Technological Innovation in Software Industry

–

Open Source Software Development

Volume I: Thesis

Ph.D. Thesis

By Kasper Edwards

Department of Manufacturing, Engineering and Management
Technical University of Denmark
Supervised by Associate Professor M.Sc. Econ. Jørgen Lindgaard Pedersen
Kgs. Lyngby
June 2003

Technological Innovation in Software Industry

Technological Innovation in Software Industry

—

Open Source Software Development

Kasper Edwards Technical University of Denmark,
Department of Manufacturing, Engineering and Management

Volume I: Thesis

Technological Innovation in Software Industry –

Open Source Software Development

Ph.D. Dissertation

Kasper Edwards

2003

Published by the Department of Manufacturing, Engineering and Management, Technical University of Denmark, Produktionstorvet, Bygning 424, 2800 Lyngby, Denmark, http://www.ipl.dtu.dk.

Phone: +45 4525 4800

E-mail: info@ipl.dtu.dk

Printed in Denmark

Typeset in Times New Roman

ISBN 87-90855-43-4

ISSN 1397-0305

IPL-039-03

Keywords: Open source software, economics, model, externalities, goods, behaviour of agents.

Preface

This project was begun in 1998 with the intention of studying technological development and recruitment in the highly competitive computer gaming industry. The working hypothesis, at that time, was that much of the development was being done by individuals, who found computers interesting and spent the better part of their time trying to come up with ways to squeeze more performance out of computer hardware. These private developers would then create short graphical sequences, demos, to illustrate their competence.

In the early days of personal computing (in the late 80'ies) these demos were circulated on floppy disks, and people would meet to show off and discuss their demos. Later, the Internet became the medium for distributing demos, and quite a culture was formed. Yearly conferences called The Party (www.theparty.dk) were held, and the community would meet for several consecutive days. The Party had prices for the best demo in several categories such as graphics, music, and others. The working hypothesis revolved around the idea of The Party serving as a place of recruitment for software developers for the computer gaming industry.

Although extensive efforts were made to make computer gaming companies interested in participating in a study, none proved to be interested. Trying to identify persons who were working on new technology and thus candidates for recruitment also proved to be troublesome. After month of trying the project was abandoned, as a more viable alternative area of research showed up.

1998 was the year when open source software became known in the press and by the venture capital firms. In the spring of 1998 Netscape released the source code for their communicator browser, and everybody shook their heads at this unprecedented move by a commercial software company. This event corresponded to the emerging awareness of the bubble economy, and many thought that this was the true sign of the coming of the new economy.

The initial research of open source software revealed this to be interesting; many persons were working on software and providing the software under a variety of licenses. The interesting element in these licenses was the demand that the source code was required to be freely available. I am a strong believer of economic incentives as a driving force of behaviour and this new phenomenon appeared puzzling and indeed suitable as a basis for this project. Consequently it was decided to research the open source phenomenon and try to come up with a reasonable explanation of why open source software was being developed.

Being a new phenomenon a lot of time and effort has gone into just investigating and understanding what open source software is, and how it is being developed. Interestingly enough, the open source software phenomenon has continued to evolve and change ever since I embarked on the project in late 1998.

In fact the open source phenomenon has changed so much that the project had to be rethought a few times on the way, as the economic environment changed and so did the open source software phenomenon. In 1998-99 open source software peaked in the media, and the papers of Eric S. Raymond was on the required reading list for anyone interested in and working with open source software. The people I talked to in those days often used the

same logic and arguments present in the papers of Eric S. Raymond. Annoyed with these answers and strongly believing that this could not be the whole truth, research began for a different answer to the question of why open source software is being developed.

In 1999 the bubble economy was at its peak and many people talked about the ways of the new economy. Open source software seemed like a natural extension to the new economy and indeed many of the dotcom's, which later in the spring of 2000 crashed with the stock market, were using open source software. This gave the impression that open source software was riding the wave of the bubble economy, and this changed some of the structures of open source software development. From relying primarily on the efforts of individuals developing software in their spare time, rapid adoption of some open source software products lead to high profile developers of open source software becoming employed by companies affiliated with open source software e.g. using and/or selling it.

Open source software firms went public, and VA Linux showed the biggest IPO (Initial Public Offering) rise in the history of NASDAQ and went from $20 a share to $300 a share. Investors and technology firms believed in open source software and many saw it as a sure sign of the reality of the new economy. Other firms like Oracle, HP, and IBM also showed interest in the open source software phenomenon and began to offer solutions based on open source software. For instance IBM announced services and enterprise level support for IBM servers running the open source web server called Apache.

The year 2000 was just as turbulent as 1999. The open source software operating system Linux was the operating system with the fastest growing market share. Also the Apache web server reached an impressive 60% market share.

This very rapid development and rise of open source software made it difficult to find a perspective from which to analyse open source software. Was it the new economy? Was it a change in the market towards firms modifying open source software to fit their needs? Was it a special epistemic community supported by the Internet? Or was it something completely different?

When the dust settled following the burst of the bubble economy in the spring of 2001 things became much clearer. The momentum carried by speculation and ridiculous high stock prices suddenly vanished, and many firms producing open source software died. Interest in open source software has since then taken a turn towards creating software and solutions, which actually had a business model and could make a profit.

It is my belief that the findings of this thesis illustrate some of the mechanisms that explain why open source software is being developed in a real economy not artificially supported by speculation. This thesis presents and discusses the incentives for individual's and firm's participation in open source software development.

This has made the project very interesting but also frustrating and demanding, as many good ideas have been proven wrong by new observations and changes in both the economic environment and the open source phenomenon.

Many persons have helped me through the project, and my deepest apologies to those who I have not mentioned in the following.

My family has been of great assistance and support throughout the process of writing this thesis. My wife Helle has been indispensable in times of despair, and this thesis would not have been completed without her. Incidentally this project also saw the birth of our two daughters, who have been a wonderful source of inspiration and of some delay. The last part of the project was very intensive and required my complete attention at the expense of

my family. Helle was instrumental in the last phase and allowed me to focus completely, I am deeply indebted to her.

I am also deeply thankful to my mother, Marianne, who has taken on the tedious task of fixing some of the countless misspellings and other language problems. This effort cannot be underestimated, as it required her full attention day and nights for fourteen days.

My supervisor, Associate Professor Jørgen Lindgaard Pedersen deserves special mentioning for having guided me through this four-year process. Jørgen possesses the unique quality of being able to completely forget anything previously read and comment on a draft as if it was never seen before, and I attribute this to competence rather than age. Jørgen has put up with endless insults before, during, and after the daily lunch, when the project drifted in the wrong direction and took the mood with it. Hidden under a grumpy mood Jørgen possesses rare pedagogical qualities, which are reflected in a subtle understanding of people. Consequently the student is slowly given greater degrees of freedom (or less Stalinist control is imposed). It has been a pleasure to work with Jørgen, and he deserves my deepest respect.

I would also like to thank the following persons, who have helped in various stages of the project. The interviewed, who deserve special thanks for wasting countless hours with me, are: Jens Axboe, Poul-Henning Kamp, Henrik Størner, Peter Makholm, Klaus Sørensen, K Christiansen, Keld Jørn Simonsen.

Kenneth Geisshirt for allowing me to record his presentation at SSLUG.

Stephen Swartz from the Danish Radio for letting me borrow his unedited interviews with Linus Torvalds and Ben Spade, chairman of the Silicon Valley Linux User Group.

Jørgen Steensgaard from the Department of Information Technology, Technical University of Denmark – for correcting factual errors regarding my understanding of object code and assembler language.

Peter Toft, Chairman of SSLUG, for letting me present my project at a meeting, thus making it possible to find possible subjects for interview.

Armando Stettner for pointing out misconceptions in the description of the history of Unix.

The editors of First Monday for providing me with an opportunity to deliver a talk at the First Monday Conference, FMCon 1.

Professor Michael Hutter and Ph.D. Student Tania Lickweg from the University of Witten, Germany for providing an opportunity to discuss preliminary result. Special thanks to Daniel Seidl for persuading Prof. Hutter to invite me to Witten.

Host master Timmy Gregers Madsen from Danish Telecom who was willing to discuss their use of operating systems.

For hints and tips with getting MS word to function correctly with sections, I would like to thank Simon Pape.

Typographical conventions

Body text (like this) in this thesis uses the Times New Roman 12 font.

Footnotes are printed with Times New Roman 10 font. References use the following format [Author year:page]. A great many references are web resources, and these are

tagged with [Website] followed with the full URL (Uniform Resource Location). Persons cited are indented using the Currier New 12 font and in quotes, like:

> "Bla. Bla. Like open source software" [Edwards 2002]

Equations and Linux commands are printed in *Italic* like:

$$Z = (z^1 + z^2 +,...,+z^H)$$

and,

bla bla bla the *tar* command is useful for unpacking.

Kasper Edwards, Lyngby, August 2003

Abstract

This is a Ph.D. thesis on open source software development. The thesis asks and answers the meta-question: "Why is open source software being developed?"

Open source software as a field of research is young, and the literature, which has been used in this project is reviewed in chapter 2. Chapter 3 begins with an analysis of the meta-question using the reviewed literature. The chapter continues to discuss how to answer the meta-question and proposes that the type of involved agents and properties of software are the two most important elements in shaping behaviour. Agents may be of two types: 1) Individuals and 2) Firms. Software properties consist of technical properties and license properties. Three economic theories are proposed for answering the meta-question, and four research questions are presented.

The license of software is central to the analysis and understanding and 10 software licenses ranging from the Microsoft EULA[1] to the GNU GPL. The licenses are compared and three licenses are selected because of their relative differences. The three licenses are then used in a three-step analysis.

In chapter 5 the first analysis in conducted, which analyses the three licenses as simple economic goods, which are produced and consumed and no feed back exist between user and producer. It is observed that this theory is sufficient for explaining the incentives for developing and marketing software licensed under the Microsoft EULA. However, it is also observed that this theoretical framework is incapable of providing any understanding of why open source software is being developed.

Chapter 6 analyse software as simple economic goods continue to focus on the importance of the licenses and begins by selection three licenses, which are to be at the centre of attention in the following analysis. The three licenses selected are the Microsoft EULA, GPL and BSD license. These three licenses are analysed as simple economic goods and the incentives for user and producer is uncovered. The theory of economic goods is found not to be able to answer the meta-question of why is open source software being developed. The theory is found to be a powerful means of gaining an understanding of the economic good constituted by each of the three licenses.

Chapter 7 develops a model for explaining the behaviour of agents using and developing the types of licenses. The model focuses on the level of the software development projects, and analyses the relationship between a maintainer and one or more user-developers. A maintainer is a person or firm taking responsibility for a software project by releasing new versions of the program. A user-developer is a person or firm who decides to use or help develop the program, which the maintainer is developing. The model uses theory of economic externalities to explain the behaviour of the maintainer and the user-developer.

As the model only focuses at the level of the individual software project chapter 8 introduce the theory of increasing returns to adoption to the analysis. This is done in an attempt to explain the dynamics of user-developers choosing which program to adopt.

[1] EULA = End User License Agreement

Chapter 9 presents the empirical evidence gathered in this project. The chapter begin with a historic account of open source software beginning with the birth of Unix in 1968. A description of open source software development and a personal test of three Linux distributions is also presented. The chapter ends with a description of a number of mini cases, which has been extracted from seven interviews with open source developers.

Chapter 10 uses the model developed in chapter 7 to analyse the mini cases and this also represents a test of the model. The test observes that indeed does the model offer a working tool for understanding the mini cases.

Chapter 11 offers a discussion of the thesis in general and results in particular.

Lastly, chapter 12 offers a conclusion.

Dansk Sammendrag

Dette er en Ph.d. afhandling om open source software udvikling. Afhandlingen stiller det overordnede forskningsspørgsmål: ”Hvorfor bliver open source software udviklet?”

Open source source software er et relativt nyt forskningsområde, og den litteratur, som er blevet anvendt i dette projekt, gennemgås i kapitel 2.

Kapitel 3 indleder med en analyse af det overordnede forskningsspørgsmål og anvender den gennemgåede litteratur til en tentativ analyse. Kapitlet fortsætter med at diskutere, hvorledes det overordnede forskningsspørgsmål skal besvares. Diskussionen resulterer i formulering af en forklaringsmodel, som danner basis for den resterende afhandling. Forklaringsmodellen udsiger, at baggrunden for at open source software bliver udviklet, skal søges i en kombination af 1) Agenternes egenskaber og 2) Egenskaber ved software. Agenternes egenskaber er bundet til hvorvidt, de er individer eller virksomheder, og egenskaber ved software er en kombination af tekniske egenskaber og licensegenskaber. Det foreslås, at tre forskellige teoretiske perspektiver skal bringes i anvendelse, og disse er teori om økonomiske goder, teori om eksternaliteter og teori om stigende skalaafkast ved adoption. Slutteligt formuleres fire separate forskningsspørgsmål, som skal besvares for at kunne besvare det overordnede forskningsspørgsmål.

Licensen på et program opfattes som central for forståelsen qua forklaringsmodellen, og kapitel 5 præsenterer og analyserer 10 forskellige softwarelicenser.

I kapitel 6 gennemføres den første analyse, som analyserer tre udvalgte licenser (Microsoft EULA, GPL og BSD) som økonomiske goder. Centralt for analysen er, at goderne opfattes som værende blot produceret og forbrugt, og der eksisterer ingen feedback mellem producent og forbruger. Det observeres, at dette teoretiske perspektiv er tilstrækkeligt for at forstå og forklare eksistensen af programmer licenseret under Microsoft EULA licensen. Men det observeres også, at der ikke, inden for rammerne af denne teori, er tilstrækkeligt belæg til at forklare eksistensen af open source software.

Kapitel 7 udvikler en model til forklaring af adfærden hos agenter, som deltager i udvikling og brug af software. Modellen dækker tre licenser: 1) Microsoft EULA, 2) GPL og 3) BSD licensen. Modellen fokuserer på de enkelte software-projekter og beskæftiger sig med forholdet mellem den agent, som kaldes en maintainer og de agenter, som omtales som bruger-udviklere (user-developers). Maintainer’en er den agent, som har påtaget sig ansvaret for et givent program, og agenten kan være både en virksomhed og et individ. Bruger-udviklere er virksomheder eller individer, som bruger og/eller bidrager til udviklingen af et program. Modellen anvender teori om eksternaliteter til at forklare forholdet og dynamikken mellem agenterne.

Kapitel 8 løfter analysen til niveauet over det enkelte software projekt og tilføjer endnu et element til forståelsen. Teori om stigende afkast til adoption anvendes til at forstå den dynamik, der ligger ud over de enkelte projekter. Dette anvendes til at forstå de kræfter, som spiller ind, når agenter skal vælge mellem flere af samme type programmer.

Kapitel 9 præsenterer den empiri, som er indsamlet i projektet. Kapitlet indleder med en gennemgang af den historiske udvikling. År 1968 er valgt som udgangspunkt, idet det var det år, hvor udviklingen af den oprindelige Unix begyndte. En personlig test af tre Linux distributioner bliver gennemgået, og der er en beskrivelse af open source software,

og hvordan det benyttes og udvikles. Kapitlet afsluttes med en gennemgang af syv mini-cases, som er resultat af de interviews, der er gennemført i forbindelse med projektet.

