages. Differentiating software on the other hand should be heavily guarded, as this is the secret sauce that makes the company stand out in the crowd. So the worst thing a company could do is spend scarce resources on developing commodity software in-house (that could readily be acquired from other sources), while releasing differentiating software to competitors – as illustrated by the shaded corners of the figure. This can be seen as an argument for buying, in the build-vs-buy dilemma, and follows the growing trend of promoting software re-use not only within an organization but also across organizational borders.

Software re-use has of course always been an important and valued practice in the field, but it is only within the last decade that the use of so-called of-the-shelf (OTS) components has appeared. The most common type is commercial of-the-shelf (COTS) components – where the software is purchased from a commercial vendor – but the use of OSS components is also an option. The acquired components are then integrated into the final product though customizations and "glue-code". The rationale for using OTS components is that leveraging tried and tested solutions, rather then rolling one's own, will result in shorter time-to-market, higher reliability, and better performance (Voas, 1998b) – but there has also been reports of problems due to lack of support and commitment from the vendor, unanticipated defects, or missing features (Voas, 1998a). Nevertheless, the use of of-the-shelf components – both open and closed source – are very popular in the software industry today (Røsdal and Hauge, 2006).

The COSI report does not go into details of why OSS should be any better than commercial of-the-shelf components, but others have voiced opinions on the possible benefits of going open source (Raymond, 1999c). The most cited reason for using open source software as of the shelf components is that the source code is provided. It is argued that by having access to the source code many of the problems of COTS are negated, because you are no longer at the mercy of a commercial vendor who only provides you with a black box. Although this may sound like a solid argument there has also been evidence that many COTS components are delivered with source code too, and that while developers usually read the source code they seldom *modify* it (Conradi and Li, 2005). Combined with the fact that costs are comparable once training and support is added to the equation has led some authors to the conclusion that using open source software as of-the-shelf components is not really that different from commercial alternatives (Di Giacomo, 2005). This hypothesis is partially supported by the findings of a recent report by Li et al. (2006), where OSS and COTS components were used with similar expectations, and in projects of similar nature.

2.7 Summary

THIS CHAPTER has presented a broad overview of the open source phenomenon – from history, licensing and motivation to development practices and business models. Some of the points we have covered are:

- How open source grew from a lone hacker's dream of an ecosystem of free software to a multi billion dollar industry, and how Linux played an important part in this.

- The difference between copyright, patents and trademarks, and how open source projects use copyright to secure the rights of the user to run the software, modify it, and redistribute it.
- Examples of some well known open source projects – to get a better feel for the variety of software developed under the open source model.
- Who partiticpates in open source projects and why they do it – showing that open source developers are motivated by both intrinsic and extrinsic factors.
- How open source developers work to build software, and what separates this way of working from traditional software engineering.
- Various business models for open source software – showing that both for-sale and for-use businesses can gain from employing an open strategy.

Chapter 3

Software Quality

This chapter presents important concepts related to software quality in general and quality assurance in particular. It starts off by discussing various approaches to software quality applied by the field of software engineering, such as quality attributes and process improvement. Then it moves on to discuss peer-reviewing, both in science and in software development, before rounding off with how open source projects handle quality assurance

3.1 Perspectives and Approaches to Quality

TO SAY that the term quality can be fuzzy and vague is a huge understatement. Ask a person to assess the quality of a book or a piece of art on a scale from one to six you'll be surprised at how confident the answer is, but have the same person convince another person of *why* that is the case, and you'll see a totally different situation. While some may consider the craftsmanship and solid structure a defining factor of quality, others may point to the esthetics and fine lines of the design. Yet they both may conclude that the quality of the product is excellent. As expressed by Robert Pirsig in his landmark book *Zen and the Art of Motorcycle Maintenance*, "Quality is not objective. It doesn't reside in the material world [. . .] Quality is not subjective. It doesn't reside merely in the mind" (Pirsig, 1974).

The view of quality presented above is but one of many perspectives on what quality really means. Kitchenham and Pfleeger (1996) reported on the ideas first expressed by David Garvin, and listed five views on software quality. The *transcendental view*, as just examplified, sees quality as something that can be recognized, but not defined. The other perspectives are more concrete than that, and perhaps better suited for software quality. One of them is the *user view*, which sees quality in terms of how good it addresses the user's needs – so called "fitness for use". What good does it make that the software is reliable with an 99.9% up-time if the user doesn't understand or can't utilize it?

A third perspective is the *manufacturing view*, where conformance to specification is key. One way of achieving this conformance is by standardizing the manufacturing process, leading to a process-centric view on quality. We will get back to this perspective in Section 3.1.2. Next is the *product view*, which sees quality as inherent in the product, determined by its many attributes. This is the most common view in the software quality literature, along with the manufacturing-view, and we will investigate this further in Section 3.1.1 below. Finally there is the *value-based view*, which weighs costs and profits against each other, reasoning that the quality of a product is determined by what the customer is willing to pay for it (Kitchenham and Pfleeger, 1996). There has been some attempts to unite these five views by providing mappings between the various factors, but this has proven to be difficult (van Vliet, 2000).

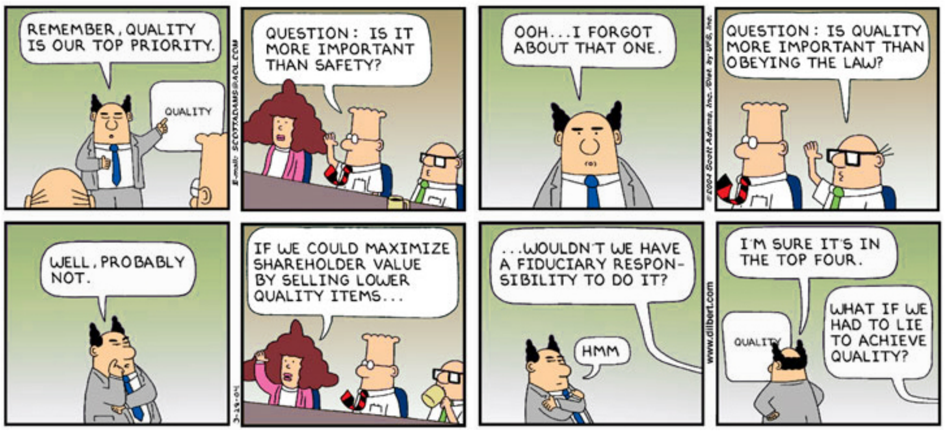

REMEMBER, QUALITY IS OUR TOP PRIORITY.
QUALITY
QUESTION: IS IT MORE IMPORTANT THAN SAFETY?
OOH...I FORGOT ABOUT THAT ONE.
QUESTION: IS QUALITY MORE IMPORTANT THAN OBEYING THE LAW?
WELL, PROBABLY NOT.
IF WE COULD MAXIMIZE SHAREHOLDER VALUE BY SELLING LOWER QUALITY ITEMS...
...WOULDN'T WE HAVE A FIDUCIARY RESPONSIBILITY TO DO IT?
HMM
I'M SURE IT'S IN THE TOP FOUR.
WHAT IF WE HAD TO LIE TO ACHIEVE QUALITY?
www.dilbert.com

Figure 3.1: Dilbert's pointy-haired boss realizes the complexity of quality.

After deciding on a fitting definition for software quality comes the problem of how to achieve it. If one settles on quality as the properties of a product then ensuring that these properties are met becomes the primary focus. On the other hand, one could reason that streamlining and standardizing the development process would result in a quality product[1]. This product vs. process dimension is complicated by the question of whether to conform or to improve. Making sure that a product or process complies with set standards is usually only the first step. One also have to look forward and figure out how to improve – to meet higher standards. These two dimensions are illustrated in Table 3.1 below.

	Conformance	**Improvement**
Product	McCall, ISO 9126	"best practices"
Process	ISO 9001	CMM, SPICE

Table 3.1: Approaches to quality (van Vliet, 2000).

Product conformance is usually achieved through the use of quality models and software metrics. This is the primary topic of Section 3.1.1. Another approach is

[1] Or perhaps just *uniformly mediocre* products – as noted by Kitchenham and Pfleeger (1996)

process conformance and improvement, which is discussed in Section 3.1.2. The cell named "best practices" includes the many software engineering techniques like object-orientation (OO), RUP, and pair programming – but these will not be discussed further in this thesis.

Attempts to unify these four approaches has been made by the strategy Total Quality Management (TQM), which advocates focus on quality in all parts of the organization. The philosophy was created in the late 20s by the American scientist Edward Deming, but didn't initially receive much attention. Only after the Japanese successfully adopted the philosophy during the after-war years did it get its recognition in the west.

While the name may sound like a middle-manager's wet dream – straight out of Dilbert or the movie Office Space – its message is not all that fat fetched. It aims for continuous improvement; sees human resources as valuable assets, not costs; and stresses the importance of the customer in the development process (van Vliet, 2000). This happens through a four-step process.

The first step, called *kaizen* (改善) is to develop a process that is measurable and reproducible. Then comes *atarimae hinshitsu* which tries to identify organizational issues that could hinder improved quality. Third is the *kansei* (歓声) step, which aims to understand how the user applies the product. By learning more about the user follows improved quality in both product and process[2]. Finally comes the *miryokuteki hinshitsu* step – targeted at broadening the management team's perception of the market (Pressman, 2001).

3.1.1 Quality Attributes

As noted in the previous section the dominant perspectives of the software engineering field has long been a combination of the product and manufacturing views – one focusing on how to define and measure quality factors and the other taking care of standardizing and improving the process. In addition there has been an increased attention in recent years towards the user-view, for example through influence from the HCI field. We will now investigate the primary topic of product-based software quality, namely *quality attributes*.

Research on quality attributes started in the late 70s, and is still on-going. The primary goal is to identify and define the factors that relate to software quality, so they then can be measured to determine the quality of a system. Unfortunately this has not been an easy job. No definition seems to be perfect, and measuring the individual factors quantitatively is in most cases impossible (van Vliet, 2000). Yet, there are no lack of taxonomies trying to refine and improve the picture.

[2] Story has it that a Japanese dish washing machine manufacturer was getting a large number of returns from customers in rural districts. After closer inspection it turned out that the farmers not only used the washing machines to clean their dishes but also to wash potatoes! The company could of course react by putting some kind of print in the manual saying "Not suited for potato-washing", but instead they decided to fix the washers to handle potatoes. This is a good example of *kansei* – satisfiying the customer's needs.

One of the earliest taxonomies was the McCall model, as described by McCall et al. (1977). The model is hierarchical, with the top level consisting of high level attributes such as reliability, efficiency, usability, testability and portability. These attributes, known as *quality factors*, are only external abstractions of internal properties of the system, so they have to be measured indirectly. This is done through the second level attributes, called *quality criteria*, which are then mapped to the individually measurable *quality metrics*. Each quality factor is a aggregate of one or more quality criteria, so by measuring and combining the relevant quality metrics one can infer the quality of a given factor. For example, the efficiency factor is dependent on both execution efficiency and storage efficiency, while the correctness factor is based on the three criteria completeness, consistency and traceability. In total there are 11 quality factors in the McCall model, which are mapped in various ways to 23 quality metrics. The factors are also categorized into three classes, based on the product phase they target, as depicted in Figure 3.2 below.

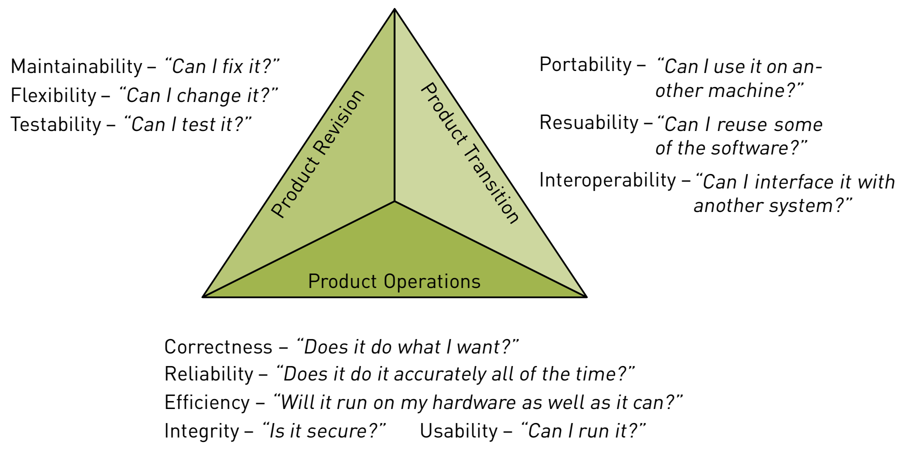

Maintainability – "Can I fix it?"
Flexibility – "Can I change it?"
Testability – "Can I test it?"
Product Revision
Product Transition
Portability – "Can I use it on another machine?"
Resuability – "Can I reuse some of the software?"
Interoperability – "Can I interface it with another system?"
Product Operations
Correctness – "Does it do what I want?"
Reliability – "Does it do it accurately all of the time?"
Efficiency – "Will it run on my hardware as well as it can?"
Integrity – "Is it secure?"
Usability – "Can I run it?"

Figure 3.2: Software Quality Factors in the McCall model. Adapted from Cavano and McCall (1978).

While the McCall model captures many of the concepts that are still used today it is not perfect, and has received its fair share of criticism (van Vliet, 2000). One of the problems is that many of the low level metrics are very subjective, which makes them hard to quantify. Another is that some metrics are binary – answered by a yes or no – which limits the richness and nuances of the model. A good example is the metric "Is all documentation structured and written clearly and simply such that procedures, functions, algorithms and so forth can easily be understood?", which exhibits both these problems (Kitchenham and Pfleeger, 1996). These problems limit how suitable the model is across projects and between teams. A third critique is that most of the factors can only be assessed after the software has been completed, so they can not help companies before development starts, or during development (van Vliet, 2000).

A more recent quality taxonomy is the ISO 9126 standard, developed by the International Organization for Standardization (ISO.org, 1991). It defines six *quality characteristics*, which are further decomposed into *sub-characteristics*. Each top

Characteristic	Description	Sub-characteristics
Functionality	The capability of the software to provide functions which meet stated and implied needs when the software is used under specified conditions	• Suitability • Accuracy • Interoperability • Security
Reliability	The capability of the software to maitain the level of performance of the system when used under specified conditions	• Maturity • Fault tolerance • Recoverability
Usability	The capability of the software to be understood, learned, used and liked by the user, when used under specified conditions	• Understandability • Learnability • Operability • Attractiveness
Efficiency	The capability of the software to provide the required performance, relative to the amount of resources used, under stated conditions	• Time behavior • Resource utilization
Maintainability	The capability of the software to be modified. Modifications may include corrections, improvements or adaption of the software to changes in environment, and in requirements and functional specifications	• Analyzability • Changeability • Stability • Testability
Portability	The capability of the software to be transferred from one environment to another	• Adaptability • Installability • Co-existence • Replaceability

Table 3.2: ISO 9126 quality characteristics and sub-characteristics.

level characteristic has several sub-characteristics, but the individual sub-characteristics are only part of one characteristic – to avoid overlap. Like the McCall model it focuses on attributes and qualities of the software product, not the development process. The full list of characteristics are presented in Table 3.2 above.

Although the McCall model and the ISO 9126 standard use different names for their concepts (quality factors vs. quality characteristics) they are in large part similar. The main differences are that the ISO 9126 standard does not have any overlap between the attributes, and that the top level characteristics are expressed as externally visible properties of the system – in line with a user-view rather than a product view.

Kitchenham and Pfleeger (1996) points to some problems that are common to both taxonomies – perhaps because they were crafted in the early days of software quality research. The first critique is that they lack rationale for the choice of attributes and sub-attributes, and mappings between these. According to Kitchenham and Pfleeger (1996) the selection seems rather arbitrary, which makes it very hard to decide if the taxonomies reflect a complete definition of quality. In addition, the lower level metrics are not very well defined – the ISO 9126 standard even skips this part – so the link to actual development practices is very vague.

One model that has tried to rectify this is the Dromey model, developed by Geoff Dromey (1995). It uses a bottom-up rather than top-down approach, by starting out with low level constructs from programming languages and then linking these to higher level attributes in the ISO 9126 standard. The result is a model which can be integrated into the every-day activities of the development process. This represents more of a manufacturing-view on software quality – employing defect counts and detailed analyses of the code to get the product right "the first time".

3.1.2 Process Improvement

The alternative to quality attributes and software metrics (or "finding the Holy Grail" as some would put it) is software process improvement. The goals are the same – getting a better product – but the means are focused on the process rather than the product.

The first thing an organization needs to be able to improve its process is some kind of *quality management system* – a set of policies, processes and procedures for tracking and ensuring quality. The ISO 9000 series, first published in 1987, describe three different models (ISO 9001-3) for standardizing these processes. The standards are not specific to any industry or product, but are targeted at manufacturing and service organizations. The ISO 9001 model describes "quality assurance in design, development, production, installation and servicing" (ISO.org, 1994), and so is the model best suited for software development. Some of the topics addressed are management responsibility, contract review, design control, document and data control, inspection and testing, and internal quality audits.

All in all there are 20 requirements that has to be fulfilled to ensure an optimal quality management system. By adhering to this standard a company can show its customers that it is capable of delivering quality product and services. It's not uncommon to find that public tenders, especially government projects, require ISO 9000 registration. Unfortunately for smaller companies, getting an official registration can be both a costly and timely process, and the registration has to be validated every six months (van Vliet, 2000).

Since the ISO 9000 standards are so generic and general there has been attempts to build models that are more suited for the software industry. Monitoring and identifying problems with the development process – commonly referred to as *software quality assurance* (SQA) – needs special attention that can not be covered by a catch-all model. The ISO 9000-3 standard tries to remedy this by acting as a guide to the ISO 9001 standard for using it in software development. Another approach is the IEEE 730 and 983 standards, which both lay out best practices for SQA.

While the above models may be best suited for verifying process conformance there are also standards that focus on improvement. The best known model in this area is the capability maturity model (CMM) – developed by the Software Engineering Institute (SEI) for the US Department of Defense in the late 80s (Humphrey, 1989). The capability maturity model framework describes five *maturity levels* that each address certain *key process areas* that must be in place for a company to be at that level. This way of organizing the requirements of the model fosters a step by step approach to improvement – always looking to move up one level. Like for the ISO 9000 standard the levels are used to indicate if a organization is suited for a particular task. Figure 3.3 on the next page lists the five levels and their corresponding process areas.

The first level is the initial level, where every company starts out. This level is characterized by by its lack of process rather than anything else, usually evident

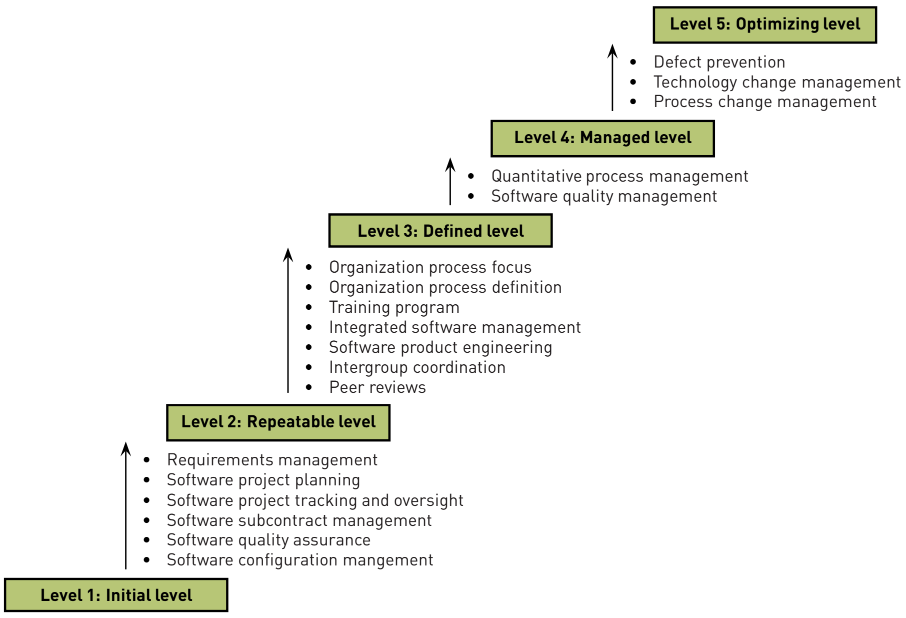

Level 5: Optimizing level
Defect prevention
Technology change management
Process change management
Level 4: Managed level
Quantitative process management
Software quality management
Level 3: Defined level
Organization process focus
Organization process definition
Training program
Integrated software management
Software product engineering
Intergroup coordination
Peer reviews
Level 2: Repeatable level
Requirements management
Software project planning
Software project tracking and oversight
Software subcontract management
Software quality assurance
Software configuration mangement
Level 1: Initial level

Figure 3.3: Maturity levels in the capability maturity model (van Vliet, 2000).

through frequent problems and deadlines that are not met. Fortunately there are ways to get past this level, but it may take as much as two years to get to just a couple of levels up the ladder. The steps include getting control of requirements, planning projects properly, tracking the progress, and basic software quality assurance. That will take you to level two, the repeatable level. Here the company has control over the basic development process, but introducing new technologies or making organizational changes is still a huge risk because it can invalidate the previous experiences and lessons. To deal with that the company must aim towards level three – the defined level – but getting there requires activities like setting up a special group for examining and defining the organization's process, starting up training programs for employers, and cooperating across projects and groups.

Now the company has a standard process for development and maintenance of software that is independent from the individual projects. Most companies stop at level three, probably because moving beyond this level has few apparent benefits and can be very costly. The last two levels are basically optimalizations of the previous activities, focusing on being proactive rather than reactive, and improving the process more than the product (van Vliet, 2000).

3.2 Peer Reviewing

The practice of reviewing, scrutinizing and critizising the works of others is not a new one. The philosophers of ancient Greece did it; 10th century physicians did it; and Galileo and his fellow scientists did it (Spier, 2002). In fact, some argue that the practice has existed for as long as people have communicated knowledge – because it is an integral part of building consensus (Rennie, 1999, citing Kronick).

3.2.1 In Scientific Publishing

The process as we know it from modern science is not that old though. It can be traced back to the mid-18th century, when journals such as *Medical Essays and Observations* and *Philosophical Transactions* started using groups of anonymous peers to review papers before they were accepted by the editor (Spier, 2002). The practice got off to a slow start, because at the time there were just not enough content for editors to be picky. But after increased diversion and specialization of journals in the mid-90's it was adopted by most major journals and conferences (Spier, 2002).

The essence of the process is the same today as it was 250 years ago. Authors write up their research into papers and send them to the appropriate journal or conference. If the editor of the journal finds that the paper passes minimum requirements such as topic and length the paper is passed on to a selected number of experts in the field (peers of the author). The identity of the reviewers are not disclosed to the original author, and the reviewers don't know who else is reviewing the paper – resulting in what is called a *blind* review. If the experts don't even know the name of the author it is called a *double-blind* review.

After receiving the paper the experts comment on it based on more thorough guidelines set by the journal, and on the general validity of the research. The comments are then returned to the editor, who is left with the decision whether to accept the paper or not (Parliamentary Office of Science and Technology, 2002). If there's strong disagreement between the reviewers on the quality of the paper the editor has to act as a middle-man – doing several iterations of the review. The point is however not to form a consensus, but to decide if the paper should be accepted or not, so the reviewers never talk directly. If the paper is not accepted for publication it is sent back to the author, usually with comments on how to improve and possibly with invitation to revise and resubmit.

The rationale for this rather bureaucratic process is that science gains from unrestricted criticism (because potential flaws can be revealed). Institutionalizing this criticism through a peer-review process will hence ensure the quality and integrity of the article, and also the journal itself (Rennie, 1999). While this may sound like a Good Thing™ the process has recived some criticism of its own. First of all it is a slow process, and manuscripts can often take months before being accepted. It's also said to be an unreliable, unfair and biased process – even when using double-blind reviews (Rennie, 1999). Using peers of the author opens up for potential conflict of interest, as the reviewers might have a personal interest in not lending credit to a competitor in the field. There is also the danger of innovative but controversial ideas being discarded because they conflict with the reviewer's perspective (Rennie, 1999). One can wonder what would have happened to Einstein's 1905 "Annus Mirabilis"-papers on mind-boggling topics such as special relativity if they had to pass through peer-review processes as stringent as the ones we have today.

Attempts to counter these problems have focused on opening up the process, for example by removing the anonymity of the reviewers, or letting the author pick or recommend suitable reviewers. Another approach is leaving the review for after publication – more in line with how normal critique of science works – or just increasing the volume of published papers. But this has only become possible in recent years, with the advent of online-based journals such as *Internet Journal of Science* and *First Monday* (FirstMonday.org, 2007).

3.2.2 In Software Development

Transferring the ideas of scientific peer review to software development puts us into the fields of software testing and reviews. Peer review can be described as a kind of informal and lightweight static testing strategy – not as rigorous as Fagan inspections and walkthroughs (Fagan, 1976), but still a valued testing technique (van Vliet, 2000). It's also part of step three (the defined level) in the CMM model.

Weinberg was one of the first to advocate looking over each others code – coining the term *egoless programming*. He argued that if people could put their ego at the door – ie. thinking of the code-base as communal property, not "my module" and "your routine" – it would create an environment where formal and informal code reviews could foster (Weinberg, 1998). Although this kind of environment is hard to achieve in practice it's important to keep the message in mind when doing peer reviews. If every criticism is taken with denial and verbal counter-attacks the whole process brakes down.

In practice software peer reviews can take many forms – covering everything from informal reading over the shoulder to double-blind reviews of programming exercises and quizzes. Van Vliet describes a process where every participant hands in two versions of a program; one "best" quality and one of lesser quality. The programs are then shuffled and distributed amongst the participants, so that every participant gets two of each kind. After the evaluation is done the participants get feedback on their programs and statistics for the whole group. Another approach is the one taken by open source projects, which we will discuss in the next sections.

3.3 Open Source Quality

WHEN proponents of free and open source software are asked about the major advantages of the open source development model they often point to the high quality of the software. The validity of this claim has been partly supported by the advent of successful projects such as Linux, Apache, and Mozilla (Miller et al., 1995; Halloran and Scherlis, 2002; Reasoning, 2003), and seems to be especially true for general-purpose, commoditized systems infrastructure used by technically sophisticated users (Schmidt and Porter, 2001).

A popular way of measuring software quality is through defect counts, which is represents typical manufacturing-view of quality. Mockus et al. (2002) found that defects were fixed faster in open source projects compared to closed source projects, and this was confirmed in a recent study by Paulson et al. (2004). Assuming that the projects were not significantly more buggy from the start then their commercial closed-source counterparts this speaks for the quality of open source software.

The classic explanation for the high rate of fixed defects is that a peer-reviewing process involving literally thousands of users and developers is bound to catch a lot more bugs than the single QA-team usually found in proprietary software-houses. This process of parallel debugging is seen as a major strength of open source development, and Raymond made it into a mantra in his seminal essay *The Cathedral & The Bazaar* when he coined the term Linus's Law; "Given enough eyeballs, all bugs are shallow" (Raymond, 1999a, p. 30).

On the other hand, McConnell argues that the practice of relying on hordes of downstream debuggers for quality assurance is perhaps no more than an economic shell-game – because the results of the debugging efforts are much easier to see than the costs. Using this practice as an excuse to ignore other parts of quality assurance will not scale very well to projects with unknown requirements, where care must be taken to avoid architectural 'bugs', and will not transfer very well to commercial vendors, where costs are very much real and visible (McConnell, 1999).

Either way, the research on *actual practices* of quality assurance in FLOSS projects is limited at best, and no major empirical work has been done to verify Raymond's anecdotes (Michlmayr et al., 2005). We will now consider the research in this field, starting with identifying common quality practices and then moving on to possible problems with the QA process in free and open source software.

3.3.1 Quality Practices

The majority of studies on quality practices in open source software has so far been surveys and interviews of developers. Zhao and Elbaum started this work seven years ago, and has been followed by a small group of authors in the same tracks (Zhao and Elbaum, 2000, 2003; Halloran and Scherlis, 2002). A recent paper by Michlmayr, Hunt, and Probert (2005) categorizes the practices into three main areas:

Infrastructure Because of their distributed nature open source projects rely heavily on infrastructure such as bug trackers and mailing lists. Major hosting sites for open source software like SourForge often provide this infrastructure as part of the package, leaving developers with little or no overhead for setting everything up. Some of the more common tools are:

- Bug trackers (e.g. Bugzilla, GNATS): these help organize and prioritize issues with the software – both defects and feature requests (Koru and Tian, 2004). Users and developers add tasks to the tracker when discovering bugs, and the task then becomes the central locus for discussions about the bug (Halloran and Scherlis, 2002). Comments can be added providing new information, a nd developers can flag relationships between bugs such such as "this bug is a duplicate of bug #1234". Housekeeping of the bug tracker is known as *triaging*, in which the developers or QA personnel sort out which bugs should be given the most attention.
- Version control systems (e.g. CVS, Subversion): these make it possible for multiple developers to work on the same code-base simultaneously. Local changes are transfer to a central repository through a process known as *check-ins*, or *commits*, and are then propagated to the other developers when they do a *check-out* or *update* on their systems. Conflicts are usually merged automatically by the software, unless the change involves the same line in a source file. The tool also allows developers to track changes historically, which can be helpful when trying to squash regression-bugs. Some tools even integrate with the bug tracking software, automatically updating and closing the corresponding tracker-task when a commit is made that fixes the bug.
- Automatic builds (e.g. Tinderbox, DejaGnu): these tools run continous or nightly builds (compile, link, package) of the latest source in the version control system, possibly on a wide range of hardware and platforms. Any problems are reported by e-mail to the developers so they can be fixed, and successful builds are frequently made available for download so that daring users can do field testing. The tool sometimes incorporate automatic testing-frameworks like unit-tests for doing continuous regression-testing.
- Mailing lists (e.g. Mailman, Majordomo): these are the primary enablers of user-to-user and developer-to-developer-communication in most open source projects. They provide a place for discussions about high level design concepts and future plans, and do also act as support channels for users experiencing problems with the software of a more general art (i.e. not bugs).

The way these tools are used of course vary from project to project. Some apply them in a very strict way, for example by only allowing check-ins by a selected number of highly trusted developers and halting development when the automatic build system finds a bug. Others are more laxed about giving away comitt-privileges and expect whoever who breaks the build to clean up within reasonable time (Michlmayr et al., 2005).

The above tools are by no means the only infrastructure used by open source projects, but a similar study by Halloran and Scherlis found largely the same list of tools in the projects they surveyed (Halloran and Scherlis, 2002). The study also showed that tools are incorporated in an incremental fashion, when they are really needed, and that new infrastructure benefits from being familiar to the developer-community. The first point is supported by findings in a survey of over 230 open source project participants made by Zhao and Elbaum. They found that although tool adoption was generally high, even in small and tiny projects, the adoption increased steadily to almost 100% for projects in the 'large'-category (Zhao and Elbaum, 2003).

Process Open source projects are famous for their rather peculiar development process – or lack of it. Vixie (1999) described FLOSS projects as having little or no system-level design, as-you-go detailed designs, and focusing on the 'fun' part of software development – the implementation. When it comes to quality assurance this trend is highlighted by the informal and casual testing techniques employed. Some elements of this process are:

- Peer review: as described in Section 3.2.2 the peer review process in software development can take many forms. Open source projects tend to cluster at the informal end – meaning they do not enforce peer review as a formalized process. Instead project members are expected to look over changes made by fellow developers as they are committed. Whether this happens in practice is not that clear (Michlmayr et al., 2005). Another form of peer review happens during the field testing stage, where users read the code and report back defects in the form of bug reports and patches.
- Testing: in addition to the day-to-day testing that happens during development most projects do some sort of directed testing before major releases. This may include everything from simulating user interaction and following step-by-step check-lists to full on regression-testing using unit-tests and similar. Once the software is released the testing falls under the item above.
- Quality assurance: some project form specialized QA teams who's job it is to make sure that the bug tracker is up to date and working in a smooth fashion. Part of this job is triaging bugs and removing old or non-applicable bugs from the system.