Kapitel 10 anvender den model, som blev udviklet I kapitel 7 til at analysere de syv mini-cases. Kapitlet repræsenterer således også en test af modellens validitet. Gennem analysen observeres det, at modellen er anvendelig som analyseredskab til afdækning af rationelle incitamenter for de involverede agenter.

Kapitel 11 diskuterer afhandlingen, og konklusionerne overføres og præsenteres i kapitel 12.

Table of Contents

1 Introduction

This thesis is about open source software development and understanding how and, in particular, why open source software is being developed.

Open source software is computer software, where the source code is available to anyone interested. Open source software may be copied, modified, and distributed by anyone who may so desire. Source code is the humanly readable instructions that a programmer writes and edits when developing computer programs. These instructions are later translated (compiled) into something that a computer can understand (a binary program).

The idea of software being available as source code for anyone to copy, modify and distribute, is embodied in the term "open source software", and not new. It is actually rather old, and has previously existed as the term "Free Software". More precisely the term open source software was coined in early February 1998[2], and has since then quite successfully been promoted by the Open Source Initiative (OSI). The much older term "Free Software" was created by Richard M. Stallman in 1982, when he also created the Free Software Foundation (FSF), an organisation with the sole purpose of promoting free software. There has been a lot of discussion in the open source software community whether to prefer open source software at the expense of free software. The FSF argues that using the term open source software removes focus from important aspects of freedom and hides the larger political issues. Whatever the effects of using one term instead of another, this thesis uses the term open source software. The reason for this is that the term open source software is widely adopted, and more persons will recognise open source software and not Free Software.

The implications of having access to a program's source code are far reaching. Given the source code for a computer program, a skilled person can read and understand how the program works and alter the program to suit his needs. A person with little skill can use the source code to build a binary program and subsequently use the program.

In contrast most software developing firms do not allow users of their software to look at the source code and certainly not to modify the source code. Many of these companies derive profits from selling their software and treat source code as a trace secret. Source code represents the result of a firm's R&D effort, and consequently it is highly protected. Most people are familiar with some of the products from Microsoft Corporation e.g. the Windows operating system and/or MS Office. Microsoft does not allow people to copy, distribute, or use their software without a licence to do so. The source code for Microsoft's products is kept secret and is not available to the public.

For many software-producing companies this model works fine, and they develop what we will call proprietary software with the intent of selling the software for profit. Proprietary software never comes with source code, as this would allow others to learn how the software was designed, and if source code is available, it is usually accompanied

[2] The "open source" label were invented at a strategy session held on February 3rd 1998 in Palo Alto, California with prominent members of the open source community. Source: The Open Source Initiative, OSI, [website] URL: http://www.opensource.org/docs/history.html

with a non-disclosure agreement. In the rare event that source code is disclosed, it is accompanied with strict non-disclosure agreements.

The value of the source code for a program cannot be understated. Providing source code is like supplying the blueprint for a building or like the Coca-Cola Company printing the complete detailed recipe on the label of Coca-Cola bottles. Anyone would then be able to mix their own Coca-Cola in the kitchen sink. It would also allow anyone to learn from the blueprints and implement the things described in the blueprint. The analogy between source code and blueprints only goes so far, and the main distinguishing feature is that software is immaterial. Thus source code allows the resulting program to be built at little or no cost, while a building requires large amounts of costly building materials. While building software does not require building materials it does require a computer and skills. It is not possible for a person with basic computing skills to build a program and certainly not modify the program. A person with basic computing skills would have to invest significant amounts of time and resource in acquiring the needed skills.

Developing software is not a trivial task, as it requires knowledge of programming and a deep understanding of the problem, which the program is intended to solve. Above all software development is time consuming, time that could have been used for other purposes such as generating an income. The capital investment for developing software is negligent compared to the labour costs.

Here we begin to grasp the need to understand why open source software is being developed. There is a cost associated with developing software, be it open source or proprietary. We have already touched upon the generic difference between open source – and proprietary software, and the difference extends further into how costs are covered. Proprietary software vendors cover their costs by selling a software product, and open source software developers allow everybody to use and distribute their work thus making it difficult to understand how they are compensated. Of course the difficulties of understanding is driven by this authors' personal failure to grasp why someone in their right mind would do professional work, and 'just' make it available to the general public without restrictions.

This observation of open source software developers making the fruits of their labour freely available would be trivial and rather uninteresting, if open source software was just a small isolated phenomenon. It could then be argued that open source software was the efforts of a small extremist group representing a counter movement to proprietary software. It could even be argued that opens source software was the result of practicing Marxists trying to share their intellectual capital in the post-industrialised society. Lastly, open source software development could be interpreted as a recreational activity - a hobby the joy of which is related to actually developing software.

However, there is nothing to indicate that the above 'easy' explanations are true. The sheer amounts of open source software projects indicate that this is not an isolated phenomenon. One way of measuring the amount of open source software projects is to have a look at SourceForge.net. SourceForge is a website that offers to host open source software projects for free, all that is required is that the projects and users are registered. Currently SourceForge hosts 36.797 projects, and has 386.903 registered users[3], quite a few projects, and users. By no means do the figures from SourceForge represent the full count of both users and projects. Many projects are not hosted at SourceForge and neither

[3] SourceForge.net keeps a counter of the number of projects and registered developers on the front page, see fttp//sourceforge.net

are many users, thus the numbers form SourceForge serve just as an indication of many people being involved, and that it is far from being an isolated event.

Open source software development is not just for individuals. Many companies are engaged in open source software either by selling hardware, which uses the software or by providing solutions utilizing open source software. Large computer vendors like HP, Compaq, SUN Microsystems, IBM, and others are marketing products using open source software, and they are also engaged in open source software development.

There are several examples of open source software competing, if not for profit then for market share, with proprietary products and dominating a particular market niche. The Apache web-server, for example, is open source software and dominating the market for web-servers. According to Netcraft, who conducts surveys of active web-servers across the web, Apache is running on 63.22% of the worlds' web-servers and Microsoft's Internet Information Server is running on 26.05% [Netcraft survey 28 May 2002]. Sendmail, a mail-server program as well as a program for routing email on the Internet is responsible for the majority of the Email transport on the Internet. BIND (the Berkeley Internet Name Damon) is a third example of open source software that is dominating its niche. BIND is a program that provides domain name resolution, which is crucial to users of the Internet. Whenever a person points his browser to an Internet site like www.linuxtoday.com, the name is resolved to an IP-address like 63.236.73.20, which is the real Internet address of the server.

Before we proceed, let us take a closer look at two open source software projects: 1) the Linux kernel and 2) the Apache web server.

1.1 The Linux Kernel

On the 25th August 1991 the following message appeared on the newsgroup comp.os.minix:

> "Hello everybody out there using minix -
>
> I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things)
>
> I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-).
>
> Linus (torvalds@kruuna.helsinki.fi)
>
> PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(. " [Torvalds 1991, (protable is a misspelling for portable)]

This was the birth of the Linux operating system, and despite being a mere hobby, Linux has been growing continuously ever since it was announced in the above mail. Today Linux is the number two most used server operating system on the web second only to Windows NT/2000, but the fastest growing [Wheeler 2001a].

When talking about Linux it should be noted that Linux is the kernel - the core of the operating system. Linux is often, mistakenly, confused with the entire operating system, which includes user interface and a lot of software. The point being that Linus Torvalds created the kernel, and not all the supporting software. The kernel is likely to be the most complex part but not the largest part of a Linux operating system. Linux has since been developed as a collaborative effort on the Internet, and the interest in developing has yet to show signs of decline. Linus Torvalds continues to be central to the development of Linux.

Linux started out as a hobby project in 1991, when Linus Torvalds had got his first computer based on the Intel 80386 processor. Linus Torvalds was interested in computers, and in particular he was interested in multitasking, the ability to run several programs at the same time, a feature which most of us take for granted. However, in 1991 the DOS operating system and MINX were the only choices for the 80386 processor. DOS was not capable of multitasking, and was in general a rather limiting computing environment. Linus Torvalds had been programming for years and was using the university computers that were running a variant on the Unix operating system. The Unix operating system and computing environment was appealing to Linus Torvalds, and the only thing that came close and would run on the 80386 was Minix. Minix was a Unix clone created as an education tool, and for this very reason it was missing many features found in larger Unix variants.

Being an educational tool Minix came with complete source code for people to modify and change for their needs. However, Andy Tannembaum, the author of Minix, wanted for educational purposes to keep Minix simple and did not include any of the changes, which many of the Minix users had made. Linus Torvalds purchased a copy of Minix and downloaded some of the enhancements that were available on the net. At some point Linus Torvalds realised that Minix just wasn't good enough, and he began developing what would become Linux.

Following the post above, where Linux was announced, people began to be interested. Not long time passed before the first comments began to arrive, and some also began to help the development by sending additions to the source code.

Since its birth Linux has been the subject of tremendous growth, which also seems to have been accelerating. The Linux kernel has grown from 1,5 million lines of source code (SLOC) to 2.4 million in just 13 month [Wheeler 2001b] - an impressing effort considering that almost ten years should pass to develop the first 1,5 million SLOC. All these lines of code have been contributed by a large amount of persons. Presently the CREDITS file of the 2.4.0 kernel contains 375 names of persons known to maintain a particular part of the kernel. The actual number of people having contributed is probably significantly larger, as the procedures of getting ones name in the "credits" file are somewhat unclear and a matter of judging when one deserves to be in. Some of these people are employed to do just kernel development, others are private persons who find kernel development interesting, and still others are in-between - in the sense that they have to modify the kernel as part of another project in their professional life. Common to all lines of code contributed is the fact that they are open source and free for anybody to use and modify!

Linus Torvalds retains full control of the development of the Linux kernel and personally approves all changes going into the kernel. The general development

methodology is that anyone is free to develop what they want and make the changes they feel are right. For the changes to become part of the official Linux kernel maintained by Linus Torvalds, the changes have to be incorporated and a new version made available by Linus Torvalds.

Changes to the kernel must be sent to the kernel development discussion forum, which is an email mailing list (mailing list). A message sent to a mailing list will automatically be resent to everyone, who subscribed to the particular mailing list. When a change is sent to the mailing list, anyone is free to criticise and make suggestions for changes. If the change is deemed to be good and valuable, the change is included in the next version of the kernel. There is a lot of discussion on the kernel mailing list, which on an average sees more than 1500 messages a week.

There is a lot of commercial interest in the Linux kernel, and firms representing a particular set of interests participate in development of the kernel. IBM, for instance, is involved in developing the Linux kernel for the s390 architecture (a particular type of computer manufactured by IBM), and naturally the code contributed by IBM is open source like the rest of the kernel.

1.2 The Apache Web Server

The already mentioned Apache web-server is a program that serves web pages on the web. Apache project grew from the National Centre for Supercomputing Applications (NCSA) HTTPD daemon (a daemon is a program that silently runs in the background without user intervention) developed by Rob McCool [Fielding 1999]. In 1994 the NCSA HTTPD was the most popular web-server software; mind you, not many web-servers existed, nor existed many web sites. It so happened that Rob McCool left NCSA, and development of NCSA HTTPD came to a sudden and almost complete halt. Many of the webmasters that used the NCSA HTTPD developed their own extensions and enhancements using the freely distributed source code. There was some staff at NCSA working on maintaining the HTTPD, but the webmasters felt a frustration getting the staff to respond to their suggested changes [Lerner & Tirole 2000]. Consequently, a group of these webmasters organised themselves via email, and began to coordinate their improvements in form of ‘patches’.

Brian Behlendorf and 5-6 other web masters agreed to collaborate on developing the web server. It was agreed that anyone was free to suggest and look at the source code, but only a few would be allowed to make changes. In august 1995 version 0.8 of the Apache web server was released, and this version was a substantial improvement on the original HTTPD. Rumour has it that the name Apache comes from the origin of the server: It was a “patch” server - derived from all the patches that had to be applied, before the NCSA HTTPD worked satisfactory.

From being just a handful of webmasters developing Apache, the organisation has grown and developed a formal structure. The essence of the original organisation, however, remains, and the actual development of Apache is done by a core team. Today, Apache is developed by the Apache Software Foundation (ASF), which is described as:

> “The Apache Software Foundation is a membership-based, not-for-profit corporation. Individuals who have demonstrated a commitment to collaborative open-source software development, through sustained participation and contributions within the Foundation's projects, are eligible for membership in the ASF. An individual is awarded membership after nomination and approval by a majority of

```
the existing ASF members.  Thus, the ASF is governed by the
community it most directly serves -- the people
collaborating within its projects.” [The Apache Foundation]
```

The ASF controls development and distribution, but it is a non-profit organisation, and no money is charged for the software. It deserves to be mentioned again that the Apache web server is the most used web server in the world, currently running on more than 60% of all web servers. One feature that makes the Apache stand out is its explicit API (application programming interface), which makes it possible for anyone interested to develop separate modules extending functionality.

1.3 A Closer Look at the Examples

It is not possible to generalise anything from just these two examples, and nor is it the intention. The two examples of open source software projects provide a brief insight into the history of the Linux kernel and the Apache web server. These two projects illustrate two very successful open source software projects. There are many other open source software projects each with its own idiosyncrasies. Looking at SourceForge there appears to be an open source project matching all proprietary software and perhaps even more. Far from all of these projects are successful, and some never gets past the initial project description stage, and are quickly abandoned.

The Linux kernel has a single person controlling development while the Apache project relies on a board of directors discussing the changes and direction of development for the Apache web-server. The board of directors use a voting system to determine which changes are incorporated and which are left out. Other projects goes beyond the voting system, and allow developers who have gained a sufficient amount of trust to make changes to the project source code. These examples illustrate a spectrum in how the projects administer changes to the project source code. The spectrum range from one person guarding the project source code to a completely distributed approach, where anyone who is trusted is allowed to make changes.

In general, open software development takes place on the web using the means of communications provided by the web. Open source software developers are situated all over the world, and thus meeting face-to-face on a regular basis is not an option. Web pages for each individual open source software project generally provide a window, where the project displays all sorts of information about the project software. Email and mailing lists are the primary means for exchanging patches and discussing development in the particular projects.

The Linux kernel and the Apache web server exemplify and indicate that the quality of some open source software is on par, and perhaps even better than proprietary competitors. It also indicates that the development effort required to create and maintain such software must be very large and require much time from developers. If, for example, a security problem was found in the Apache web server, it would be expected that a fix was made available soon after the problem was discovered. Actually, The Apache project has a history of responding to problems very quickly, often within 24 hours. This sort of response requires someone to dive into the source code and identify and fix the problem. Bear in mind that there are no firm with staff on standby to do it, and most if not all is done by volunteers.

In short, open source software is being developed and made available to anyone interested, who may use, modify, copy, and distribute the software without any restrictions. There are a great many open source software projects, and a tremendous number of people

participating in open source software development. The quality of some of this software is on par and, argued by some, perhaps even better than their commercial counterparts. An indication of this is the market dominance of some open source software in their respective market segment like the Apache web server. Firms participate side by side with private developers in open source software development. IBM, for instance, has donated large amounts of code to the Linux kernel, and SUN has donated a journaling file system also for the Linux kernel.