Of the three items described above many would say that the peer reviewing is the heart and soul of open source quality practices (Stark, 2002). This is largely due to Eric Raymond's anecdotes about the open source development process, which focused on the collective and distributed nature of debugging – resulting in peer-reviewing becoming somewhat of an 'icon' of the open source model. It is also used to highlight the contrast to commercial closed-source projects, which are widely criticized for not involving the user enough (Ehn, 1993).

Zhao and Elbaum (2003) reported that user participation and feedback in open source projects were in deed very high, which confirms some of Raymond's claims. For example did user suggestions generate over 20% of the changes in nearly 50% of the projects examined. The suggestions were for the most part regarded as valuable by the developers, and many respondents (44%) thought that users were capable of finding 'hard' bugs that developers would otherwise overlook. In larger projects this trend was magnified, but for suggestions bordering to feature requests the larger projects would sometimes brush them off as "not fitting into my design". This finding was not evident in the smaller projects – indicating a more open attitude towards incorporating user suggestions (Zhao and Elbaum, 2003). All of these findings describe peer review 'in the wild', i.e. after the software is released to users. Studies of how peer review works between developers during development seems to be missing from the picture.

With regard to testing Zhao and Elbaum found that the amount of time spent on this task was very high compared to commercial software development projects (Zhao and Elbaum, 2003). These statistics may be skewed by respondents incorporating peer-reviewed field testing into the figure – which is not comparable to the in-house systems-testing performed in the commercial studies. The fact that larger projects spend less time than smaller projects on testing may also suggest that the formal pre-release testing is replaced by peer-reviewing as the project builds a user-base. The testing techniques employed were typically imitating users (68%), testing border-line values (25%), and using code-assertion frameworks like JUnit (25%). One notable discovery was that only 48% of the projects had some kind of regression-testing-suite, which is surprising considering how FLOSS projects focus on frequent releases (Zhao and Elbaum, 2003).

Documentation The last category described by Michlmayr et al. is documentation practices. Vixie (1999) stated that open source projects tend to have no formal documentation of requirements or system-design. It has been suggested that this is because doing documentation is not 'sexy' enough compared to programming (Bonaccorsi and Rossi, 2003). Still there are some forms of documentation that are more common than others, such as coding style guidelines and build instructions. An example of the former is the various different brace- and indentation-styles – which are often policed with striking enthusiasm:

Listing 3.1: Example of the BSD/Allman style. The braces are placed on the next line, indented to the same level as the control statement.

```
if (!isDone)
{
    doSomething();
}
else
{
    System.err.println("Finished");
}
```

Listing 3.2: Example of the Sun/K&R style, also known as the One True Brace Style. The braces are placed on the same line as the control statement.

```
if (!isDone) {
    doSomething();
} else {
    System.err.println("Finished");
}
```

Zhao and Elbaum (2003) reported numbers that confirm Vixie's statement. For example did 84% of the projects use simple 'TODO'-lists, and 62% had build instructions and guidelines, but only 32% of the projects had design documents, and a mere 20% had release-schedules with planned features.

3.3.2 Possible Problems

In addition to the many quality practices identified in the literature there are also authors who point out possible problems with the current approach. We will now consider the most imminent and serious problems for open source quality assurance.

Short development cycles Porter et al. (2006) described the seemingly conflicting goals of providing frequent releases with cutting edge features/bug fixes and that of building a stable product through thorough testing. New versions of the software that fixes a couple of bugs may also introduce new bugs or regression-bugs if the software has not been properly tested, and this may be unacceptable for users who want stable releases. Considering that Zhao and Elbaum found surprisingly low levels of regression-testing this may be a serious problem. Porter et al. did also point out that regression-testing may not always be very effective either, as users who run the tests do not typically send them back to the developers. It has been suggested that focusing more on release-management and doing time-based releases may remedy some of these problems (Michlmayr, 2005).

Configuration-complexity Another problem suggested by Porter et al. is that the open nature of FLOSS software makes it vulnerable to configuration bloat. With so many combinations of features, platforms, and optional plug-ins doing test-coverage of all this becomes very hard (Michlmayr et al., 2005). This is magnified by the fact that most developers only have access to only one or two systems, so they have no practical way of testing all possible configurations (Porter et al., 2006).

Reliance on individuals Open source projects also rely heavily on experienced core-developers for major new features. This can be a problem if a core developer decides to quit the project, or otherwise becomes unavailable for an extended period of time, as this will leave the code orphaned. This means it will not be maintained in parallel with the rest of the code-base, and will sooner or later fall behind unless someone else takes over. This single-point-of-failure can be very dangerous to any software project (Weinberg, 1998). Michlmayr and Hill (2003) showed how this can be a problem for package-maintainers in Debian.

Quality of user-contributions Finally, Porter et al. pointed out that not all user contributions are as valuable as one may think. Developers complain that suggestions or patches are often written without the big picture in mind, resulting in either fragile patch-work code or developers having to rewrite the patch before committing it. Bug-reports are also frequently incomplete or vague (Michlmayr et al., 2005), and regression-tests are reported back without specifying the context of the tests (Porter et al., 2006). This may be because of lack of formal documentation on how to do these tests, as noted by Michlmayr et al. (2005).

3.4 Summary

THIS CHAPTER has introduced the concept of software quality from several different perspectives – starting with the traditional software engineering approach and then moving on to peer reviewing and how open source projects deal with quality. Some of the points we have covered are:

- How quality attributes have been used to try to capture essential parts of the quality of a software product.
- How process improvement models have been used to guide companies in ensuring stable quality over time.
- How the peer-reviewing process evolved in scientific publishing, and the various ways it is used in software development.
- How open source deveopers think about quality, the quality practices that they employ, and some of the possible problems of these practices.

Chapter 4

Knowledge Management

This chapter gives a brief introduction to the field of knowledge management. We will start off by looking at how the field has evolved in the past two decades. Then we will consider different typologies of knowledge – introducing the central concept of tacit knowledge – before moving on to ways of transferring knowledge, for example through communities of practice. Finally we will round off with a look at how information systems has played a part in shaping the direction of knowledge management initiatives.

4.1 A Short History

KNOWLEDGE is the foundation for a wide range of businesses today, and phrases such as *knowledge-workers* and *knowledge-intensive organizations* highlight the shift from classic assembly line manufacturing to businesses where knowledge is the primary 'goods'. This vision of the post-industrial society was popularized by Daniel Bell in his book *The Coming of Post-Industrial Society: A Venture in Social Forecasting*, and although it has been criticized for being utopian and exaggerating the uniqueness of the knowledge worker most people will agree that there has been at least *some* change to our economy, and that knowledge plays an important role in modern businesses (Hislop, 2005).

At the same time it has been shown that capturing, processing and transferring knowledge is a difficult task (McDermott, 1999). This stems partly from the fact that knowledge as a concept is vague and not very well understood. For example, how does the human brain represent knowledge; how do we 'capture' new knowledge; and how do we convey that knowledge to other human beings without loss or risk of misinterperation? These and other issues have given rise to the field of *knowledge management*, which has seen a significant growth in the past two decades (Scarbrough and Swan, 2001; Baskerville and Dulipovici, 2006).

In dealing with the management of *knowledge* it is difficult to not overlap somewhat with classical philosophy – especially the field of *epistemology*, which tries to explain the nature of knowledge. Questions such as "what can we possibly know about something", and "how can we obtain that knowledge" are central to this field. Traditionally there has been two opposing main camps in this debate: positivism and phenomenology – with the former focusing on objective methods for measuring and obtaining knowledge about the world, and the latter focusing on social constructs and the "meanings that people place upon their experience" (Easterby-Smith et al., 1991).

With knowledge management being so closely related to this debate it is not surprising to find that there are two broad perspectives to knowledge management too. Using the words of Hislop (2005) there's the *objectivist* perspective on one side, and the *practiced-based* perspective on the other.

The former builds on a positivistic epistemology, and regarding knowledge as a something that exists independently of people – an entity that can be extracted and codified. This was the dominant perspective in the early years of the knowledge management literature (approximately up to 1998) and has later been termed the *first generation* of knowledge management (Scarbrough and Carter, 2000).

Coming from fields such as economy and management the objectivist perspective was largely motivated by a desire to develop techniques for identifying and transferring knowledge in organizations (Werr and Stjernberg, 2003). The results were numerous IT-project with the sole goal of coding and storing knowledge in databases and schemas – all based on the assumption that knowledge was in fact codifiable and that people would be happy to share what they knew (Hislop, 2005). The first generation knowledge management literature has later been criticized for being to technical and ignoring the social and cultural sides to knowledge.

In recent years the practiced-based perspective has gained more and more momentum, with social sciences like sociology leading the way (Empson, 2001). By focusing on socio-cultural factors they hope to rectify the errors made by the first generation literature. Business managers too are starting to realize that there is more to knowledge management than codifying experiences and dumping them in a database. The practiced-based perspective is based on the assumption that knowledge is inseparable from practice, or as expressed by Maturana and Varela (1987) "all doing is knowing, and all knowing is doing". One example of how this 'second generation' literature recognizes social interaction as a bearer of knowledge is the popular theory of communities of practice, presented later in Section 4.3.2.

4.2 Typologies of Knowledge

ONE OF the central questions of knowledge management is what knowledge really is. The two perspectives base much of their assumptions on the underlying epistemological foundation – with the objectivist perspective drawing largely from positivism, and the practice-based perspective ranging from a classic interpretive standpoint to situated learning theory and actor network theory (Hislop, 2005).

When trying to understand what knowledge is it is sometimes helpful to distinguish is from other related concepts such as data and information. Zack (1999) describes the former as mere observations or facts, without any context, and therefore without any direct meaning. Once a context is added the data becomes information, for example expressed through messages.

The leap from information to knowledge comes when large amounts of information are absorbed, processed, and linked to other bodies of knowledge by human beings – forming understandings, or as described by Nonaka (1994) "justified true beliefs". This shows us that knowledge involves both the entities which we 'store' in our minds, and the process of turning a flow of information into knowledge (Zack, 1999).

4.2.1 Tacit and Explicit Knowledge

A popular frameworks for theorizing about the nature of knowledge is the tacit-explicit dichotomy, which distinguishes between *tacit knowledge* and *explicit knowledge*. These terms were first described by the philosopher Michael Polaniy in his book *The Tacit Dimension* (1966), and were later popularized by the works of Nonaka (1994) and Nonaka and Takeuchi (1998).

Tacit Knowledge Tacit knowledge refers to knowledge that is personal and subconsciously understood, and hence hard to articulate. Polanyi's famous aphorisms "we know more than we can tell" is often used to illustrate the nature of this knowledge (Polanyi, 1966). Similarly Nonaka and Takeuchi (1998) uses the metaphor of an iceberg – comparing explicit knowledge to the tip of the iceberg and tacit knowledge to the base. This is a nice approximation of the likely ratio between the two, but is at the same time a quite vague description of the concept because it lumps everything from subjective insights to hunches and intuition in there.

A more detailed picture is obtained by examining the way Nonaka and Takeuchi describes the properties of tacit knowledge. First and foremost it has "a personal quality, which makes it hard to formalize or communicate" (Nonaka, 1994). The personal quality is rooted in an invidiual's experiences, emotions, values and ideals, so part of the problem of communicating tacit knowledge is making sure that the context can be generalized to suit other individuals (Kogut and Zander, 1996).

Nonaka and Takeuchi also decompose tacit knowledge into two dimensions. The first is the technical dimension, which deals with skills or crafts, such as those used by bakers or potters. While these workers may be masters of their crafts, the 'know-how' that they possess is difficult to articulate – even in non-technical terms (Nonaka and Takeuchi, 1998). The second dimension is the cognitive dimension, which covers our image of reality and or visions for the future. These 'schemas' of perception shape how we interact with the world around us, and how we interpret new information (Nonaka and Takeuchi, 1998).

Explicit knowledge Explicit knowledge on the other hand is easily articulated. This is the kind of knowledge that we find in journals, reports, books and manuals – i.e. *codified* knowledge. It's easily communicated and shared because of its formal nature – analogous to digital information. Zack (1999) describes how explicit knowledge plays an important part of knowledge-work in businesses and organizations, and Nonaka and Takeuchi (1998) notes that this is especially true for western organizations, where the view of knowledge management as merely "information processing" is deeply integrated in management traditions.

The perspectives introduced in Section 4.1 differ in how they interpret these two concepts. The objectivist perspective considers tacit and explicit knowledge to be completely separate types of knowledge: you either have explicit knowledge in for example a document or database, or you have it subconsciously within you as tacit knowledge (Hislop, 2005). Some authors even go as far as to not discriminate between data and information and knowledge, which exemplifies the position that knowledge is rational and objective.

The practiced-based perspective on the other hand regards tacit and explicit knowledge to be inseparable – basically two aspects of the same concept. This implies that you will never have explicit knowledge without a tacit element. Walsham (2001), in discussing the works of Polanyi (1966), points out that his original message was that explicit knowledge can only exist in the context of tacit knowledge. Without context the knowledge becomes meaningless, and hence there are no objective 'truths' (Walsham, 2001).

4.2.2 Individual and Social Knowledge

Another complimenting approach to the nature of knowledge is whether it exists on an individual or social level. Nonaka (1994) argues that knowledge is purely individual, but authors such as Spender (1996) base their discussion on the premise that social knowledge (group-knowledge) plays an important part of the picture. By combining the tacit-explicit dichotomy with that of individual-social we end up with the table below:

	Invidual	**Social**
Explicit	Conscious	Objectified
Tacit	Automatic	Collective

Table 4.1: Two dichotomies of knowledge (Hislop, 2005).

Spender (1996) describes *objectified* knowledge as existing in "libraries, data banks, standard operating procedures [and] rule-based production systems" – basically static repositories with no element of learning – while *collective* knowledge is embedded in the organization and shared through "social processes of the collective, such as teamwork" (Spender, 1996, p. 71). We will get back to this interesting cross between social and tacit knowledge when presenting the theory of communities of practice later in Section 4.3.2.

4.3 Managing and Sharing Knowledge

FACED with the realization that knowledge is not simply – to use the words of Walsham (2001) – "quantifiable tradable assets", it becomes imminent to understand how to best manage and share this knowledge between individuals and across mediums. Nonaka and Takeuchi feels that Japanese businesses have an edge in this area, because they have already made this realization and are leveraging the vast amounts of tacit knowledge in the organization (Nonaka and Takeuchi, 1998).

The opposing interpretations of the two perspectives on what the relation between explicit and tacit knowledge is has implications for how they advocate management and sharing of knowledge. Although the objectivist perspective considers tacit knowledge to be difficult to articulate there is still a strong belief that 'conversion' to explicit knowledge is possible (Hislop, 2005). Once the knowledge is codified into explicit knowledge it is by definition purely objective and hence easily transferable. A consequence of this view is that the objectivist perspective has a tendency to downplay tacit knowledge as something that just has to be converted. Its main focus is how to enable effective transfers of explicit knowledge (Hislop, 2005).

This contrasts with the practice-based perspective, which sees tacit knowledge as an important component in the actual transfer, not just something you convert and then discard. With the assumption that you can never fully convert tacit knowledge into explicit knowledge it becomes meaningless to try to 'extract' every last piece and bit of information from the employees of a company and dump it in a database. Instead, the focus is moved to social interaction as a way to enable knowledge transfer. Recognizing and fostering communities of practice is one way to do this.

We will now look at two approaches to knowledge sharing. First up is Nonaka's spiral of organizational knowledge creation and transfer (which has its roots in the objectivist perspective), and then comes a quick overview the theory of communities of practice (which springs out of the practice-based perspective).

4.3.1 The Spiral of Knowledge

The works of Nonaka were done with a background in business management, and so one of his goals was to relate the philosophical ideas of Polanyi to real-world management issues – giving advice on how facilitate knowledge transfer in organizations. That led Nonaka to propose four 'patterns', or modes, of conversion between tacit and explicit knowledge, which together form a theoretical framework of knowledge transfer. The four patterns are detailed below, and then summarized in Table 4.2.

Socialization This mode describes transfer of tacit knowledge from one individual to another. The transfer comes from interaction, observation, imitation and practice, which all help to establish a shared experience. A good example is how apprentices learn a new craft from their mentors. Though observation and imitation the apprentice picks up the subtle details of the craft that the master may not be able to articulate, and this helps to build a shared platform for future learning though the other modes of interaction. Although this apprentice-mentor situation is often accompanied by guiding explanations Nonaka points out that the element of language is not a prerequisite for tacit to tacit knowledge transfer (Nonaka, 1994).

Externalization The process of externalization happens when a person tries to articulate tacit knowledge, in an attempt to produce explicit knowledge. Since tacit knowledge is expressed in a 'richer' language (because of its contextual/personal quality) than most natural languages the conversion often involves using metaphors, analogies, concepts and models to convey the full meaning of the knowledge, and ensure that contextual information is not lost or misinterpreted.

Combination This mode of conversion deals with explicit to explicit knowledge transfer. By combining, sorting, and systemizing already existing sources of explicit knowledge it's possible to create knew knowledge. Nonaka and Takeuchi (1998) uses the example of managers who may combine knowledge about product concepts and business strategy with an explicit company vision to inspire new products and visions.

Internalization Finally we have the conversion from explicit to tacit knowledge, which bears resemblance to the classic notion of learning. By absorbing explicit knowledge and combining it with existing tacit knowledge the person internalizes the knowledge and can use it to build or refine mental models and concepts. In a way this mode is similar to that of socialization, except the knowledge source is explicit rather than tacit. If the explicit knowledge is also well written and easily absorbable, for example by using diagrams, or gripping stories that "put you there", this will increase the rate of internalization (Nonaka and Takeuchi, 1998).

Name	Mode	Description
Socialization	Tacit → Tacit	Learning by observation and experience.
Externalization	Tacit → Explicit	Articulation/conceptualization of tacit knowledge, for example through the use of analogies and metaphors.
Combination	Explicit → Explicit	Collection of already existing expressed knowledge, for example by categorizing it.
Internalization	Explicit → Tacit	Integration of documentation and other explicit knowledge into personal mental models and work routines.

Table 4.2: Types of knowledge transfer according to Nonaka and Takeuchi (1998)

Together these four modes interact in what Nonaka called the *spiral of knowledge-creation*, where knowledge is transfered from tacit to explicit and back again in cycles. Interaction between individuals fosters socialization, which again triggers externalization of the tacit knowledge. This explicit knowledge is then combined with existing explicit knowledge, and the results are then internalized for a new cycle in the transfer-circle.

Whether the interaction and transfer of knowledge really happens in the ordered fashion described by Nonaka and Takeuchi is not that evident, but the main message is still valid: knowledge transfer and interaction consists of more than just combination of explicit knowledge, and by making that realization a company can structure the organization to facilitate and optimize knowledge transfer.

Nonaka and Takeuchi also present five conditions that are required for an organization to successfully facilitate the knowledge-spiral. Not all of them are relevant for this thesis, so I will concentrate on just two of them: *autonomy* and *redundancy*.

Autonomy This condition focuses on employees 'individual freedom' to work in their own tempo, set their own goals, and act autonomously – all within the greater vision of the company of course. By allowing autonomy Nonaka and Takeuchi argues that the organization will see increased creation of new knowledge. One way of achieving this autonomy is by using cross-functional self-organized teams, where the participants may be inspired to challenge the knowledge of others by bringing new ideas and perspectives to the table.

Redundancy This condition deals with the tendency of western managers to view information processing in terms of reduction of complexity and increased efficiency – much like the principles of database schema optimization. By skimming away everything that looks like overlap and redundant information western companies are missing out on a big part of the knowledge transfer process.

Sharing redundant information promotes the sharing of tacit knowledge, because individuals can sense what others are trying to articulate.

Nonaka and Takeuchi makes the point that although redundant information may be of no use at the time, it helps to build a shared understanding of the business and products. This can be especially valuable in concept-development, where ideas are floating around and tacit knowledge is shared. Trying to restrain and reduce new ideas into a fixed 'format' is counter-productive to the brainstorming process, and although redundancy can cause information overload in the short run Nonaka and Takeuchi considers it an advantage for the knowledge creation process (Nonaka and Takeuchi, 1998).

4.3.2 Communities of Practice

As noted in Section 4.2.1 the theory of *communities of practice* zooms in on the cross between social and tacit knowledge – also referred to as collective knowledge (Spender, 1996). The theory was introduced by Lave and Wenger (1991), who argued that learning is not a solitude activity – it involves engagement in communities of practice where experiences are exchanged and discussed, creating a shared understanding and repertoire of knowledge. This is very much in line with the practice-based perspective on knowledge, because it emphasizes the 'doing' part of sharing knowledge.

This process of collective learning can be found around us in our daily lives, and we are all part of multiple communities of practice, even if we don't recognize them as such. Communities are formed out of common activities such as field of work, or social interests, and they range from the fluid and informal – almost 'invisible' – to strictly formal. We join them by participating in activities that match that of the community, and as we increase our interactions with the community we gradually step up the ladder to full participation.

One example of the dynamics of communities-of-practice can be found in Brown and Duguid's discussion of the work of Julian Orr, who did ethnographic studies of service technicians at Xerox. The technicians met in informal settings such as lunch or a quick coffee and frequently discussed problems they had encountered during the day. These "war stories" formed the basis for collective 'detective-work' where the technicians would diagnose the problem together, and they also built a body of knowledge in each technician that could not be achieved by reading operations manuals or through formal training (Brown and Duguid, 1991).

In the terms of Nonaka and Takeuchi this could be described as tacit knowledge being externalized through social interaction, and then internalized by each technician. Similar to the spiral of knowledge the point of communities of practice as a theoretical concept is is to recognize that they exist and thus support the growth of these communities in organizations rather than break them down.

4.4 Knowledge and Informatics

THE CONCEPTS described in the previous sections were rather abstract and philosophical. We will now look at what influences the knowledge management literature has had on the field of informatics, and how failed attempts at using IT to foster knowledge sharing has resulted in new ways of thinking about knowledge management.

The major theme of the first generation knowledge management literature was how groupware systems could help managers get an overview of the 'combined knowledge' of the organization and help employees share experiences and 'best-practices'. Network and database technologies was seen as a revolutionary new way to boos the speed and effectiveness of knowledge sharing, and applications suites such as Lotus Notes and SAP were central in providing the technological back-end for numerous large scale ITC-systems.

Over the years it became more and more evident that dumping explicit knowledge into databases was not the silver bullet everyone had hoped for. Despite all the good intentions and huge investments in ambitious knowledge-initiatives companies were still having a hard time leveraging knowledge in the organization (McDermott, 1999). As described by Walsham (2001) managers were tired of building data-warehouses that nobody would visit. This shifted the focus to recognizing the tacit elements of knowledge, and how the content of intranets and groupware-systems could be a lot more potent if combined with social interactions and context (Walsham, 2001).

In later years the focus has been turned towards supporting communities of practice through open and fluid communication-technologies like e-mail and instant messaging. The idea is that by not restricting how knowledge is shared, the chance of tacit knowledge being diffused in the organization increases. This approach is a stark contrast to the fixed-field databases of the first generation knowledge management literature and IT-initiatives.

An interesting example of the fine line between supporting and constraining the growth of communities of practice is presented in a recent paper by Thompson (2005). The author did a study of a small sub-group of web-designers within a large UK-based IT-company, where the group had been put together in a rather untraditional way. Instead of being an integrated part of the parent company the group operated independently – without the constraints of the parent company. This perceived freedom from bureaucracy – combined with a laboratory-style informal work-environment filled with 'props' such as pool-tables and bean-bags – resulted an atmosphere where people felt motivated and inspired to work.

The prevalence of fluid, lean-and-mean communication-tools was high, with a wiki-like system for creative brainstorming, and e-mail and instant messaging being used for quick exchange of information. Thompson (2005) refers to these factors as *seeding* structures – indirect enablers of social interaction – and notes that such structures can play an important role in fostering communities of practice. What's unique about these structures is that they are non-imposing, ie. not trying to control people's actions.

Controlling structures on the other hand are built with the premise that it is beneficial to control people's behavior. Examples are religious adherence to so called 'best-practices' and distinguishing between billable and non-billable hours. Thompson argues that introducing these kinds of structures into a project is bound to fail – which was exactly what happened in the case of the web-designers. The parent company – inspired by the apparent success of the small group – decided to open up the research-facilities for other employees, and started to introduce control-structures to manage the increased number of personnel involved. This killed much of the spirit that had started the whole community, which in turn resulted in discouraged employees and a notable decrease in productivity.

4.5 Summary

THIS CHAPTER has presented the field of knowledge management – starting with a brief history, and then discussing typologies of knowledge and knowledge transfer, before rounding off with a look at how knowledge has played a part in informatics. Some of the points we have covered are:

- How the field has moved from understanding knowledge as something that can be easily extracted and codified – the so called first generation knowledge literature – to recognizing that knowledge has a large social and contextual element.
- How one way of looking at knowledge is thought the tacit-explicit dicthtomy, and how the old and new generation knowledge literature view this distinction.
- How knowledge can be transfered from tacit to explicit and back again using the spiral of knowledge
- How collective knowledge is dispersed in communities of practice, and how we can build structures to support these communities.
- How knowledge-initiatives in IT organizations has moved from relying on databases and data warehouses to including the contextual and social elements of knowledge.

Part II

Case Study

Chapter 5

Methodology

This chapter outlines the methodological choices in my study, and how they played out in practice. It starts off with an introduction to research in informatics – clarifying widely used terms and concepts – before moving on to the actual research process. After highlighting issues such as case selection and data processing I finish off with an evaluation of my own work, based on the principles of Klein and Myers (1999).

The majority of the material presented in this chapter is based on the book *Project Research in Information Systems* by Cornford and Smithson (1996), and I will use their terms for the various concepts when possible.

5.1 Information Systems Research

INFORMATION systems is a diverse field with influences from both organizational theory, management, sociology and psychology. While the field is taught primarily in business schools in the United States, the European approach has been more technical – grouping information systems together with computer science and software engineering. Cornford and Smithson (1996) relate these fields to each other by saying that computer science (CS) deals with how software and computers function, software engineering (SE) with how to build technical systems from a specification, and information systems (IS) with how these systems affect humans and organizations. Following this distinction information systems is an applied field, whereas computer science is primarily theoretical (and software engineering falls somewhere in between).

Within informations systems we find a range of perspectives, from *technism* – seeing the computer as an instrument of progress – to *radical criticism* – which sees the computer as out of control and threat to humanity. Between these two extremes we find the more natural *progressive individualism* and *pluralism* – the latter being strongly linked to the socio-technical perspective so popular in the Scandinavian information systems tradition (Mumford, 2000). The main message there is that computer systems are social systems, which means that technology must always be analyzed in terms of its organizational and social context.

With all these different perspectives and schools of thought I find it helpful to go back to the basics for a moment – to clarify some of the 'ground rules', so to speak. I will now present concepts such as ontology and epistemology, and discuss various approaches to research, before returning my own position in this landscape in the next section.

5.1.1 Ontology and epistemology

Up through history researchers have always debated the usefulness of different research topics and the validity of each others results. But the core issue usually goes a little deeper than statistical significance and proper interviewing techniques. It deals with the underlying philosophy of it all: what exists in this world; what can we observe and measure; are there any undisputable truths out there; and how do we as researchers affect all of this in our work?

Philosophers quickly picked up on these questions and put them into more defined theories and models. *Ontology* refers to how we see reality, or the properties of the world, usually ranging from realism – that the world exists independently of what we think about it and has external 'real' properties – to nominalism – that the world is shaped and interpreted by each individual (Cornford and Smithson, 1996). Most researchers fall somewhere between these two extremes, and the point is not to make a stand, but rather to think about how this issue affects ones work.

Related to the question of reality is the term *epistemology*, which deals with the nature of knowledge: what can we possibly know about something, and how can we obtain that knowledge. Positivism and anti-positivism are the two main camps here – the former focusing on objective methods for measuring and obtaining knowledge about the world, and the latter focusing on social constructs and the "meanings that people place upon their experience" (Easterby-Smith et al., 1991).

A key point of anti-positivism is that the positivist stance of 'value-freedom' (that one can objectively determine what to study) is naive, because people will always be affected by their own and others expectations and wishes. As a consequence scientists should be more honest about their motivations for choosing a certain topic or case, and take this into consideration when creating research designs and analyzing results. Another term for the anti-positivist perspective is *interpretivism*, which highlights the fact that the researcher is never a neutral observer.

These two terms, ontology and epistemology, are obviously closely related, and you can often find that positivists lean towards objectivism, while interpretivists lean towards realism. Still, there is much room for interpretation along these axes, and there is seldom a one-to-one relationship between a chosen ontology and epistemology (Easterby-Smith et al., 1991). Collectively these philosophical assumptions about the research process form what is referred to as research *methodologies*. It is important to note that one methodology is not necessarily 'better' than the other, but that they each have their uses, depending on the type of research.

5.1.2 Approaches and Methods

After settling on a methodology the focus is usually turned to the research design and the *methods* to use. Methods are more hands-on techniques for data collection and analysis – although they also influence the research design. Typical examples include experiments, surveys, and case studies. The research design must also consider issues such as whether the study should be empirical (involving observations and data) or non-empirical (forming theories and ideas), cross-sectional (snapshots of a population) or longitudinal (repeated measures over time), and whether to follow a quantitative or qualitative approach. The former aims to extract quantifiable metrics which can later be processed using statistical analysis, while the latter focuses on providing rich insights and understanding of a phenomenon in its original context. Most people regard these two approaches to be compliments – each suitable to extract information not available to the other – but some also take the position that their favorite approach is the 'one true way' and that the other merely gives a shallow picture of the real phenomenon (Cornford and Smithson, 1996).

Based on the issues above we can categorize three broad styles of research:

Constructive research is non-empirical in nature and focuses on creating frameworks, concepts and guidelines for design and development (design-oriented research). The information systems literature is filled with studies in this style, in particular on development methodologies.

Nomotheic research uses empirical data to test hypotheses and formulate general laws and theories. It has roots in scientific tradition and often follows a quantitative approach. Methods like formal mathematical analysis, experiments, and surveys are common.

Idiographic research is also empirical in nature, but focuses on exploring phenomena in their natural environment – leaving a "rich picture of what transpires" (Cornford and Smithson, 1996). Methods like case studies and action research are typical for this style.

Although the choice of methodology does not dictate the research style to use there is often a correlation between the two. Cornford and Smithson (1996) notes that the qualitative approach is "strongly associated with an interpretivist and relativist position". Klein and Myers (1999) on the other hand points out that there is nothing wrong with doing qualitative research based on a positivistic background. The field of information systems – which has traditionally been concerned with people's experiences in organizational contexts (i.e an interpretive stance) – has a strong use of case/field studies and conceptual analysis. Similarly, in the computer science field, which focuses mostly on formulating algorithms and 'unified' processes (based on a positivistic stance) the method of choice is mathematical analysis (Glass et al., 2004).

The choice of methods in information system research is a hot topic, and while some advocate the use of more statistics and experiments (Tichy, 1998), others feel that interpretive methods might give better results (Galliers and Land, 1987; Walsham, 1995). The common theme seems to be that information systems academics should vary their methods and be more applied in their work, as to strengthen the credibility of the field.

5.2 The Research Process

NOW THAT we have considered various methodical perspectives and approaches to research it is time to zoom in on my own part in this. I will first place my research within the theoretical frameworks presented in the previous section, and then recount the individual 'stages' of my research.