When participating in open software development there is, at least, opportunity costs associated with the time spent. Since open source software allows unrestricted distribution and modification, it is difficult to understand how the opportunity costs are covered. At least firms have to pay salaries to employees developing open source software.

Given the opportunity costs of open source software and no obvious way of covering these costs, the question that comes to mind is: "Why is open source software being developed?"". This is the meta-question, which this thesis tries to answer, and it will be subdivided into research questions in chapter 3 "Research Questions and Methodology".

1.4 Open Source Software as a Field of Research

Research into open source software is a fairly recent endeavour, as noted the term open source software is rather young and was coined in 1998. The principles of sharing software and contributing to other persons' software are, however, as old as computers. The development of the Unix operating system in the 1970'ties is a history of collaboration between the developing firm AT&T and universities, which freely used the operating system and had access to the source code. The younger term free software was coined by the Free Software Foundation and dates back to 1982. However, these early examples of software sharing that, can - to a degree - be compared to open source software having received little attention by researchers. This is probably mainly because the phenomenon was of relative small scale and happened in the research community. This early phenomenon seemed satisfactorily explained by the fact that it was research institutions, which provided funding for having the software developed.

As open source software began to become widely known and adopted by important corporations, academic interest in the phenomenon grew. It appeared that the size of the phenomenon grew to such a size that the educational tool explanation was insufficient. Today there is a lot of research in open source software trying to understand it from different perspectives. So far the amount of papers published in academic journals are scarce, but many working papers have been produced.

The following chapter (Chapter 2) will present a review of some of the articles and working papers that have been produced. So far there is little consensus about stylised facts, and indeed little consensus about what theoretical perspectives to use for understanding and explaining open source software development. Usually we stand on the shoulders of giants in academic research, within this particular field the giants have yet to emerge.

In open source software research there is little prior art and the basic understanding derived from the existence of prior art, which leads to focus questions, becomes troublesome. In the previous pages I have tried to explain why I find open source software interesting. Basically it boils down to explaining why a lot of individuals and firms provide valuable software to anyone interested without requiring anything in reward or payment.

I have chosen to abandon a very narrow focus, which is usually associated with a Ph.D. thesis, and pursue a broader understanding of open source software. I feel that given the circumstance this is the right approach. However, this thesis would not be completely without focus. We will distinguish between focus in analysis and focused questions. The analysis will be focused, while the initial question is broad and perhaps rather unfocused. However, focused questions require a good understanding of the research subject, which did not exist at the time of writing. The literature, which does exist, will be reviewed and used as the basis for choosing a theoretical perspective, and this will in turn help to form more focused research questions.

Having a focused analysis and a broad question has the implication that the broad question will be answered from a specific and narrow perspective defined by the theoretical approach. As such there may be other answers to the same question based on a different understanding and different theoretical perspective. This does not weaken the results of this thesis. Rather, it serves as a contribution to the common understanding of open source software development and as an inspiration/deterrent to other researchers.

1.5 The Aim of this Thesis

The aim of this thesis is to answer the meta-question: Why is open source software being developed? The question will be answered using different types of theory and both interviews and written empirical sources. It is further the aim of this thesis to develop a qualitative model for understanding open source software development. The model would have to go beyond the individual developer and account for incentives when using and developing open source software.

Apart from the pure academic aim this thesis also serves as a demonstration of the skills, which I have acquired during this Ph.D. project. It is required that the Ph.D. student demonstrates knowledge of the research field in which he is engaged and advance the research in the area. This thesis will demonstrate both. The following chapter (Chapter 2) in particular and this thesis in general demonstrate understanding of open source software research.

The advances made in this thesis are identified by the use of a new theoretical approach and independently collected empirical evidence. Furthermore an independent model for understanding open source software has been developed during this project.

1.6 Structure of the Thesis

This thesis uses a methodology that can best be described as a combination of an explorative study and a hypothetical deductive approach. As such the empirical evidence consisting of interviews has indeed been explorative. The thesis of this project is the belief that licenses of open source software are important when understanding why open source software is being developed. This has implications for the structure of the thesis.

This thesis is divided into three parts: 1) Existing Knowledge and Research Questions, 2) Developing a Model, and 3) Testing the Model.

Following this chapter the first part of this thesis begins. Part I presents only existing knowledge, and creates research questions. The next chapter (Chapter 2) presents some of the existing literature on open source software and is reviewed and briefly discussed. This serves as a background for developing the research questions, and within same chapter a discussion of the methodological issues related to the research questions is conducted. The

methodology chapter (Chapter 3) also presents the questions used in the extensive interview conducted as part of this research and how they were conducted.

Following the methodology chapter the theoretical chapter (Chapter 4) explains the theory that is to be the tools for understanding why open source software is being developed. The chapter presents three theoretical perspectives: 1) Economic Goods, 2) Theory of Economic Externalities, and 3) Theory of Competing Technologies.

Part II contains all that is new, and is a direct result of the work during this project encompassing both a model for understanding open source software and the gathered empirical evidence. It is also here the hybrid methodology begins to show evidence of its existence. Since licenses are believed to be an important factor, and a selection of licenses are analysed as the first chapter (Chapter 5) in part II. This is a in itself a collection of empirical evidence and analysis, which serves as a basis for choosing three licenses that are used in the following three chapters (Chapters 6, 7, 8). Chapter 6 analyse three licenses as simple economic good in order to test if the theory of economic goods are capable of explaining why open source software is being developed. Chapter 7 develops a model for software development and consumption. This is the main contribution made in the thesis: A model for understanding behaviour of agents adopting or developing open source software. In chapter 8 the model developed in placed in a context intended to illustrate the selection process, which precedes the behaviour modelled in the model of software development and consumption.

Following development of the model, another empirical chapter (Chapter 9) is presented. This chapter contains a brief history of open source software, and a description of what open source software is i.e. a description of installation and working with three Linux distributions. The chapter also contain the results from in-depth interviews with open source software developers, which are to be used for testing the validity of the model.

Part III contains a chapter (Chapter 10), which tests the developed model, and a chapter (Chapter 11) containing a discussion of both the test and the results of the thesis in general. Part III ends with a general conclusion (Chapter 12) for the thesis.

The thesis consists of three separate volumes. The first is the thesis, The second is the appendix containing all the licenses reviewed, a timeline of the development of Unix and Linux beginning 1968, and the log from a test installation of three Linux distributions. The third volume contains the raw transcripts from interviews conducted with open source software developers. The appendix is distributed with this thesis, and the interview transcripts are only available on direct request and it must be noted that the interview transcripts are in Danish.

1.7 Terms: Open Source Software, Free Software or Libre Software

The term open source software is used throughout this thesis at the expense of free software or libre software. Open source is chosen as the preferred term, simply because it is most widely used, but also because free software is a subset of open source software.

The term open source software was coined in 1998 and received instant fame, when Netscape released the source code for the Communicator browser. Before that free software was the term used. However, some felt that free software was far too ambiguous, as it has the dual meaning of free like in free speech i.e. the freedom to reuse and modify the source code, but also the meaning of free as in free of charge. In 1998 some people in the open source community felt that the term free software was an obstacle for gaining wide corporate adoption.

The problem was some of the wording in the GPL license, which states:

> "By 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." [GNU General Public license Version2: Preample]

Many thought that in particular this preamble was alienating companies from adopting the concept of free software as business opportunity. Consequently the term open source software was adopted, and naturally this resulted in a very heated debate, as there are some important differences between free software i.e. the GPL and LGPL licenses and open source software licenses. Technically open source refers to the open source definition (see section 5.5.2 for a description), which states a set of minimum requirements for a software license in order for the license to be classified as an open source license.

The term libre software arose in the academic community as a term describing the phenomenon but not having all the political baggage of the other two terms.

However, in hindsight the re-labelling of free software to open source software has helped the adoption of open source software. Open source software was the term that caught the attention of the media and myself, and thus it is the term used in this thesis.

Free software is a subcategory of open source software in the sense that free software satisfies the demands of an open source software license. However, free software is stricter in its licensing terms than is open source software. The difference between open source software and free software will become clear in chapter 5, where a selection of licenses is analysed.

1.8 Summary

Software is big business, and some of the richest companies in the world are software companies. The companies developing software protect their investment and intellectual property by selling the software with strict licenses not allowing the buyers to modify, copy, or distribute their software.

Open source software, on the other hand, is software like its commercial counterpart but with the license of open source software it allows anybody to modify, copy, and distribute the software. The developers of open source software are individuals, groups, and firms, who develop the software and distribute it freely. Open source software has in some areas complete market dominance, a testimony to its level of quality.

Given this level of market appeal, some open source software is clearly a candidate for being as sold as a commercial product and thus generating a profit. This is, however, not the case, and the question arises: Why is open source software being developed? This question is interesting given the fact that software development is time consuming, and that some open source software could potentially be sold for a profit.

Part I: Existing Knowledge and Research Questions

The first part of the thesis contains the existing knowledge, which formed the basis for this thesis and the research questions. Chapter 2 contains a literature review and chapter 3 uses the literature to develop a number of research questions. The research question in turn specifies the theoretical basis for this thesis, which are presented in chapter 4.

2 The literature of Open Source Software

This chapter presents a review of some of the contributions, which have been made to the understanding of open source software development. The chapter serves two purposes, 1) as a literature review and 2) as a description of open source software. The literature review has the positive side effects of also providing a general introduction to open source software.

Open source software is a fairly young area of research, and consensus of what books articles, etc. which should be on the required reading list, have yet to be established. Still references in recent articles point in the direction of a few articles that have been a starting point for many a discussion. It is far from certain that these articles provide the correct answers or are per se right in their assumptions and/or conclusions. Still these articles were some of the first to analyse, discuss, and advance our understanding of open source software - that fact alone makes them interesting. Theses articles are also some of the contributions that have had the greatest bearing on this project.

The past and recent discussion of open source software is directed at several different areas of the open source phenomenon. The open source phenomenon is the general term encompassing development, use, and the community surrounding open source software. As the interest in the open source phenomenon is relatively new, a large part of the initial effort has gone into characterising and describing the phenomenon. An important task, and this makes it possible to begin identifying the phenomenon as being comprised by different elements.

The literature review begins with Fred P. Brooks' "The Mythical Man Month", a book, which does a marvellous job of analysing and describing software development in firms. Brooks also characterises different types of programs and relates them to the time it takes to develop each of the types of programs, and this will be used later to analyse open source software development.

We then proceed to the writings of Eric S. Raymond, who has had significant influence on our understanding of open source software. Two papers by Eric S. Raymond are included in this review. Following these papers an internal Microsoft memorandum is presented, which describes and analyses open source software as a threat to Microsoft.

A paper by Rishab Ghosh is then reviewed and it is focused on a more general concept of sharing knowledge on the Internet. Professors Lerner & Tirole have analysed a few open source software projects and present an economic analysis in "The Simple Economics of Open Source Software". This paper and the following two papers have been a source of inspiration when developing the model of software development and consumption in chapter 7.

The following two papers by Justin Pappas Johnson and James Bessen analyse open source software from a perspective of public goods.

The last paper reviewed is a paper by Maureen McKelvey, which analyses open source software from a perspective of economics of innovation.

While the formal and systematic search for literature ended in 2002 emerging papers and other resources were continuously followed through out the project.

2.1 Fred Brooks' "The Mythical Man Month"

The Mythical Man Month by Frederic P. Brooks Jr. (Brooks) is by many perceived as a classic and is both highly cited and used [Bezurokov 1999b]. The insights presented in the book are based on Brooks' own experience as project manager of the IBM operating system/360. The book was reprinted at its 20th anniversary in 1995, and 4 additional chapters were added. "The Mythical Man Month" is interesting, as it provides good insight into the process of developing software in a company setting and presents some of the mechanisms at work in software projects.

Programming or coding, the preferred term by software developers, is the art of writing a sequence of instructions that a computer may execute and thus perform the tasks wanted. The creation of a program may be described as a blend between rigid syntactic, perfection and the creativity of building a castle out of nothing. Each and every command has to be stated according to correct usage, as any misspellings or wrong semantics will not be understood by a compiler. A compiler does the job of translating the human readable source code into something, which a computer can understand and execute. Coding gives the programmer absolute control, anything can be created, the programmer just has to know what he wants, and then there are infinitely different ways in which the program may be coded. Of course there are good and bad ways to implement a program depending on the given situations.

The actual task of writing a program is done in the same way as a text document is typed and edited in a word processor. Program instructions are entered in an editor, edited and the resulting source code is saved in a file and compiled into a program that can be executed on a computer. There exist large number of different kinds of programming languages, which a programmer may use to get his job done. The languages have different features making some languages better suited to solve certain problems than others. The development of the GNU/Linux operating system and much of the related software are done in the language C. The C programming language was a spin-off from the development of Unix at AT&T in the beginning of the 1970'ies (see appendix 2). C is also noted to be an abstraction just above the hardware layer meaning that the programmer has much control of the hardware, but is still accessing the hardware through a language abstraction. C is praised to be simple but yet powerful, and sometimes the word beautiful even comes up.

Brooks provide a good description of why it is interesting and personally rewarding to write computer programs, and he calls it the joy of programming. Programming in itself is interesting if not downright exiting. Brooks mentions five points that describe the "joys of the craft". 1) Programming is a creative activity, where results are constructed from pure thoughts and it is possible to create anything in software. This is the simple joy of making things. 2) Also, software is mostly made to be useful not just to the programmer but also to other people. Making something that is useful to other people is rewarding in itself. 3) Then there is the engineering issue, the joy of tinkering with something complex with many interdependent systems each crafted carefully to perform its little part of the task and work smoothly with the other parts. 4) Programming is also a way of always learning, that once a problem has been solved, the same implementation will not have to be carried out again and can just be copied. If one encounters a similar problem again, problem will be situated in a different context, and thus in essence making the problem a new one. While the theoretical solution may be the same the practicalities are not and vice versa. 5) There is a joy of working with a medium that may be adapted to all sorts of purposes by creating from pure thought.

There are also problems of the craft, namely the inherent precision of computers and programming. The programmer must create the syntactically perfect program for it to function. A comma or space placed wrongly, and the program will not function. Even though programs must be syntactically perfect, there are other sources of errors in programs. Programs can have errors in the sense that they behave in a different way than intended, which may cause all sorts of other problems. The source of these errors may be a misconception of the problem to be solved, architectural i.e. using the wrong approach to solve a problem. (To a hammer the world are nothing but nails).

Programmers developing large systems, where other programmers are involved, find that they are dependent on programs from the other programmers. These programs are sometimes poorly written, poorly implemented, or do simply not follow the agreed specifications. Finding and fixing these bugs are difficult, as it requires one to go over large amounts of source code, written by another programmer to find the culprit. Fixing bugs are tedious and not very interesting compared to developing a new cutting edge program.