5.2.1 Choosing Methods

Coming from a largely technical education, and being raised to think of natural sciences – illustrated by the stereotype of a mad scientist in a white lab-coat experimenting with bubbling potions – as the only way to do 'proper' research, my introduction to information systems was a refreshing new perspective. During the past two years my subscription to a deterministic world view has been constantly challenged, and I now adopt an interpretivist stance – recognizing that reality is socially constructed and that objective 'truths' are few and far between.

Based on the research questions presented in Chapter 1, and grounded in an interpretivist perspective, the choice of an ideographic style of research came very natural. Answering questions like "How do developers determine the nature and proper solution for a bug?" is a perfect candidate for an empirical approach, and since the majority of previous studies in this area were quantitative surveys I felt that doing a case study would provide a supplement to the existing literature.

A case study is identified as being an in-depth investigation of a phenomenon in is real-life context – typically when the boundaries between the phenomenon and the context are blurry, and the researcher has little control over events (Yin, 1994). The method may be based on either a positivistic (Yin, 1994) or interpretivist (Walsham, 1995) viewpoint, and does not discriminate between qualitative and quantitative data. That being said, qualitative studies are largely predominant, and the cases are usually longitudinal in nature.

Yin (1994) further decomposes case studies into *exploratory*-, *explanatory*-, and *descriptive* designs, while Stake (1995) describes the difference between *intrinsic* cases (aimed at revealing something about that particular case) and *instrumental* cases (aimed at general understanding). All of these 'sub-genres' can also be either single-case or multiple-case studies.

Of course there were other viable methods that I could have successfully employed to answer the research questions – for example surveys or action research. But, as noted earlier, the existing literature on the subject already covered surveys, and doing action research would most likely involve a considerable risk, due to the many factors affecting participation in open source projects. With time as a very real constraint I opted for the safer approach of using a case study.

5.2.2 Case Selection

After deciding on case study as my research approach came the process of finding one or more suitable cases to study. This process started in the fall of 2006, with the goal of finalizing the choice before the spring semester of 2007. During that period i surveys several sources for possible candidates – including the two open source websites SourceForge and Freshmeat. The former provides hosting service (bug-tracker, source control, website) for open source projects, while the latter is an index of projects hosted by other services.

However, many of the projects on SourceForge actually employ external hosting, and only use SourceForge as a way to relieve the main download servers and archive old versions of the software, so in many ways SourceForge can be used as an index too. Both sites have the ability to sort project by various criteria such as popularity (measured by activity and downloads), license, and field of usage.

Before starting the survey I put down a number of criteria to help me narrow the search. These were:

- Must employ an open source license (OSI approved)
- Must be relativly mature (beta at least)
- Must be relativly active
- Must have a minimum number of 10 developers
- Must employ peer-reviewing in their process
- Must run on either Windows or Linux
- Preferably be written in either Java or C^{++}
- Preferably not surveyed before in the literature
- Preferably run in a graphical environment (GUI)

The rationale for the last point was a hypothesis that graphical end-user applications would have a larger number of users, and that those would in general be less technically inclined then your average kernel hacker. This would in turn result in more 'low quality' patches that would really put the peer review process to the test.

Based on these criteria I used the Freshmeat website to narrow down the search to about ten projects, and these were then observed over a period of one month. During that time I used pragmatic considerations such as ease of access, activity level, and personal preference to rule out projects, until I was left with three possible candidates: the Linux music player Amarok, the web-based image gallery Gallery, and the Linux instant messaging client Gaim.

At this point I had decided that basing my case study on two separate projects would give me some leverage in terms of variations in the activity levels, and also allow me to do a comparison between characteristics in the quality assurance practices, so I proceeded to send out introductionary e-mails to the two most promising projects – Amarok and Gallery. The message was worded in a open tone, as the purpose was to get a feel for the willingness of the project developers to take part in my study:

```
Hi!

My name is Tor Arne and I'm a graduate student in infor-
mation system (IS) at the Norwegian University of Science
and Technology. I'm writing my master thesis on communication
and cooperation in open source projects, and I'm currently
in the process of finding a suitable case for my study.

That's where you guys come in :) You see, I've been
following Gallery for a while now, both on #gallery and
by browsing e-mail archives and bug reports, and I think
this project would be ideal for my research topic. I'm
specifically interested in how open source projects do
quality control, i.e. making sure that what's integrated
into the code base is both bug free and does the job. This
is interesting because if Eric Raymond's statement that
"all bugs are shallow" is true, then the need for some
mechanism to make sure a random patch doesn't introduce
new bugs is imperative. The classical explanation is that
the process of peer-reviewing takes care of quality, but I
would like to investigate this further in a longitudinal
(~6 months) empirical case study.

What this would mean for you is that I would use already
existing archives (e-mails, bug-reports, forums, IRC-logs)
to try to get a clear picture of the day to day activities
in Gallery, and how they contribute to this quality control.
I would also hope to get some interviews (through IRC or
Skype) in the later stages of my study, but only as an
addition to the other data, and on a volunteer basis.

Although my PHP experience is limited I consider myself a
competent programmer, and would be happy to contribute my
part in making Gallery an even better product. I'm also an
eager amateur photographer, so Gallery is a project that
really excites me - not just something I picked on random ;)

Of course all of the normal scientific 'oaths' would apply
for my final write-up and results. Names would be anony-
mized and credit given where appropriate. So, does this
sound like something you would be OK with? I don't want
to step on anyone's toes - that's why i sent you this
e-mail - so feel free to tell me to get lost :)

Hoping to hear from you!

Tor Arne Vestbø / torarne@#gallery
```

Alias	Role
AmarokDev01	Core developer
AmarokDev02	Core developer
AmarokDev03	Core developer
AmarokDev04	Core developer
AmarokDev05	Core developer
AmarokDev06	Developer
AmarokDev07	Developer
AmarokUser01	User

Alias	Role
GalleryDev01	Core developer
GalleryDev02	Core developer
GalleryDev03	Core developer
GalleryDev04	Core developer
GalleryDev05	Developer
GalleryDev06	Developer
GalleryDev07	Developer
WPG2Dev01	WPG2 developer

Table 5.1: Informant aliases for the Amarok and Gallery projects respectively.

The immediate feedback I got indicated a positive attitude to my study, and the two projects seemed to compliment each other well in terms of the approach to quality assurance, so I settled on them and proceeded to start data collection.

One thing to note about the e-mail was how I tried to make it clear that the results would be anonymized, to protect the identities of the individual informants. This is particularly important for studies like this where most of the material is publicly available on the Internet, which makes cross-analyzing data much easier. To still be able to track individual informants between vignettes and quotes in my work I generated mappings between each informant and a coded alias (see Table 5.1).

5.2.3 Data Collection

Data collection was conducted in the period from January 2006 to August 2006, using a wide range of sources. Yin (1994) describes six possible sources of evidence for case studies, and my study employed five of them. The following discussion present these sources and how I used them:

Documents The first source i surveyed when initiating my data collection was official and unofficial documents that could give me a broader understanding of the two cases. This ranged from project websites and wikis to meeting minutes, status reports, and release notes. I also looked for news-media interviews of the developer team, as well as personal blogs, to get a feel for the people involved and how they viewed the project.

Archival records After getting a good overview of the nature of the two projects I started surveying the various archival records. Because the time frame of my data collection was limited, historical records like e-mail archives and bug reports were central to getting enough material for my analysis. I initially set out to survey everything from January 1st 2006 up till today – i.e. one extra year of data – and this turned out to be plenty enough.

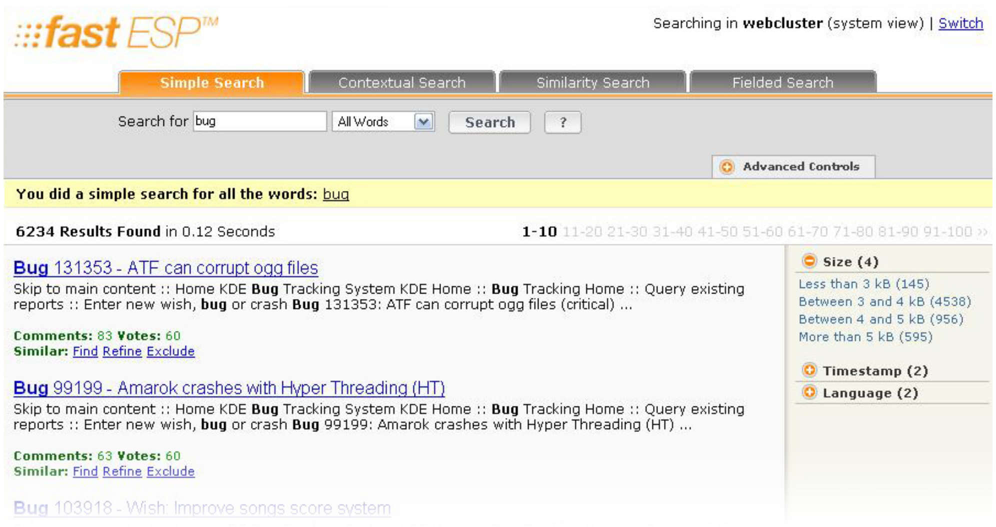
Most of the archives had sufficient functionality to navigate the data easily, for example by sorting by a certain topic or searching for a particular date or keyword, but in one case I had to employ special customized software. The problem was that one of the bug-trackers did not allow sorting by the number of comments or number of votes (which seemed like good indicators of controversial topics).

This issue was solved by employing the Fast Enterprise Search Platform (FastSearch.com, 2007) to index all of the bugs in the tracker, and then process each report through a custom filter that extracted the necessary data for querying in the search engine. The filtering was a matter of a few regular expressions:

Listing 5.1: Custom filter that was used to process each document and extract the number of comments and votes for each bug.

```python
from docproc import Processor, ProcessorStatus, ConfigurationError
from docproc.Document import Document, TextChunks, FromDocML
import re

class BugzillaExtractor(Processor.Processor):
    def ConfigurationChanged(self, parameters):
        self.commentRe = re.compile('<a name="c(\d+)"', \
            re.IGNORECASE)
        self.votesRe = re.compile('<td>(\d+)&nbsp;', \
            re.IGNORECASE)

    def Process(self, docid, document):
        html = document.Get("html")
        match = self.commentRe.findall(html)
        if match :
            document.Set("igeneric1", int(match[-1]))

        match = self.votesRe.findall(html)
        if match:
            document.Set("igeneric2",  int(match[-1]))

        return ProcessorStatus.OK
```

Using a huge software package like Fast ESP for a simple task like extracting a few numbers from a data-set may seem like overkill, but the platform was familiar turf for me, and did the job:



Searching in **webcluster** (system view) | Switch

Search for: bug

You did a simple search for all the words: bug

6234 Results Found in 0.12 Seconds

1-10 11-20 21-30 31-40 41-50 51-60 61-70 71-80 81-90 91-100 »

Bug 131353 - ATF can corrupt ogg files
Skip to main content :: Home KDE **Bug** Tracking System KDE Home :: **Bug** Tracking Home :: Query existing reports :: Enter new wish, **bug** or crash **Bug** 131353: ATF can corrupt ogg files (critical) ...

Comments: 83 **Votes:** 60
Similar: Find Refine Exclude

Bug 99199 - Amarok crashes with Hyper Threading (HT)
Skip to main content :: Home KDE **Bug** Tracking System KDE Home :: **Bug** Tracking Home :: Query existing reports :: Enter new wish, **bug** or crash **Bug** 99199: Amarok crashes with Hyper Threading (HT) ...

Comments: 63 **Votes:** 60
Similar: Find Refine Exclude

Bug 103918 - Wish: Improve songs score system

Size (4)
Less than 3 kB (145)
Between 3 and 4 kB (4538)
Between 4 and 5 kB (956)
More than 5 kB (595)

Timestamp (2)
Language (2)

Figure 5.1: Screenshot showing a search for the word "bug", sorted by number of comments.

Physical artifacts Next to documents and archival records the source code itself was vital in tracking changes – sometimes even several years back. It also helped getting an overview of the dynamics of the development process. Both project of course had their version control systems (Subversion) open for public read-only access, which allowed me to download complete histories of the changes in the project (going back to day one), using the command "`svn log`".

Using this history as a basis I hacked[1] up a small tool that would aggregate the individual log messages into statistics of the number of commits and number of developers per month – plus who the most active developers were. The central part of this tool looks looks like this:

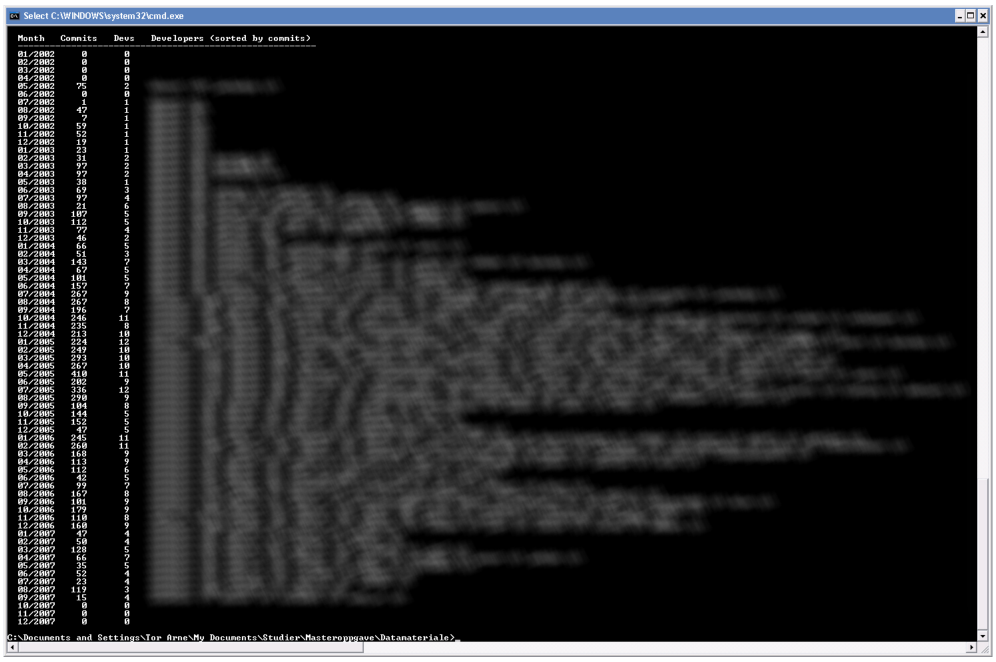
Listing 5.2: Custom filter that was used to process each document and extract the number of comments and votes for each bug.

```python
def parseSubversionLog(filename):
    # Compile the regex first, to speed things up a bit
    pattern = re.compile('^r(?P<revision>\d+?)␣\|␣' \
        '(?P<username>.+?)␣\|␣' \
        '(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2}).*\|␣' \
        '(?P<lines>\d+?)␣lines$')

    data = {}
    logFile = open(filename, 'r')
    for line in logFile:
        # Sanitize and skip uninteresting lines
        line = line.strip()
        if not line.startswith('r'): continue
        if not line.endswith('lines'): continue
        match = pattern.match(line)
        if not match: continue

        rev, user, year, month, day, lines = match.groups()
        year = int(year)
        month = int(month)

        # Make sure our data structure is ready
        if not data.has_key(year):
            data[year] = {}
        if not data[year].has_key(month):
            data[year][month] = {'commits' : 0, \
                'lines' : 0, \
                'developers' : {}}
        if not data[year][month]['developers'].has_key(user):
            data[year][month]['developers'][user] = 0

        # Update the stats
        data[year][month]['commits'] += 1
        data[year][month]['lines'] += int(lines)
        data[year][month]['developers'][user] += 1

    logFile.close()
    return data
```

[1] In this context, "hacking" implies that I disregarded common engineering practices to get something up and running quickly.

Combining these numbers with statistics over the number of downloads in each project (SourceForge.net, 2007a,c) allowed me to generate the charts in Chapter 6 – showing activity levels and downloads over time. It also gave me a nice overview of developers coming and going, and who the core developers were:

```
Month    Commits    Devs    Developers (sorted by commits)
---------------------------------------------------------
01/2006    245       11     GalleryDev02 (99) GalleryDev03 (91)
02/2006    260       11     GalleryDev02 (109) GalleryDev03 (67)
03/2006    168        9     GalleryDev02 (117) GalleryDev03 (23)
04/2006    113        9     GalleryDev02 (76) GalleryDev01 (26)
05/2006    112        6     GalleryDev02 (93) GalleryDev01 (11)
06/2006     42        5     GalleryDev02 (24) GalleryDev01 (12)
07/2006     99        7     GalleryDev02 (74) GalleryDev07 (10)
08/2006    167        8     GalleryDev02 (144) GalleryDev07 (7)
09/2006    101        9     GalleryDev02 (71) GalleryDev01 (12)
10/2006    179        9     GalleryDev02 (100) GalleryDev01 (31)
11/2006    110        8     GalleryDev03 (41) GalleryDev02 (22)
12/2006    160        9     GalleryDev02 (100) GalleryDev07 (24)
01/2007     47        4     GalleryDev02 (26) GalleryDev01 (9)
02/2007     50        4     GalleryDev02 (22) GalleryDev03 (15)
03/2007    128        5     GalleryDev02 (84) GalleryDev01 (28)
04/2007     66        7     GalleryDev02 (31) GalleryDev03 (16)
05/2007     35        5     GalleryDev02 (18) GalleryDev03 (12)
06/2007     52        4     GalleryDev02 (32) GalleryDev03 (12)

[...]
```



```
Month    Commits   Devs   Developers (sorted by commits)
--------------------------------------------------------
01/2002      0       0
02/2002      0       0
03/2002      0       0
04/2002      0       0
05/2002     75       2
06/2002      0       0
07/2002      1       1
08/2002     47       1
09/2002      7       1
10/2002     59       1
11/2002     52       1
12/2002     19       1
01/2003     23       1
02/2003     31       2
03/2003     97       2
04/2003     97       2
05/2003     38       1
06/2003     69       3
07/2003     97       4
08/2003     21       6
09/2003    107       5
10/2003    112       5
11/2003     77       4
12/2003     46       2
01/2004     66       5
02/2004     51       3
03/2004    143       7
04/2004     67       5
05/2004    101       5
06/2004    157       7
07/2004    267       9
08/2004    267       8
09/2004    196       7
10/2004    246      11
11/2004    235       8
12/2004    213      10
01/2005    224      12
02/2005    249      10
03/2005    293      10
04/2005    267      10
05/2005    410      11
06/2005    202       9
07/2005    336      12
08/2005    290       9
09/2005    104       8
10/2005    144       5
11/2005    152       5
12/2005     47       5
01/2006    245      11
02/2006    260      11
03/2006    168       9
04/2006    113       9
05/2006    112       6
06/2006     42       5
07/2006     99       7
08/2006    167       8
09/2006    101       9
10/2006    179       9
11/2006    110       8
12/2006    160       9
01/2007     47       4
02/2007     50       4
03/2007    128       5
04/2007     66       7
05/2007     35       5
06/2007     52       4
07/2007     23       4
08/2007    119       3
09/2007     15       4
10/2007      0       0
11/2007      0       0
12/2007      0       0

C:\Documents and Settings\Tor Arne\My Documents\Studier\Masteroppgave\Datamateriale>_
```

Figure 5.2: Screenshot showing the complete output of running the subversion-tool on the Gallery2 source code tree.

Direct observation In parallel with surveying documents, archival records, and physical artifacts, I performed daily direct observation of the two projects. This consisted of reading any new messages on the mailing lists, and checking out recent activity in the bug tracker and commit logs. I also kept an IRC client open on a separate screen attached to my university workstation, so that I could easily glance over and follow any interesting discussions as they happened.

It was during one of these sessions that I discovered a separate IRC channels for the Amarok developers, named `#amarok.dev`. This channel had not been mentioned in any of the documents or archival records that I so far had surveyed – in fact in all cases where new developers were encouraged to join the discussion on IRC they were told to go to the normal `#amarok` channel. I proceeded to join this channel, but was quickly told that it was a private room – for developers only – and that I would have to leave. After enquiring about the reason for this I was explained that the policy was intended to improve the signal to noise ratio for the regular Amarok developers – which is very understandable considering the high level of noise in the normal `#amarok` channel.

I followed up with an e-mail to the project where I pleaded for the developers to reconsider the policy in this particular case – noting that I would keep very quiet and not contribute any noise. I also stressed – just like in the introductionary e-mail – that anything I observed would be fully anonymized. But, after not receiving a reply for a week, I concluded that I would have to accept this source as a blind spot in my data material. According to Walsham (1995) these problems are common for direct observation because the researcher is not considered a participant in the project.

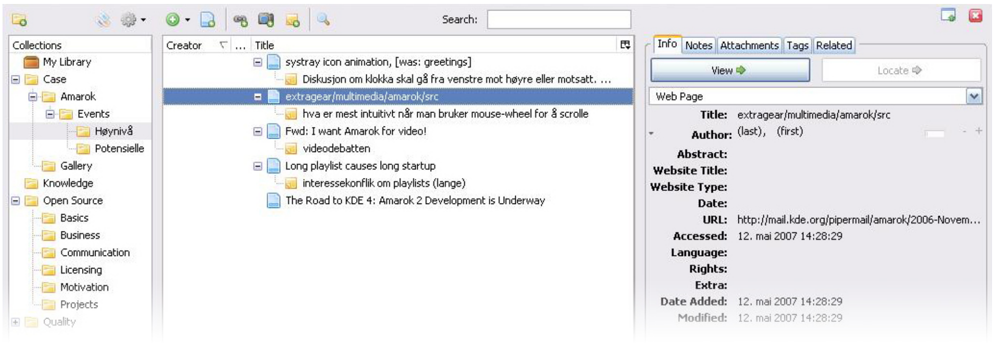
Interviews This last source was only touched upon briefly, by sending out a few open-ended questions to key developers in the two projects (four developers from each project). This was done in the later stages of my study, to get feedback on some of my earlier observations, and to validate them against the developers' own views. The response rate was pretty good, with two developers from each project replying within a day or two.

In addition to describing six sources for evidence in case studies Yin (1994) also proposes three principles of data collection that should help ensure validity and reliability for case studies. The first involves using multiple sources of evidence, to better triangulate the observed events. This was the primary reasons for choosing so many sources in this study – although not all of them were used to their full extent. The second principle advocates the use of a case study database where all observations, surveys, and notes are stored – the motivation being that critical readers should be able to "review the evidence directly and not be limited to the written reports" (Yin, 1994, p. 95). Following in the same line of thinking the third principle calls for the researcher to maintain a chain of evidence for all conclusions – so that readers can trace them back to their origin. One way of doing this is according to Yin (1994) to cite material from the case study database throughout the report.

These two last principles point to an important aspect of case study reporting, namely the accountability of the researcher in providing sufficient evidence for his conclusions. But the solutions presented by Yin (1994) are not without their problems – especially in the context of this particular study. First of all it is highly unlikely that one researcher will be able to replicate the results of another just by having access to the same case study material. In assuming that, Yin (1994) also implies that the researcher is an objective observer – which reveals his positivistic stance.

Secondly, when the case material is publicly available on the Internet, like for the two projects in this study, having all vignettes, observations, and quotes referenced back to the original source quickly compromises the anonymity of the informants – because following the chain of evidence is suddenly as easy as clicking a hyperlink. Walsham (2006) touches upon this dilemma briefly – in the context of disclosing the name of the organization under study – and argues that non-disclosure is sometimes necessary, but should be accompanied by good contextual information.

For this particular study it can be argued that because the case material is publicly available and up for the taking, everything is really in place for doing secondary analysis' (for those who actually take the time to go to the source) – hence fulfilling the part of providing access to the original data material. In addition I employed a variant of a case study database where I recorded themes, discussions, and notes, by using the Zotero citation manager (Zotero.org, 2007), as illustrated in Figure 5.3 below.



- My Library
- Case
  - Amarok
    - Events
      - Høynivå
      - Potensielle
    - Gallery
- Knowledge
- Open Source
  - Basics
  - Business
  - Communication
  - Licensing
  - Motivation
  - Projects
- Quality

- systray icon animation, [was: greetings]
  - Diskusjon om klokka skal gå fra venstre mot høyre eller motsatt. ...
- extragear/multimedia/amarok/src
  - hva er mest intuitivt når man bruker mouse-wheel for å scrolle
- Fwd: I want Amarok for video!
  - videodebatten
- Long playlist causes long startup
  - interessekonflik om playlists (lange)
- The Road to KDE 4: Amarok 2 Development is Underway

Web Page
Title: extragear/multimedia/amarok/src
Author: (last), (first)
Abstract:
Website Title:
Website Type:
Date:
URL: http://mail.kde.org/pipermail/amarok/2006-Novem...
Accessed: 12. mai 2007 14:28:29
Language:
Rights:
Extra:
Date Added: 12. mai 2007 14:28:29
Modified: 12. mai 2007 14:28:29

Figure 5.3: Screenshot showing how Zotero was used to keep track of events and topics in the data material.

5.2.4 Data Analysis

As the data collection moved forward I gradually started to analyze the incoming data – interpreting what I saw and trying to put it all together. The problem with data like these – typically referred to as *process data* – is that they are 'messy' and eclectic in nature – involving multiple levels and units of analysis with ambiguous boundaries (Langley, 1999). To help solve this problem Langley (1999) presents seven strategies for making sense of process data, summarized in Table 5.2 on the next page. I will now present the strategies employed in this study, namely the grounded theory strategy and the narrative strategy.

Strategy	Description	Output	Strenghts
Narrative	Construction of a detailed story from raw data	Stories, meanings, mechanisms	A+, S-, G-
Quantification	Coding qualitative incidents for statistical analysis	Patterns, mechanisms	S+, A, G
Alternate Templates	Explanation through multiple alternative theories	Mechanisms	A+, S-, G-
Grounded Theory	Coding data bottom-up into emerging categories	Meanings, patterns	A+, S, G-
Visual Mapping	Visual representation of events and concepts	Patterns	A, S, G
Temporal Bracketing	Time slicing events into units of analysis	Mechanisms	A+, S, G
Synthetic	Operationalizing events to variables	Predictions	A, S, G

Table 5.2: Summary of the sensemaking strategies from Langley (1999). Strengths are defined as Accuracy (A), Simplcity (S) and Generality (G), ranging from + to - (high/low).

As noted above the data were collected and cross-referenced using the Zotero citation manager (see Figure 5.3 on the previous page). This allowed me to keep track of the original source, while also adding my own notes and related material. Using categories and labels/tags to code the data I gradually built up recurring themes and related episodes that would later form the basis of my analysis.

This process was inspired by the grounded theory strategy (Glaser and Strauss, 1967), but was not executed as rigidly as the original authors advocate – especially in terms of following very discrete steps and not having a theoretical backdrop for the study. Glaser and Strauss have later disagreed on how to handle mental baggage such as literature reviews and existing theories, but the dominant approach in information systems has so far been in line with Strauss: allowing the use of seed categories and drawing on previous knowledge and experience (Hughes and Jones, 2004). To me this pragmatic attitude seems quite reasonable in contrast to Glasser's almost utopian world of no preconceived ideas.

In fact I constantly moved back and forth between the observed data and the theories from my initial literature review – combining a bottom-up and top-down approach. Walsham (1995) describes this as one of three ways of using theory (the other two being as an initial guide, and as a final product). During the analysis I revised my problem definition and research questions several times as the results were crystallizing – which is typically part of the process in interpretive studies like this (Walsham, 1995).

Secondly, during the later stages of the analysis, I employed the narrative strategy, which aims to build a story-like description of all the details in the raw data (Langley, 1999). This approach can both work as a way of organizing and describing the detailed data along a time line – in so called "thick descriptions" – or as an analytic tool, where the many links and connections between events are examined. For me it was a combination of both, as I used this strategy to flesh out vignettes from the cases (see Chapter 7) which were then used as a basis for my analysis. In writing these vignettes I aimed at a *narrative-biography* approach, as described by van der Blonk (2003). This approach is identified by how it allows the complexities of the case to show, while still presenting them as a monologue from the researcher. The whole study follows a normal linear-analytic structure, as detailed by Yin (1994).

5.3 Evaluation

An important part of doing research is assessing the quality of our work, and how it generalizes to other situations. We touched upon this briefly in the previous section when discussing Yin's three principles for data collection (Yin, 1994), but this section will take a more systematic approach.

Traditionally quality has been operationalized through the concepts *validity* (are we measuring what we intended to measure) and *reliability* (can the measurements be repeated by others), but these two definitions are not particularly well suited for evaluating interpretive research. One way of solving this is by redefining the concepts, as advocated by Golafshani (2003). Another is to try to adhere to certain methodical principles and guidelines, which will in turn increase the chances of producing a satisfactory result.

Such principles can also be used for *post-hoc* analysis of research, which is what I will do next. As advocated by Walsham (2006) I will base my discussion on the seven principles of interpretive field studies, formulated by Klein and Myers (1999). Their structured nature makes it fairly simple to do a step by step evaluation of works such as this, but I will limit my discussion to principles that are particularly relevant to this thesis.

The fundamental principle of the hermeneutic circle It is hard to talk about any of the other principles without first looking at the principle of the hermeneutic circle, which acts like a basis for all the other principles. It summarizes the hermeneutic philosophy of "understanding a complex whole from preconceptions about the meanings of its parts and their interrelationships" (Klein and Myers, 1999, p. 71), and promotes applying this thought to the other principles in repeating cycles – from elements to whole, and back to elements. The philosophy was originally aimed at textual interpretations, but the concept of understanding a complex whole from its parts is known from other fields too, for example gestalt psychology and object-oriented software engineering.

Through my work I have tried to follow this principle by always keeping an eye on the big picture, while also revisiting each of other six principles at regular intervals.

The principle of contextualization This principle advocates that the "subject matter be set in its social and historical context so that the intended audience can see how the current situation under investigation emerged" (Klein and Myers, 1999, p. 73). Getting a common understanding between the reader and the author is important because it is not really possible to 'reset' an organization and start a study from scratch – so the context for *this particular study* has to be explained.

To fulfill this principle I have dedicated a separate chapter to introduce the two cases in detail – focusing on everything from history and the market they operate in to project structure and quality practices. This should give a solid understanding of the context for this study.

The principle of interaction between the researchers and the subjects This principle deals with how the 'extraction' of data never happens in a vacuum, but rather that the data forms out of the research process. Klein and Myers illustrates this nicely in the following quote:

> *In social research, the "data" are not just sitting there waiting to be gathered, like rocks on the seashore. Rather, interpretivism suggests that the facts are produced as part and parcel of the social interaction of the researchers with the participants.*

This has two implications. First, the researcher must be aware of how he or she affects the participants, thereby changing the results of the observation. In my case the observations were as direct as they can be – because I did not involve myself in the discussions on the mailing list or in the IRC channels. But, I may still have affected the results by contacting the projects in the first place – making the participants a bit more weary.

Secondly, the researcher must avoid constructing 'meanings' out of the observed data based on preconceptions about the case and its participants. Being eager to come up with interesting results, I may have misinterpreted some observations, or tried to fit them into theories that they didn't belong in. I tried to offset this danger by conducting a few brief interviews in the later stages of my study – to balance and verify my results with the project participants – and this worked out well.

The principle of abstraction and generalization This principle calls for the researcher to abstract theoretical, general concepts out of the empirical material, and to show the steps that took place in shaping these insights. Walsham (1995) describes four different ways of generalizing from interpretive research, and I employed two of these in my thesis. First of all, I provided rich insights that may inspire future work, and secondly, I discussed implications of the results for open source projects and for he wider software engineering community. Hopefully I also managed to present a plausible and persuading reasoning for my results, so that the conclusion actually matters.

The principle of dialogical reasoning This last principle focuses on identifying the fundamental assumptions that governed the research, and making them transparent to the reader. While I do explain my background and philosophical roots earlier in this chapter, there is a gaping hole in the part of explaining my initial intellectual basis and how it evolved during the study. That is perhaps the biggest down-point of this study, as I fail to show the various side-roads and detours that helped form the final ‘highway’.

One part of the explanation is probably that I was a relatively blank canvas when starting on my study. Except for some practical experience with open source projects, and rough knowledge about quality in software engineering and knowledge management issues, I had no theoretical framework to build on. This evolved as I was doing the literature review, but it still took some time before it set in and started to affect the work I was doing on the case material.

Another part of the explanation was perhaps that I unconsciously was trying to present my work in a linear and objective manner, to conform to a more positivistic view of ‘proper’ research. Now that I think about it I remember asking my supervisor early on if it wasn’t ‘cheating’ to not have my problem definition and research questions set in stone before starting data collection.

If I were to go back in time and do this study all over again, this principle would definitely get a lot more attention.

Chapter 6

Research Setting

This chapter aims to give a broad contextual overview of the two case study projects. First we look at their market segment, how they were founded, and the history up till today. Then we proceed to organizational issues such as project and code structure, before rounding of with a look at quality assurance practices and the tools employed for communication and cooperation.

6.1 Amarok

AMAROK is a music player for Linux and other Unix variants. The main responsibility of a music player is playback of audio, either from media files or from regular audio CDs. Media files are typically bought in online music stores, downloaded from file sharing networks, or ripped[1] by the users themselves.

The raw audio is usually encoded in a format designed to reduce file size, while maintaining audio quality. Examples include MP3, WMA, RealAudio, Vorbis, AAC and Flac – with MP3 being the most successful so far. The massive success of the MP3 format can largely be attributed to early support by audio players such as Winamp, and the lack of DRM restrictions inherent in some of the other codecs. Despite its reputation as a 'free' format the MP3 codec is actually patented and requires licensing. This has moved proponents of free software to recommend using the patent-free (and supposedly technical superior) Ogg Vorbis format instead.

6.1.1 The Codec Wars

The MP3 versus Ogg Vorbis battle is an example of one of the core issue of the digital audio landscape today, namely the right to fair use – both as defined by local laws, and in the minds of the general population. The digital audio market

[1] The process of transferring music from a regular audio CD to a computer.

is very competitive, with large players fighting for their share. Both hardware vendors, software vendors, and content providers want to secure their position and maximize their profits, which regularly results in format wars such as the classic Betamax versus VHS battle. The results for the end user are more often than not negative – in the form of vendor lock-in and incompatibilities; not to mention wasted money spent on systems or technologies that are depreciated faster than you can say MiniDisc.

One example of the former is how Apple iPods are notoriously difficult to use without Apple's own music software iTunes. Third-party software vendors are left with having to play catch-up each time Apple makes changes to the proprietary format of the iPod music database. The latest straw in the wind came when Apple recently introduced a new range of iPods, tagging along a change to the database-format that effectively locked the database to a specific device and prevented third-party software from modifying any of the data. Interestingly it only took the open source community two days to reverse-engineer the format and work around the issue, which shows how important it is for the community to be able to control their own media devices. Related to this issue is the on-going case (initiated by the Norwegian Consumer Council) against Apple, regarding its digital right management (DRM) system FairPlay. The DRM system prevents music bought in the online iTunes Music Store from being played at devices other than the iPod, which has been ruled as illegal by the Norwegian Consumer Council. The outcome of this case has yet to be settled, but the verdict will definitely mark an important precedence for similar cases in other countries.

Both of the cases above would be considered fair use – if not by a court of law (*de jure*) then surely by the average consumer (*de facto*). Managing the iPod's music library on a computer running Linux, or buying music from the iTunes Music Store and then listening to it on a generic brand portable player would seem like natural things to do for many people, but fact is often they can't. To be fair to Apple it should be noted that they are hardly the only company to employ these tactics to control the marked. On the other side of the pond is Microsoft, with their Windows Media Audio (WMA) codec and Zune media player. Adding to this are all the 'creative' DRM schemes employed by the music industry to control how consumers use the content. Going into the gory details of DRM technologies in general though would be too much for this thesis, so I'm going to leave it at that. For more information about these topics I refer to web-site of the Electronic Frontier Foundation – an organization dedicated to defending users freedom in the digital world (EFF.org, 2007).

Many of the observations above can be interpreted as companies trying to build and protect closed 'standards' to support their business model (rather than building open and shared infrastructures for everyone to use). By taking advantage of mechanisms like path dependency and network externalization (Hanseth, 2000) to increase the install-base of their product they hope to gain increased profits and a dominance in the marked. For example, talking major record companies into selling music exclusively in your format gives positive feedback for users to buy a media player supporting that format, which again creates a path dependency the next time the user needs to upgrade any of his software or hardware (because all his music is now in that format).

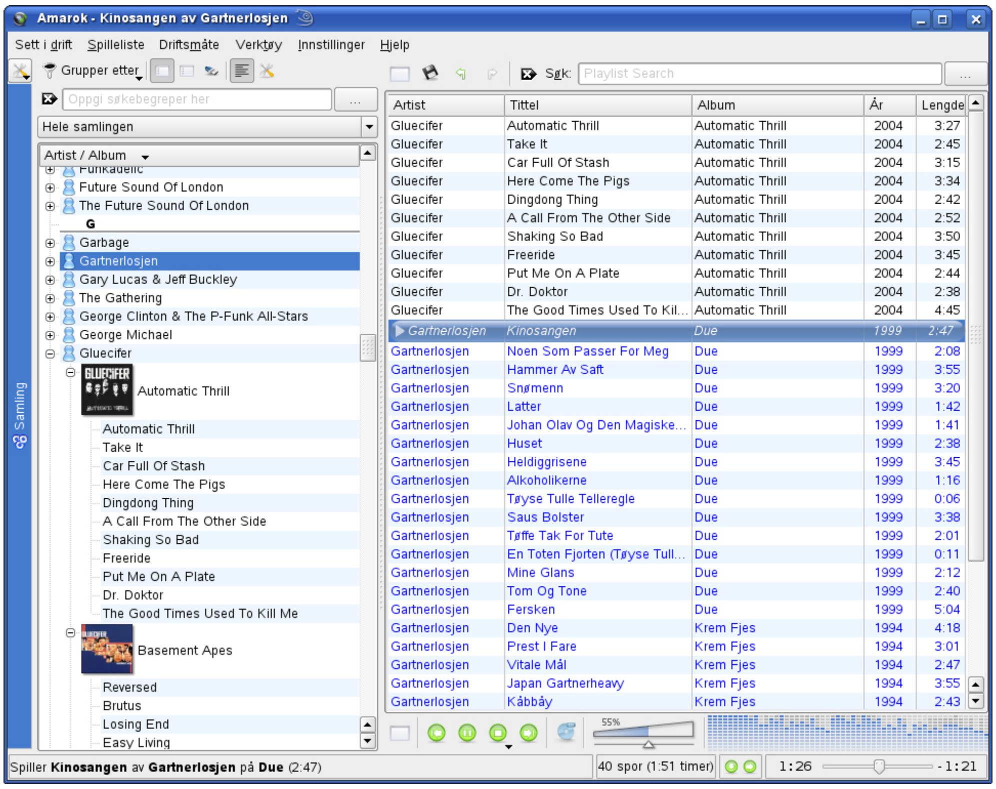
Considering all these road blocks to enjoying (legally purchased) digital music where-, when-, and how-ever one wishes it is easy to understand why open source media players and open formats thrive in this segment. They are an important counter-balance to restricted content and crippled software, because they allow users to play back their media files without jumping through all the technical and legal hoops of the big commercial vendors. In a way it creates a buffer between the user and the quickly changing dynamics of the market, as users are shielded from the effects of latest DRM fad or device-interoperability-problems.

6.1.2 Media Players in General

The earliest music players were very simple and usually only supported one or two formats. As people's digital music collections grew – much thanks to the success of the MP3 format – music players started adding features such as media management (browsing/navigating), meta-data editors (adding titles, track numbers and cover art to the media files), and playlist management. With the advent of the Internet music players added support for streamable radio stations and the ability to automatically fetch and update the media meta-data. Finally, with the recent explosion in portable media players, came the ability to synchronize music between devices.

Today there are a vast number of music players in the market, ranging from the minimalistic and simple, with limited but solid functionality, to the ones filled with so many features that they could even replace your toaster. On the Windows platform the dominant player is Windows Media Player, much thanks to Microsoft bundling it with their operating system. This practice has been criticized as being unfair and monopolistic, and in 2004 Microsoft was ordered to pay a €497 million fine in a European Union antitrust suit because of this. Their appeal was recently rejected in the union's second-highest court, and Microsoft has yet to decide if they will take the case all the way to the European Court of Justice.

Similarly, on OS X, the dominant player is iTunes – which as discussed earlier is also the player of least resistance for people owning an iPod. On Linux the situation is a bit more fragmented, but popular players include Amarok, XMMS and Rhythmbox. Amarok is usually the player bundled with the Linux desktop environment KDE, while the GNOME desktop environment includes players such as Banshee and Rhythmbox. A recent user poll on the Linux website LinuxQutestions.org landed Amarok the award for "Audio Media Player Application of the Year" with 57% of all votes (LinuxQuestions.org, 2006).



Funkadelic
Future Sound Of London
The Future Sound Of London
G
Garbage
Gartnerlosjen
Gary Lucas & Jeff Buckley
The Gathering
George Clinton & The P-Funk All-Stars
George Michael
Gluecifer
Automatic Thrill
Automatic Thrill
Take It
Car Full Of Stash
Here Come The Pigs
Dingdong Thing
A Call From The Other Side
Shaking So Bad
Freeride
Put Me On A Plate
Dr. Doktor
The Good Times Used To Kill Me
Basement Apes
Reversed
Brutus
Losing End
Easy Living

Artist	Tittel	Album	År	Lengde
Gluecifer	Automatic Thrill	Automatic Thrill	2004	3:27
Gluecifer	Take It	Automatic Thrill	2004	2:45
Gluecifer	Car Full Of Stash	Automatic Thrill	2004	3:15
Gluecifer	Here Come The Pigs	Automatic Thrill	2004	3:34
Gluecifer	Dingdong Thing	Automatic Thrill	2004	2:42
Gluecifer	A Call From The Other Side	Automatic Thrill	2004	2:52
Gluecifer	Shaking So Bad	Automatic Thrill	2004	3:50
Gluecifer	Freeride	Automatic Thrill	2004	3:45
Gluecifer	Put Me On A Plate	Automatic Thrill	2004	2:44
Gluecifer	Dr. Doktor	Automatic Thrill	2004	2:38
Gluecifer	The Good Times Used To Kil...	Automatic Thrill	2004	4:45
Gartnerlosjen	*Kinosangen*	*Due*	*1999*	*2:47*
Gartnerlosjen	Noen Som Passer For Meg	Due	1999	2:08
Gartnerlosjen	Hammer Av Saft	Due	1999	3:55
Gartnerlosjen	Snømenn	Due	1999	3:20
Gartnerlosjen	Latter	Due	1999	1:42
Gartnerlosjen	Johan Olav Og Den Magiske...	Due	1999	1:41
Gartnerlosjen	Huset	Due	1999	2:38
Gartnerlosjen	Heldiggrisene	Due	1999	3:45
Gartnerlosjen	Alkoholikerne	Due	1999	1:16
Gartnerlosjen	Tøyse Tulle Telleregle	Due	1999	0:06
Gartnerlosjen	Saus Bolster	Due	1999	3:38
Gartnerlosjen	Tøffe Tak For Tute	Due	1999	2:01
Gartnerlosjen	En Toten Fjorten (Tøyse Tull...	Due	1999	0:11
Gartnerlosjen	Mine Glans	Due	1999	2:12
Gartnerlosjen	Tom Og Tone	Due	1999	2:40
Gartnerlosjen	Fersken	Due	1999	5:04
Gartnerlosjen	Den Nye	Krem Fjes	1994	4:18
Gartnerlosjen	Prest I Fare	Krem Fjes	1994	3:01
Gartnerlosjen	Vitale Mål	Krem Fjes	1994	2:47
Gartnerlosjen	Japan Gartnerheavy	Krem Fjes	1994	3:55
Gartnerlosjen	Kåbbåy	Krem Fjes	1994	2:43

Spiller **Kinosangen** av **Gartnerlosjen** på **Due** (2:47)
40 spor (1:51 timer)
1:26
-1:21

Figure 6.1: Screenshot of Amarok version 1.4, showing the two-pane layout 'signature' UI of the application.

6.1.3 Background and History of Amarok

Amarok was founded in 2002 by Mark Kretschmann, who, frustrated about the minimalistic nature of the XMMS player, decided to roll his own[2]. The original amaroK was built on the design of the popular file manager Midnight Commander, with its two-pane drag and drop layout. The idea was to have the the music collection in the left pane, and the current playlist in the right, letting the user drag songs from the collection to the playlist to queue them up for playing. Figure 6.1 above shows a screenshot of the current stable version of Amarok. It is fairly easy to see that the two-pane layout has lived on to this day. The music collection to left is represented as a tree-structure with artists and albums as sub-nodes, and the playlist to the right lists all the tracks that are queued up for playing, represented by a table with columns such as title, album and year.

After working on the code in solitude for a couple of months Kretschmann released the first public version of amaroK – a so called "pre-alpha" release – in the summer of 2003. It was followed up with version 0.51 later that year. The application was pretty basic and very unstable – miles away from its current state. But, it attracted enough fresh developers to continue the development towards a stable 1.0 release. Figure 6.2 on the next page shows a time-line of how the

[2] Story has it that the XMMS 'play' button was only 1x1 pixel in size, so he kept hitting stop instead of play when he wanted to listen to music.

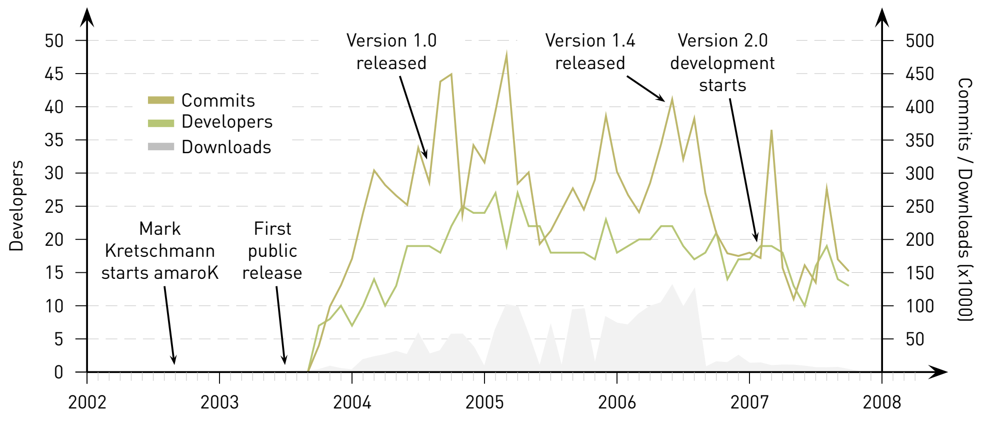


Figure 6.2: Key events and phases of the Amarok project (SourceForge.net, 2007a). The green line shows the number of developers with commit access to the source code repository, and the brown line shows number of commits that month. The gray area shows the number of source code downloads from the open source website SourceForge.net each month (in thousands), but only represents part of the picture since Amarok utilizes additional repositories for download and is typically distributed in binary form with Linux distributions. The drop in downloads in September 2006 is due to the Amarok project no longer hosting downloads at SourceForge.net.

project has developed over the years. Following the green line we can see how the project quickly grew past the 10-developers mark, and had almost 20 developers at the time of the 1.0 release in the summer of 2004.

The important 1.0 release included many of the features that are now part of the core functionality of amaroK, like a collection browser and the ability to edit meta information for tracks. The release was also based mainly on the sound streaming back-end GStreamer[3], but had basic support for other popular streaming engines like Xine and aRts. Subsequent minor releases in the months that followed saw a gradual shift towards more mature engines, and in version 1.3, released August 2005, the the GStreamer engine was finally replaced by the Xine engine as the default recommended sound back-end. The 1.3 version also included features such as Wikipedia artist lookup, dynamic playlists, and support for podcast streaming and downloads.

Especially the Wikipedia artist lookup feature was a natural step towards building a more contextual music player, as it automatically downloaded the biography and discography of the band you were listing to and presented them in a sidebar of the player. The dynamic playlist feature was a compliment to the already existing feature called smart playlists. Together these two allowed the user to create endless playlists based on queries such as “tracks that I have rated with four or five stars but haven’t listened to in the last three months” – a feature adopted from competing music players such as iTunes.

[3] Amarok doesn’t actually play any raw audio itself, but acts as a front-end for the various low level codecs that do the audio processing and playback. GStreamer is one such low-level audio back-end for Linux.

The most recent minor release of amaroK came in the summer of 2006 with the 1.4 series – titled “Fast Forward” – which added support for portable media devices, music sharing via DAAP, radio streaming from online radio stations like Last.fm and Shoutcast, and integration with the Magnatune online music store. The project was also renamed to Amarok, without the upper-case ‘K’ at the end.

Support for portable media devices was of course an important feature considering the marked growth of the iPod and similar players, and thanks to the reverse-engineering efforts of the open source community Amarok could now synchronize music just as easily as Apple’s own iTunes software. Music sharing over the DAAP protocol was another important feature, as it allowed Amarok to play music shared by other applications – among them iTunes. Unfortunately Apple changed the encryption scheme in the latest version of iTunes (version 7.0), incendentally breaking the compability with third party players in the process. This issue has still not been resolved by the community.

Today Amarok can best be described as a feature-rich music player, sporting functionality such as an ‘intelligent’ collection browser (showing you related bands and facts from Wikipedia on the fly), support for various portable media devices, automatic tagging of music, several integrated online music stores, and of course support for most audio formats you can think of. One feature that Amarok has purposely avoided though is video playback. This has probably helped the project focus on what it does best – music. The Amarok tag-line is “Rediscover Your Music”, and considering its many contextual features it seems like a fitting description of what it does.

The next step for Amarok will be the important 2.0 release, which will be based on the forthcoming KDE 4.0 project and Qt 4. This will allow Amarok to be run on both Windows and OS X too, thanks to the cross-platform features of the graphics toolkit Qt. Moving to a new version of both the graphic toolkit and the base libraries of KDE involves a pretty major rewrite of the application, and while this is a risk in itself, it also allows the developers to re-think the various parts of the application – redesigning the GUI and adding features that might be long missed. Ideally the Amarok-team would like to release version 2.0 of Amarok at the same time as KDE 4.0 is released: December 11, 2007. This deadline, combined with a slight decrease in the number of active developers in the last year, puts some extra pressure on the developers, but so far the spirit seems good and progress is going well (despite the project not receiving any Google Summer of Code developers this year).

As far as users go the user base has grown significantly since the initial 1.0 release, and Amarok is now included by default in most serious Linux distributions. The project also builds their own stipped down Linux distribution called Amarok Live, which aims to provide an environment where Amarok can be booted directly from a CD and run without installing anything. This will allow users to test Amarok before installing it, or bring music to friends for parties or social gatherings with a free software player included.

6.1.4 Project Structure and Organization

Amarok has traditionally been developed by the "holy trinity of core developers" – Mark Kretschmann, Max Howell, and Christian Muehlhaeuser – often referred to as only MMM. These developers are still very active, but the project has over the years grown to a core of about 8-10 developers. Adding to that are more peripheral developers, plus users creating scripts for the built in plug-in system. Amarok also has an artist team of about nine members who do graphics for icons, splash screens and themes, as well as creating promotional material for the promotion team called Rokymotion. This team consists of ten members. There is some degree of overlap between these teams, but most contributers have a pretty focused role in the project.

The source code itself is currently branched into two separate trees. One is the 1.4.x branch, which contains the stable and working version of Amarok, polished through multiple iterations since the initial 1.4 release. The other is the development version of Amarok 2.0, which is based on a brand new version of the KDE and Qt libraries. This branch is mainly for developers, but gutsy users are encouraged to give it a go and report bugs. Both branches are hosted on KDE community servers, using the source code revision system Subversion. This allows concurrent development and easy comparison between different versions of the same file.

The code is also structured into components by functionality, improving the modularization of the project. A complete list of the Amarok components are listed in Table 6.1 to the right, together with the number of open bugs for each component as of 14. April 2007. This should give a rough indication of their relative size, assuming that the complexity and bug density of the individual components are comparable. As we can see the "general" component has the most number of open bugs, but this can in many cases be traced back to bug reporters not knowing or caring which component a bug is reported for. Many of these bugs are also duplicates of the same issue.

Component	Crash	Normal	Wishlist	Total
AFT		1	1	2
Collection	2	35	26	63
CollectionBrowser	1	14	24	39
ContextBrowser	1	10	25	36
DAAP		3	1	4
Engines	1	6	1	8
FileBrowser		3	2	5
Helix		1		1
Magnatune		3	4	7
Mediabrowser	6	23	24	53
Moodbar		1	1	2
OSD		5	8	13
Playlist	2	17	29	48
PlaylistBrowser		10	19	29
Podcast	2	7	14	23
general	31	186	233	450
scripts	1	7	4	12
xine		7	1	8

Table 6.1: Components in Amarok, and the number of open bugs in each category. The types Crash, Normal, and Wishlist note the severity of the bug.

6.1.5 Quality Practices and Development Infrastructure

Amarok is a project which strives to provide its users with not only a solid music player but also cutting edge features. In a highly competitive market such as media players this can sometimes lead to prioritizing, where stability or security is sacrificed for a new feature which might set the product apart from the rest of the crowd. In Amarok this is part of the general consensus of how development should work. As put by one developer: "[although] we fix whatever bugs we find, we don't do a lot of heavy unit/regression testing when we add new features. In many ways, we rely on our own use of Amarok and of those that use Amarok built from SVN to help us find any bugs that get introduced" (AmarokDev02).

This development model resonates well with how open source development has been described in the literature (Raymond, 1999a; Vixie, 1999; Zhao and Elbaum, 2003), but also emphasizes the importance of having ways for users and developers to communicate about the bugs that are found.

Currently Amarok utilizes four tools to support communication between users and developers:

Forum The discussion forum located at `http://amarok.kde.org/forum/` is perhaps the most user-friendly of the four communication mediums. The user interface is web-based and follows normal interaction patterns for such forums. Reading the forum is open to anyone, but posting new topics or replying to old ones requires registration. The registration process is simple, fairly anonymous, and does not bear the disadvantage of being flooded by messages unrelated to your topic, like for example e-mail lists. Most of the discussions in the forums are support-related, and requests for new features or actual bug reports are usually referred to the bug-tracker or mailing list.

Mailing-list The mailing list `amarok@kde.org` has much of the same content as the the forums, but leans more towards development discussions rather than support requests. The list is archived and available on the web, but most people subscribe to the list and receive new posts in their e-mail inbox. Contrary to the forums the mailing list does not require registration to post new messages. This can sometimes lead to trouble when people reply only to the list and not to the original poster, not realizing that the original poster does not subscribe to the list.

Recently another list was created called `amarok-dev@kde.org` – probably in an attempt to get away from some of the noise and spam the general list was seeing. In addition to these two lists it is also possible to subscribe to changes in the Subversion repository, which makes it a lot easier go get a "feel" for the dynamics of the code base.

IRC Amarok also utilizes IRC, short for *Internet Relay Chat*, to provide synchronous chat between users and developers. The channel #amarok hosted on the IRC network Freenode averages about 150 users, and does not require any registration other than a basic IRC client. Most discussions are pure support requests, and the ratio of 'noise' is fairly high – exemplified by the frequent quiz competitions, sporadic off-topic discussions, and 'bots' (automated scripts) replying to unknowing users in classic Eliza-style phrases[4]. This level of noise is not uncommon for the IRC medium, and the lack of developer discussions is also understandable in light of the discovery of a second developer channel, as discussed earlier in Section 5.2.3.

Bug-tracker The bug tracker at `https://bugs.kde.org/` is the medium of choice for technical discussions about the code, as well as for more high-level design problems and feature requests. The bug tracker service is hosed as part of the larger KDE project, so any changes to the code base is reflected in weekly KDE commit digests (Commit-Digest.com, 2007).

Submitting new bugs requires registration, but the registration process is simplified into entering your name and a valid e-mail address – probably to lower the barrier for reporting new bugs. After registration a wizard will guide you through the steps of reporting the bug, making sure the user first searches for existing reports of the same bug, and that he or she includes important information such as application version and build environment. Special fields are used to signal the nature of the issue (bug, feature request) and if the issue has been resolved.

In addition to the communication-tools described above the project also hosts a Wiki with solutions to common problems, information for new developers, and design proposals and mockups (graphical sketches) of new features. Amarok also benefits from the KDE project's automated code checker and API validator EBN (English Breakfast Network), which continuously scans the source code repository for common programming pitfalls, as well as esoteric API uses that might fail once in a blue moon.

6.2 Gallery

GALLERY is a web-based image gallery/photo sharing application written in the PHP programming language. An image gallery is the digital equivalent of a traditional photo album – a way to store and present images in a visually pleasing yet practical and manageable fashion. The user typically imports images from sources such as digital cameras or scanners onto his hard drive, and then incrementally adds these to the image gallery to build a portfolio or presentation of their life.

Although digital editing and manipulation of images is very common these days, this step is usually performed by special software (like Photoshop, or the Linux equivalent The GIMP), leaving the main responsibility of the image gallery to taking care of management and presentation of the edited images. The purpose of a web-based image gallery is to not only keep a local copy of your gallery but to also share the content with the rest of the world, through a web site. By making

[4] ELIZA, designed by Joseph Weizenbaum in 1966, was one of the first AI programs to simulate a human beeing. By paroding a Rogerian therapist it rephrased any input from the user as a new question, making it seem "intelligent" (until you realized its simple algorithm).

the images accessible on the Internet the user can easily point friends and family to the pictures from his latest vacation without having to print extra copies or lend out the originals.

Sharing images on a web site is usually accomplished by uploading the images to a web server and then referencing the image by using the HTML image tag:

```
<img src="vacation-photo-01.jpg" alt="New York, New York">
```

This would of course be very tiresome for large sets of images, and that's where the image gallery software helps out. By automating the image presentation the user is left with the easy task of uploading new images to the web server. The image gallery software runs on the web server, reads the uploaded photos, and presents them on the Web. Apart from this basic functionality the features of image galleries varies enormously – from the basic collect-and-present approach to fully customized and themeable software packages with advanced features such as meta-data presentation and per-user-access sections.

6.2.1 Image Galleries in General

The first online image galleries came about as a result of photo finishing businesses introducing online print ordering in the late 1990s. By digitizing the analogue film, or uploading the digital media directly, the customer could then pick and choose which photos to order – specifying the size and number of copies on the fly. Because the photos were kept on the site even after the initial order the customer could go back and re-order photos shot months or even years earlier without doing any extra work.

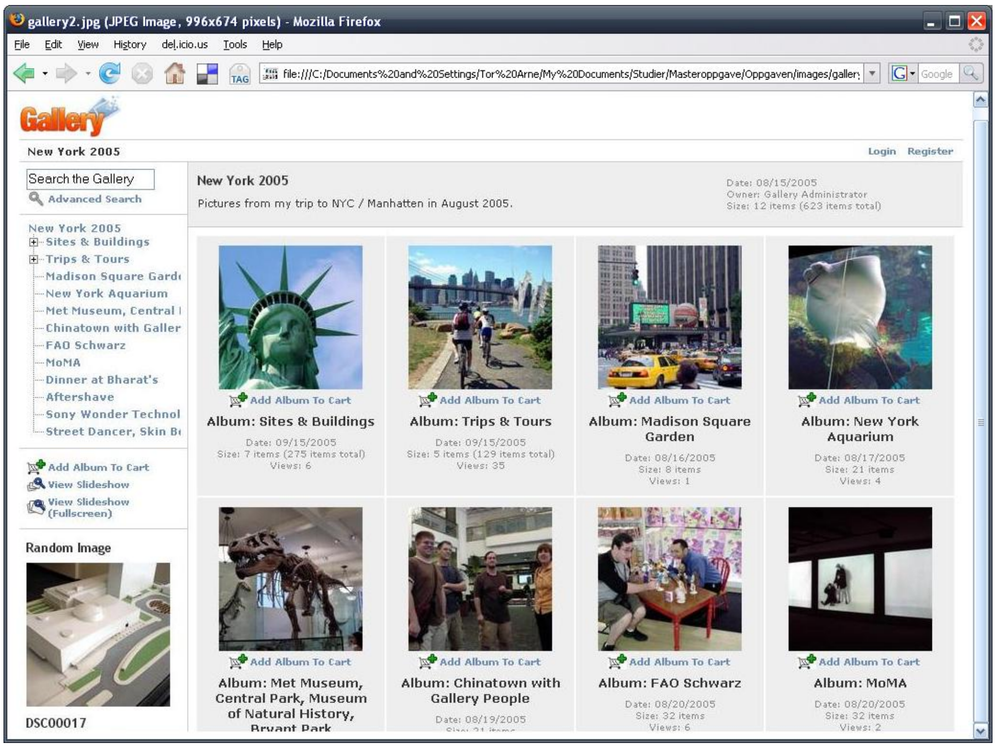
During the early 2000s the Web was flooded with image gallery applications in all shapes and sizes. Some of them were web-based, running as server-side scripts on a web host, generating content on the fly, while others used the approach of generating the HTML for the presentation off-line, at the user's computer, and then uploading the static content to a web server. The quick growth in image gallery software at the time can probably be attributed to the combination of a lack of existing proper software with the fact that everyone wanted to write their own implementation. Image galleries for home pages (that's what they were called back then) was *the* home-grown software of the early Web, along with classics such as guestbooks and hit counters.

After a few years of the free-for-all mayhem the larger and more organized projects drew enough developers to become stable entities in the image gallery software landscape, while the smaller one-man-bands either joined the larger projects or settled with the thought that their image gallery would never reach critical mass and subsequent worldwide domination.

Along the the way a number of commercial interests had picked up on the trend, and soon commercial image hosting services started to pop up. Not only did they provide the user with ready to use gallery software, but they also offered the disk space needed to store all those images. This was a important selling point, as most free web hosting services didn't provide enough disk space to host any sensible number of images.

Today professional image hosting services and stand alone image gallery software live (somewhat) happily side by side. In many ways they cater to two different crowds. The former are usually simple, not very customizable, and cost money if you want to host large galleries. At the same time you don't have to do much work to get images up on the Web, which is a big plus for many people. The latter are more complex, but easily customizable and integrates well with other Web applications such as blogs. You are also in total control of your images since you host them yourselves. The downside is of course that you have to set things up manually, so the initial barrier is a bit higher. It all comes down to personal preference.

Alexa, one of the largest web traffic analyzers today, list Photobucket, Flickr, and Yahoo! Photos as the top three image hosting/photo sharing services (Alexa.com, 2007). The third of these is no longer active, as Yahoo recently bought up Flickr and are now marketing this as their new image hosting service. When it comes to image gallery software the major players are Gallery, Coppermine, and LinPHA (SourceForge.net, 2007b), but a search for "gallery" on SourceForge.net, sorted by descending registration date, reveals that the grass root one-man-bands that lay the foundation for these projects almost a decade ago are still very much alive even today.



New York 2005

Pictures from my trip to NYC / Manhatten in August 2005.