Programs are not just programs! Some programs are large, some are small, others again are very complex. Here, to clarify a bit, programs are divided into different types, and in "The Mythical Man-Month" Fred Brooks outlines four types of programs, which will be used throughout this thesis:

1) A little program
2) A Programming product (Generalisation, testing, documentation)
3) A Programming System, (Interfaces and system integration)
4) A Programming systems product, (Combination of 2 and 3)

A little program is an entity in itself, and it can be compiled and run on a standard system. The little program is the creation of the individual, little comments and remarks are made in the source code. Understanding of the program design is retained in the mind of the programmer, and there is a lack of explicit documentation. This often implies that the structure of the program is less clear, and the programmer has no defined coding strategy. Elaborating on the program by others than the author is severely hampered hereby. Programming is however stimulated by the apparent progress, as many lines of code are written without wasting time on structure and documentation. More often than not the little program solves a specific problem and is not designed with general usage in mind.

The programming product is a program that has been extended in such a way that any programmer can maintain the program by making changes in the code. The program is now more structured and commented on, so that other programmers understand the design and choice of the specific programming technique in different parts of the program. The usability of the program has been enhanced into something that can be used to solve several types of problems. The programming product is tested in different contexts i.e. in conjunction with other programs and under various configurations. Testing is very time consuming and requires substantial resources. Still the programming product is a single program that is not dependent on other programs to perform its function. Along with the commented source code the programming product is documented. Estimated project time is 3 times that of a little program with a similar amount of instructions.

A programming system differs from the above by using several different programs to perform its function. It is a system of programs interacting and working together to perform its tasks. The programming system appears, to the user, to be a single program

though it is many little programs performing different tasks. Often the programming system is the result of several people working together, and the different tasks have been delegated as individual programs. The challenge for the programming system is requirement of the individual programs to conform to standard for input and output. For data to pass between individual programs, components have to have a standard describing the interface between them. This has to be done in detail defining interface, syntax, and semantics. Co-ordination between involved parties is often a problem in the programming system, where initiative and progress may cause incompatibility between the individual programs in the system. The number of programming components complicates testing of the system, and the amount of errors rises with the combination of programming components. Estimated project time is 3 times that of a little program with a similar amount of instructions.

A programming systems product has the all the characteristics of the above. It is programming product, where structured programming has been applied and good testing and documentation have been conducted. In addition it is a programming system with many interacting individual programs, where elaborate testing has been conducted. Estimated project time is 9 times that of a little program with a similar amount of instructions. The estimated time is derived from the combination of a programming product (3 times) and a programming system (3 times).

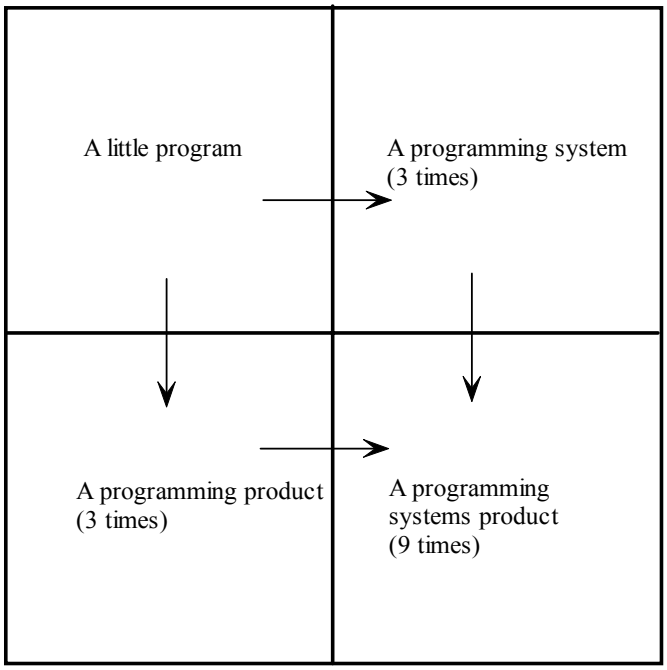


Figure 1: The amount of work required to produce a program of the same functionality [Brooks 1995].

Another important insight from “The Mythical Man-Month” in relation to the open source software development process is the actual concept of man-month and programming software. Many software projects have, and still do, run into trouble due to lack of time. In production management, the remedy to a project falling behind schedule would be to add manpower to the late project. Production managers are well aware that diminishing returns to adding manpower exist. However, in software development it is observed that there are negative returns to adding manpower to a software project. This

observation leads Brooks to state Brooks’ Law: “Adding manpower to a late software project only makes it later” [Brooks 1995].

A programming project that will require the work of one good programmer for 12 month to complete is from the man-month logic equal to 12 programmers working in parallel for one month: 1 man * 12 month = 12 man-month ⇔ 12 man * 1 month = 12 man-month. This, however, is far from the truth, and Brooks goes on to show that there are diminishing returns to adding programmers to a project

The mechanism observed is related to the special properties of software development. When manpower is added to a project, the existing project staff will have to educate the newcomer to the project. Software development is very complex, and to some extent software development displays artistic traits. Thus software projects tend to be very idiosyncratic, and newcomers must get to know the project software before being able to contribute. Coordination and communication become very time consuming, as the number of programmers raise, and so does the possible number of communication paths by n(n-1)/2. In this understanding each individual programmer must coordinate his effort with the rest of the team. Debugging software takes its toll in making the programming effort unrelated to the number of programmer. As Brooks notes, work tasks cannot be partitioned, as the task of debugging has a sequential nature [1995:17].

Brooks goes into great detail in describing software development and how to make sure that projects are delivered within the specified time frame. Brooks offers many partial explanations as to why software projects are late and also partial solutions. However, Books concludes that there is “No silver bullet” to software development.

Brooks’ point of “adding manpower to a late software project only makes it later” is very interesting to open source software development. One of the traits of open source software development is the fact that anyone interested has the opportunity to participate in actual development. There is a clear understanding in the open source community that many people do actually participate in development. Accepting Brooks’ law as true raises the question of why it does not apply to open source software development projects.

Eric S. Raymond, Linux hacker and self-proclaimed anthropologist of the open source community, has tried to answer that very question along with several others.

2.2 Eric S. Raymond “The Cathedral and the Bazaar”

Eric S. Raymond [1999a] has been active in the open source movement since the early days of the Free Software Foundation in the mid 1980’ies, where Eric S. Raymond wrote different modes for the Emacs editor. Emacs is a highly programmable text editor, often touted as the king of all editors. In fact Emacs was the first software written by Richard M. Stallman, when he started the Free Software Foundation [Stallman 1999:58].

The Cathedral and the Bazaar gained instant fame, when Netscape lost the browser war to Microsoft Internet Explorer at the end of 1997. Netscape collapsed, and the management was looking for alternative ways to either reconstruct the company or to continue the life of the once successful Netscape browser. On January 23rd, 1998, Netscape made the announcement that they would give away the source code for the next generation of their Netscape Communicator suite [Hammerly et al. 1999]. The idea for this move was attributed to the paper “The Cathedral and the Bazaar”. The Cathedral and the Bazaar had been the basis for an internal Netscape document outlining how to give away the source code. Later that same year a meeting was held between the Netscape executives and some of the more prominent figures of the open source community including Linus Torvalds and

Eric S. Raymond. The meeting, among others, discussed licensing problems, and what the open source community was interested in. With much publicity the Netscape source code was released in the 31st March 1998 during an event formed as a synchronous party taking place at several locations around the world and transmitted live on the Internet. The code released from Netscape needed a place to live, where developers could collaborate and consequently Mozilla.org was created. Mozilla is a web site, where developers can interact, get the latest version of the source code, and also submit their changed source code. Undoubtedly, the publicity of Netscape using The Cathedral and the Bazaar as a basis for their decisions has helped both the paper and Eric S. Raymond gaining a high public profile.

The Cathedral and Bazaar has been the subject and point of departure of many a discussion and paper. The Cathedral and the Bazaar has had a wide impact on the general discussion, and the views of Eric S. Raymond have seen wide adoption. The impact of these writings has been evident in many discussions in the authors' local Linux user group. Skåne Sjælland Linux User Group (SSLUG) is a 6000+ member user group, and the arguments and rhetoric of the writings have been repeated in many forms. However, the paper has also been the subject of severe critique from, among others, Nikolai Bezurokov [1999a].

To this project, the Cathedral and the Bazaar has also been of importance, as the paper presented a first approach to understanding open source software. One might even venture that the first part of this research has been tainted by the views of Eric S. Raymond's writings. When the project began in late 1998, the effect of Netscape using the paper was still profound.

The Cathedral and the Bazaar is the first, and perhaps because of this it became a very influential paper written about open source software development. The Cathedral and the Bazaar tries to explain why open source software development is so successful. In The Cathedral and the Bazaar, Eric S. Raymond uses his personal experience from the fetchmail project to illustrate no less than 19 points.

Eric S. Raymond was impressed that open source development was able to actually develop good stable operating systems. Creating operating systems or other complex software for that sake was thought to require some kind of centralised development, where a selected few knew the system and were allowed to work on the system. These systems grew in the hands of their creators with long periods of time between new versions. A great deal of this time was devoted to fixing bugs in the software and creating high quality software with few bugs, before each new version was released. Open source software development as exemplified by Linux illustrated a different approach to developing software.

The fetchmail project began in 1993, when Eric S. Raymond was running the technical side of a small Internet service provider. Using email a lot Eric S. Raymond wanted to fetch the email from the servers of the Internet service provider to his home computer and handle the email using his regular software. However, in those days there were problems getting the email from the server to the home system using a dial-up line. The SMTP (simple mail transfer protocol), used to route email on the Internet, was designed to work with computers that were connected to the Internet all the time. Using a

dial-up connection to fetch ones mail from the Internet service provider was not possible, and instead one had to telnet[4] to the mail server and read the mail on the server.

What was needed was a program that could fetch the email from the Internet service provider (ISP) and deliver the mail to the home computer. Such programs existed, but all of them lacked one or more features. Eric S. Raymond decided to use the open source program called popclient as the basis for adding his own features and modifying the program. It so happened that the author of popclient, Chris Harris, had lost interest in the program and was willing to pass over responsibility. An added benefit to taking over popclient was the user base, which followed. Popclient had many users, who were using the program on a regular basis, and who had an interest in the continued development of popclient.

Eric S. Raymond was then the official maintainer of popclient, and immediately he began to modify popclient to fit his own needs. A feature that was missing was the ability to reply directly to the author of an incoming email. Popclient got the address wrong and of course with continuous use of email this became a serious nuisance needing to be remedied as soon as possible. Popclient began to change, and new versions of popclient were released on a regular basis, and each new version included bug-fixes and/or new features.

Popclient began to grow, and the high development activity attracted users and also to interest existing users. Users began to submit bug-reports and also to suggest changes and new features. Some users even made their own changes to popclient and submitted these to the maintainer (Eric S. Raymond), who included the changes that were fit for the project.

According to the Cathedral and Bazaar there are several reasons for the success of open source software. The points and reasons related to open source software development are interesting and explain some of the mechanisms believed to be present in open source software development.

Brooks’ law, as mentioned, states that adding manpower to a late project only makes it later. Eric S. Raymond believes that this effect is not present in open source software development. The reason for this is that in open source software development debugging can be done in parallel, which contradicts Brooks, who states that debugging is sequential. This is attributed to the open source motto: “Release early, release often”, where new versions of the software are released, when even minor changes are made. This makes it possible for eager co-developers to test new versions of the software, identify bugs, and perhaps suggest changes, which are included in the next release. In the early days of the Linux development new versions were released every day and even twice a day – in glaring contrast to commercial development, where new versions are released twice a year.

The motivation for participating in open source software development and starting an open source software project is explained with “Every good work of software starts by scratching the developer’s personal itch”. If a person needs a program, and the program is not available, the person will soon find himself developing the needed software – provided he possesses the required skills.

In the process of developing open source software the Cathedral and Bazaar suggests that the maintainer must treat users of a project as co-developers. Treating users as co-developers means listening to the users and using their suggestions, in effect making them

[4] Telnet is a simple protocol for communicating with a remote computer.

part of the project development. This produces interested users, who feel responsibility for the project, and this is a motivating factor, which will improve the quality of the code and the speed of development.

In open source software development it is important that the project shows a plausible promise. That is, the software may be crude and buggy and not functioning well, but it still shows promise. Eric S. Raymond notes: "What it must not fail to do is (a) run, and (b) convince potential co-developers that it can be evolved into something really neat in a foreseeable future.

2.3 Eric S. Raymond "Homesteading the Noosphere"

Homesteading the Noosphere [Raymond 1999b] presents a contradiction between so-called hacker ideology and actually observed behaviour. This observation is used as a starting point for analysing the customs that pertain to ownership of open source software. It is demonstrated that these customs imply a kind of underlying theory of property rights, which are believed to resemble the theory of land tenure as presented by John Locke. This is related to the so-called hacker culture – open source software culture – that becomes a gift culture. In the hacker gift culture hackers compete for fame and glory by giving away software, which represents an investment in time and energy. Lastly this analysis is related to conflict resolution.

The preliminary remarks about differences in the observed practice and the official hacker ideology is of little interest to this thesis, however, the latter observations are interesting and describe and explain the practice and also present an analysis.

An important question asked in Homesteading the Noosphere is: What does ownership mean in open source software? Open source software is infinitely reproducible, and the source code is freely available. The question is answered quite straightforwardly, the owner of an open source software project is the person, who is recognised by the community to release new versions of the project software.

This calls for some explanation, as most open source software development is organised around a series of projects. Each open source software project has a declared goal; for instance, the Apache project aims at providing an open source web server. To the users of the software it is important to know where the latest updates came from, and even more it is important to know if the updates came from the owner of the project. In open source anyone can distribute new versions and patches, and it is a real and present danger that other persons besides the project owner distribute new versions, and that these versions contain malicious code (e.g. computer virus). The project owner is trusted by the users of the software, and serves as a guarantee that a new version does not contain malicious code. New versions and updates released by the owner are regarded as the 'official' versions. Updates and new versions distributed by others than the owner are referred to as 'rouge' patches and are often not trusted.

There are in general three ways in which ownership of a project can be acquired. 1) By founding the project. 2) By having the project handed over. On occasion the owner looses interest in his project and decides to pass on responsibility to the next interested person in line. Linus Torvalds has passed on the responsibility for the 2.4.x kernel series to Marcelo Tosatti [Brown 2001]. 3) By observing that a project needs work, and that the project is no longer being maintained. An orphaned project is free for anyone to acquire, however, before doing so it is customary to make proper announcements before doing so.

These customs are related to Anglo-American common law and the theory of land tenure. In this theory there are three different ways of acquiring ownership of land: 1) On the frontier where land has never been owned (not mentioning the natives), land could be acquired through homesteading. Homesteading means acquiring ownership by defending the not owned land and ones title. 2) Transfer of ownership was done using transfer of title, in which case the previous owner transfers the deed to the new owner. 3) Land might be lost or abandoned, and thus it could be homesteaded by adverse possession – that is, by improving the land. The point is that the costums observed in the open source culture are evolving is the same way as the old Anglo-American common law, and as such it is a natural process.

From this point the argument turns to economics to explain the motivation for defending ones land and open source project. The logic is that hackers in open source act and uphold the customs, because they defend an expected return from their effort. The explanation proposed is that the hacker milieu is in fact a gift culture.

Gift cultures arise around abundance and exist in cultures, which have no problems with survival. There are sufficient resources for the day-to-day living, and the level of material goods are sufficient for the culture. A premise for the understanding of open source software development as a gift economy is that humans have an innate drive for social status. In the gift economy social status is determined by what you can give away. In the open source software culture the necessities for survival are disk space, computing power, and network bandwidth. And there is no serious shortage of these. Software is open source and thus freely shared among those who are interested, and this creates abundance. In this situation the only measure of success is reputation gained by writing good code and giving the code away.

Open source software development is a reputation game, where the players compete for reputation by giving away the fruits of their labour. This by no means ignores the joy of developing software, as Fred Brooks mentioned, and it remains true. It also points in the direction of the open source culture as a craftsmanship culture, and does not change the rationale of the culture being driven by a reputation game. Any craftsmanship culture must eventually organise itself through reputation.

Reputation is worth striving for, and there are several reasons: 1) Reputation among one’s peers is intrinsically rewarding, 2) Prestige and reputation have spill-over effects, and therefore people will want association to the reputed person. It is thus easier to convince others that they will gain from association with the reputed person. 3) Should the gift-economy be connected with the regular economy, the spill-over effects are also present and may earn the developer a higher status and salary.

The reputation game logic has implications for the motivation, incentives, and reward structure in open source software development. The prestige or reputation gained from founding or contributing to projects is directly related to the amount of innovation put into the project. Consequently there is little reputation to be gained from developing software, which is copying other software and acting as “me-too” projects infringing on the success of the original project. Also, there are different gains from being a contributor than from being a founder. There is much more reputation to be gained from being the original author, as it is the name of the author, which people associate with the software.

In the short review presented here the main arguments, which have bearing on this thesis, have been presented. The main point is that developing software is by itself rewarding, but the prestige and reputation, which come with contributing software to the

community are extra benefits to the developer. Reputation can then be used when founding new projects and seeking help from peers.

The writings of Eric S. Raymond attracted a great deal of attention, when the so-called Halloween documents were made public. Microsoft employee Vinod Valloppillil wrote the Halloween documents in august 1998, in which he analyses the open source software movement and discusses the possible ways, in which Microsoft may counter the threat of Linux to Microsoft core business.

2.4 Vinod Valloppillil "The Halloween Documents"

The Halloween Documents are two internal Microsoft memorandums written by Microsoft employee Vinod Valloppillil and dated August 1998. The memorandums where leaked to the open source community and the press and attracted a great deal of attention. The leaked memos were named the "Halloween documents" by Eric S. Raymond. The name is a reference to the date, when it was annotated in the weekend of 31st Oct - 1st Nov 1998, and the hope was that it would be Microsoft's worst nightmare[5].

The New York Times ran the story [Harmon and Markoff 1998] on the 3rdth of November 1998, and so did several other news outlets[6]. The authenticity of the documents has been confirmed, but Microsoft claims that the documents are merely technical reports by one of their employees and does not reflect Microsoft's position on the subject [Halloween3 1998]. There has been speculation that the Halloween documents were released purposely to appease the Department of Justice in the antitrust case against Microsoft. Allegedly the Halloween would establish that Linux was a threat to Microsoft's Windows business and thus weakening the argument made by the Department of Justice. This, however, does not seem very likely, as the threat established is also countered using the same tactics that has made Microsoft subject to the antitrust lawsuit. The documents are available at opensource.org containing extensive annotation by Eric S. Raymond, and he discusses the various points made by the author. When Halloween1 was leaked and published, the open source community rejoiced and read the document as a proof of Microsoft recognising Linux as a threat. This, of course, implied to the community that they where doing a good job.

Whatever the official Microsoft position, the Halloween documents are interesting reading, as it is the Microsoft perspective and analysis of open source software. The first of the two Halloween documents discusses open source software as a development methodology, and the second Halloween document goes into the details of Linux itself. As the focus in this thesis is open source software in general and the development hereof, the first Halloween document is the most interesting.

The main points are the description and analysis of the open source development process, the perceived threat of the open source software, and how Microsoft should counter the threat.

[5] See the history section of the Halloween document: http://mirror.opensource.dk/halloween/halloween1.php

[6] For a list of links covering the press coverage of the Halloween documents see: http://www.opensource.org/halloween/links.html

2.4.1 Open Source Software - A (New?) Development Methodology

“Open Source Software - A (New?) Development Methodology” is the original title of the Halloween document, which points to the development process as being the most important aspect of open source software development. The development of open source software is directly related to the license of the software. In the software world several different types of licenses exist, and some of these licenses are open source licenses (see chapter 5 for a description of a selection of software licenses). A lot of open source software is distributed under the GPL-license requiring people to make their changes available to the public, if they are distributed. This is seen as a critical step further from other open source software licenses, which allow people to use the project source code for their own purposes, and do not require the changes to be available. The GPL-license is the typical license used in association with the Linux kernel. It should be noted that Apache does not use the GPL-license, but a BSD-style license.

The Halloween document concludes that open source software, Linux and Apache in particular, has achieved commercial quality, and this is opening the door to customer environments. It is observed that open source projects are also undertaking programming tasks of a size and scale not hitherto perceived and of a complexity not manageable by open source software projects.

Open source software projects depend on having access to a large mass of developers serving as a massive distribute development effort and this makes the open source software development process unique. The interest is rooted in the hobbyist culture of open source software, which began life in the scientist community and was typified by ad hoc exchange of source code by users and developers [Valloppillil 1998:11]. This is underlined by the lack of monetary motivation; the projects rarely (if ever) have an explicit commercial perspective.

Open source developers are connected through the Internet and geographically far flung; there are literally developers all over the planet. Coordination of open source development employs the native forms of communication to the Internet i.e. email lists, newsgroups, websites. The complex and large projects are dependent on a sufficiently large group of people attacking and solving the problems. As a consequence there is a direct correlation between the size of the project, which the open source software development process can handle, and the growth of the Internet.

Common goals make distributed decision-making and distributed development possible. A clear statement like “Recreate a free version of Unix” is very efficient in directing the efforts of open source developers. Also common precedents are important for understanding the success of open source development. Open source software draws heavily from a the history of Unix providing almost 30 years of programming experience in what worked, and what did not work.

The skill-set required to participate in open source development is rather small, and so are the barriers to entry into open source development.

> “I think the whole process wouldn't work if the barrier to entry were much higher than it is ... a modestly skilled UNIX programmer can grow into doing great things with Linux and many OSS products.” [NatBro in Valloppillil 1998][7]

[7] NatBro is a Microsoft employee referred to in Valloppillil [1998:15]

Parallel debugging is possible because the developers are hobbyists, and consequently the costs of parallel debugging are close to zero. It is noteworthy that this requires a large group of individuals who are willing to submit source code and that the source code is modularised. Modularisation makes it possible for developers to work on different modules of the same program and not having to coordinate their effort – unless, of course, the module interfaces are changed. Debugging is also helped in open source development by impulse debugging. A person, who stumbles across a bug, will have the possibility to fix this bug, as all Linux distributions come with a full set of development tools.

Open source development is motivated by several things: Solving the problem at hand is one of the strong reasons. Education is significant in two respects: 1) Developers learning for themselves and 2) Open source software used for educational purposes in the institutions. Ego gratification is the developers' personal joy from being respected by fellow developers. Lastly Homesteading the Noosphere refer to the act of acquiring property in the sense of being the first to create a new program or solve a programming problem. Originally homesteading was the act of acquiring property by being the first to occupy the property.

The extensive use of the GPL-license (see section 5.5.3 for an review of the GPL-license) has wide impact on the economic viability of open source software. As the code must be available under the GPL-license there is no long-term economic gain from open source software.

In the long run open source software has credibility to its user base. This is related to the source code being freely available, and thus if the project dies, someone depending on the software is still able to fix and extent the software. The use of the GPL-license also helps credibility, as the GPL-license to some extent prevents code forking. Code forking happens when a project is split up into competing projects. The GPL-license requires the availability of the source code, and thus development in one part of the fork is available to the other fork. Thus extensive sharing will take place even if a fork should happen

The weaknesses fall into three areas: 1) Management costs, 2) Process issues, and 3) Organisational Credibility. The rate of growth in development effort requires significant management costs. Linux and other open source projects have caught up and are on par with their commercial competitors. The development has usually relied on being able to "chase the tail lights" of the competition and thus required little coordination and management. Being on par changes the development game and requires open source developers to coordinate their efforts much more, than they have been used to.

The open source development process results in a high rate of new versions released, which are not applauded by the commercial users of software – they want fewer releases. Open source software is typical software by hackers for hackers resulting in good technical feedback but little non-expert feedback. This makes it hard for open source software to penetrate the corporate sphere.

The open source organisation has problems addressing the needs of commercial adopters/customers. Commercial customers have a keen interest, support issues, and they also know the strategic direction of the development. Both issues are not addressed in the open source community.

In itself the Linux operating system poses a short-term threat to the Microsoft NT server market. It is relatively easy to use Linux to replace discrete networking services performed by Windows NT with Linux.

However, Microsoft has several possible strategies to counter the threat of Linux. Use of proprietary extensions to networking protocols will make it difficult for Linux to serve Windows clients on a network. There is no central company responsible for neither Linux nor other open source software. Therefore it makes sense to target on open source development process rather than on a single company.

It is interesting to read the perspective of Microsoft, and how they perceive Linux and open source software. Today, three years after being published, the Halloween document is still interesting and presents observations not yet incorporated in academic work.

Leaving the Halloween document the review now proceeds to the more theoretically based papers that have tried to offer an explanation for the open source phenomenon.

2.5 Rishab Ghosh “Cooking Pot Markets”

Rishab Aiyer Ghosh, managing editor of FirstMonday[8] presents observations on the nature of free goods on the Internet. The paper was written in 1998 following Netscape’s release of the source code for their browser. Ghosh is influenced by Netscape’s move and believes that this is the sign that the corporate world may begin to profit from open source software. Ghosh is interested in the Internet as an economy and in explaining how this economy works.

The Internet has attracted countless users and likewise countless producers, who post to news groups, create homepages or contribute software or information. All of these are economic activities that involve value but no money. Newsgroups, for example, are an economic activity, where people meet and share ideas. All these ideas and small notes not fit for publishing are welcome on the Internet and useful to someone. On the Internet our criteria for what is useful is very different from our criteria when buying books and other information, in fact, these criteria do not matter. What matters is the fact that people post information, on the Internet, as this is a product of someone’s labour. It has been created and distributed, and it may be consumed – and someone might even find it to be of value. This is particularly true in the knowledge economy, where all sorts of knowledge should be treated as economic goods.

> “Every snippet posted to a discussion group, every little web page, every skim through a FAQ list and every little snoop into an on-line chat session is an act of production or consumption or both” [Ghosh 1998]

In this perspective every exchange of information or knowledge is an act of trade. However, the value of the goods used and produced are hard to determine, and Ghosh [1998] argue that value lie in the willingness of people to consume a good.

People go to the Internet looking for people of same mind, and likewise developers of open source software are looking for other open source software developers. This illustrates the trade that is happing on the Internet. When people post messages to a news group, or a developer makes available a new piece of software, trade is happening. The other persons in that news group trade messages for their own messages. In this situation people are trading their own work for the work of others.

[8] FirstMonday is an Internet based monthly journal, which has published much research on open source software. see www.firstmonday.org

Ideas are sold in exchange for other ideas, but reputation is also a part of the equation. Posting messages that are valued by the readers will gain the writer an increased reputation, which could later be bought or sold. Reputation is different from ideas by being a by-product that is a result of producing ideas.

In the real world money is involved in most kinds of trade, and that is not the case of the Internet. On the Internet every exchange is a direct and equal exchange, which share a close resemblance to barter. There are two sides to the perception of value and exactly where the value of the good lies. In contrast, monetary transactions assume that both sides of a trade situation perceive the value as fixes.

On the Internet the two sides of a transaction change place and form two simultaneous transactions with interchanging roles of producer and consumer. In a barter economy both sides of a trade situation will have to provide something that is of value to the other part. Since money is not involved in the exchange, there is no medium in the exchange that is universal or widely accepted. Goods cannot be priced, and the valuation of a good happens at the time of transaction.

We now know that two things are exchanged in transactions on the Internet, and that they motivate production: 1) Ideas and 2) Reputation. However, since the Internet exchange is based on barter, the value gained is not readily exchanged for goods required to sustain life. Reputation gains fortunately spill-over into real life and can be exchanged for jobs or business opportunities, as has been the case for many of the top open source software developers.

2.5.1 Cooking Pot Markets

The understanding of the economy that drives the interaction of people on the Internet is one of cooking-pots. Everyone throws in small bits and pieces, and likewise everyone can take out as much as they please. There is a motivation to give something back after taking out a lot. This has nothing to do with altruism, and if it had, it would not work. The thing that is traded is information on the Internet, which can be reproduced indefinitely at no cost. Everybody can have their full share of the cooking pot, and giving a little back to the pot increases the potential value and motivation to contribute back to everybody. A person reading an interesting post on the Internet is motivated to write a new post. This post might be indifferent to all of the persons on the Internet but one, who then submits a new post, and thus the process continues.

A person is not giving away information for free; rather goods are given away to millions of others in exchange for at least one copy of information useful to that person. Since the copies provided to the millions of others do not cost anything, the producer looses nothing giving away his information. This is what makes the cooking pot function.

People participating in open source software development, news group discussions, or otherwise adding content to the Internet, are not adverse to money or altruistic. They are acting in a way that resemble rational self-interest. The cooking pot model provides a rational incentive for the reason why people produce goods and provide these for free on the Internet. Ideas (software, posts and other information released on the Internet) are produced in exchange for other ideas. A side effect of producing ideas is reputation, which the producers gain. Reputation enhances incentives to produce and may spill-over into the real world.

2.6 Lerner & Tirole “The simple economics of open source software”

Professors Josh Lerner and Jean Tirole (Lerner & Tirole) were among the first to advance the theoretical understanding of open source software development. “The Simple Economics of Open Source” [Lerner & Tirole 2000] analyses open source software using four cases: Apache, Linux, Perl, and Sendmail.

The introduction mentioned Apache and Linux, but Perl (Practical Extension and Reporting Language) is a very successful open source scripting language as well. Perl is used for administrative tasks, on web pages, and for automating tasks; Perl is often referred to as the glue of the Internet, underlining its strong ability to make different systems, databases, and the likes communicate. Sendmail is the grand old mail transport software, which runs most of the Internets mail systems. Sendmail is believed to be responsible for approximately 75% of the Internet mail. The four cases are used to provide proof for the theories and observations advanced by Lerner & Tirole.

The paper uses an economic perspective to understand open source software development. The questions, which the paper addresses are: 1) Why do people participate? 2) Why are there open source projects in the first place? 3) How do commercial vendors react to the open source movement? The first two questions are treated within the same chapter, whereas the third and last question is treated separately. The two first questions reflect the basic questions that interest economists: 1) What are the incentives for people’s actions, and 2) why do firms and companies arise. The third question reflects an understanding of the open source phenomenon as competing with and acting in the same arena as commercial companies, but under a different set of rules. The paper is interesting, as it presents four interesting cases and a sound rational economic treatment.