Date: 08/15/2005
Owner: Gallery Administrator
Size: 12 items (623 items total)

Album: Sites & Buildings
Date: 09/15/2005
Size: 7 items (275 items total)
Views: 6

Album: Trips & Tours
Date: 09/15/2005
Size: 5 items (129 items total)
Views: 35

Album: Madison Square Garden
Date: 08/16/2005
Size: 8 items
Views: 1

Album: New York Aquarium
Date: 08/17/2005
Size: 21 items
Views: 4

Album: Met Museum, Central Park, Museum of Natural History, Bryant Park

Album: Chinatown with Gallery People
Date: 08/19/2005

Album: FAO Schwarz
Date: 08/20/2005
Size: 32 items
Views: 6

Album: MoMA
Date: 08/20/2005
Size: 32 items
Views: 2

Figure 6.3: Screenshot of Gallery2 in action, showing an album from New York.

6.2.2 Birth and Development of the Gallery Project

Gallery was started by Bharat Mediratta in May 2000, after buying a digital camera and thinking "wouldn't it be nice if I could put these photos somewhere". His initial attempts to publish them on the Web was a painful experience, so he created a few PHP script to handle thumbnails and navigation – and before he knew it he had a small image gallery on his hands. The first version was published in June the same year (after a friend asked for a copy) and it quickly attracted additional developers. Suddenly Bharat found himself leading a growing community of developers who were excited by the possibility of hosting their own image gallery. The Gallery project was born.

After a couple of initial beta releases Bharat finally released version 1.0 of Gallery in April 2001 (see Figure 6.4 below for a time-line of releases and activity-levels in the project). The release was followed by a number minor releases in the year that followed – adding support for features such as nested albums, photo-commenting, EXIF meta-data, and the ability to host Gallery on web servers running the Windows operating system.

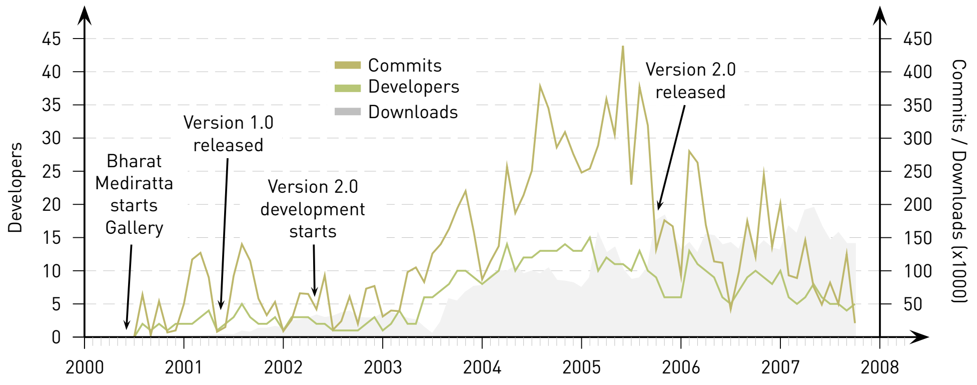


Figure 6.4: Key events and phases of the Gallery project (SourceForge.net, 2007c). The green line shows the number of developers with commit access to the source code repository, and the brown line shows number of commits that month. The gray area shows the number of source code downloads from the open source website SourceForge.net (in thousands). The statistics represent both the Gallery 1.x branch and the Gallery2 branch combined.

The nested albums feature was a welcomed addition, as it let users create top-level albums such as 'Concert Photography', 'Life/Misc', or 'Maco Shots', and then use the top level albums to categorize the individual photo-sessions. Photo commenting was also a popular feature, as it added the element of feedback to posting photos. A lot of photographers welcome critique and tips/tricks from more experienced shooters, and having comment-fields on your photo-pages allowed exactly that. Finally, the EXIF meta data feature allowed essential information about how the photo was captured to be displayed in the margin or underneath each image. Typical data included shutter-speed, f-stop, ISO value, and focal length – all great tools to have when dissecting how the photo was created.

By the time of the 1.3 version, released in May 2002, the project had grown from a one-man show to a steady core of 3-4 four developers, and the project saw tens of thousands of downloads each month. The increase in both the number of users and developers highlighted a growing problem with the web-forums; they were becoming a bottleneck in the day-to-day communication in the project. As a result Bharat saw the need to abandon the web forums as primary means of communication, and instead opted to create a mailing list for user-to-user and user-to-developer interaction.

This was not the only problem arising from the growing interest in the project though. Over the year it had became more and more apparent that Bahrat's initial lack of experience with the PHP programming language and fundamental database concepts had made the source code a complete mess. Adding new features to the Gallery 1.x branch had become a slow, complex, and painful process. The development team therefore decided (after much internal discussion) to scrap everything and start over with a clean slate – using the community's new-found knowledge as a new basis. This resulted in the Gallery 2.x series, which saw its first stable major release, version 2.0, in September 2005. The development of Gallery 1.x did not halt completely though, and is currently maintained and improved in parallel with Gallery2.

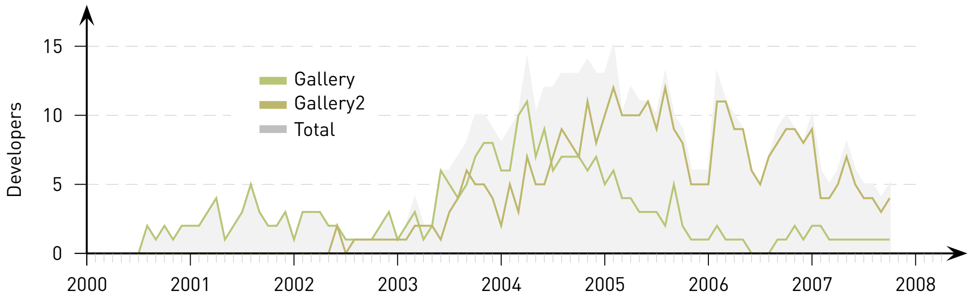


Figure 6.5: The graph shows the number of developers each month for the entire life-span of the Gallery project, broken down into the two sub-projects Gallery and Gallery2 (SourceForge.net, 2007c).

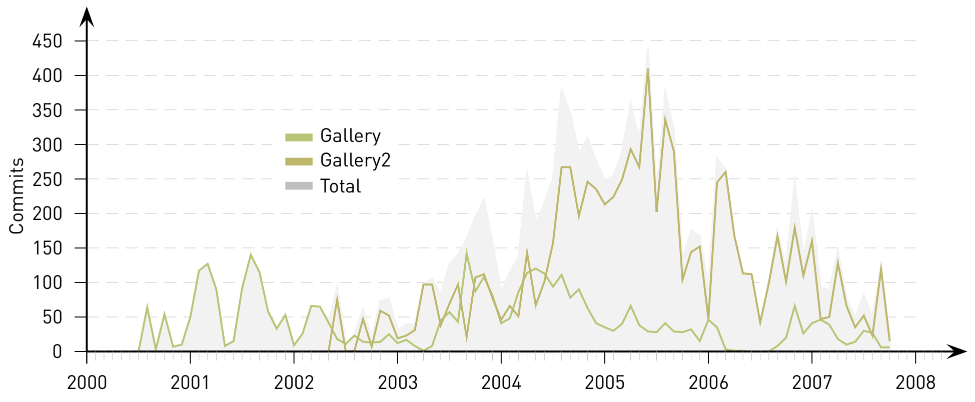


Figure 6.6: The graph shows the number of commits each month for the entire life span of the Gallery project, broken down into the two sub-projects Gallery and Gallery2 (SourceForge.net, 2007c).

Today Gallery has around five active core developers, and (according to the founders) over 300 000 users around the world. The software includes features such as per-user-permissions, a module system for easy customizeability, and drag and drop upload for easy photo management. In effect, Gallery does not only cater to the users that require the gallery to blend naturally with their existing web-site-layout, but also to those who's main goal is to get their pictures online without too much fuzz. That's probably one of the reasons for its success and widespread use.

6.2.3 Project Structure and Organization

The Gallery project consists of several teams that work together to build the complete solution. First of all there is the Gallery 2.x team, who deals primarily with the core functionality of Gallery2. This team has about 30 active members, where five of them are core developers. Complimenting this team is the Gallery 1.x team, who's job is to maintain the old 1.x series of Gallery. This team has about five active members, but only one primary maintainer who do all the commits.

Both these teams work with the Gallery Remote team, who's job it is to maintain a client-side Java application for configuring the two Gallery versions. Finally there's a support team, which also do paid supports, and a team who deals with the Gallery website. Many of the core Gallery developers have multiple roles in the project, and my act as both Gallery 1.x and 2.x developers while also providing support.

The Gallery code is naturally separated into two development trees – one for Gallery 1.x and one for Gallery2. There is a also a branch for Gallery Remote, the configuration client. The Gallery2 branch structures functionality into modules, with each module having a specific responsibility, and layout and design is put into user-editable themes. The modules talk to the Gallery API when they need database access or other low level features, and this is provided by the core module. This organization of code makes it very easy to extend Gallery with new functionality without messing up the base system. The full version of Gallery is shipped with 68 modules and 9 themes, providing a solid basis for customization. Table 6.2 on the opposite page lists some of the more popular modules.

6.2.4 Quality Practices and Development Infrastructure

Compared to Amarok the Gallery project has a slightly different take on where to draw the line between just enough and too much quality control. Where Amarok prioritizes adding new features quickly and then doing bug squashing as they go along, Gallery follows a thorough process of peer reviews and regression tests that need to be passed before any new code is entered into the code base – even to the point where it may create a bottleneck in getting new code into the project.

First of all, any changes to the code base are peer reviewed by a very small group of core developers, and the reviews are usually very strict. Then the code has to

Name	Description
Core	Gallery core functionality. Required module for all other modules.
Members	Members List and Profiles
Rearrange	Rearrange the order of album items all at once
Search	Search your Gallery
URL Rewrite	Enables short URLs using Apache mod_rewrite, ISAPI Rewrite, or PathInfo
User Albums	Create an album for each new user
Gd	Gd Graphics Toolkit
Zip Download	Download cart items in a zip file
eCard	Send photos as eCards
Dynamic Albums	Dynamic albums for newest, most viewed or random items
Flash Video	Enable display of Flash video files
ImageFrame	Render frames around images
Keyword Albums	Dynamic albums based on keyword search
New Items	Highlight new/updated Gallery items
Panorama	View wide jpeg/gif images in a java applet viewer
Slideshow Applet	Fullscreen slideshow using a Java applet
Square Thumbnails	Build all thumbnails so they are square
Watermark	Watermark your images
RSS	RSS feed
Comments	User commenting system
EXIF/IPTC	Extract EXIF/IPTC data from JPEG photos
MultiLanguage	Support item captions in multiple languages
Rating	Item Rating Interface
Add Items	Add items from local server or the web
Archive Upload	Extract items from uploaded zip files
Picasa	Import for Picasa 2 XML-Exports
Publish XP	Publish photos to Gallery directly from Windows XP
Remote	Implementation for the remote control protocol

Table 6.2: Popular Gallery modules. This list only represents a subset of the 68 modules available.

pass all of the roughly 2500 unit tests in the project – for all platforms and environments. This test array is especially useful around major releases. In addition the project hires external security auditors for major releases to ensure that the project has no bugs which could compromise the security of a web server.

One advantage of this through approach is, as explained by one of the core developers, that "because we keep our svn very stable and clean, we have quite a lot of users that use the svn version in production" (GalleryDev03). This should in theory increase the number of potential "eyes" that will scan the code for bugs that may have slipped through the internal peer review and regression tests. This fairly rigid approach is more in line with classic software engineering – especially the practice of doing elaborate testing of new code. At the same time the project draws from the "open source-effect" of having users spot and fix bugs in the nightly builds.

The infrastructure tools used by Gallery are very similar to those of Amarok, but how they are used varies slightly:

Forum Gallery's web forum hosted at http://gallery.menalto.com/forum has about the same functionality as the Amarok counterpart. Although the majority of topics are support-related – just as for Amarok – there are also sections for general discussion about Gallery's features and direction, plus sections for topics about 3rd party modules, which can sometimes be of a more technical form. Just as for the Amarok forums any error reports and feature requests are directed to the bug tracker.

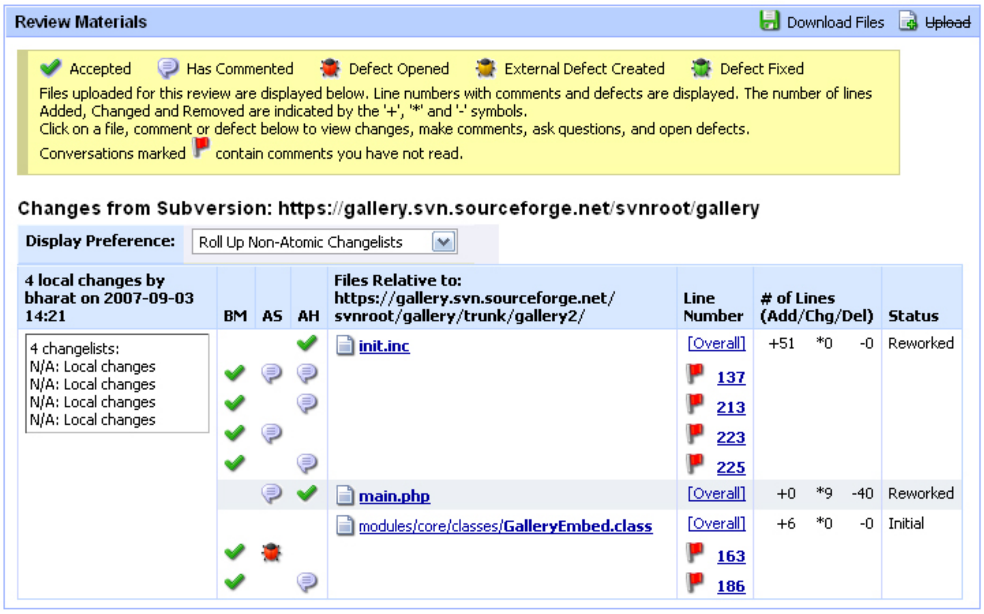
Mailing-list The mailing lists are hosted by SourceForge.net – the number one provider of free hosting services (bug trackers, mailing-lists, forums) for open source projects. There are several lists: one for announcements, one for the documentation team, and one for the translation team, but the most active lists are the development list and the source control checkins list. The former is where most discussions about new features and implementation details are taking place, and the latter provides developers with a notice every time someone makes a change to the source code repository. In addition to these public lists there is a mailing list for core developers only, but this is primarily used for security-sensitive discussions.

IRC Just as in the Amarok project Gallery uses IRC to provide synchronous communication between developers and users. The major difference between the two projects when it comes to this medium is that the support channel, #gallery-support, has a much higher signal-to-noise ratio, and that the development channel, #gallery, is open to anyone. This may be because a lot of 3rd party developers and integrators are dependent on the APIs that Gallery provides, and thus needs to be able to communicate with the developers, even though they are not part of the Gallery project themselves.

Bug-tracker The bug tracker is also hosted as part of the services offered by SourceForge.net, and is divided into four separate trackers: one for bugs, one for requests for new features, one for patches, and one for translations. This is a more clear-cut separation between types of issues than what the Amarok project uses, but at the same time this makes it a bit more cumbersome to re-classify a bug-report as a feature-request or vice versa.

Peer-review system In addition to the four mediums discussed above Gallery also uses a dedicated system to handle peer reviews. The review system, located at `http://reviews.gallery2.org/`, requires registration both to view and contribute to peer reviews, but registration is open to anyone. The introduction of this system to the project is fairly recent, but the team makes and effort to use it as often as possible – especially for external patches. The system itself allows developers to add code for review, and other developers can then sign up as reviewers and comment on the whole patch or even individual lines. Figure 6.7 to the right shows an example of this.

In addition to the tools mentioned above Gallery also uses a Wiki for information about everything from development practices and API details to user documentation and pictures of the lead developers. The use of wikis for documentation seems to be a popular approach among open source projects, perhaps because it increases the chance that someone will contribute to the common repository



Review Materials Download Files ~~Upload~~

✓ Accepted 💬 Has Commented 🐞 Defect Opened 🐞 External Defect Created 🐞 Defect Fixed

Files uploaded for this review are displayed below. Line numbers with comments and defects are displayed. The number of lines Added, Changed and Removed are indicated by the '+', '*' and '-' symbols.
Click on a file, comment or defect below to view changes, make comments, ask questions, and open defects.
Conversations marked 🚩 contain comments you have not read.

Changes from Subversion: https://gallery.svn.sourceforge.net/svnroot/gallery

Display Preference: Roll Up Non-Atomic Changelists

4 local changes by bharat on 2007-09-03 14:21	BM	AS	AH	Files Relative to: https://gallery.svn.sourceforge.net/svnroot/gallery/trunk/gallery2/	Line Number	# of Lines (Add/Chg/Del)	Status
4 changelists: N/A: Local changes N/A: Local changes N/A: Local changes N/A: Local changes			✓	init.inc	[Overall]	+51 *0 -0	Reworked
	✓	💬	💬		🚩 137		
	✓		💬		🚩 213		
	✓	💬			🚩 223		
	✓		💬		🚩 225		
		💬	✓	main.php	[Overall]	+0 *9 -40	Reworked
				modules/core/classes/GalleryEmbed.class	[Overall]	+6 *0 -0	Initial
	✓	🐞			🚩 163		
	✓		💬		🚩 186		

Figure 6.7: Gallery peer review system in action. Notice how the contributed code has been commented by several reviewers, but that there still is a red "bug" left (signaling issues with the code) that needs to be fixed before the patch can be signed off.

of knowledge instead of hosting their own little tutorial in a small corner of the Internet (effectively fragmenting the knowledge).

6.3 Summary

THIS CHAPTER has presented various sides of the two case projects – ranging from the market they operate in and their history to organizational issues, quality practices and development infrastructure. Some of the points we have covered are:

- How the media player market is very competative and saturated with roadblocks for the average user to enjoy his media.
- How the image gallery market has evolved from home-grown solutions to big companies providing hosting services.
- How the two projects were formed, and how they evolved into the software packages they are today.
- How development is organized, both in terms of teams and code structure.
- How the two project differ somehwat in their view of software quality.
- How the two projects employ tools in their daily quality assurance work.

Chapter 7

Case Vignettes

This chapter presents a small number of vignettes from the two cases presented in the previous chapter. Each vignette was selected on the basis of illustrating and highlighting recurring observations in the case material. Although a chronological presentation of all the vignettes would be preferable the asynchronous nature of e-mail lists and bug trackers makes this difficult. To remedy this problem each section starts off with an overview of the time-span of each vignette.

7.1 Amarok

As described in Chapter 6.1 the Amarok project has for the past year been busy polishing the 1.4 "Fast Forward" series, and working on the brand new 2.0 release. Because the status of the 2.0 branch is still quite experimental and not heavily used by the user community the vignettes for Amarok focus on the period leading up to the 1.4 release and the months following the release.

The following graph shows a rough overview of the time-span of the vignettes presented in this section:

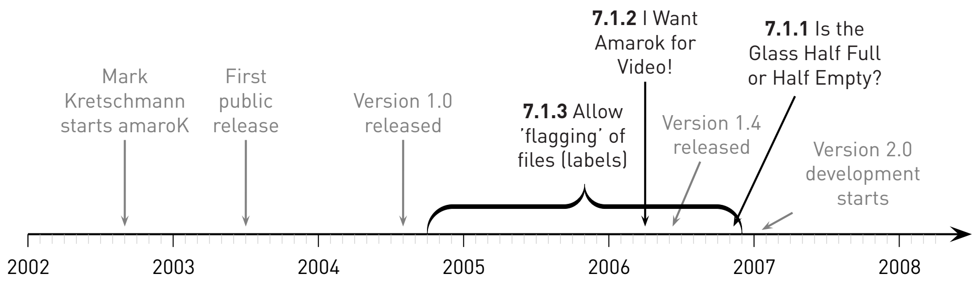
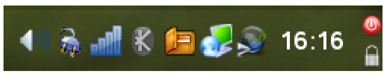

Mark Kretschmann starts amaroK
First public release
Version 1.0 released
7.1.3 Allow 'flagging' of files (labels)
7.1.2 I Want Amarok for Video!
Version 1.4 released
7.1.1 Is the Glass Half Full or Half Empty?
Version 2.0 development starts
2002
2003
2004
2005
2006
2007
2008

Figure 7.1: Time-span of the three vignettes presented in this section. Two of them are as short as one day, while one spans over two years.

7.1.1 Is the Glass Half Full or Half Empty?

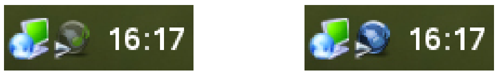
A typical use-case for Amarok involves the user loading up the application, adding some music to the playlist, and then starting playback. What happens next is usually that the user minimizes the application and starts doing something else on the computer. But, since the application is not closed – only minimized – the music keeps on playing in the background. This way of using the application helps reduce clutter on the desktop, as the application window is invisible and does not get in the way of normal work-flow, but still provides an important service to the user.



Applications designed to be used in the background like this sometimes have a feature that provides the user with quick access to the running application. The feature is usually implemented as an icon in the system tray of the operating system (the area in the lower right corner of the screen). Amarok follows this trend by having an icon that lets the user do common tasks such as skipping to the next track in the playlist or adjusting the volume – all while the application window is hidden. If more advanced adjustments are required the window is restored by double-clicking on the icon.

Most applications who adhere to this 'standard' use static icons, but some go beyond that and use animations to signal important events or activity. A network monitor application can for example signal network flow by flashing the icon, or changing it to another color based on the network load. The Amarok team added a variant of this feature in September 2004, in the form of track play-time visualization.

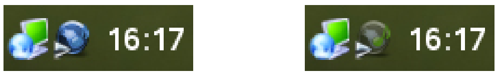
Instead of just showing the normal static Amarok icon like before, the new implementation gradually changed the icon as the track progressed. At the start of a track the icon was rendered in gray-scale, with no saturation (see Figure 7.2(a)). As the track progressed, the icon was gradually rendered with more and more color, from the bottom to the top – almost like pouring a glass of water. At the end of the track the icon looked exactly like the old static Amarok icon (see Figure 7.2(b)).



(a) Start of track (b) End of track

Figure 7.2: First version of play-time visualization. Icon is animated from 'empty' to 'full' – like pouring a glass of water.

The new feature was well received by the user community, as it let the user see how much of the current track had progressed by just glancing down at the Amarok system tray icon. The feature lived a happy life and remained unchanged for almost two years, until one day in July 2006 when core developer AmarokDev03 decided to change the direction of the animation. Instead of going from 'empty' to 'full', like before, the icon was now rendered at full color at the *start* of the track – gradually moving towards gray-scale as the track progressed (see Figure 7.3 below). In the commit message for the change the developer wrote: "let systray run out of saturation just like a sand timer runs out of sand", which is a good metaphor for what's happening.



(a) Start of track (b) End of track

Figure 7.3: Second version of play-time visualization. Icon is now animated from 'full' to 'empty' – like sand running out of an hourglass.

The change went unnoticed for a couple of months, but then reports started surfacing that questioned the change. A forum post on the Amarok forums asked how to revert the change, and on the 12th of October 2006 a user posted the following message to the Amarok mailing list:

```
[...]

I have a question for the developpers:
it is possible that there will be a "switch control" to
configure the icon in the "systemtray" (i hope that this
is the right name). now in the 1.4.3 the icon during the
song go to full to empty... and that's ok, but someone
don't (doesn't????) like this. so if it is possible can you
put a switch "on/off" and "full to empty/empty to full"?
```

A few hours later core developer AmarokDev01 replied, explaining that adding an option for this was not an alternative. Instead he appealed to the community to come up with a consensus on what would be the best option:

```
[...]

No, we don't really plan to make this optional. We think
adding "micro-options" like this makes the app cluttered
and reduces usability. Instead, we try to choose something
that works well for us and the majority of our users.

In this case, we can discuss what the best behavior for the
systray icon is. If others agree with your opinion, we
might possibly change it.

Asking the mailing list: What's everyone's opinion
regarding the systray icon?
```

[AmarokDev01]

The post spawned an interesting discussion on the mailing list. Some argued for the superiority of their favourite direction while others continued to press for a configuration option. Other approaches were brought up too, for example using the metaphor of a clock, or changing the icon completely.

In the end the discussion died, and no consensus was formed. The latest version of Amarok still uses the hourglass metaphor, going from 'full' to 'empty' and users seem to have settled with this choice.

7.1.2 I Want Amarok for Video!

As mentioned in Chapter 6.1 the Amarok team has always been reluctant to touch the field of video playback. Even though music and video are both digital media, the set of features required to play video is very different to that of playing music. For example, modern video playback is pretty useless without scaling/cropping functionality, subtitle support, and the ability to adjust and skip frames dynamically when CPU load is heavy.

The argument against transforming Amarok into a combined music and video player has always been that adding these video-only-features would complicate maintenance and turn Amarok into a huge 'beast' of an application. Apparently the Unix philosophy of writing applications that do one thing and do it well still lives on – even in inherently complex applications like a music player.

Naturally, with the history of Amarok on video support in mind, many eyebrows were raised when the following message appeared in the weekly KDE commit digest of 4th March 2006:

```
This week...

[...]

Initial video support in Amarok, with heavy interface
redevelopment underway for version 2.0.

[...]
```

Not surprisingly the report revived the debate on video support in Amarok, and people started posting messages to the mailing list stating their opinion on the subject. Some were happy that their favourite feature was finally going to be implemented, while others were scratching their heads – wondering why the Amarok team had turned completely on such an important issue. One user described his skepticism like this:

```
I'm not sure I like the way things are heading [...]. I
don't see why Amarok needs to be a universal media player.
In fact, I think Amarok is bordering on becoming a little
too heavy.

I do agree that there should be an application that can
index video files. It would be really useful with tags for
videos, similar to the ID tags that music tracks have. And
I've always thought that video files should have an
internal index of what different time intervals contain.
That way, the location bar in a video player could be
completely redefined, and one could automatically skip
credits or intros in tv shows for example.

BUT that is NOT a job for Amarok. Video indexing and all
that stuff should be done by a separate, dedicated
application. I want Amarok for music. I want it to be as
fast, light, and easy to manage as possible. Music and
video are two completely different media.
```

I didn't take long for one of the Amarok developers to step in and defuse the whole situation though. In a short message to the mailing list (that was also echoed to the various online message boards that had picked up on the story) he explained the reasoning behind the video change – assuring everyone that it did not mark the beginning of a new era for Amarok:

```
Just to clear things up here a bit, the video support which
will be included in Amarok is very simple and will (unlike
most people's impression) add exactly zero overhead. The
reason for this is that the engines already basically
handle the video streams (except for the decoding and
showing of them), and the way the actual visual will be
presented to the user is just like that: As a visual. If
you have the visualiser shown, then if Amarok detects a
video stream in the file, the visualisation will be
replaced with that video stream.

Think of it this way: Amarok is a music player. If your
music files have a video stream, then it can be argued that
this video stream is a visualisation of the sound, and as
such should supercede the generated visual.
                                           [AmarokDev06]
```

It turned out that the change merely built on one of Amarok's existing features: the visualization window. This feature had existed in Amarok ever since version 1.0, allowing add-on modules to 'tap in' on the audio stream and render fancy representations of the audio data to a separate window (an example of the *libvisual* library's *gforce* plugin can be seen to the right). The much debated code change basically just added functionality so that any time a music file included an embedded video stream it would be shown in the visualizer window (instead of the fancy hippie-animations). Situations where this would be natural includes files with embedded music videos, or slides for an audio presentation.

Despite this reasonable explanation some users probably feared that the innocent video change was a sign of things to come. To put the issue to rest once and for all the Amarok team followed up with a message in their weekly Amarok newsletter:

```
As agreed by the developers, Amarok will provide basic
video support, just enough to let you watch music videos
added to playlist. No advanced video stuff, like DVDs or
subtitles, for reasons discussed too many times.
```

Considering all the fuzz that was created because of one small change to the code it is ironic to note that the functionality was removed about a month later, with the following commit message: "Remove the video stuff completely, it will be better to start from scratch further down the road". What this means for video in Amarok in the future one can only speculate, but I for one hope that the developers stick to their initial plan – to create the best *music* player on the planet.

7.1.3 Allow 'flagging' of files (labels)

The human brain has a natural tendency to group stimuli into 'categories', or 'schemas', to aid the processing of these stimuli. Psychology uses this mechanism to explain a broad range of human behaviors – from stereotypes and prejudices to fight and flight responses (Aronson et al., 2001). Whether or not a result of the same mechanism, people have always enjoyed categorizing music into genres and styles, for example 'classic rock', 'techno', and 'J-pop'.

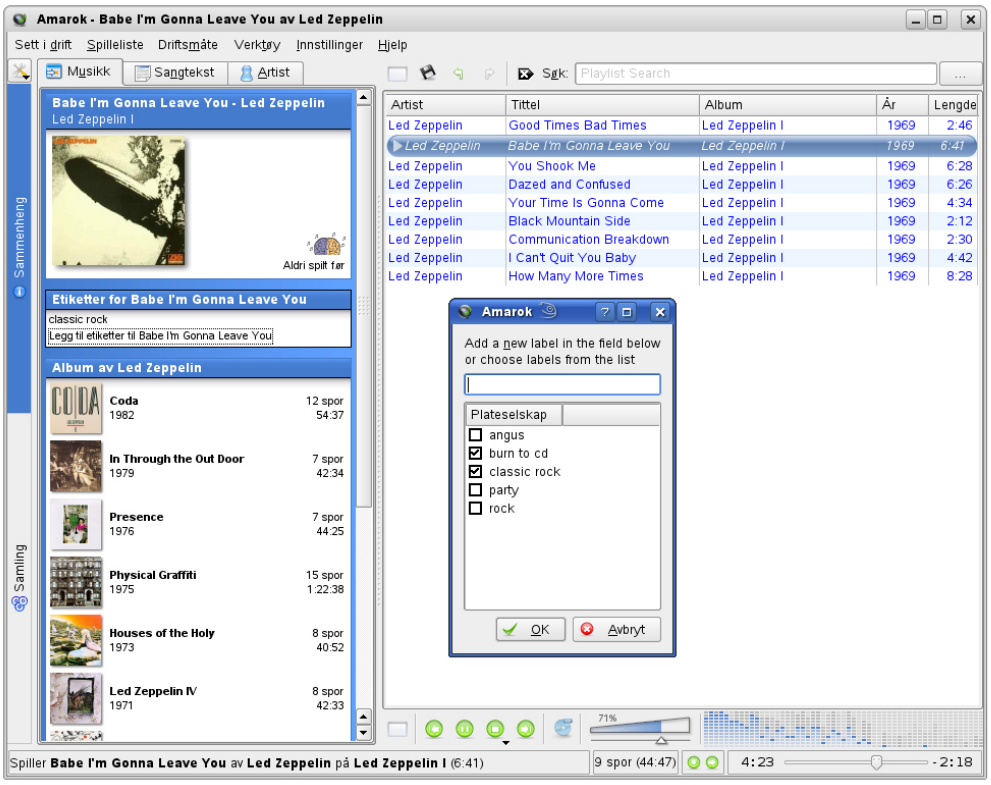
Although broad categories like 'rock' can fit a wide array of artists, the process of using top-down taxonomies like this for a highly subjective matter such as music is asking for trouble. What appears to you as 'low-fi industrial grungecore' might appear to me as 'alternative baroque nu-metal', or even 'ambient acid-rock'. As a consequence many applications started allowing users to label their music with arbitrary labels or *tags*, instead of pre-defined categories. In recent years this trend has evolved into online collaborative labeling/social tagging, also known as *folksonomies*. The idea being that the combined efforts of thousands of users will converge the labels to a subset of common 'accepted' categories – bottom-up instead of top-down.