2.6.1 What Motivates Programmers?

A programmer will only participate in open source software development, if the programmer derives a net benefit = (immediate payoff – immediate cost) + (delayed payoff – delayed cost).

A programmer in an open source software project incurs various costs and benefits. There is an opportunity cost associated with the time spent on the project – this time cannot be spent elsewhere. This opportunity cost is related to the programmer’s affiliation. Being an independent programmer, the developer would forego monetary income by working on open source software. Should the programmer be employed, the programmer will incur an opportunity cost from not focusing on the job, which he was employed to do. This results in a slow down in performance that could result in a delayed cost from not completing the task within a given timeframe. However, the immediate payoff might counter this effect. If the programmer were employed as a systems administrator using open source software in his job function. In this situation working on open source software and fixing problems might indeed improve his job performance, as the tools used in the job would be improved.

The delayed reward consists of two different incentives, 1) Career concern incentive, and 2) Ego gratification incentive. There is a possible delayed reward to be found in career concern, as working on open source software may lead to job offers in future open source software projects. It is noted that there are extensive examples of open source software developers being funded by companies to do open source software development. Ego gratification, on the other hand, does not include the perspective of future jobs or monetary reward from open source software development. This rather stems from a desire to gain peer reputation! However ego gratifiers may wish to signal their abilities to others than the

peer group and as such qualify for a job outside open source software development. These two incentives are grouped together as signalling incentives.

Signalling incentives are stronger:

a) The more visible the performance to the relevant audience (peers, labour market, venture capital community),

b) The higher the impact of effort on performance, and

c) The more informative the performance about talent.

This gives rise to strategic complementarities, as programmers will want to work on software projects that will attract a large number of other programmers. The higher the signalling power, the more programmers will be interested in the project. Signalling incentives also have the effect of directing the effort of open source software development. According to Lerner & Tirole developers will tend to work in areas with the highest signalling effects, which are often performance related areas. Performance has the value of providing a direct comparison between two pieces of software solving the same problem.

It is questioned why leaders of open source projects return code of high value to the community, when this code might serve as a platform for monetary reward. From a pure economic standpoint it would seem that most would have problems resisting the temptation to take valuable technology private and reap the benefits. A compelling reason is not provided, and it is argued that the leaders' employment might play a part in this, as the employer would not allow a developer to profit from something that might have been developed while working on company time.

2.6.2 Comparing Open Source and Closed Source Programming Incentives

When comparing open source and closed source programming incentives, the short term rewards are first considered. In the short term commercial companies have a clear advantage over open source projects, as these are able to actually pay salaries. Commercial companies produce proprietary code that generates income, and this makes it possible for commercial companies to offer salaries.

In contrast open source projects offer two effects that lower the cost for the programmer, 1) The alumni effect and 2) Benefits from customizing and bug-fixing.

The alumni effect describes the situation in universities and schools, where code from open source software are used for learning purposes. Thereby the programmers are already familiar with the code and thus the cost of programming.

Benefits from customizing and bug fixing may lower the cost of contributing to open source software, if there are private benefits to be gained.

Delayed reward from signalling incentives also pose some advantages to open source software development over closed source software development. Signalling incentives may be stronger in open sources software development for three reasons, 1) Open source software provides opportunity for others to evaluate the quality of the code being developed, which is not possible with closed source software. 2) The open source software programmer has the full initiative and full responsibility, and therefore the programmer's performance can be judged more precisely. 3) The labour market of open source software is more fluid. Extensive sharing of code in open source software development makes the programmer and accumulated knowledge more easily adapted to a new environment.

2.6.3 Evidence of Individual Incentives

Users’ personal benefit is the key to a number of open source software projects, and often personal problems motivate contributions to open source software development. In the Apache project, the development was a direct consequence of the problems experienced by website administrators. The problems prompted the combined effort that resulted in development of the Apache server. Several other software projects exemplify how a single individual has been instrumental in the development of software.

Recognition of the developer’s effort as reputation is thought to have real effects. It has influence on peers and increases the chances of getting a job. Reputation also impacts the possibility of accessing venture capital.

2.6.4 Organization and Governance

For open source software projects to be successful it is important that the project code is modularised, i.e. the source code is divided in smaller and well-defined chunks. The project must present fun challenges for interested programmers to pursue. The leader of a particular open source project is responsible for these tasks, and further the leader must provide a vision, attract other programmers, and keep the project together.

The distributed nature of open source software development has the effect that projects must be modularised. In this way programmers can work in parallel on different modules of the code without having to extensively coordinate their efforts.

The initial leader of an open source software project must also assemble critical mass in terms of source code. The code must show the project as doable and interesting to other programmers and thereby attract programmers. Programmers are reluctant to join an effort, unless they are able to identify a challenge. It is often interesting for programmers to join in the early stages, as contributions in this phase will be very visible in later stages of the project.

The success of open source projects is also related to the nature of leadership in the projects. The governance structures in open source projects vary, some projects have an undisputed leader, others rely on a core group, and even a voting system has been used. Often leaders are the ones who developed the initial code for the project, but as the project evolves, the leader does less programming and more leading.

The leader in open source projects has little formal power but a lot of real authority, and the recommendations of the leader tend to be followed. The leader provides the vision, and must ensure that the vision is updated on a regular basis. This gives the project a sense of direction that also helps to prevent forking.

Trust is the key to successful leadership; developers must trust the leader’s decision and believe that their goals are aligned with the leader’s goal. Charisma will help leader prevent forking by signalling a strong leadership that will make indecisive programmers rally behind the leader.

2.7 Justin Pappas Johnson “Economics of Open Source Software”

Like Lerner & Tirole Justin Pappas Johnson uses economic theory to explain the existence of open source software. However, the methodology and perspective in this paper differ from the one of Lerner & Tirole. In the Economics of Open Source Software the methodology is strict economic modelling and uses the economics of public goods as the central theoretical body. More specifically it is the private provision of public goods

that is used as a basis for the model. The mathematical approach used in the paper will not be reproduced in this thesis, as it is the qualitative understanding presented that is of interest.

Like the other papers on open source software this paper refers to Eric S. Raymond [1999a], Netscape's choice to distribute their browser as open source, and the success of Apache. These are becoming the examples along with Linux that typify open source software development as having proven its success.

2.7.1 The Size of the Developer Base and Welfare

User-developers are individuals, who use open source software and potentially those, who develop and contribute code to open source software projects. It is observed that different open source software projects have varying size of their user-developer base. Some projects are more visible, more interesting or attract developers for other reasons. The number of user-developers influences the probability of development, redundant effort, and welfare.

Contrary to what one might expect, the probability of development is lowered, as the size of the developer base increases. This is because the number of developers increases, and each developer believes that the other developers will be fixing a particular problem. However, the decrease is small, and all agents prefer a large user base with potential user-developers. As the user base increases, so does the temptation to free ride, which may lower the overall probability of development. It is demonstrated that even though the incentive to free ride increases with the number of user-developers, the probability of development is only affected minimally.

Extending the above argument would suggest that an unlimited large user-developer base would reduce the probability of development to zero. However, when extending the user-developer base to infinite, the probability of the other agents developing to a single agent is less than one. Thus it is not certain that a particular feature will be developed, and therefore the user-developer with the highest value-to-cost ration will have the incentive to develop.

As a note an infinite number of user-developers might not lead to development. Given the lack of co-ordination in open source software development there is a huge potential for duplicate effort. In this model the duplicate effort does not grow without bounds. The incentive to free ride is sufficiently strong, and user-developers will restrict the amount of redundant effort.

2.7.2 Stylised Facts in Context Of the Model

It is observed that some types of software are not developed in the open source community. For instance the open source community has for a long time lacked a powerful word processor that would stand up in comparison to Microsoft Word. In contrast hundreds of different open source applications and tools exist and with a much more narrow and technical aim. There is a close relation between human capital and the production technology, which influence the production of open source software. Those that value networking utilities are also the most likely to be the ones to develop and make extensions to existing utilities. Also cost of the utility and its value play a part, as the most valuable utility is most likely to be developed.

In the open source software community the modular structure of the software is often hailed as a precondition for the distributed development to function. It is argued that small

modules and even small tasks within the modules make it possible and attractive for individuals to contribute to development. This increases the aggregated development of open source software. The basic question becomes whether a modular design as present in open source software is more productive than a non-modular design. It is discovered that the size of the developer base is an important part of the question. Given a sufficiently large developer base, the modular environment will outperform the non-modular. In contrast, with a smaller base of potential developers, the modular environment has the uppers hand. The rationale should be found in the potential developers' willingness to contribute to the project. All potential developers value the project, however, they might show no interest in modules, which require development. Since they value the project software, and the software is only useful when all modules are functioning, they have a personal incentive to contribute to the development.

One often hears the argument that people are not willing to use open source software because it's not complete. It is the impression that features are missing, and in general open source software tends to be less complete compared to its commercial counterparts. Theoretically this is true, as the probability of all modules of a project being completed approaches zero, as the number of modules grows. Commercial companies, on the other hand, have a clear incentive to complete all modules. Commercial companies care about the users' (buyers') expectations, which must be met with a complete product, where all features and modules are implemented. Developers of open source software do not share this incentive and are more interested in satisfying their own interest in the project.

The understanding offer by Johnson is illustrates the superiority of the open source development method on some cases but also highlights the problem of free riding inherent in the provision of public goods.

2.8 James Bessen "OSS: Free Provision of Complex Public Goods"

This paper [Bessen 2001] builds on Lerner & Tirole [2000] and Justin Pappas Johnson [2000] but adopts a different approach in which debugging, testing, and enhancement are the focus. Thus this complements the approach by Lerner & Tirole, which focuses on reputation as a motivating factor. The reputation value in fixing bugs, and enhancing existing software is assumed to be a lot less reputation driven, which places the focus outside the scope of Lerner & Tirole [2000]. Johnson [2000], as we have just seen, explores some of the problems that arise, if developers choose a free ride and wait to see, if other developers should develop the software first. The object of study is complex software goods, their different property rights, and different forms of provision.

The model uses a two-stage game, where the agents and companies enter during the first stage and decide whether to enter the second stage If they enter the game, they decide to develop software and provide in whatever way they please. In the second stage the produced software is offered as either a product or as open source software. The product is, of course, sold at a price, whereas open source software is offered for free. If the product is sold, the producers compete on price.

The paper examines two types of software goods 1) simple goods and 2) complex goods. This is general distinction under which three different forms of provision are analysed. The three different forms of provision are: 1) Commercial software, 2) Custom programming/self-development and 3) Open source software.

1) Commercial software is provided by firms with the purpose of selling and making a profit.

2) Custom programming/self-development are both more dependent on self-selection than software provided by commercial companies. Agents may choose to either develop the software themselves or contract the development. The software may be resold to other parties once developed. Custom programming may be provided as both product and open source software.

3) In open source software agents develop software themselves and provide it (license) to others at no charge.

Further the paper examines how different forms of intellectual property protection influence provision.

2.8.1 Simple Goods

When simple software goods are provided as commercial software, it is not possible to discriminate on price, as the software is simple and comparable among producers. If the market is sufficiently large, and patent protection is available, the provision is certain. However, if it is only a question of copyright and trade secret protection, there is a positive probability of provision. In the event that markets are small, commercial companies will not provide the software. In small markets commercial provision fails, as companies only have limited knowledge of the consumers' needs. There might be customers, who are willing to pay, but the producing company does not know this for sure.

With custom programming and self-development the producer obtains intimate knowledge of the users' individual preferences. In this situation the agents' preferences are private information, and thus custom programming by a third party developer will require transaction. Transaction costs alone, given that this is a simple software, result in no one contacting a third party developer, and therefore this is a case of self-programming. The software is not existing, and there are no existing patents to block entry. When the software has been developed, the software may be offered to anyone interested.

Agents' decisions in the first stage depend on the agents' belief in other agents' behaviour. Other agents may also choose to develop the software, and the profit from the developed software will depend on whether other agents have entered the first stage. If an agent decides not to enter the first stage and develop software, the agent may late in the second stage be able to acquire the software from other agents. Therefore agents have an incentive to wait. Provision by self-development will happen in both large and small markets. Comparing self-development to commercial provision reveals that self-provision is superior in small markets.

Open source provision is similar to self-development, but the open source license ensures that agents have no gain from sales. It is expected that no patents hinder agents in entering the first stage. A utility function similar to the one used by Johnson [2000] is employed, and there is a positive probability of provision. It is noted that agents motivated by narrow self-interest would prefer self-development due to the possibility of reselling the software to other agents and gaining a profit.

For simple software intellectual property protection such as copyright and trade secrets generates a weak improvement in the probability of software being developed. Patents only improve probability of development in large markets. There are two different ways in which open source software is affected by free riding. 1) If the cost of development exceeds personal gain from use, there is no incentive to develop. 2) When waiting costs are low, there is an incentive for strategic waiting.

2.8.2 Complex Goods

In this model it is considered how the amount of features in a product affect its provision. A product is assigned a number of features of which a base number provides basic value of the product. Additional features only increase the value of the product to some but not all users. The model now includes the cost of adding extra features to the software, and the notion of a use-product is introduced. A use-product is an exact combination of features, which a user will want and use in combination. The specific combination of features in a use-product must be tested and debugged separately, as the unique combination of features interacts in a particular way. The same product in the hands of different users thus makes up many separate use-products. However, testing and debugging are not free, and there is a cost associated with these activities. The cost of debugging is assumed to be independent of the total complexity of the product, as it is only a particular set of features that are used in each use-product. Debugging also includes enhancing the product. A feature that is not functioning correctly or implemented poorly is a bug, and features missing and not implemented is a serious bug.

The game has now three stages. 1) Agents decide whether to enter or not, 2) Agents decide which features to develop, and 3) They compete on price.

For commercial provision the situation assumed is one where a monopolist with a patent is to provide the software. It is observed that the monopolist is in a position where he may price discriminate extensively i.e. offer the exact number of features which consumer desire and thereby maximise profits. It is interesting that as the number of features grow larger and finally becomes very large even the monopolist fail to provide the program with all features. This is due to the cost of debugging all the features in the program and the cost of debugging by far exceeds the cost of development. The monopolist may therefore offer a program containing only a subset of the possible number of features and thereby profit from providing the program.

For open source software it is assumed that a base product exists, but any enhancements has yet to be developed and any use-products to be debugged. In this setting it is presumed that the base product core version is motivated by altruism or desire for reputation and is available for free under the viral GPL license (see appendix 1.2 for a copy of the GPL license). The GPL license requires that the software remains free, and that any software that incorporates GPL software also becomes GPL, hence the viral effect. In this setting all future enhancements also become GPL and may be distributed for free.

Debugging a particular use-product will be considered, if the utility is greater than the average cost of testing and debugging. Utility is equal consumption value plus the aggregate value of features in the particular use-product. If the number of agents wanting a particular use-product is much lower than the total number of agents, there is a positive probability of the feature being provided, as strategic waiting becomes less profitable. Interestingly, open source software developers are in the same situation as commercial companies in a small market, where there is little knowledge of what other agents want. Agents in open source software development, however, will have some indication based on their knowledge of other agents, although these only represent a much smaller number than the total number of agents. Consequently, the number of features does not affect open source software in the same way as commercial companies, and the number of features may be arbitrarily large.

2.9 Maureen McKelvey "Internet Entrepreneurship…"

This paper by Professor Maureen McKelvey [2001] departs from the path set by the previous authors. The paper is rooted in the tradition of evolutionary economics and makes an effort of using that very body of theory instead of public goods, economics of career concern, or the reputation game understanding. Empirically the paper makes extensive use of the writings of Eric S. Raymond, but also other sources are used. Interestingly these sources differ from the empirical sources used in the previously mentioned papers.

The paper introduces and defines the concept of "internet entrepreneurship", which is defined by five characteristics:

1) That multiple persons are distributed organisationally and/or geographically but can still interact in real time to create novelty;
2) That one person can be both user and developer but s/he does not necessarily combine both roles;
3) That copying and distributing information may be costless or it may be costly, depending in the situation;
4) That distributed persons contribute to innovation through the investment of their resources (time and effort) – without necessarily being 'paid' for their labour.
5) That instantaneous worldwide distribution of software and communication over the Internet or World Wide Web enable an identifiably different process of knowledge creation from organisation-based innovation.

It is argued that the development of the Linux kernel exhibits the above characteristics of Internet entrepreneurship. The Linux kernel is believed to be a general case of open source software development, and the arguments for the Linux kernel being a case of Internet entrepreneurship will be outlined in the following.

Characteristic 1, multiple persons are distributed organisationally and/or geographically but can still interact in real time to create novelty. The very nature of open source software, i.e. the source code is freely available on the Internet, has historically meant that anyone interested has been able to access and modify the source code. Given the existence of the Internet, people may be located anywhere on the planet and still be able to participate in developing open source software.

Characteristic 2, one person can be both user and developer but s/he does not necessarily combine both roles. As noted, anyone can participate and contribute to open source software development. This makes it possible for users to also act as developers and contribute source code. Testing is an integral part of software development and necessary for the software to become usable and reliable. However, users who want to contribute and become developers must be knowledgeable and possess the skills required for developing software. It must be noted that it is the individual, who decides whether to participate in development.

Characteristic 3, copying and distributing information may be costless, or it may be costly, depending on the situation. As the user does not have to pay for open source software, the distribution is costless. It is interesting that both market and non-market forms of distribution exist side by side. The Internet supports some of the non-market forms of distribution by offering free download of software and source code. Companies

take care of the market forms of distribution and sell boxed sets of CDs with a particular combination of software that is easily accessible.

The user’s time is a factor in the decision whether to buy or create one’s private software combination. Also search costs should be taken into account, as a user must be expected to spend a large amount of time in search for all relevant software. In contrast boxed versions offer a pre-configured installation and a selection of software that are configured and known to function in conjunction with each other.

Non-market forms of distribution also entail the printed media. Journals and magazines have been quick to enter the open source software market and have been instrumental in diffusing both open source software and knowledge of how to use it. Many magazines include a cover CD or DVD that distributes the latest updates and software.

Characteristic 4, distributed persons contributing to innovation through the investment of their resources (time and effort) – without necessarily being ‘paid’ for their labour. Many users spend time and resources testing new versions of software, in particular testing new versions of the Linux kernel. Unpaid individuals perform this activity, which is motivated by an interest in “The latest thing”, or are able to contribute by either identifying problems or fixing them. The latter two also have reputation effects, which may increase motivation.

Characteristic 5, instantaneous worldwide distribution of software and communication over the Internet, or World Wide Web enables an identifiably different process of knowledge creation from organisation-based innovation. The diffusion of knowledge of open source software more resembles the diffusion patterns associated with basic science and knowledge communities, where institutions are definable, and where there are incentives to produce certain kinds of knowledge. There are no firms visible in the development of Linux, and the development process seems very different from the closed processes in companies.

2.9.1 Coordination and Decision-Making

The definition of Internet Entrepreneurship states that the process of innovation will be different from the processes taking place in R&D departments within the firm, and makes no predictions about how it will differ.

The open source software development process is an evolutionary process, where survival of the fittest (code) is the selection mechanism. The fittest code is viewed in relation to the environment and the alternative code presented.

The selection mechanisms in Linux and open source software are comparable to R&D within firms, where selection takes place outside the market. Linux and open source takes this one step further, and selection takes place outside an organisation. The lack of firms means that some selection is made without strategic economic perspective. Most often selection is based on technical arguments, which again underlines the resemblance with autonomous scientific research.

Although open source software is a distributed form of development, there are specific persons (the maintainers) who have a final say. This organisational structure has clear limitations, as the maintainer will become bottleneck, as the organisation grows. In Linux delegating to leading programmers in their particular fields circumvents this limitation.

2.9.2 Dynamics Beyond the Initial Group of Users

In the beginning of the development Linux depended on the users being capable and also capable developers. It was the effort of the initial group that brought Linux to a state, where it began to have a wider appeal. The skilled users were more like developers, as they added code to the project and tested new code from other developers. In a company context these users were not buyers, and the point is that firms need users who are buyers.

There are a number of factors that contributes to the initial development of Linux:

The first factor is the cool factor from being part of the development of a technically challenging operating system. There is prestige to be gained from knowing about or developing a particular part of the Linux operating system. This is an explanation based on Linux being fashionable within certain groups of people.

The second factor is the visibility of Linux and open source software. Magazines have been very active to support open source software and Linux, and in turn this helps to attract users. Since Netscape released the source code for their browser as open source the in 1998 Linux and open source software have been exposed in most of the computer news outlets.

The third factor behind diffusion rates is a consequence of the technical nature of the software. The modulised structure of the source code in open source software was important to make it easy for people to contribute.

The fourth factor is the amount of people available, and who possessed the skills necessary to contribute to the development of software. Unix has since the early 1970'ies been thriving in universities, and thus students and the academic community in general have been exposed to the principles. Linux bares a close resemblance to Unix, and they are somewhat compatible. This lowers the barriers to enter development, as the basic principles are known, and existing applications can be reused.

The fifth factor is the availability of Linux on many sorts of hardware and its modest hardware requirements - Linux will run without problems on older machines.

The sixth factor is that Linux fits into a context of public domain software. It will be discussed and demonstrated in section 2.10.8 that the comparison between public domain software and open source software is misguided.

The seventh factor is the effect of market based distribution of various goods and services that has helped the rapid diffusion of Linux. Companies seem to prefer to have a clear agreement of where to get help to their installation in case of trouble.

The eighth factor is that companies packaging and distributing Linux were already involved in making Linux accessible to less skilled users, when the wider interest for Linux caught on in 1998.

These eight factors together form a system of positive reinforcement in which diffusion can be explained. The factors have been dependant on continuing use and testing of Linux by potential users.

The conclusion to be drawn from this paper is that Internet Entrepreneurship as defined by the five characteristics can be identified in open source software. This fuels the initial assumption that a new way of innovating exists in the modern economy. The question, however, still remains whether this new way of innovating will have any significant impact on the economy. Internet entrepreneurship in the case of Linux has

clearly had an impact, as an alternative operating system, which is able to compete with the operating systems produced by Microsoft.

2.10 Discussing the Contributions

Having referred the eight authors and nine contributions, there is of course room for a brief discussion. The literature shows clear signs of a progression in the research in open source software. Brooks does not concern himself with the distinction between open and closed software, as the terms open source software and free software are of a later date. Brooks is interested in understanding the software development from the perspective of a corporation. The contributions made by Brooks form a basic understanding of software development, and some of the insights presented have gained wide acceptance. This is evident from the other papers, many of which carry reference to Brooks.

2.10.1 The Cathedral and the Bazaar

The basic empirical understanding is presented by Eric S. Raymond, who appears to be the first to reach a wide audience with a description of open source software development. The papers are well written and compelling, but written with the enthusiasm on an insider. The Cathedral and the Bazaar has been instrumental in the initial discussion of open source software development. The Cathedral and the Bazaar painted a picture of open source software as a Bazaar style development, a kind of bottom-up form of development. In this view all the involved parties have influence on the direction of development. In contrast, commercial companies a la Microsoft use a Cathedral style of development – a top-down strictly controlled development.

The main contribution of The Cathedral and the Bazaar is the empirical hands-on insights, which leave the reader with an understanding of how open source software development works. It also offers some explanation of why open source software is being developed, mostly pointing to personal need (Every good work of software starts by scratching a developers own itch [Raymond 1999a]) as the motivation for developing open source software. But it is also noted that open source software developers are motivated by ego satisfaction and peer reputation.

The Cathedral and the Bazaar also countered “The Mythical Man-Month” by stating that given enough co-developers all problems will be characterised quickly and the solution obvious to someone. Part of the reason for this was later described in the Halloween documents, where it was observed that since developer time was a cheap resource that did not impose a cost to other than the individual developer spending the time. However, Eric Raymond’s claim that debugging is parallelisable does not seem a compete truism, as some types of bugs are dependant on more shallow bugs, and thus to solve a problem developers must uncover the above layers before solving the real problem. This type of debugging is not parallelisable.

2.10.2 Homesteading the Noosphere

Homesteading the Noosphere also by Eric S. Raymond is more theoretical and focuses on what mechanisms make open source software development work. The Lockean theory of land tenure is shown to match the behaviour of open source developers in the situation of acquiring projects. Attached to this is the value perspective that is introduced to explain the incentives of open source developers. In this view the incentives of open source developers are based on reputation that can be gained from giving code (gifts) to the community. The community is a surplus community, and thus standard economics do not

apply, and therefore gifts are the real currency. However, one important resource is scarce: Time. Compared to the work that needs to be done, developers only have very limited time at their hands. Whereas the paper goes a long way in explaining the mechanisms related to so-called ownership of open source software projects, the reputation/gift incentive seems simplified.

The ideas presented in the Cathedral and Bazaar and Homesteading the Noosphere are recognised in all other literature reviewed.

2.10.3 The Halloween document

The Halloween document makes extensive use of the writings of Eric S. Raymond, but makes independent contributions to the understanding of open source software development. As mentioned, the economics of debugging in open source software are explained. Further it is noted that the speed of fixing bugs are important to open source software, as other developers need these fixes to move on to solve other problems. "Impulse Debugging" [Valloppillil 1998:17] is an interesting contribution to the understanding that explains, how the availability if source code invites developers to try to fix problems that they personally experience.

Coordinating many programmers not formally organised is difficult to understand, when the resulting product is as complex as an operating system. The answer should be found in the extensive history of Unix (Linux is a Unix clone), where existing implementations of Unix have served as a target for the distributed development effort. This has been described as following the taillights of the car in front, which is much easier than finding the way yourself. The Halloween document also shed light on the GPL license, and how it provides credibility in the market. The free nature of the source code ensures that the software will always be available, no matter the fate of any supporting company.

The Halloween documents contribute little in terms of theoretical understanding and are based on the author's personal experience and perception. Microsoft's possible strategies to counter the threat of Linux show the networking theoretical nature of operating systems. Operating systems are communication systems that are dependent on underlying standards. If these communication standards change, the operating system must also change, but if the changes are made by a private vendor, the changes may be kept a secret. In this scenario Microsoft's dominating installed base is important, as it makes it possible to dictate changes in communication standards.

2.10.4 Cooking Pot Markets

Rishab A. Ghosh uses economic logic to understand the production and consumption of information goods on the Internet. Ghosh's contribution is to underline that production and consumption of information goods are economic activities. It might be that there is no money involved in transactions, but still it is an economic activity, as goods are produced and consumed. The economy is based on exchange that resembles barter, as the goods do not have a fixed value like money. The information goods that are made available for others also have a reputation value attached. The higher the quality or the more demand for the good, the higher the reputation for the producer. Reputation in turn may spill-over into the real world and offer monetary compensation in form of job or business opportunities. These reputation effects are also seen in Eric S. Raymond's paper Homesteading the Noospehere. The difference being that Ghosh does not see reputation spill-over as a factor that directly influences behaviour, rather it is a consequence of trading information on the

net. Later as the reputation spill-over generates a profit, it may indeed become a primary motivating factor.

The problem in Ghosh's paper is that it discusses information goods and uses postings to news groups as examples. Information goods in Ghosh's paper could easily have a very short life, as they are part of an ongoing discussion. The ability to search for information goods does prolong the life of the information goods. In contrast open source software should be compared to durable goods, which have a much longer lifespan than information in news groups. The durable nature of open source software imply a much more stable relationship between producer and user. Users are likely to check the web site of the open source software that they use regularly. In news groups the different categories of discussions are rather stable, but the course of development in the individual topics are impossible to predict. The cooking pot analogy does go some way in explaining the nature and continued development of open source software. The ability for everyone to pour from the pot and get a full share without having to give anything back to the pot. If one decides to contribute to the pot, everybody will be better off, as this contribution will also be available. This is hardly a precise explanation of why open source software exists and is still being developed. It is a description of the general mechanism of open source software development and the accumulation of freely available information goods on the Internet.

2.10.5 The Simple Economics of Open Source Software

The Simple Economics of Open Source by Lerner & Tirole is a compelling account that contributes to the understanding of both coordination of open source software projects and developers' motivation. With regard to coordination Lerner & Tirole has a rather rigid understanding of how open source software projects work.

In a paper [Edwards 2000a] published in FirstMonday I have discussed the very perception of leadership, and how leaders lead open source software projects presented by Lerner & Tirole. The details of the discussion can be found in the mentioned paper. However, some points from the critique are worth mentioning. It is important to understand that leaders in open source software projects have little means of directing the efforts of co-developers in open source projects. Co-developers are free to participate in projects as they see fit. Almost all co-developers are working for free, and their motivation stem from personal interest, personal need, being part of a community, wish for reputation or desire to be respected by a particular peer group. There may be other motivational factors, and motivation will be discussed in greater detail at a later stage. It seems that co-developers are motivated by many different things but not by so-called leaders in opens source projects. It might be an exception, when a co-developer is seeking recognition by a particular project leader, and does so by following his every wish. There is little evidence to support a theory of direct leadership in open source development, it rather seems that there is evidence of indirect leadership.

Indirect leadership is the maintainer performing his duty of overseeing the development of the project software. The maintainer receives bug-fixes and patches containing new features from co-developers, who are interested in the software. The maintainer serves as a gatekeeper for the project, and he single-handedly decides which patches and bug-fixes are accepted into the software. This position gives the maintainer a high degree of control of the direction of development. This is the indirect leadership that a maintainer may exert – a choice between suggested improvements to the project software.

The misconception of leadership does not affect the treatment of co-developers' motivation in open source software development. It seems fruitful that Lerner & Tirole

introduce a distinction between immediate and delayed payoff and cost. This distinction allows for an analysis of the incentives of open source software developers. The weakness in the analysis and the problem of the conclusions are the narrow focus on the individual and his utility function. There shall be little doubt that open source software development is a group effort, and this perspective seems to be missing in the analysis. Delayed payoff and costs imply strategic behaviour on part of the developers, as they seek to increase their personal visibility by joining the high-profile projects. As there are only few projects like kernel development that attract wide spread attention, this in turn imply that developers would only join these projects. This does not seem to be the case. Indeed 'sexy' projects have no problems finding developers. But many other projects seem not to suffer from developers fleeing to high-profile projects. The evidence to support this can be found at the SourceForge[9], where projects are constantly formed, and developers can join existing projects. The implication seems to be that motivation is not driven by long-term strategic behaviour to maximise personal visibility (although it may be true in individual cases), but rather personal need to solve a problem and perhaps other motivating factors.