This is an ingenious solution to many of the problems of standardization identified in Bowker and Star's case study of the International Classification of Diseases coding scheme (Bowker and Star, 1994). The "tension between the desire to standardize [. . .] and the drive of each [user] to produce and use its own specific [local] list" is effectively eliminated by allowing tags to not only be defined locally but also be aggregated into statistics and broad categories. A handy side-effect to this technique is that if you find a user with similar tastes in music you can often assume that his labels will overlap with your own definition of the music – which then allows you to navigate from user to user and artist to artist, discovering new music in the process.

Online music services like Last.fm are built entirely around the concept of folksonomies – enabling users to play custom radio stations based on both local and global tags, and of course Amarok has adopted some of these features too. Each track can have an arbitrary number of labels associated to it, which can later be user to build smart playlists such as "tracks not listened to in the last month and labeled as '*burn to cd*'". The Last.fm service has also been fully integrated into the player – allowing users to listen to radio streams using the same interface as they normally use for music playback.

But these handy features have not always been part of the product. In September 2004 a wish was filed in the Amarok bug tracker, asking for a variant of this functionality:

```
It'd be great to be able to right-click on a file and "Flag
as..." [and] then select from some user-defined list of
flags, which are accessible in a sidebar.

My user-interaction involves a lot of "hey, I like this, I
think i'll burn it to CD" or it's great for the gym so I go
'it's going on the mp3 player', or "i've already got this
```



Babe I'm Gonna Leave You - Led Zeppelin
Led Zeppelin I

Etiketter for Babe I'm Gonna Leave You
classic rock
Legg til etiketter til Babe I'm Gonna Leave You

Album av Led Zeppelin

Coda	1982	12 spor 54:37
In Through the Out Door	1979	7 spor 42:34
Presence	1976	7 spor 44:25
Physical Graffiti	1975	15 spor 1:22:38
Houses of the Holy	1973	8 spor 40:52
Led Zeppelin IV	1971	8 spor 42:33

Artist	Tittel	Album	År	Lengde
Led Zeppelin	Good Times Bad Times	Led Zeppelin I	1969	2:46
Led Zeppelin	Babe I'm Gonna Leave You	Led Zeppelin I	1969	6:41
Led Zeppelin	You Shook Me	Led Zeppelin I	1969	6:28
Led Zeppelin	Dazed and Confused	Led Zeppelin I	1969	6:26
Led Zeppelin	Your Time Is Gonna Come	Led Zeppelin I	1969	4:34
Led Zeppelin	Black Mountain Side	Led Zeppelin I	1969	2:12
Led Zeppelin	Communication Breakdown	Led Zeppelin I	1969	2:30
Led Zeppelin	I Can't Quit You Baby	Led Zeppelin I	1969	4:42
Led Zeppelin	How Many More Times	Led Zeppelin I	1969	8:28

Amarok

Add a new label in the field below or choose labels from the list

Plateselskap
- [ ] angus
- [x] burn to cd
- [x] classic rock
- [ ] party
- [ ] rock

Spiller **Babe I'm Gonna Leave You** av **Led Zeppelin** på **Led Zeppelin I** (6:41) | 9 spor (44:47) | 4:23 | - 2:18

Figure 7.4: Example of how Amarok uses labels to tag tracks for various purposes. (An interesting curiosity in this screenshot is the fact that the person who created the Norwegian translation of Amarok has interpreted the string 'Label' as meaning 'record company' – translating it into 'Plateselskap' instead of 'Merkelapp' or 'Etikett').

```
song, but i'm too lazy to figure out if this one or the
other has a higher quality, but i should get to that
someday'.

It'd be great to have a sidebar of 'Flags' (or whatever
name is appropriate), and then the context menu with "Flag
as ...", a submenu with all the Flags (ie. "Burn to CD",
"Copy to MP3", or "Check for dups", and "New flag"), and
when you want to look at all the flagged files, go to the
Flags sidebar.
```

Although the use-case put forward by the user only describes situations where tracks are flagged for later actions (burn to CD, find a better version) the general case would be labeling tracks for any purpose – even genre classification.

In the months that followed the only comments added to the report were Amarok developers marking other bug reports as duplicates of the original bug – noted by lines such as "Bug 104132 has been marked as a duplicate of this bug". This is a common way for open source projects to keep their bug trackers 'tidy', as it centralizes any discussion about a bug to one place.

It took almost a year for someone to provide a more detailed input to the report, which happened in August 2005:

```
As I've said in duplicate 110868, another option would be
to synchronize with the "tags" from the last.fm service.
That way you could save your tags in case of a system
failure as well as use the ones you've already filled in
last.fm.
```

The sudden attention to the report was not coincidental. Web-sites based on the folksonomy-principle were starting to take off – among them the Last.fm music service. It was only natural for people to start wondering and brain-storming about how this new technology could be incorporated into Amarok. Over an eight-month period the bug report saw several comments on everything from use-cases to technical implementation details.

At one point someone queried the Amarok developers about the status of the feature request, and if it would make it into the upcoming 1.4 version of Amarok, but was left with no answer. That cooled down the interest a bit, and it took five more months before someone added to the discussion. This time it was a user providing a series of detailed design propositions (over 1100 words long) for how the feature could work:

```
Here are some design suggestions I thought I should make to
help out, comments are welcome:

[...]

Oh, more design suggestions I forgot:

[...]

More musings:

[...]                                          [AmarokUser01]
```

In the comments he compared Amarok to various other applications of similar nature – which already had labeling/tagging functionality – and tried to use lessons and best-practices from these applications to form a good design for Amarok.

Aparantly someone in the Amarok developer-team had been working on a labeling feature for some time, because one month later, in October 2006, the following comment was added to the bug report:

```
SVN has now support for labels                  [AmarokDev07]
```

This was good news for everyone, and in most cases the bug report would be closed as 'fixed' right there and then, and never touched again. But a week later the user who added the detailed design propositions commented:

```
It doesn't seem advisable to close this request yet; there
are unimplemented details which would benefit from
discussion (e.g. the two pages of useful implementation
notes I took the time the generate above, which outlines
a nice way to make smart playlists and such as well as
other things).
                                               [AmarokUser01]
```

This seemed reasonable – after all he had put in some work in these proposals, and if the report were to be closed they would be 'lost' in the constant noise of open bugs. But the reply from one of the Amarok core developers was:

```
[...] as a general rule, we really dislike bug report
essays. (still, we really appreciate the effort, and make
time to read them).
```
[AmarokDev05]

Aparantly there was a limit to how much details to provide in a design proposition, and this particular user had crossed it. The report was closed for good, and did not receive any additional comments.

Although I personally don't think the Amarok developers were ignoring the propositions purposely (after all, the core developer did stress that they do appreciate the effort), it says something about how the bug tracker is used compared to the other tools and mediums available. We will get back to this discussion in Chapter 8.

7.2 Gallery

When it comes to Gallery the project developers have for the past two years focused most of their attention on improving Gallery2, or the 2.0 series, which was released back in September 2005. One or two developers have been maintaining the 1.0 branch, but because the activity level for the 2.0 branch has been much higher all the vignettes for this case stem from that branch.

The following graph shows a rough overview of the time-span of the vignettes presented in this section:

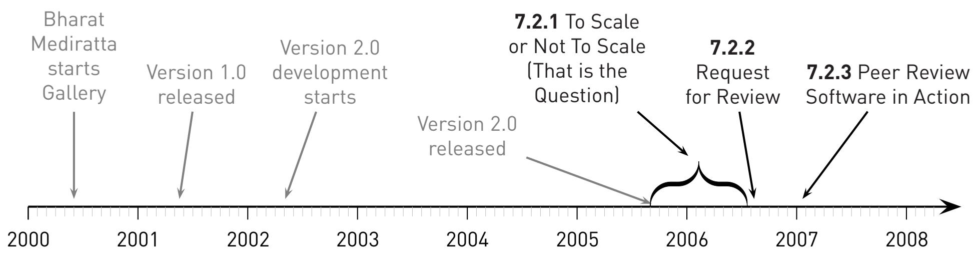

Bharat Mediratta starts Gallery
Version 1.0 released
Version 2.0 development starts
Version 2.0 released
7.2.1 To Scale or Not To Scale (That is the Question)
7.2.2 Request for Review
7.2.3 Peer Review Software in Action
2000
2001
2002
2003
2004
2005
2006
2007
2008

Figure 7.5: Time-span of the three vignettes presented in this section. Two of them are as short as one day, while one spans over two years.

7.2.1 To Scale or Not To Scale (That is the Question)

As described in Chapter 6 Gallery is structured into *modules*, to help organize the code into manageable pieces and to prevent tight coupling between the various features. Some modules provide core functionality, and these are always bundled with Gallery and activated by default, no matter the distribution. Others provide more esoteric features, and have to be downloaded and activated manually by the user.

One such module is the ImageBlock module, which extends the Gallery third-party API to include a function for retrieving a block of random (or recent) images from the image gallery. This is a very handy function for people who want to integrate their photos into the layout of 'external' websites without doing a full-fledged integration of the complete Gallery installation. A common example is showing a few recent photos in the sidebar of a blog, as illustrated in Figure 7.6 to the right:

The original implementation in Gallery of this function had the following (simplified) signature

```
getImageBlock(..., fullSize=false, maxSize=NULL, ...)
```

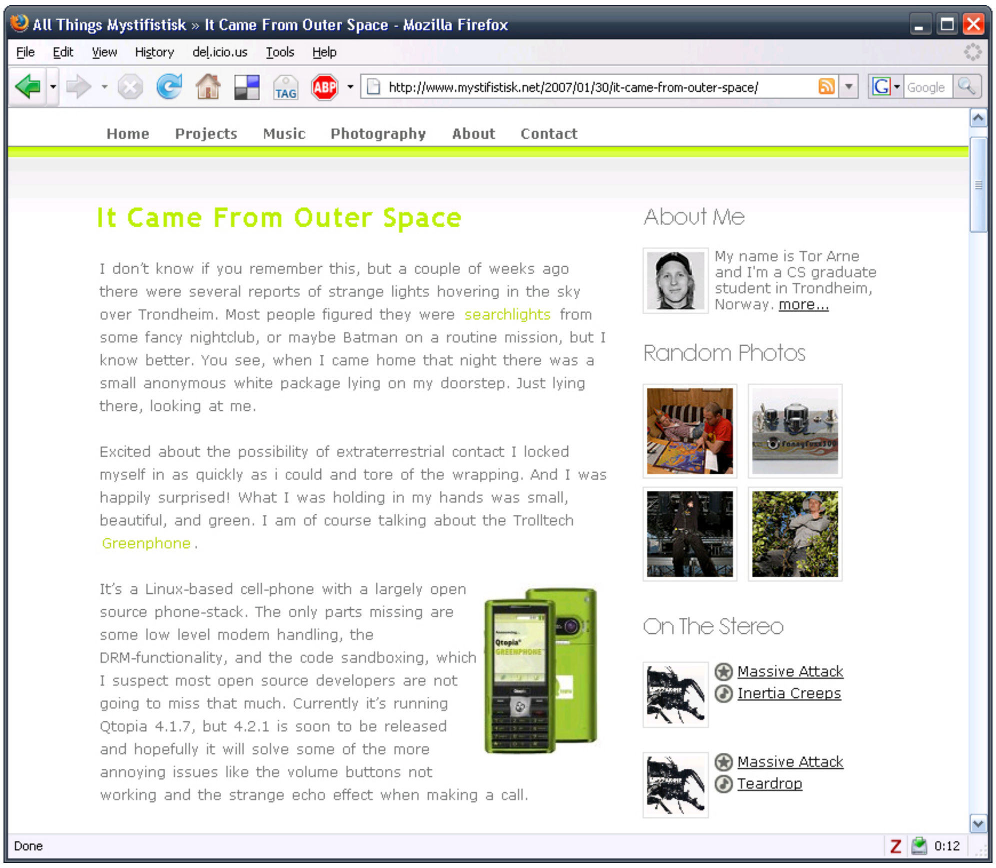
By calling `getImageBlock` with the default arguments you would end up with random images at the size of a thumbnail[1]. Adding the argument `fullSize=true` would replace the thumbnails with full size images, whatever that might be (if the user had uploaded images directly from their digital camera they would typically be 3000x2000 pixels, i.e. much too large for a normal PC screen). And if you added both `fullSize=true` and `maxSize=200` the function would return full size images, but scale them to 200x200 using the HTML image tag `width` and `height` parameters.

The problem with this implementation was that the thumbnail-sized images returned by the default arguments was often too small for any real-life use. And although larger images could be retrieved by specifying the `fullSize` and `maxSize` parameters it wasn't always obvious to users how to use these parameters (due to poor documentation of the API).

As a consequence many users assumed that the only size that could be returned was the small thumbnail size, and those who did take advantage of the extra parameters to increase the image size ended up with always having to download full size images, even though the requested size was much smaller (e.g. 200x200). This was a serious problem – not only because it wasted bandwidth for users, but also because the image scaling algorithms employed by most browsers produce terrible visual results.

The limitations of the original implementation annoyed many users, including the founder and maintainer of the popular Wordpress Gallery2 integration project (WPG2). The project provides easy integration of Gallery into the Wordpress blog system by taking advantage of the ImageBlock module. Unfortunately the limitations of the `getImageBlock` function were affecting the WPG2 project, so in August 2005 the WGP2 maintainer added the following request for enhancement (RFE) to the Gallery tracker:

[1] Not an actual nail, but small previews of the original photo, usually sized 100x100 pixels or lower.


All Things Mystifistisk » It Came From Outer Space - Mozilla Firefox
File Edit View History del.icio.us Tools Help
http://www.mystifistisk.net/2007/01/30/it-came-from-outer-space/
Google
Home Projects Music Photography About Contact
Done
0:12

It Came From Outer Space

I don't know if you remember this, but a couple of weeks ago there were several reports of strange lights hovering in the sky over Trondheim. Most people figured they were searchlights from some fancy nightclub, or maybe Batman on a routine mission, but I know better. You see, when I came home that night there was a small anonymous white package lying on my doorstep. Just lying there, looking at me.

Excited about the possibility of extraterrestrial contact I locked myself in as quickly as i could and tore of the wrapping. And I was happily surprised! What I was holding in my hands was small, beautiful, and green. I am of course talking about the Trolltech Greenphone .

It's a Linux-based cell-phone with a largely open source phone-stack. The only parts missing are some low level modem handling, the DRM-functionality, and the code sandboxing, which I suspect most open source developers are not going to miss that much. Currently it's running Qtopia 4.1.7, but 4.2.1 is soon to be released and hopefully it will solve some of the more annoying issues like the volume buttons not working and the strange echo effect when making a call.

About Me

My name is Tor Arne and I'm a CS graduate student in Trondheim, Norway. more...

Random Photos

On The Stereo

Massive Attack
Inertia Creeps

Massive Attack
Teardrop

Figure 7.6: Example of integrating images into a website. The sidebar to the right contains four random images from the full image gallery, blended nicely into the overall design of the site. `http://www.mystifistisk.net/2007/01/30/it-came-from-outer-space/`.

```
[G2] Imageblock Support for Midsized Images

Imageblock currently returns two types of images (thumbnail
or "fullsize".) In the case of fullsize the image size
being returned can be further limited by the maxsize
parameter. In the case of very large images, the suggested
enhancement would rather than the imageblock blindly
returning the fullsized image, instead the imageblock
returns the closest sized image to the parameter maxsized
thus dramatically reducing the amount of item it would
take to process the image..
```
[WPG2Dev01]

What the developer wanted was for the `getImageBlock` function to take advantage of the fact that Gallery pre-generates several copies of the original image, at intermediate sizes. These sizes are used internally when displaying images in the normal Gallery user interface, but they were not exposed through the ImageBlock module. By returning one of these images instead of the full size image the problem of bandwidth and browser resizing artifacts would be minimized.

The RFE was accepted by the Gallery team, but due to lack of time and manpower the implementation was delayed again and again. Almost a year later, after yet another inquiry on the Gallery Embedded forums by someone trying to increase the size of their image-block previews, a user finally stepped up and provided a patch adding most of the functionality from the SFE:

```
I have hacked up the imageblock module to something that's
usable for me until something better is done.

[...] Usage now works as follows:

If you choose FullSize in the WPG2 plugin in WordPress
there will be no change. The fullsize image will be sent
to the browser with width and height tags for the maximum
size you set, and the browser will resize it.

If you don't have FullSize checked, it will find the
biggest resized image with both dimensions less than or
equal to the maximum size set and use that one. If the
fullsize image is within this range, that will be used.
If there are no resized images in the range, the thumbnail
will be used.
```

Basically the patch allowed the `maxSize` parameter to be used not only when the `fullSize` parameter was set to `true` (which was the old behaviour), but also when it was set to `false`. The `getImageBlock` function would then choose the largest pre-scaled intermediate copy within the bounds of the `maxSize` parameter, i.e. *equal or smaller*.

For example, if the `getImageBlock` function was called with `fullSize=false` and `maxSize=300`, and Gallery had already rendered intermediate copies for sizes 75x75 (thumbnail), 200x200, 500x500 and 3000x3000 (full size[2]), the function would return the image sized 200x200. Note however that it would not scale that image up to 300x300 (using HTML image tag parameters) because scaling up to a larger size produces even worse results than scaling down.

The patch was of course applauded by the user community, and it quickly found its way into the RFE. The WPG2 maintainer also sent an e-mail to the Gallery developer mailing list, asking for the patch to be applied. Aparantly the added attention helped, because one month later a Gallery developer committed a piece of code with the following commit log:

```
* Fix RFE #1256288:  ImageBlock maxSize parameter (when
  specified  without fullSize) now selects the closest
  possible derivative to the requested size.  In the case
  where two derivatives are equidistant from the requested
  size, the larger of the two is taken for quality's sake.

  Since this is simply a mathematical test, it doesn't do
  any fudging. If a 100px thumb and a 500px resize are
  available, and a maxSize of 299 is specified, the thumb
  is the derivative returned.

  A maxSize of 300 or greater would return
  the 500px derivative.                          [GalleryDev05]
```

[2] Normally photos have a size ratio of 3:2, but to simplify the example we assume they are square.

The committed code worked exactly like the forum patch, except for one crucial point: it also considered pre-scaled copies *larger* than the requested `maxSize` when deciding which image to return. If the final selection turned out to be larger than the requested `maxSize` the algorithm would scale it down using the HTML image tag parameters (like for full size images), but if the image was smaller than `maxSize` it would not do any scaling (just like the forum patch implementation).

It didn't take long for a response to appear on the Gallery mailing list:

```
You have no idea just how happy I am to see this commit!!
:) The [issue has] been around now since Gallery 2.0 days
as has been one of the weak points of Gallery2 for a very
very long time (we get a HEAP of messages over the lack of
midsized photos!!)

Thanks Heaps :)                                  [WPG2Dev01]
```

But, the joy didn't last for long. A couple of hours later, after having a closer look at the implementation, he realized that he was not perfectly happy with how it worked:

```
The issue is, imageblock is still delivering undersized
photos [...]

The logic you have coded [...] is 99% of the way, the only
change that needs to be made is to return closest image
size that is GREATER than or equal to the maxsize (if one
exists) so we are not returning an undersized photo.
                                                 [WPG2Dev01]
```

He was objecting to the fact that the implementation would in some cases return an image smaller than `maxSize`, which conflicted with how he had imagined the behaviour. His original intent behind the RFE was to prevent 'holes' in the layout of external websites – a situation that would often arise if a web designer made room for a 300x300 image in the layout but was only blessed with a 75x75 thumbnail. The recent commit by GalleryDev05 didn't address this fully, as it sometimes would choose the smaller of two mid-sized images and end up with a 200x200 image when the `maxSize` parameter was set to 300.

His view was countered by the Gallery developer, who felt that his implementation was both correct and true to the original RFE:

```
I disagree. The point of the bug [...] was to find the
closest possible derivative, regardless of which one it
was.  The primary reason being that full sized images, and
even high-quality resizes, can easily be in the multiple-
megabyte range, and nobody wants to download that just for
an imageblock.  I think that the logic that's in there is
correct.
                                              [GalleryDev05]
```

It turned out the Gallery developer had focused on the bandwidth part of the problem – trying to eliminate excessive transfers of large images – and not realized that there was another side to the issue. In that context it made perfect sense to return images smaller than `maxSize`. His view was backed up by other core Gallery developers, who argued that the current behavior covered 99% of the possible use-cases for the `getImageBlock` function.

This was apparently a case of two sides with opposing interpretations of the requirements, which escalated in a conflict over who was right and who was wrong. In the end the WPG2 developer got his wish, but not without a fight.

7.2.2 Request for Review

Most web applications that manage content of non-trivial size and complexity provide some sort of search facility to its users, and Gallery is no exception. Ever since the Gallery 1.x days the search box has been an integral part of the user interface, and every time meta-data features such as user comments and keywords were added the search functionality was extended to include search in the new meta-data. Figure 7.7 below shows a screenshot of how search works in Gallery:

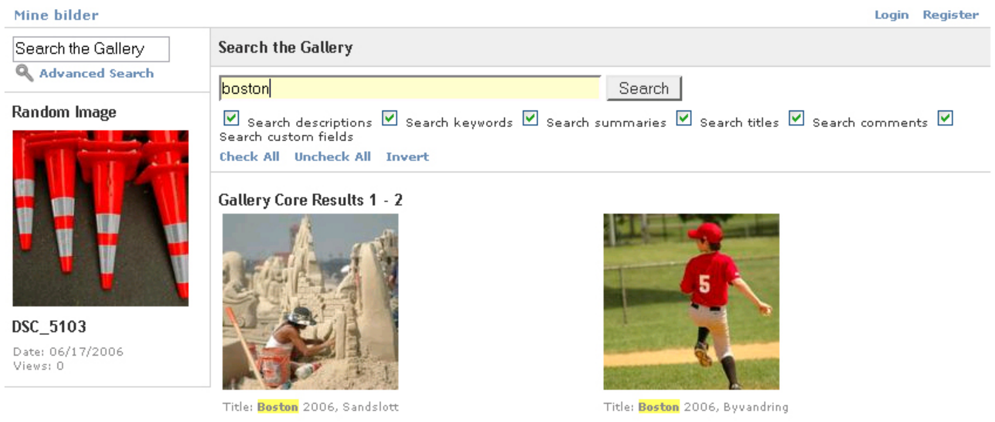


Figure 7.7: A search in Gallery2 for "boston" correctly returns two albums.

The user enters keywords in the top left search box, and when the form is submitted the albums and images matching the search terms are displayed to the right. Matching search terms are highlighted in yellow. The query can easily be altered by editing the text in focused query field, and the results can be narrowed down by unchecking the various checkboxes below it. One problem with the Gallery search feature though is that the query language is very limited.

Actually, limited is an understatement. A more accurate description would be 'non-existent'. The only supported query format is phrased search – meaning all terms have to be found in the potential hit, and they all have to be in the same order, with no intermediate terms. As long you search for "bananas" you will be fine, but once you start looking for "yellow bananas" you will suddenly miss out on photos of "yellow ripe bananas" or "bananas that are yellow".

Most people would probably expect a search for "yellow bananas" to imply "yellow `or` bananas", as this is how popular search engines like Google work. In some cases the query could be interpreted as “yellow and bananas”, for example when the or-query returns too many hits to be useful. But, almost never is the default interpretation “exact(yellow bananas)”, as this basically requires the user to know the full text of the result before searching. Adding to the lack of boolean operators the search implementation also had problems with case-insensitive queries, and would sometimes give you different results depending on the case of the terms in your query.

The demand for more advanced search functionality has been raised several times by the user community, and while unofficial third party patches exist that improve the issue somewhat there are still many who feel that this should be part of the core gallery distribution. A testament to how important this issue is for the community is that the feature request for boolean search is currently ranked seventh in the list of open feature requests (out of over 500).

Luckily there are Gallery developers working on this, and about a year ago the Gallery mailing list saw the following message:

```
Finally I was able to complete the AND search capabilty
for G2. Phew!

Please find the patch as text, compressed archive and
zipped at: [...]

I ask for a review of the patch soon. I kept this change
working since a year, resolving all conflicts coming in in
the meantime. This was due to lack of time to finish the
missing tests. You might imagine my pain not beeing allowed
to commit until the unit tests were finished, while the
feature itself worked well.
```
[GalleryDev06]

The developer was asking for code reviews of a patch that he had been working on for almost a year. The feature itself – adding boolean ‘AND’ search – had been functional for some time, but the Gallery core developers had also asked him to provide unit tests for the feature, and this had apparently delayed the patch.

After a couple of days with no reply from any of the core developers he posted the following follow up to the mailing list:

```
Nobody answered, I assume the patch is commonly accepted
and of minor interest. But the enhanced search is important
for me, so I will commit the changes in 2 days, unless
someone complains.
```
[GalleryDev06]

Aparantly this woke up the sleeping beasts because he soon got a reply:

```
Please hold off on committing this until either I,
GalleryDev02 or GalleryDev03 can review it. I've got a pretty
big backlog, not sure about GalleryDev02 and GalleryDev03 ..
but my guess is that we're going to need a little time to
get around to it. Regular reminders wouldn't go amiss :-)
```
[GalleryDev01]

Basically he was asked to not commit anything before there had been a proper peer review of the code – which he dutifully agreed to, replying that the 'threat' of committing un-reviewed code was made tongue-in-cheek, to place an emphasized reminder.

This short vignette shows a clear example of how stringent the Gallery project is in doing peer reviews and testing on any new code, and resonates well with the earlier descriptions of the project presented in Chapter 6. Doing pro-active peer reviewing like this is both a contrast to how the Amarok project works, and more importantly to how quality assurance and peer reviewing in FLOSS projects are normally presented. We will get back to this discussion in Chapter 8.

During the following weeks he got several reviews of his patch, which uncovered some minor bugs and helped clean up the implementation. In the end the code was actually not committed, for reasons outside of the scope of this thesis, but the code will probably make its way into Gallery2 in a later stage.

7.2.3 Peer Review Software in Action

Gallery – being a web application – comes with some very specific features geared towards the web medium. One of them is support for the popular *mod-rewrite* module of the Apache web-server, which rewrites URLs on the fly to more a more meaningful representation. For example, a URL like this:

```
http://www.mystifistisk.net/gallery2/main.php?g2_itemId=810
```

Is turned into:

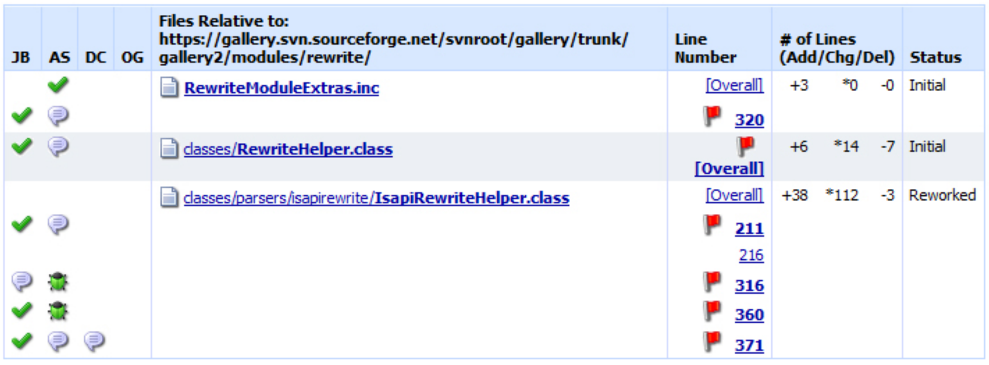
```
http://www.mystifistisk.net/gallery2/Personal/Grillings/
```

This obviously makes the URL easier to read for users of the web site, but it also makes it easier to index for web crawlers – which is a big plus if you host an image gallery that you want traffic to.

Unfortunately the piece of code in Gallery that supported this functionality had a few flaws which in edge cases manifested into errors for the end user. These flaws were first discovered by the maintainer of the WPG2 Gallery plug-in (previously discussed in Section 7.2.1), and a Gallery developer then provided a patch for the bug. Hoping to get the patch included in the Gallery code base the developer posted a a new entry in the Gallery peer review system in January 2007:

```
''Generate URL rewrite substitutions using UrlGenerator''

This patch fixes a bug with the WebDAV OPTIONS URL rewrite
rule found by [the WPG2 maintainer]. It updates the URL
rewrite internal regex rule data structure to support
reusing code from GalleryUrlGenerator.

[... detailed description of the implementation ...]

This patch passes a full unit test run, including code
audit tests. I manually tested with mod_rewrite and Path
Info.
                                          [GalleryDev07]
```

Bundled with the review was a set of files that he had changed in order to fix the bugs. A screenshot of how this looks in the peer review software can be seen in Figure 7.8 below:



JB	AS	DC	OG	Files Relative to: https://gallery.svn.sourceforge.net/svnroot/gallery/trunk/ gallery2/modules/rewrite/	Line Number	# of Lines (Add/Chg/Del)	Status
	✓			RewriteModuleExtras.inc	[Overall]	+3 *0 -0	Initial
✓	💬				🚩 320		
✓	💬			classes/RewriteHelper.class	🚩 [Overall]	+6 *14 -7	Initial
				classes/parsers/isapirewrite/IsapiRewriteHelper.class	[Overall]	+38 *112 -3	Reworked
✓	💬				🚩 211		
					216		
💬	🐞				🚩 316		
✓	🐞				🚩 360		
✓	💬	💬			🚩 371		

Figure 7.8: Example of how the Gallery peer review system lists files in a patch change-set. The four columns to the left represent the participants in the review, and mark any comments or defects they have added to the file (with line numbers shown to the right). Clicking on the line numbers brings up the file with comments in the margins.

Although the developer provided a detailed description of how the patch worked he failed to provide a simple summary of what the patch actually addressed. This made difficult to get a review going, as noted by one of the more senior Gallery developers:

```
1. Is there a bug id? Could you explain the bug?

Your description is detailed on the detail level, but I
don't understand what exactly was wrong/needs to be fixed.

2. What's a high level summary/description of this
patch/bugfix?
```

[GalleryDev03]

Apparently GalleryDev03 got his wish, because his next comment in the review was the following:

```
Ok, done. GalleryDev07 explained the problem and
the idea behind the solution to me via IRC.
```

[GalleryDev03]

What had been going on 'behind the scenes' on IRC in between the two comments was this:

```
<WPG2Dev01> will you get a chance to do a review on
[entry number] 55 as well GalleryDev03?

<GalleryDev03> see my comment in that review. i requested a
description of the bug etc. once i understand the problem,
i'll take a look at the review.

<WPG2Dev01> oh I can supply you with that

<GalleryDev03> i'll add you as observer to the review

<WPG2Dev01> ok
```

The WPG2 developer – eager to get the patch committed – queries one of the Gallery developers if he can do a review of the patch. He is told that the reviewer needs a clarification of the issue first, so the WPG2 developers volunteers to make that clarification:

```
<WPG2Dev01> using the current rewrite rules I get the
following when webdav connecting in my error_log -> Request
exceeded the limit of 10 internal redirects due to probable
configuration error. Use 'LimitInternalRecursion' to
increase the limit if necessary. Use 'LogLevel debug' to
get a backtrace.

<WPG2Dev01> GalleryDev07 found a number of concerns in the
rewrite rules

<GalleryDev03> @infinite recursion: that's a symptom,
what's the bug?

<WPG2Dev01> incorrect rewrite rule let me look up the log

<WPG2Dev01> sorry mate did not log the actual changes
to the rewrite

<WPG2Dev01> [pastes URL to patch]

<WPG2Dev01> was the initial patch

<GalleryDev03> it's already too large. i'd like to understand
the problem without spending 1 hour reading the patch
```

The WPG2 developer's initial attempts to clarify the issue fails because he is missing a log that details some of the core changes. The two developers then agree to wait until the author of the patch can explain the issue.

Later that night the patch author joins the discussion and provides the details:

```
<GalleryDev03> maybe it's obvious, i just don't see it

<GalleryDev07> no, you got it. it should redirect to
modules/webdav/data/options/, but currently it redirects
to main.php?g2_href=modules/webdav/data/options/

<GalleryDev07> this is fixed by using GalleryUrlGenerator
instead of manually building query strings in the URL
rewrite module

<GalleryDev07> thus short URLs are redirected to the URL
GalleryUrlGenerator /would/ have generated if we're not
using RewriteUrlGenerator

<GalleryDev03> i don't quite get it yet. if you do
$urlGen->generateUrl(array('href' => ...) it does the right
thing, no matter if urlGen is GalleryUrlGenerator or
RewriteUrlGenerator, right?

<GalleryDev07> no. this is a dumb example, but say you have
a rule 'match'=>array('href'=>'yyy'),'pattern'=>'xxx%href%zzz'

<GalleryDev07> and you generateUrl(array('href' => 'yyy))

<GalleryDev07> with GalleryUrlGenerator, you get /gallery2/yyy
```

```
<GalleryDev07> so you'd expect the same effective url with
RewriteUrlGenerator

<GalleryDev03> GalleryDev07: i see, so in that edge case it
will do the wrong thing i guess.
```

After understanding the issue at hand GalleryDev03 also confirms that the patch under review actually fixes the issue:

```
<GalleryDev03> is that what your review 55 is about?

<GalleryDev07> but RewriteUrlGenerator makes a .htacces
which redirects /gallery2/xxxyyyzzz to main.php?g2_href=yyy

<GalleryDev03> gotcha, i see it's a bug. is that another
issue or is that this review 55?

* GalleryDev03 reads the description of review 55 again

<GalleryDev07> 55 fixes this bug

<GalleryDev07> 55 addresses a problem with webdav and
php-cgi, which is a consequence of the above bug

<GalleryDev07> by using GalleryUrlGenerator instead of
manually building query strings in the URL rewrite module,
we escape these edge cases

<GalleryDev03> ok, so that's the motivation. it was difficult
for me to follow the description and to understand why we
are doing this. a description of the bug, maybe an example
would have helped.

<GalleryDev07> GalleryDev03: it was on my list of things to
do, since i read your initial comments :)
```

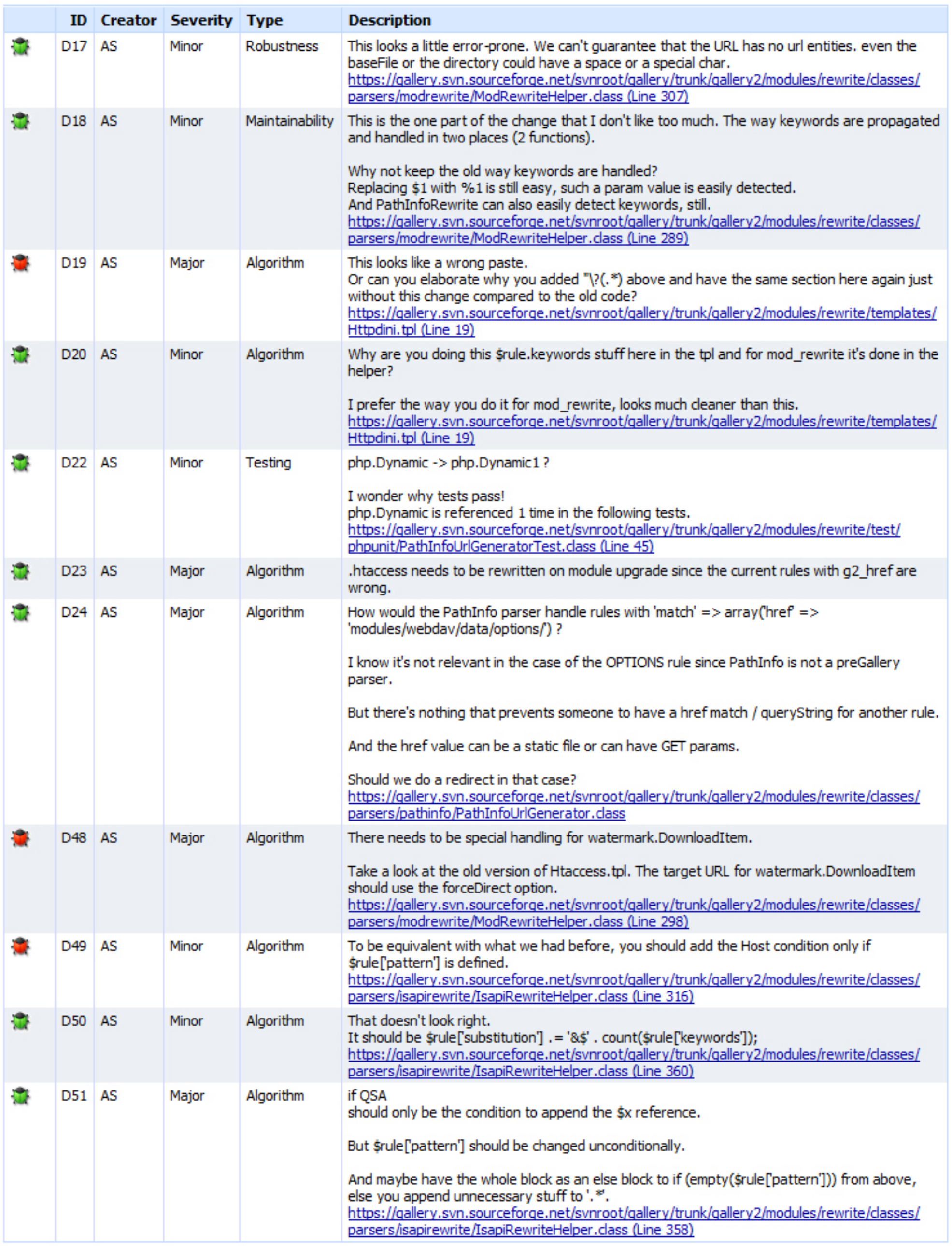
Through about 30 minutes of IRC interaction with the WPG2 maintainer who discovered the bug and the Gallery developer who provided a patch, the Gallery developer who took the job of reviewing the patch was able to build an understanding of what the patch addressed.

What followed was a back-and-forth iterative debugging process in the peer review system; the reviewer opening new defects based on bugs in the patch, and the patch author providing new and improved versions of the code. During the process the peer review system always kept a running list of all open defects, as exemplified in Figure 7.9 on the next page.

In between iterations the patch author got encouragements like "You're almost done, it's looking good.", and finally – after having to rework five of the 11 source code files in the patch, thereby fixing 11 minor and major defects – the Gallery developer commented:

```
Ok, you're done. Thanks! Please commit the change.

And then I'll follow up with extensive IIS testing to
ensure that things still work. I can do small follow-up
patches if necessary.
```

[GalleryDev03]



	ID	Creator	Severity	Type	Description
🟩	D17	AS	Minor	Robustness	This looks a little error-prone. We can't guarantee that the URL has no url entities. even the baseFile or the directory could have a space or a special char. https://gallery.svn.sourceforge.net/svnroot/gallery/trunk/gallery2/modules/rewrite/classes/parsers/modrewrite/ModRewriteHelper.class (Line 307)
🟩	D18	AS	Minor	Maintainability	This is the one part of the change that I don't like too much. The way keywords are propagated and handled in two places (2 functions). Why not keep the old way keywords are handled? Replacing $1 with %1 is still easy, such a param value is easily detected. And PathInfoRewrite can also easily detect keywords, still. https://gallery.svn.sourceforge.net/svnroot/gallery/trunk/gallery2/modules/rewrite/classes/parsers/modrewrite/ModRewriteHelper.class (Line 289)
🟥	D19	AS	Major	Algorithm	This looks like a wrong paste. Or can you elaborate why you added "\?(.*) above and have the same section here again just without this change compared to the old code? https://gallery.svn.sourceforge.net/svnroot/gallery/trunk/gallery2/modules/rewrite/templates/Httpdini.tpl (Line 19)
🟩	D20	AS	Minor	Algorithm	Why are you doing this $rule.keywords stuff here in the tpl and for mod_rewrite it's done in the helper? I prefer the way you do it for mod_rewrite, looks much cleaner than this. https://gallery.svn.sourceforge.net/svnroot/gallery/trunk/gallery2/modules/rewrite/templates/Httpdini.tpl (Line 19)
🟩	D22	AS	Minor	Testing	php.Dynamic -> php.Dynamic1 ? I wonder why tests pass! php.Dynamic is referenced 1 time in the following tests. https://gallery.svn.sourceforge.net/svnroot/gallery/trunk/gallery2/modules/rewrite/test/phpunit/PathInfoUrlGeneratorTest.class (Line 45)
🟩	D23	AS	Major	Algorithm	.htaccess needs to be rewritten on module upgrade since the current rules with g2_href are wrong.
🟩	D24	AS	Major	Algorithm	How would the PathInfo parser handle rules with 'match' => array('href' => 'modules/webdav/data/options/') ? I know it's not relevant in the case of the OPTIONS rule since PathInfo is not a preGallery parser. But there's nothing that prevents someone to have a href match / queryString for another rule. And the href value can be a static file or can have GET params. Should we do a redirect in that case? https://gallery.svn.sourceforge.net/svnroot/gallery/trunk/gallery2/modules/rewrite/classes/parsers/pathinfo/PathInfoUrlGenerator.class
🟥	D48	AS	Major	Algorithm	There needs to be special handling for watermark.DownloadItem. Take a look at the old version of Htaccess.tpl. The target URL for watermark.DownloadItem should use the forceDirect option. https://gallery.svn.sourceforge.net/svnroot/gallery/trunk/gallery2/modules/rewrite/classes/parsers/modrewrite/ModRewriteHelper.class (Line 298)
🟥	D49	AS	Minor	Algorithm	To be equivalent with what we had before, you should add the Host condition only if $rule['pattern'] is defined. https://gallery.svn.sourceforge.net/svnroot/gallery/trunk/gallery2/modules/rewrite/classes/parsers/isapirewrite/IsapiRewriteHelper.class (Line 316)
🟩	D50	AS	Minor	Algorithm	That doesn't look right. It should be $rule['substitution'] .= '&$' . count($rule['keywords']); https://gallery.svn.sourceforge.net/svnroot/gallery/trunk/gallery2/modules/rewrite/classes/parsers/isapirewrite/IsapiRewriteHelper.class (Line 360)
🟩	D51	AS	Major	Algorithm	if QSA should only be the condition to append the $x reference. But $rule['pattern'] should be changed unconditionally. And maybe have the whole block as an else block to if (empty($rule['pattern'])) from above, else you append unnecessary stuff to '.*'. https://gallery.svn.sourceforge.net/svnroot/gallery/trunk/gallery2/modules/rewrite/classes/parsers/isapirewrite/IsapiRewriteHelper.class (Line 358)

Figure 7.9: Running list of defects for review number 55 – "Generate URL rewrite substitutions using GalleryUrlGenerator". As defects are fixed they are marked with green bugs (instead of red ones) and the corresponding text is crossed over (removed in this example for readability).

The patch was finally committed on the 21st of January 2007 – two days after the review was opened, which all things considered is a pretty good track record for such a thorough process.

Part III

Analysis

Chapter 8

Discussion

Software quality has traditionally been viewed as something that can be achieved through careful specification and rigid processes – as evident by the massive focus on quality attributes and process conformance in the software engineering literature (Cavano and McCall, 1978; ISO.org, 1991, 1994; Pressman, 2001). Even thought software metrics have been criticized for not making the jump between reality and theory (Kitchenham and Pfleeger, 1996), and software projects are occasionally failing despite conformance to three-letter acronyms, the underlying view that if we just uncover all the factors and control them we will end up with a high quality product is still dominant (van Vliet, 2000).

This view is reflected in open source development too, but with the twist that we can solve the quality issue by massively parallelizing the debugging process – taking advantage of the many-eyeballs-effect. Bugs and other issues are still viewed as factual in nature – we just need to find and fix them all. Critics have noted that this is perhaps more of an economic shell-game than an actual change in how we develop and debug software (McConnell, 1999), which in turn limits its suitability for commercial development (where costs are very much real).

From a knowledge perspective this description also seems questionable – not because it does not transfer well to commercial development, but because the premise of bugs being factual is highly dubious. Not only are the majority of bugs inherently complex and non-trivial, but quashing them requires more than spotting them and checking off the correct solution – it also involves a large degree of sensemaking, conflicting interests, and politics.

We will now look at three themes from the case material that highlight how debugging in open source projects is in fact highly intensive knowledge work. Section 8.1 deals with how bugs are classified and conceptualized, from being reported to marked as 'fixed'; Section 8.2 shows that the peer-reviewing process in open source projects has many elements of traditional software engineering practices; and Section 8.3 discusses how communication-tools and mediums reflect the task at hand and the subject matter that is being discussed.

8.1 Conceptualizing Bugs

ON THE SURFACE the parallel debugging process in open source projects seems beautifully simple, yet effective. Jørgensen (2001, p. 335) describes the process as "see bug, fix bug, see bug fixed" and Mockus et al. (2002, p. 343) note that "most of the effort in bug fixing is generally in tracking down the source of the problem", and that "the tasks of finding and reporting bugs are completely free of interdependencies".

Apparently, finding bugs is a matter of prolonged effort, and once you've found one the implementation of a fix is perceived to be a fairly straight-forward engineering effort. While this is certainly true for bugs of limited size and complexity, the observations from the two cases indicate that the picture is a lot more nuanced – both in terms of getting a collective understanding and consensus on the nature of a bug, and finding a way to solve it that pleases everyone.

One example is the vignette presented in Chapter 7.1.1 – *Is the Glass Half Full or Half Empty?* – where there was disagreement on how to best represent track progression using the tray icon. While the discussion ended with the Amarok developers not making any changes to the original behaviour, the issue caused a pretty heated debate on the mailing list over what the most logical solution was. The actual code under discussion was very small – maybe 10 lines of code – which by traditional explanation should be "an easy fix", but yet it spawned a discussion of very polarized sides. As one user reflected:

```
It is a way of looking at a piece of music :

a - There is no music at the beginning of the track. So the
indicator is empty. As the track plays along, the song goes
on, the more you listen the more the song is complete in
your head. When you have listened to all of it,the song is
complete and the indicator is full.

b - At the beginning, there is the whole track to play. As
it plays, you consume it, the more you consume the less
there is to. So the indicator is more empty as you approach
the end of the song.

It is not a matter of right or wrong, it's a matter of
perception and of course it differs from person to person.
Therefore it could be configurable.
```

The user points to an important realization: that it is not always a matter of right or wrong. In fact, deciding on the 'correct' solution is more often than not a matter of subjective assessment. A given behaviour in the software may be considered a bug by some, while others may regard it a feature and applaud its existence. Halloran and Scherlis (2002) notes that the distinction between "bug report" and "feature request" in open source projects is not very sharp, because many of the users are also developers. This tendency was observed in the case material too, where issues frequently changed status from bug to feature request and back again, depending on the direction of the discussion. One possible explanation is perhaps that developers often see issues not only for their direct effect on the software, but also for for their potential to solve a more general problem and improve the software. At other times there may be strong agreement that something is not right, but coming up with a solution that pleases everyone is still very hard.

In this particular example many users argued for the feature to be configurable – due to its inherent conflicting nature. One of them even proposed a compromise between adding new code and letting users configure the behaviour, by taking advantage of an already existing option in the application:

```
I think that the animation direction should depend on the
way the user has set up the timer (which is to the right
of the trackbar) -- to measure the elapsed time or the
time left. That IMO would be optimal
```

The proposition referred to the fact that Amarok already allows users to toggle how the progress of a track is displayed in the full application window – as either time elapsed (2:43) or time remaining (-0:23). Switching the direction of the icon animation based on this option would be a natural extension of this feature.

But, the suggestion was quickly rejected by one of the core developers:

```
> Yes this seems sensible and intuitive. Also it's probably
> quite easy to implement (not that I'd know - i'm just
> assuming!) :)

Smells like code bloat. Does anyone like the
current direction?
                                        [AmarokDev04]
```

The term 'code bloat' refers to code that does provide some sort of useful functionality, but where the functionality is not deemed important enough to warrant an increase in the size of the code base. More code means more stuff to wade through when trying to fix bugs or add new functionality – or using the words of the software engineering literature: it negatively affects maintainability.

Code bloat is of course a subjective argument – just like any of the other statements in the discussion – but because it has a flair of engineering tradition to it, it is sometimes misused as a disguised way of saying "I don't like this piece of code". By stating that a piece of code is bloated you effectively disarm your debate-opponents, because no-one wants to argue that maintainability is not important. The only solution for the other side is to call the bluff, or to go into a technical argument for why the code is *not* negatively affecting maintainability – which takes a lot of time, effort, and knowledge about the issue. More often than not people tend to leave it at that. Whether this particular incident was a misuse of this argument is hard to say, but it did cool off the discussion, and eventually it ended by the Amarok developers getting it they way they wanted.

This is another interesting observation, especially in light of the findings of Mockus et al. (2002) on who does what in open source projects. They reported that the core developers (operationalized as the top 15 developers) contributed as much as 83% of the code dealing with new functionality, while the wider community focused more on defect repair. Even when taking into consideration that the border between defect repair and adding new functionality blurred, it still seems likely that the core developers have a more pronounced influence on major functional changes to the code base, to steer the direction of the project.

This resonates with Halloran and Scherlis's notion that quality assurance in open source projects involves a metaphorical 'wall' around the project – designed to "maximize outgoing information flow and simultaneously to strictly limit and control incoming information flow" (Halloran and Scherlis, 2002, p. 2). The wall is implemented using various tools such as bug trackers, mailing lists, and source code control, which together have to provide the right balance between incoming and outgoing flow – to keep both users and developers happy. Consequently, as open as open source software is, there is no doubt that the project leaders are in some sort of control.

This was evident in the vignette presented in Chapter 7.1.2 (*I Want Amarok for Video!*), where the developers trumped their own vision of a music player, ignoring the many voices asking for real video playback. Similarly, the developers would be perfectly in their rights to introduce video playback in a future version of Amarok, ignoring all the users who supported the old mantra.

Another example of how a seemingly simple bug turned out to be the subject of intense discussion is the vignette presented in Chapter 7.2.1 – *To Scale or Not To Scale (That is the Question)*. Even though the patch provided by the Gallery developer included all of the features of the original forum patch (and even added a few) it didn't match the expectations of the WPG2 developer who initially requested the feature. When he tried to explain the remaining issues to the Gallery developers he was met with the attitude that the patch was correct, which in turn triggered a very emotional response from the developer:

```
I feel pretty let down by the RFE process as I have raised
this concern for the better part of 12 months and at the
end of the process the solution been handed will only meet
some of the requirement and just been told I need to re-
raise another RFE..

The simple fact is the developer interpretation of MY RFE
does not meet the needs of the external embedded
developers, returning a smaller image is NOT the solution,
such a solution once again cripples the imageblock essential
functionality of returning images to an embedded application.

I am pretty disillusioned over this and will no longer be
developing the WPG2 plugin as it is clear that the Gallery2
developers are not listening to the developers of external
embedded projects when they are trying to represent the
needs and requirements of the communities they support and
without that support is it pointless to continue
development of our embedded applications..
                                                  [WPG2Dev01]
```

His outburst was at first met with polite but unwavering replies, explaining that although they were sorry that he felt this way they still thought that the final outcome of the RFE was the correct one. But, after a while one of the primary core developers stepped in and apologized for the whole mess:

```
I'm pretty unhappy with this situation. I feel your pain,
I think that our RFE process needs work.  [Our] voting
module gives users a voice in picking which ones are the
most important to work on, but a lot of the problems we
have with fulfilling RFEs comes from the fact that we're
short staffed. In this particular case we definitely did
you a disservice by making you wait for 10 months and then
implementing only part of what you need and not doing the
rest. 9 or 10 months ago I promised you that I would
implement this feature for you when I had time, and I
intend to make good on that promise.  Now is clearly the
time to do it. [...]

In the meantime, as I understand it, what you want is to
make sure that if there's a 200x200 hole in your layout,
you fill it with an image that is as close to 200x200 as
possible while wasting the smallest amount of bandwidth
and minimizing on white space. It's reasonable to have an
option that allows users to decide between optimizing for
filling their layout vs. saving bandwidth. I'm not worried
about the bandwidth issue because I think that it's up to
the album owner (or site admin) to make that decision. If
you agree that this is what you want, then I'll start
working on this today.
                                           [GalleryDev01]
```

Coming from one of the main Gallery developers this disarmed the situation enough to keep the discussion going about how the feature could be implemented. Soon after GalleryDev01 committed a patch adding a new parameter to the `getImageBlock` function that would let a user specify the *exact* dimensions of the requested image – fulfilling the concerns of the WPG2 developer. In the end the bug report was closed as 'fixed', but to say that the process of solving the imageblock-issue was simple is far from the truth.

This is yet another example of the observation that bugs are not binary in nature. Determining if a bug is a bug at all, what is causing it, and what the best solution is, are all on-going discussions that shape the day-to-day activities of open source quality assurance. Bugs also take shifting forms, moving from small and easy to fix one-liners to 'larger' bugs such as usability issues, conflicting requirements, or the direction of the project. When users and developers are discussing these points they are effectively dispersing knowledge about the issue between the participants of the discussion – making tacit knowledge explicit and combining explicit knowledge into a deeper understanding about the problem.

Looking at it from the point of communities of practice this situation has much in common with the service technicians of Julian Orr, as described by Brown and Duguid (1991), who constantly and fluidly moved between basing their problem-analysis on canonical operation manuals, their own experiences, and the advice of their peers (which was in turn based on real life situations). And just as the Xerox technicians had to join forces to solve the intricate and complex problems, open source users and developers act in a symbiosis to deal with the wide range of 'bugs' in open source project – small and large.

Another perspective on this comes from Østerlie and Wang (2006), who looked at how software developers comprehend software failures in integrated systems. Based on an ethnographic study of the Gentoo Linux distribution they argue that comprehension is a collective socio-technical process of sensemaking, where the goal is not necessarily to find the 'perfect' solution, but to reach a closure. We witnessed an example of such a closure earlier when the `getImageBlock` bug-report was closed because the Gallery developers considered it fixed. Weick (1995) argues that sufficiency and plausibility take precedence over accuracy, and that is exactly what happened in the imageblock-case – except that the accuracy was so low and the WPG2 developer so persistent that the report was opened again, and *then* closed for good.

Chapter 4 described how the knowledge management field has moved from understanding knowledge as simple objective truths – easily stored in databases and schemas – to seeing it for its complex and contextual nature. Similarly, assuming that quality assurance is as simple as feeding issues into a bug tracker, and then ticking off each bug as they get processed through the fact-based and deductive debugging machinery, is ignoring all the complex knowledge-work that goes on behind the scenes. At the end of the day a bug may get marked as 'fixed', and appear to the outside world as an easy fix, but the road there may have been very bumpy – involving several levels of discussion, sensemaking, and conflicting interests.

8.2 Proactive vs. Reactive Peer Reviewing

The debugging practice in open source projects is commonly referred to as distributed peer-reviewing – because thousands of end users will inspect the code for bugs and other issues. This description contrasts with how peer-reviewing is presented in the software engineering literature, where the process is a bit more rigorous (van Vliet, 2000; Pressman, 2001).

Peer reviewing is for example part of step three in the capability maturity model (CMM), where it is described as a "disciplined engineering practice for detecting and correcting defects in software artifacts, and preventing their leakage into field operations" (Humphrey, 1989). This description is closer to the original concept of code inspections (Fagan, 1976) than the informal looking-over-shoulder reviewing advocated by Weinberg (1998). One way to put it is that the software engineering approach is *pro-active* – intended to weed out bugs before the code reaches the users – while the open source approach is *re-active* – designed to catch anything the developers overlooked.

This raises the question of whether users of open source software are actually *reviewing* the code (line by line), or if they are merely running compiled executables, and using the code as reference when debugging issues found at runtime. According to Zhao and Elbaum (2000) 75% of the the open source developers in their study expected users to check the source code, but when queries about the effect of proactive and reactive peer reviewing one of my informants noted:

```
I'm not familiar with the reactive external reviews that
you mention. [Our project] is open source so anyone can
look at the code, but I've never really seen someone
```

```
reviewing code that has already been committed except
when investigating a bug or working on a feature upgrade.
```

[GalleryDev04]

From his point of view users only dived into the source when they were trying to find or fix an *already observed* bug – not as part of a targeted code inspection process. This may seem like splitting hairs (after all users are finding, reporting, and fixing bugs), but I find it interesting to note that reactive peer-reviewing, or the 'eyeball'-effect, perhaps has more in common with ordinary large scale field-testing (sending out binary beta-versions of the software to selected customers, who then report back bugs), than it does with peer-reviewing in the traditional software engineering sense. Getting some empirical data on how often bug reports are actually solved by the initial reporting user (compared to other users and project developers), could perhaps enlighten this issue some more.

Another question is whether open source development is synonymous with reactive peer reviewing, or if proactive peer reviewing has a place in all of this. As described in Chapter 6 the two projects differ somewhat in their approach to quality assurance. While Amarok focuses on getting new code integrated as fast as possible to stay on the edge, and hence base the majority of their QA on the reactive peer review model, the Gallery project runs a tight operation with stringent review-processes – more in line with a proactive model.

We saw an example of the latter in the vignette presented in Chapter 7.2.2 – *Request for Review* – where a Gallery developer was asked to hold on committing his code until it had been reviewed. This strictness is highlighted by one of the Gallery developer's own description of their approach:

```
Internal review is very important to the quality of
[Gallery]. Actually, I'd call it completely essential.
It ensures correctness, consistency of coding standards,
and coverage of tests.
```

[GalleryDev04]

Amarok on the other hand takes both models into account, but prefers the reactive approach:

```
Both reviewing techniques are essential. In my personal
opinion, reactive peer reviews are most effective for a
few reasons:

 - No barrier to entry: the submitter does not need to wait
   and see if the code is accepted

 - Code which sits waiting for review before submission
   rarely gets the testing required. Reactive reviews may
   introduce bugs into the code-base more often, but it is
   guaranteed to be tested. In a similar vein, better code
   also gets submitted more often.

Proactive reviews rarely consist of thorough analysis of
source code. Rather the main task is to determine the
processes and the logic which is being employed by the coder.

Ultimately, proactive reviews cause delays in getting
patches into the code base, and many patches often get
```

```
lost or forgotten about - ergo, reactive reviews
contribute more positively to [Amarok].
```

[AmarokDev05]

The point of this comparison is not to make a judgment of which of the two projects has the 'better' approach to quality assurance (especially since both projects has proven to of very high quality), but rather to highlight that there is variation to how open source projects do peer reviewing, and that the proactive model also has a role in in this regard – as illustrated by how Gallery operate.

Based on the observation that open source projects mix and match the proactive and reactive peer reviewing models, and the fact that the reactive model has strong elements of traditional field-testing, it is tempting to question the normative description of the open source development model as something new and revolutionary (Raymond, 1999a; Dibona et al., 1999) – at least the part of quality assurance practices. This point has been raised by authors such as Fuggetta (2003, 2004), who argue that many of the peculiarities of open source development such as daily/nightly builds and so called agile methods can be traced back to traditional commercial development.

The nuances of peer reviewing also has implications for studies of open source development practices, as mixing several concepts into one can cause skewed results. For example, when Zhao and Elbaum (2003) surveyed the level and types of testing performed in open source projects some respondents may have considered reactive peer-reviewing a testing strategy, while others may have limited their answers to traditional unit and regression tests, and left peer-reviewing for another category. The mix-up can also cause problems when the success of open source – explained through the edge of massive peer-reviewing – is used as an argument for more peer reviewing in commercial software development, without being specific about the actual practice.

8.3 Division of Labor

A FINAL observation that calls for closer analysis is the notable difference in intensity and topics for the various communications mediums and tools. Perhaps the biggest contrast was found between IRC and the bug trackers, where the former was regularly being used for off-topic general chatter and even quiz-shows, while the latter employed a very minimalist style of one-liner comments and short summaries of the issue at hand.

As presented in Chapter 6 both projects employed IRC for synchronous chat between users and developers – divided into one channel for developer-related discussion and one for support-issues and general chatter. Unfortunately I was not able to gain access to the developer-only channel of the Amarok project, so the following discussion is based on the data from the two support-channels and one of the developer-channels.

All three channels had a relatively high intensity – with several active users at all times of the hour – but the signal to noise ratio varied enormously from channel to channel. While the developer-channel stayed mostly on topic, the two support channels had a tendency to drift off topic – adding 'noise'. One example was an automated *bot* (short for robot) that resided in one of the support channels:

```
<NewUser> How do I install Amarok 2.0 on my Kubuntu? I
got KDE4 installed as well ..

<RoBot> You can do to install kubuntu.

<NewUser> ?

<NewUser> do what to .. :-s

<ResidentUser> RoBot: confuse him

<RoBot> ResidentUser: Lol i just deleted the files and
such. I want him to the toilet, if you add it to show
him how to leave for my looks and behaves pretty slick.

<ResidentUser> RoBot: confuse him more

<RoBot> ResidentUser: What distro are you one that also
loves to complain about state-owned companies, there's
still more than a few emails, saying that him and he had
slept with her brother. I love him in the past 6 months
ago but missed that little bugger confuse you.

<Developer> RoBot: confuse NewUser

<RoBot> Developer: Man oh man women confuse me.

<ResidentUser> lol

<NewUser> Ah bot :P

<NewUser> riiiiiiiight

<NewUser> It nearly had me :P
```

The bot was set up to record phrases from random conversations in the channel, and then later reply to questions using a mix of earlier recorded material. In the example above a resident user – familiar with the bot and its ways – clears up the situation by showing that the bot is easily manipulated. This kind of interaction happened quite often, as new users would enter the channel and not understand the answers given by the bot. Another bot in then same channel acted as a resident quiz-master, and could also be taunted into saying various embarrassing quotes by issuing special commands like `!confess`. Sometimes these two bots would start triggering each other, resulting in ten to twenty lines of random noise.

While having chatter-bots like this surely must be confusing for new users, it also seemed to provide an amusing pastime for regulars and developers – almost like a pet. Having rituals like this helps build a sense of a community, and that is probably why the project has kept these two bots despite their annoyance to new users.

As for the bug trackers they tended to be very condensed – almost to the point of feeling like a telegram from the trenches of some forgotten war. Status updates like "`Bug 103642 has been marked as a duplicate of this bug`", and "`+1 please fix this`" were common (the latter indicating that the user experiences the same problem as the original poster), and lengthy posts were usually the result of stack traces (history of function calls just before a crash) being added to the comment, to provide the developers with clues about what may have caused the bug.

The variation in how the many communication-mediums were used seemed to appear quite natural and unproblematic to the project developers. A good example of this was when the developers moved seamlessly between the peer-review system and IRC to solve the issue at hand in Chapter 7.2.3. When asked about this the developers confirmed that the separation between mediums was at least partly intentional, which explains their comfort in choosing the right medium for the right job. The advantages of IRC were explained like this:

```
IRC is for real-time discussion where you need / prefer
instant replies. And IRC is for community building. A lot
of developers and power-users hang around in IRC. It's
where people meet. It's also a substitute for meeting in
an office face to face, since we can't do that due to the
geographical distribution of our team members.

Important team members are available almost every day in
IRC for several hours. It's important to know that key
figures are available right when you need to talk to them.

If there are things discussed in IRC that need the
attention of all developers / the community, we attempt
to ensure that this is also communicated through one of
the official mailing lists.
                                            [GalleryDev03]
```

This supports the findings of Gutwin et al. (2004), who argued that tools such as e-mail and IRC are important for maintaining awareness in open source projects. In this case having the developers available on IRC for instant feedback is similar to how mailing lists are used to ask questions with no specific recipient. The people who know the answer will reply, and other parties will pick up bits and pieces of the communication – enough to get a feel for the status of the project.

The minimalist nature of the bug tracker was explained like this:

```
Bug tracker: issues. we use this more for a not-fixed/fixed
indication of things and usually discussion of issues is
kept to other places. Occasionally, the issue will need
clarification or verification and this is done in
the bug tracker
                                            [GalleryDev04]

Support issues are mostly discussed in the forums and then
escalated to filing a bug sometimes. Replies in the bug
tracker are usually very concise and solution-oriented.
Communication is minimal since often there's not much
uncertainty about it.

Then there's the code review process at reviews.gallery2.org
where we discuss the bug fixes more in depth.
                                            [GalleryDev03]
```

Apparently the developers see bug trackers as a way to keep track of the status of the project and which tasks need attention – not the place to go into details about how to solve each bug. This is interesting considering that one common problem with bug trackers is that users report incomplete information. Without proper information about the issue, the environment the software was running in, and stack tracer of the crash, the issue becomes very hard to debug (Monteiro et al., 2004).

Another problem of maintaining a divide like this is that new users may not be aware of the fine details of where to post what, and how to behave in the various mediums. This effect is magnified by the fact that all of the major tools (e-mail, IRC, bug tracker, forum) are free-form mediums where the user can type anything they like. An example of how this can turn into confusion was presented in in Chapter 7.1.3 – *Allow 'flagging' of files (labels)* – where the users failed to realize that the bug tracker was not the best place for elaborate discussions about architecture and design.

Getting to know where to post the right content can be described as part of the joining scripts of the projects (von Krogh et al., 2003). New developers pick up on the finer details by observing the communications for some time, and after a while the divide becomes second nature. This calls for open source project to be clear on these details, for example in documented guides, so that the costs of joining the project are reduced.

A different approach would be to tighten up on how the various tools are used, for example by limiting the length of the comment-fields in the bug tracker. This would of course have to be accompanied with clear instructions on where to move any elaborate discussions – to ensure that they are not lost in the process. While consolidation the mediums like this would seem beneficial, it also has the disadvantage of controlling how people interact. According to Thompson (2005) adding controlling structures like this will have a negative effect on knowledge sharing in the project.

In the end it boils down to the fact that communication-tools and mediums reflect the subject matter that is being discussed. All the tools surveyed in this thesis can be used to support all kinds of discussions, because of their free-form nature, but as we have seen the context decides which one is preferred at the time. As observed in Section 8.1 the daily discussions of open source developers range from small and easy to fix issues to 'large' bugs with implications for UI design or the direction of the project, and the transition between them is very fluid. It is not only natural that the tools and mediums used reflects the fluid nature of the discussions, but open source projects should strive to keep it this way. One way of doing this, as touched upon earlier in the discussion, is to be clear and verbal about what the primary use for a medium is, but *still allow people to use it as they see fit.*

Chapter 9

Conclusion

The preceding discussion has shown us that quality assurance in open source projects involves more than just checking off bugs as they are discovered and fixed by hordes of helpful eyeballs. First of all, most bugs are not binary in nature. On the contrary, they are often complex, non-trivial, and open for subjective interpretations – meaning that there is no right or wrong solution, and 'fixed' is a relative term.

Secondly, the level of sensemaking involved in finding workable solutions to bugs makes it highly intensive knowledge work, with all the implications that follow. Developers are constantly trying to combine and internalize the explicit knowledge coming from users, fellow developers, stack traces, and the code itself, while at the same time juggling the tacit knowledge diffused through discussions about the bug – all to come up with a solution that not only solves the bug from a purely technical perspective, but also keeps everyone happy and follows the overall goals of the project[1].

One way to put it is that the core developers – who we have seen have a strong influence on the direction of the project – are like the mangers of the traditional knowledge dilemma: struggling to get enough knowledge from the workers below them to make informed decisions one how to run the organization. In our case the knowledge involves possible causes and solutions to the problem, and the decision comes down to balancing the various solutions against the wishes of the users and the long term goals of the project.

Perhaps the reason why the open source development model has managed to produce highly successful projects such as Linux, Apache, and Mozilla is not because it is a revolutionary new way of doing quality assurance (based on the observations in Section 8.2 it is not), but because open source developers more or less consciously have recognized that debugging is very demanding knowledge-work, and have taken the consequences of this – for example by adopting open and free-form communication tools and mediums, to easier disperse the knowledge within the project.

[1] Phew! No wonder debugging can be both frustrating and exhausting at times.

If this is the case then it has implications for both open source developers, as well as for the wider software community. First of all, open source projects should recognize this advantage and exploit it to the fullest – for example by building seeding structures that foster communities of practice and open flow of knowledge. This lesson also applies across projects boundaries and within the open source community as a whole.

Secondly, software engineering academics and practitioners need to reconsider the notion that careful specification of software metrics and quality attributes are reliable determinants for the quality and success of a project. As evident in this case study there are factors well beyond measurable and factual software properties that contribute to the quality of the end product.

Extending this argument to include software development in general we must conclude that if debugging is knowledge work then surely design and implementation must be too. Developers usually like to think of their work as being highly intellectual and creative – Graham (2004) even compares hacking to painting – and they typically welcome knowledge transfer through informal social interaction. Software managers on the other hand tend to believe that knowledge can be successfully transferred through formalized routines and quality systems, and spend their time trying to impose these routines on frustrated developers.

This paradox has been highlighted by several authors (Conradi and Dybå, 2001; Glass, 2006), and interestingly the solution seems to be to "create a more cooperative and open work atmosphere, with strong developer participation in designing and promoting future quality systems." (Conradi and Dybå, 2001). If that's not a call to look to open source I don't know what is.

Going even further, maybe all work has elements of knowledge and creativity – even the most mundane tasks. This is not a novel idea, but goes back to the critique of the knowledge society as focusing too much on theoretical knowledge over other types of knowledge (Hislop, 2005). Perhaps this is something to keep in mind next time we black-box something as simple and then wonder why the output does not match the expectations.

9.1 Further Work

THE IDEAS presented in the above conclusion are far from conclusive, and should be investigated further to validate and refine the results from this study. One way of doing that would be to do additional case studies, using the same theoretical basis, to see if the results manifest in other situations.

Another way would be to approach the issue from a slightly different angle – for example using the computer supported cooperative work literature as a basis to investigate if open source developers somehow have overcome the various problems of these tools (Bowers, 1994), thereby increasing the rate of knowledge transfer. As interesting as it sounds, I will leave that for someone else.

Bibliography

Literature

ARONSON, E., WILSON, T. D. AND AKERT, R. M. *Social Psychology*, 4th ed. Prentice Hall, June 2001. ISBN 0130288640.

BASKERVILLE, R. AND DULIPOVICI, A. The theoretical foundations of knowledge management. *Knowledge Management Research & Practice 4*, 2 (2006), pp. 83–105.

BERGQUIST, M. AND LJUNGBERG, J. The power of gifts: organizing social relationships in open source communities. *Information Systems Journal 11*, 4 (2001), pp. 305–320.

BEZROUKOV, N. Open source software development as a special type of academic research (critique of vulgar raymondism). *First Monday 4*, 10 (1999). Available online from `http://firstmonday.org/issues/issue4_10/bezroukov/`; last accessed November 8, 2007.

BONACCORSI, A. AND ROSSI, C. Why open source software can succeed. *Research Policy*, 32 (2003), pp. 1243–1258.

BOWERS, J. The work to make a network work: studying CSCW in action. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work* (New York, NY, USA, 1994), ACM Press, pp. 287–298. ISBN 0-89791-689-1.

BOWKER, G. AND STAR, S. L. Knowledge and infrastructure in international information management: Problems of classification and coding. In *Information Acumen: the Understanding and Use of Knowledge in Modern Business*, L. Bud, Ed., pp. 187–213. Routledge, London, 1994.

BROOKS, F. P. *The Mythical Man-Month: Essays on Software Engineering*, 20th anniversary edition ed. Addison-Wesley Professional, Aug. 1995. ISBN 0201835959.

BROWN, J. AND DUGUID, P. Organizational learning and communities-of-practice: Toward a unified view of working, learning, and innovation. *Organization Science 2*, 1 (1991), pp. 40–57.

BURTON, D. A. Software developers want changes in patent and copyright law. In *2nd. Mich. Telecomm. Tech. L. Rev. 87* (1996).

CAVANO, J. P. AND MCCALL, J. A. A framework for the measurement of software quality. *ACM SIGMETRICS Performance Evaluation Review 7*, 3-4 (1978), pp. 133–139.

CONRADI, R. AND DYBÅ, T. An empirical study on the utility of formal routines to transfer knowledge and experience. In *Proc. European Software Eng. Conf.* (Vienna, Austria, 2001), ACM Press, pp. 268–276. ISBN 1-58113-390-1.

CONRADI, R. AND LI, J. Observations on versioning of off-the-shelf components in industrial projects (short paper). In *Proceedings of the 12th international workshop on Software configuration management* (Lisbon, Portugal, 2005), ACM Press, pp. 33–42. ISBN 1-59593-310-7.

CORNFORD, T. AND SMITHSON, S. *Project Research in Information Systems: A Student's Guide.* Palgrave, 1996.

DAHLANDER, L. AND MAGNUSSON, M. G. Relationships between open source software companies and communities: Observations from nordic firms. *Research Policy 34*, 4 (2005), pp. 481–493.

DI GIACOMO, P. COTS and open source software components: Are they really different on the battlefield? *Proceedings of the Fourth International Conference on COTS-Based Software Systems* (2005), pp. 301–310.

DIBONA, C., OCKMAN, S. AND STONE, M., Eds. *Open Sources: Voices from the Open Source Revolution*, 1 ed. O'Reilly Media, 1999. ISBN 1565925823.

DIBONA, C., COOPER, D. AND STONE, M., Eds. *Open Sources 2.0*, 1 ed. O'Reilly Media, Nov. 2005. ISBN 0596008023.

DROMEY, R. G. A model for software product quality. *IEEE Transactions on Software Engineering 21*, 2 (1995), pp. 146–162.

EASTERBY-SMITH, M., THORPE, R. AND LOWE, A. The philosophy of research design. In *Management Research: An Introduction*, ch. 3. Sage Publications, Ltd., 1991.

EHN, P. Scandinavian design: on participation and skill. In *Participatory design : principles and practices*, D. Schuler and A. Namioka, Eds., pp. 41–78. Lawrence Erlbaum Ltd., 1993.

EMPSON, L. Knowledge management in professional service firms. *Human Relations 54*, 7 (2001), pp. 811–817.

FAGAN, M. E. Design and code inspections to reduce errors in program development. *IBM Systems Journal 15*, 3 (1976), pp. 182–211.

FELLER, J. AND FITZGERALD, B. A framework analysis of the open source software development paradigm. In *Proceedings of the twenty first international conference on Information systems* (Brisbane, Queensland, Australia, 2000), Association for Information Systems, pp. 58–69. ISBN ICIS2000-X.

FITZGERALD, B. The transformation of open source software. *MIS Quarterly 30*, 2 (2006), pp. 587–598.

FOGEL, K. F. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, Nov. 2005. ISBN 0596007590.

FUGGETTA, A. Open source software ? an evaluation. *The Journal of Systems & Software 66*, 1 (2003), pp. 77–90.

FUGGETTA, A. Open source and free software: A new model for the software development process? *The European Journal for the Informatics Professional 5*, 5 (2004), pp. 22–26.

GALLIERS, R. D. AND LAND, F. F. Viewpoint: choosing appropriate information systems research methodologies. *Communications of the ACM 30*, 11 (1987), pp. 901–902.

GHOSH, R. A. The impact of free/libre/open source software on innovation and competitiveness of the european union. Technical report, The European Commission, Brussels, 2007. Available online from http://www.flossimpact.eu/; last accessed November 15, 2007.

GHOSH, R. A. Free/libre and open source software: Survey and study. Technical report, University of Maastricht, The Netherlands, 2002. Available online from http://www.infonomics.nl/FLOSS/report/; last accessed April 25, 2007.

GLASER, B. G. AND STRAUSS, A. L. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Transaction, 1967.

GLASS, R. L. *Software Creativity 2.0*. developer.* Books, Nov. 2006. ISBN 0977213315.

GLASS, R. L., RAMESH, V. AND VESSEY, I. An analysis of research in computing disciplines. *Communications of the ACM 47*, 6 (2004), pp. 89–94.

GOLAFSHANI, N. Understanding reliability and validity in qualitative research. *The Qualitative Report 8*, 4 (2003), pp. 597–607.

GRAHAM, P. Hackers and painters. In *Hackers and Painters: Big Ideas from the Computer Age*, pp. 18–33. O'Reilly Media, Inc., May 2004.

GUTWIN, C., PENNER, R. AND SCHNEIDER, K. Group awareness in distributed software development. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work* (Chicago, Illinois, USA, 2004), ACM Press, pp. 72–81. ISBN 1-58113-810-5.

HALLORAN, T. J. AND SCHERLIS, W. L. High quality and open source software practices. In *Meeting Challenges and Surviving Success: The 2nd Workshop on Open Source Software Engineering* (2002), pp. 26–28.

HANNEMYR, G. The art and craft of hacking. *Scandinavian Journal of Information Systems*, 10 (1998), pp. 255–262.

HANSETH, O. The economics of standards. In *From Control to Drift*, C. Ciborra, Ed., pp. 56–70. Oxford Univ. Press, 2000.

HARS, A. AND OU, S. Working for free? Motivations of participating in open source projects. In *Proceedings of the 34th Annual Hawaii International Conference on Systems Sciences* (2001), System Sciences.

HECKER, F. Setting up shop: The business of open-source software. *IEEE Software 16*, 1 (1999), pp. 45–51.

HERTEL, G., NIEDNER, S. AND HERRMANN, S. Motivation of software developers in open source projects: an internet-based survey of contributors to the linux kernel. *Research Policy 32*, 7 (July 2003), pp. 1159–1177.

HISLOP, D. *Knowledge Management in Organizations: A Critical Introduction.* Oxford University Press, USA, 2005. ISBN 0199262063.

HOEPMAN, J.-H. AND JACOBS, B. Increased security through open source. *Communications of the ACM 50*, 1 (2007), pp. 79–83.

HUGHES, J. AND JONES, S. Reflections on the use of grounded theory in interpretive information systems research. *Electronic Journal of Information Systems Evaluation 6*, 1 (2004).

HUMPHREY, W. S. *Managing the Software Process.* Addison-Wesley Professional, 1989. ISBN 0201180952.

JØRGENSEN, N. H. Putting it all in the trunk: incremental software development in the freebsd open source project. *Information Systems Journal 11*, 4 (2001), pp. 321–336.

KARELS, M. J. Commercializing open source software. *Queue 1*, 5 (2003), pp. 46–55.

KITCHENHAM, B. AND PFLEEGER, S. L. Software quality: The elusive target. *IEEE Software 13*, 1 (1996), pp. 12–21.

KLEIN, H. K. AND MYERS, M. D. A set of principles for conducting and evaluating interpretive field studies in information systems. *MIS Quarterly 23*, 1 (1999), pp. 67–93.

KOGUT, B. AND ZANDER, U. What firms do? Coordination, identity, and learning. *Organization Science 7*, 5 (1996), pp. 502–518.

KORU, A. G. AND TIAN, J. Defect handling in medium and large open source projects. *IEEE Software 21*, 4 (2004), pp. 54–61.

KRISHNAMURTHY, S. An analysis of open source business models. In *Making Sense of the Bazaar: Perspectives on Free and Open Source Software*, J. Feller, S. A. Hissam, B. Fitzgerald, and K. R. Lakhani, Eds., pp. 279–297. MIT Press, 2005.

LAKHANI, K. R. AND WOLF, R. G. Why hackers do what they do: Understanding motivation and effort in free/open source software projects. In *Making Sense of the Bazaar: Perspectives on Free and Open Source Software*, pp. 3–21. MIT Press, 2005.

LANGLEY, A. Strategies for theorizing from process data. *Academy of Management Research 24*, 4 (1999), pp. 691–710.

LAURENT, A. M. S. *Understanding Open Source and Free Software Licensing*, 1 ed. O'Reilly Media, Aug. 2004. ISBN 0596005814.

LAVE, J. AND WENGER, E. *Situated Learning: Legitimate Peripheral Participation.* Cambridge University Press, Sept. 1991. ISBN 0521423740.

LERNER, J. AND TIROLE, J. The open source movement: Key research questions. *European Economic Review 45*, 4-6 (May 2001), pp. 819–826.

LI, J., CONRADI, R., SLYNGSTAD, O. P. N., BUNSE, C., TORCHIANO, M. AND MORISIO, M. An empirical study on decision making in off-the-shelf component-based development. In *Proceeding of the 28th international conference on Software engineering* (Shanghai, China, 2006), ACM Press, pp. 897–900. ISBN 1-59593-375-1.

MATURANA, H. AND VARELA, F. *The Tree of Knowledge: The Biological Roots of Human Understanding.* Shambhala, 1987.

MCCALL, J., RICHARDS, P. AND WALTERS, G. *Factors in Software Quality*, vol. 1,2 & 3. Rome Air Development Center, Nov. 1977.

MCCONNELL, S. Open source methodology: Ready for prime time? *IEEE Software 16*, 4 (1999), pp. 6–8.

MCDERMOTT, R. Why information technology inspired but cannot deliver knowledge management. *California Management Review 41*, 4 (1999), pp. 103–117.

MICHLMAYR, M. Quality improvement in volunteer free software projects: Exploring the impact of release management. In *Proceedings of the First International Conference on Open Source Systems* (Genova, Italy, 2005), M. Scotto and G. Succi, Eds., pp. 309–310.

MICHLMAYR, M. AND HILL, B. M. Quality and the reliance on individuals in free software projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering* (Portland, OR, USA, 2003), pp. 105–109.

MICHLMAYR, M., HUNT, F. AND PROBERT, D. Quality practices and problems in free software projects. In *Proceedings of the First International Conference on Open Source Systems* (Genova, Italy, 2005), M. Scotto and G. Succi, Eds., pp. 24–28.

MILLER, B., KOSKI, D., LEE, C. P., MAGANTY, V., MURTHY, R., NATARAJAN, A. AND STEIDL, J. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin-Madison, Apr. 1995.

MOCKUS, A., FIELDING, R. T. AND HERBSLEB, J. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology 11*, 3 (2002), pp. 309–346.

MONTEIRO, E., ØSTERLIE, T., ROLLAND, K. H. AND RØYRVIK, E. Keeping it going: The everyday practices of open source software. Unpublished. *Norwegian University of Science and Technology* (2004). Available online from http://www.idi.ntnu.no/~thomasos/paper/keeping_it_going.pdf; last accessed November 2, 2007.

MUMFORD, E. A socio-technical approach to systems design. *Requirements Engineering 5*, 2 (2000), pp. 125–133.

NONAKA, I. A dynamic theory of organizational knowledge creation. *Organization Science 5*, 1 (1994), pp. 14–37.

NONAKA, I. AND TAKEUCHI, H. A theory of the firm's knowledge-creating dynamics. In *The Dynamic Firm. The role of technology, strategy organization and regions.*, ch. 10. Oxford Univ. Press, 1998.

PARLIAMENTARY OFFICE OF SCIENCE AND TECHNOLOGY. Peer review. *Postnote*, 182 (2002), pp. 1–4. Available online from http://www.parliament.uk/post/pn182.pdf; last accessed May 17, 2007.

PAULSON, J., SUCCI, G. AND EBERLEIN, A. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering 30*, 4 (2004), pp. 246–256.

PIRSIG, R. *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values.* Bantam, 1974. ISBN 0553277472.

POLANYI, M. *The Tacit Dimension.* DoubleDay, 1966. ISBN 038506988X.

PORTER, A., YILMAZ, C., MEMON, A. M., KRISHNA, A. S., SCHMIDT, D. C. AND GOKHALE, A. Techniques and processes for improving the quality and performance of open-source software. *Software Process: Improvement and Practice 11*, 2 (2006), pp. 163–176.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*, 5th ed. McGraw-Hill, Nov. 2001. ISBN 0072496681.

RAYMOND, E. S. The cathedral and the bazaar. In *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, pp. 21–63. 1999a. Available online from http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/; last accessed May 20, 2007.

RAYMOND, E. S. Homesteading the noosphere. In *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, pp. 65–111. 1999b. Available online from http://www.catb.org/~esr/writings/cathedral-bazaar/homesteading/; last accessed May 20, 2007.

RAYMOND, E. S. The magic cauldron. In *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, pp. 113–166. 1999c. Available online from http://www.catb.org/~esr/writings/magic-cauldron/; last accessed May 20, 2007.

REASONING. How open source and commercial software compare: A quantitative analysis of TCP/IP implementations in commercial software and in the Linux kernel. Technical report, Reasoning, Inc., 2003. Available online from http://www.reasoning.com/pdf/Open_Source_White_Paper_v1.1.pdf; last accessed May 20, 2007.

RENNIE, D. Editorial peer review: its development and rationale. In *Peer Review in Health Sciences*, F. Godlee and T. Jefferson, Eds., pp. 3–13. BMJ Books, London, England, 1999.

RØSDAL, A. AND HAUGE, Ø. *A Survey of Industrial Involvement in Open Source*. Master's thesis, Norwegian University of Science and Technology, 2006.

SCARBROUGH, H. AND CARTER, C. *Investigating Knowledge Management*. Chartered Institute of Personnel & Development, 2000. ISBN 0852928998.

SCARBROUGH, H. AND SWAN, J. Explaining the diffusion of knowledge management: The role of fashion. *British Journal of Management 12*, 1 (2001), pp. 3–12.

SCHMIDT, D. AND PORTER, A. Leveraging open-source communities to improve the quality and performance of open-source software. In *Proceedings of the 1st Workshop on Open-Source Software Engineering* (Toronto, Canada, 2001), ICSE.

SHAIKH, S. A. AND CERONE, A. Towards a quality model for open source software (oss). In *1st International Workshop on Fondations and Techniques for Open Source Software Certification* (Braga, Portugal, 2007).

SPENDER, J. Organizational knowledge, learning and memory: three concepts in search of a theory. *Journal of Organizational Change Management 9*, 1 (1996), pp. 63–78.

SPIER, R. The history of the peer-review process. *Trends Biotechnol 20*, 8 (2002), pp. 357–8.

STAKE, R. E. *The Art Of Case Study Research*, 1 ed. Sage Publications, Inc, Apr. 1995. ISBN 080395767X.

STALLMAN, R. M. The GNU manifesto. In *GNU Emacs Manual*, pp. 475–484. Free Software Foundation, Boston, 1985.

STARK, J. Peer reviews as a quality management technique in open-source software development projects. In *Proceedings of the 7th International Conference on Software Quality* (Helsinki, Finland, June 2002), vol. 2349 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 340–350. ISBN 0302-9743.

THOMPSON, M. Structural and epistemic parameters in communities of practice. *Organization Science 16*, 2 (Mar. 2005), pp. 151–164.

TICHY, W. F. Should computer scientists experiment more? *Computer 31*, 5 (1998), pp. 32–40.

TORVALDS, L. AND DIAMOND, D. *Just for Fun: The Story of an Accidental Revolutionary.* Collins, 2002. ISBN 0066620732.

VAN DER BLONK, H. Writing case studies in information systems research. *Journal of Information Technology 18* (2003), pp. 45–52.

VAN VLIET, H. *Software engineering: Principles and Practice*, 2nd ed. John Wiley & Sons, Inc., 2000. ISBN 0-471-97508-7.

VIXIE, P. Software engineering. In *Open Sources: Voices from the Open Source Revolution* Dibona et al. (1999), pp. 91–100.

VOAS, J. The challenges of using cots software in component-based development. *Computer 31*, 6 (1998a), pp. 44–45.

VOAS, J. COTS software: The economical choice? *IEEE Software 15*, 2 (1998b), pp. 16–19.

VON HIPPEL, E. AND VON KROGH, G. Open source software and the 'private-collective' innovation model: issues for organization science. *Organization Science 14*, 2 (2003), pp. 209–223.

VON KROGH, G., SPAETH, S. AND LAKHANI, K. Community, joining, and specialization in open source software innovation: a case study. *Research Policy 32*, 7 (2003), pp. 1217–1241.

WALSHAM, G. Interpretive case studies in is research: nature and method. *European Journal of Information Systems 4* (1995), pp. 74–81.

WALSHAM, G. Knowledge management: The benefits and limitations of computer systems. *European Management Journal 19*, 6 (2001), pp. 599–608.

WALSHAM, G. Doing interpretive research. *European Journal of Information Systems 15*, 3 (June 2006), pp. 320–330.

WEICK, K. E. *Sensemaking in Organizations.* Sage Publications, Inc, May 1995. ISBN 080397177X.

WEINBERG, G. M. *The Psychology of Computer Programming*, silver anniversary ed. Dorset House Publishing Company, Sept. 1998. ISBN 0932633420.

WERR, A. AND STJERNBERG, T. Exploring management consulting firms as knowledge systems. *Organization Studies 24*, 6 (2003), p. 881.

WHEELER, D. A. Why open source software/free software (OSS/FS)? Look at the numbers! Available online from `http://www.dwheeler.com/oss_fs_why.html`; last accessed November 16, 2007.

WITTEN, B., LANDWEHR, C. AND CALOYANNIDES, M. Does open source improve system security? *IEEE Software 18*, 5 (2001), pp. 57–61.

YIN, R. K. *Case Study Research: Design and Methods*, 2nd ed. Sage Publications, Inc, 1994. ISBN 0-8039-5662-2.

ZACK, M. Managing codified knowledge. *Sloan Management Review 40*, 4 (1999), pp. 45–58.

ZHAO, L. AND ELBAUM, S. A survey on quality related activities in open source. *ACM SIGSOFT Software Engineering Notes 25*, 3 (2000), pp. 54–57.

ZHAO, L. AND ELBAUM, S. Quality assurance under the open source development model. *Journal of Systems and Software 66*, 1 (Apr. 2003), pp. 65–75.

ØSTERLIE, T. AND WANG, K. H. Systems comprehension during corrective maintenance: Making sense of failures in integrated systems. Unpublished. *Norwegian University of Science and Technology* (2006). Available online from http://www.idi.ntnu.no/~thomasos/paper/Osterlie-SystemsComp.pdf; last accessed November 2, 2007.

Websites

ALEXA.COM. Alexa - Sites in: Photo Sharing, 2007. http://www.alexa.com/browse/general?&CategoryID=241159&SortBy=Popularity; last accessed September 12, 2007.

CATB.ORG. Halloween Document, 1998. http://www.catb.org/~esr/halloween/; last accessed November 5, 2007.

COMMIT-DIGEST.COM. KDE Commit-Digest, 2007. http://www.commit-digest.org/; last accessed September 24, 2007.

ECONOMIST.COM. Netscape breaks free, Mar. 1998. http://www.economist.com/business/displaystory.cfm?story_id=E1_TVJDDR; last accessed November 1, 2007.

EFF.ORG. EFF: Homepage, 2007. http://www.eff.org/; last accessed September 19, 2007.

EQUALAREA.COM. I oppose Amazon.com's 1-Click Patent, 1996. http://www.equalarea.com/paul/amazon-1click.html; last accessed November 5, 2007.

FASTSEARCH.COM. FAST - Solutions - Platforms - FAST ESP, 2007. http://www.fastsearch.com/thesolution.aspx?m=376; last accessed November 19, 2007.

FIRSTMONDAY.ORG. First Monday, 2007. http://www.firstmonday.org/; last accessed November 8, 2007.

FRESHMEAT.NET. freshmeat.net: Statistics and Top 20, 2007. http://freshmeat.net/stats/; last accessed March 26, 2007.

GNU.ORG. Why you shouldn't use the Lesser GPL for your next library - GNU Project - Free Software Foundation (FSF), 1999. http://www.gnu.org/licenses/why-not-lgpl.html; last accessed November 5, 2007.

GOOGLE.COM. Google Trends: "open source", "free software", 2007. http://www.google.com/trends?q="open+source",+"free+software"; last accessed November 5, 2007.

ISO.ORG. ISO 9001:2000 - Quality management systems – Requirements, 1994. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=21823; last accessed November 7, 2007.

ISO.ORG. ISO/IEC 9126-1:2001 - Software engineering – Product quality – Part 1: Quality model, 1991. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749; last accessed November 7, 2007.

ITEA-COSI.ORG. COSI - Wiki, 2007. http://www.itea-cosi.org/; last accessed November 5, 2007.

LINUXQUESTIONS.ORG. Audio Media Player Application of the Year - LinuxQuestions.org, 2006. http://www.linuxquestions.org/questions/showthread.php?t=514978; last accessed September 19, 2007.

LI.ORG. Linux Counter: Home Page, 2007. http://counter.li.org/; last accessed November 5, 2007.

NOSOFTWAREPATENTS.COM. No Software Patents!, 2007. http://www.nosoftwarepatents.com/; last accessed November 5, 2007.

OPENSOURCE.ORG. The Open Source Definition (Annotated) | Open Source Initiative, 2006a. http://www.opensource.org/docs/definition.php; last accessed November 5, 2007.

OPENSOURCE.ORG. Open Source Licenses by Category | Open Source Initiative, 2006b. http://www.opensource.org/licenses/category; last accessed March 26, 2007.

SOURCEFORGE.NET. SourceForge.net: Project Statistics For Amarok, 2007a. http://sourceforge.net/project/stats/index.php?group_id=84099&ugn=amarok; last accessed September 24, 2007.

SOURCEFORGE.NET. SourceForge.net: Welcome to SourceForge.net. http://sourceforge.net/; last accessed November 22, 2007.

SOURCEFORGE.NET. SourceForge.net: Software Map, 2007b. http://sourceforge.net/softwaremap/trove_list.php?form_cat=92; last accessed September 12, 2007.

SOURCEFORGE.NET. SourceForge.net: Project Statistics For Gallery, 2007c. http://sourceforge.net/project/stats/index.php?group_id=7130&ugn=gallery; last accessed September 4, 2007.

ZOTERO.ORG. Zotero - The Next-Generation Research Tool, 2007. http://www.zotero.org/; last accessed November 20, 2007.

Appendix A

Open Source Licenses

This appendix presents verbatim copies of the most relevant open source licenses discussed in this thesis. The licenses were gathered from `http://opensource.org/licenses/category`, and have not been modified in any way other than simple formatting for inclusion into this text.

A.1 The Open Source Definition

Version 1.9

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.

A.2 The BSD License

Copyright © <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 OWNER 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.

A.3 The MIT/X License

Copyright © <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.

A.4 The Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the

Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License.

   Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License.

   Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution.

   You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   1. You must give any other recipients of the Work or Derivative Works a copy of this License; and

   2. You must cause any modified files to carry prominent notices stating that You changed the files; and

   3. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   4. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must in-

clude a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions.

   Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks.

   This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty.

   Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability.

   In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability.

   While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

How to apply the Apache License to your work

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

> Copyright [yyyy] [name of copyright owner]
>
> Licensed under the Apache License, Version 2.0 (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.apache.org/licenses/LICENSE-2.0`
>
> Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

A.5 The GNU General Public License

Version 2, June 1991

> Copyright © 1989, 1991 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
>
> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether grates or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without

limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the 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 Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

    NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

A.6 The GNU Lesser General Public License

Version 2.1, February 1999

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages–typically libraries–of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other

authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) The modified work must itself be a software library.

   (b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

   (c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

   (d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

   Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

   This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

   However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

   When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

   If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

   Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

   You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

   (a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the

Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

(b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

(c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

(d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

(e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. 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 not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

(a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

(b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

    If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

    It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.