The conclusion is that the incentive model presented by Lerner & Tirole is not sufficient to explain the existence and continued development of open source software. It does however present a contribution to understanding some of the individual incentives. These incentives are not a sufficient explanation on their own and must be complemented by other incentives that relate to the interaction between developers in a community.

2.10.6 Economics of Open Source Software

The treatment by Justin Pappas Johnson demonstrates that simple modelling of open source software development goes a long way in explaining behaviour. However, Pappas uses a different terminology, where the user-developer is the central figure. It is very true that the distinction between users and developers is blurred in open source software development. Still this distinction is useful and should be upheld. In this thesis the distinction will be upheld, and both users and developers will be referred to. Users are just users and developers are agents who develop code and contribute code to projects.

The strength of the formal approach is the ability to make a strict analysis and understand how different properties affect the model. The weakness is the fact that open source software is a young area of research, and our understanding of behaviour is far from robust. It is doubtful that the behaviour modelled is in fact the real behaviour of open source software development.

2.10.7 OSS: Free Provision of Complex Public Goods

Building on the idea of privately provided public goods, James Bessen, contribute to our understanding by demonstrating that the complexity of the good provided influences provision. Free riding is a central element in understanding of open source software provision, and free riding does not matter, if the software product is sufficiently complex. In this situation the actual usage pattern matters, and user preferences are an integral part of the testing, debugging and development. Markets are not very good at handling this situation, as it requires extensive communication between developer and user. In open source software the developer is the actual user, and thus preferences are intimately known and may be implemented at will.

[9] www.sourceforge.org

Like the “Economics of open source software” I believe that although the models are intriguing, modelling assumes that agents’ behaviour is consistent. The focus on particular use-products results in a situation, where only competent users have incentive to develop features, as their cost of implementing new features are much lower than that of the plain user with no programming experience. The graphical user-interfaces for Linux exemplify that features for plain users are fast being developed by competent users, who add features that are of specific interest to plain users. For instance the KDE (K Desktop Environment) has conducted usability tests, where KDE developers watched users in an Internet café.

2.10.8 Internet Entrepreneurship

Maureen McKelvey contributes to our understanding of open source software development by applying a different theoretical perspective than previously seen. The paper introduces the concept of Internet Entrepreneurship, and it is argued that open source software development as exemplified by Linux is an example of this concept. It is, however, unclear how the newly defined concept is used or influences analysis and understanding of open source software. The theoretical understanding of Internet Entrepreneurship is absent in the sections of the paper that introduces and explains the concept.

The paper does, however, advance our understanding of coordination and decision-making and the dynamics of open source software. In particular it is interesting that Maureen McKelvey attempts an explanation of the dynamics that are beyond that peer-group of developers. The previous papers have focused mainly on dynamics within the peer-group. Eight factors are presented, which positively influenced the diffusion of Linux, and these indeed appear plausible.

Maureen McKelvey claims that open source software fits the description of non-firm software production known as public domain software. At this point the paper makes the error of associating Linux and open source software with public domain software. Public domain software is software in the public domain; that is, no one is claiming copyright for the software. Open source software relies completely on copyright to protect the software from changing its license. Public domain software may be re-licensed by anyone to whatever license they choose. This changes the incentives for contributing to development completely, and there is little incentive for co-developers to contribute code to the project. Any code contributed may later be taken private by third party individuals, who may place and enforce a license on the software. Open source software licenses ensure that the code remains open source, no matter the amount of contributions from others.

At a later stage the paper argues that Linus Torvalds has made no attempt to license, patent or in other ways prosper form the development of Linux. Again, the paper fails to grasp the nature of open source software and in particular the nature of the GPL license applied to Linux. Linus Torvalds retains full copyright to every line of code he has written. However, Linus Torvalds has applied a particular license that grants licensees (users) extensive privileges. With this license it is not possible to patent or change the license on the already released version. He may choose to license new versions differently, but the previous versions will retain their license and thus remain open source software. It is true, however, that Linus Torvalds did not apply for patents in connection with the development of Linux.

2.11 Common Themes

The present review, reading of many other news group postings, papers and participation in discussions suggest that some stylised facts are emerging. Research into open source software also suggests that common themes are emerging. In this section stylised facts about open source software development and common themes of research are presented. Three different themes can be identified:

2.11.1 Incentives and Motivation

Open source software is valuable software, and some like the Apache web server and Linux operating system are indeed valuable to the point, where the software is revenue generating production platform. Still many contribute to the development of this software and do not receive any monetary compensation for their efforts.

For economists, at least, it is puzzling that individuals spend time and resources developing a product that is provided for free to anyone interested.

2.11.2 Coordination and Communication

Open source software development is done by countless developers situated in different parts of the world communicating via. Email, chat and other Internet based forms of communication. Developing software and in particular large systems like the Linux kernel, Apache web server, and graphical user interfaces is highly complex. This raises the question of how the efforts of all these developers are coordinated. Poor or no coordination is expected to lead to duplicate efforts and destructive development in the sense that one developer may change something that is the basis for other developers work. Resulting in even more work, when fixing what is now broken.

The speed of development, as seen in many open source software projects, suggests that coordination does indeed exist and is efficient.

2.11.3 Understanding the Process

The third research theme is focused on understanding the actual development process and sees the process as a key to understanding, why open source software is being developed. Within this theme lies a discussion of tacit knowledge and learning in open source software. Some developers in open source software are extremely knowledgeable about the software, they have developed, however, documenting and disseminating this knowledge seem to be a problem.

2.12 Summary

In this chapter a selection of papers has been reviewed and discussed. These papers form a basic understanding of open source software each providing an answer to why open source software exists and is being developed. These papers have served as the foundation on which this thesis is build.

Some of the papers are filled with empirical insight and others pursue a theoretical perspective to understand open source software. The theoretical papers have used economic theory to understand open source software.

3 Research Questions and Methodology

Until now the meta-question has been the ambiguous: “Why is open source software being developed?” Answering this question also requires understanding of how open source software is being developed. It is it the purpose of this chapter to develop and define research questions, which must be answered in this thesis.

This chapter begins with a discussion of how open source software should be understood. In course of the discussion, the relevant agents who participate in open source software development are identified.

The two types of agents identified are then used as focal points for a tentative analysis of open source software. The tentative analysis employs the literature presented in the previous chapter to investigate whether the contributions can be used to answer the meta-question. The analysis reveals the shortcomings and strengths of the contributions and leads to the following section of this chapter, where the meta-question is refined and research questions stated.

3.1 Understanding Open Source Software

From the introduction and brief description of open source software in general and Linux and Apache in particular it is clear that the existence and continued development of open source software seem both special and interesting. The most important features of open source software are:

- Open source software is free - anyone may use, redistribute, and make improvements.
- Software development is costly and requires times and knowledge as a minimum of input.
- Both private unpaid developers and firms participate in open source software development.

One analytical problem of open source software is the observation that both firms and private developers participate in open source software development. Looking elsewhere to other industries it is not possible to identify the same behaviour. Rarely, if ever, are private persons engaged in actual product development together with firms. This mix of firms and individuals engaged in open source software development and thus often contributing to the same open source software projects makes it difficult to choose a theoretical framework, which is capable of handling all agents and explain their behaviour. A theory, able to explain the incentives of those participating in open source software development, will stumble on the fact that firms and unpaid individual developers are subject to very different incentive structures.

Unpaid voluntary development of open source software has been the driving force from the beginning. This fact leads to the assumption that unpaid voluntary development of open source software will continue to be a central part for some time to come. It is, however, possible that widespread adoption of open source in firms will increase the share of contributions coming from firms.

Firms must cover their costs in order to pay salaries and fixed costs. Therefore firms, even though they are developing and providing open source software, must relate their open source activities to some kind of revenue generating perspective. This perspective can be both short and very long, and network properties of software operating systems predict that the perspective be long.

Intuitively, it seems fruitful to adopt the observed distinction between firms and unpaid voluntary developers as the analytic distinction between two different types of agents. This distinction is based on incentives and the assumption that firms and individuals must have different incentives. Firms must pay salaries and thus generate revenues. In contrast, an unpaid voluntary developer does not have to generate an income from open source software development. I have chosen to refer to unpaid voluntary developers as private developers to convey a picture of private persons sitting in their homes contributing to the development of open source software.

Incentives and motivation for contributing to open source software development must be understood as covering a broad spectrum. In one end there is the cynical profit maximising perspective, where open source software development is the foundation for revenue generating activities. In the other end of the spectrum we find the incentive of just participating for fun.

Firms might participate for purely research purposes, where a firm is experimenting with open source software to either develop a new business model or a new product. In this perspective open source software is viewed as a possible opportunity that must be measured. Firms might rely on open source software for some of their own computing needs, and therefore they have an incentive to improve the very open source software they use.

Private developers are per definition not relying on open source software as a source of income. Thus, the incentive spectrum for private developers is different and range from participating in open source software development for fun in one end, and in the other end of the spectrum we find an incentive to develop open source software to demonstrate a particular job qualifying skill. In-between lie personal needs for some particular software and the drive to be part of a particular group of persons, who develop open source software.

The two groups of agents may choose to either use and/or modify their open source software. Users are individuals or firms, who just use open source software and do not make modifications to the software. Developers make modifications and contribute to development of open source software by adding code to the project, fixing errors or otherwise helping to understand the programming problem at hand. Often developers are also users of the very software they help to develop.

3.2 Tentative Analysis

Before developing research questions, a tentative analysis of the meta-question in light of the contributions presented in the previous chapter is conducted. The purpose of the tentative analysis is to determine whether the existing literature could provide a satisfactory answer to the meta-question of this thesis. The tentative analysis tries to answer the meta-question: “Why is open source software being developed?” and uses the distinction between agents developed in the previous section.

3.2.1 Fred Brooks "The Mythical Man month"

Using Brooks to answer the question poses a problem, as Brooks does not treat open source software in particular. However, Brooks does have a deeper understanding of software development.

Brooks is concerned with understanding the software development process when developing large systems in a firm. Brooks worked on IBM's OS/360, which at its peak had thousand people working, and in the period 1963-66 approximately 5.000 man-years went into the project. From this perspective, a firm has an obvious motivation to make every developer work in the same direction in a coordinated fashion, maximising productivity.

Open source software development is different from development in a company, as developers are rarely hired by the same organisation and are often volunteers working for free, and there is no guarantee that developers work to achieve the same goal.

But, Brooks does provide insight, which sheds light on open source software development. Brooks was the first to describe "The Joys of the Craft", which are the intrinsically motivating factors, and they should also apply without restrictions to open source development. Programming is a creative activity, where developers are creating useful software from pure thoughts. This software is useful not only to the programmer, but also to other persons, and to make something that is useful to others, is rewarding in itself. Apart from being a creative activity, software development is also an engineering activity, where the developer is playing with something that is complex. Programming is also a learning activity, and learning in itself is motivating. In open source software this is amplified, as many developers are not paid for their work, and they develop open source software, only out of interest.

Using Brooks makes it possible to account for personal motivation for developing open source software. If this logic is extended, it can also be used to explain why developers are contributing software to open source projects. If a programmer finds it enjoyable to work on a project, then the programmer will want to share the fruits of his effort. Sharing the software will make other people use the software, which the programmer has created, and this is intrinsically rewarding.

It is not possible to account for the behaviour of firms using the work of Brooks, as it is assumed that firms are working for profit and must deliver at a specific date. Although engagement in a business that the employees find interesting might boost productivity, this is of little interest, if the business cannot support the salaries of the employees.

Brooks discusses the cost of coordination, which is related to the number of communication paths n(n-1)/2 and the cost of adding programmers to a project (cost of training). The reason for the complexity and rising number of communication paths' is the fundamental problem of software: It is an immaterial being. Unlike a building or other material goods it is impossible for software developers working on the same project to easily illustrate how the different parts of a program relate to each other. When programmers discuss their changes these will have to be communicated to all the other programmers in the project and this is the cost of coordination according to Brooks.

Using the insight of Brooks to understand open source software leads one to the conclusion that open source software should not be possible. Open source software projects are characterised by many developers entering and leaving projects without restrictions. Developers are constantly added to projects, which according to Brooks should severely hamper any progress in the projects, as all resources would be allocated to

educate newcomers, and the number of communication paths would be varying. Increasing the number of developers increases the number of communication paths and thereby increasing the cost of communication. It seems that both development of the Linux kernel and the Apache web server defy the insight presented by Brook by their very existence, and the fact that many developers continuously join and leave the projects. In this respect the insight offered by Brooks does not help our understanding of coordination, nor does it even begin to explain the involvement of firms in open source software development.

3.2.2 The Cathedral and the Bazaar

The Cathedral and the Bazaar by Eric S. Raymond was written to explain open source software development, and it should be expected to help solve the meta-question of this thesis. According to the Cathedral and the Bazaar, developers are motivated by personal interest or a personal problem. This explains the motivation for individuals to participate in open source software development.

The Cathedral and the Bazaar does not offer any direct answers to why firms are developing open source software. However, personal need could also fit the description of a firm, which needs a particular type of software. Firms, however, would not be expected to give away software that might provide the firm with a competitive advantage.

3.2.3 Homesteading the Noosphere

Homesteading the Noosphere, also written by Eric S. Raymond goes further than the previous contributions in providing an answer to the meta-question. Like Brooks, Homesteading the Noosphere recognises the pure artistic satisfaction and craftsmanship of programming as a motivating factor. But the relationship between developers also affects incentives. Open source software developers are part of a reputation game, where status is derived from contributions and the quality hereof. This provides an incentive for developers to create open source software. In this perspective developers are maximising their reputation incentives.

Homesteading the Noosphere does not offer any explicit insight to the participation of firms in open source software development. But, the logic may be used to explain the involvement of firms: Firms may enjoy benefits from reputation effects when participating in open source software development. However, reputation tends to be bound to a particular individual and the achievements of this particular individual. A firm may benefit from reputation effects by being able to attract skilled staff and enjoy collaboration from unpaid open source software developers. In this way, firms would enter a strategy of maximising the returns on investment i.e. contribution, which the firm has made to open source software. Reputation effects may have direct revenue generating effects, as the firm is associated with a particular skill that can be marketed and sold. For instance, Cygnus Solutions provide support solutions and customisation to open source compilers [Thiemann 1999]. Cygnus Solutions maintain a number of open source software packages, and this makes Cygnus the obvious choice when choosing a support company for one of their software packages.

3.2.4 The Halloween Document

Halloween Document explains motivation for participating in open source software development as solving the problem at hand, education, and ego gratification. Solving the problem at hand explains the immediate motivation for developers, but does not explain the incentive to release the resulting software as open source software. Education is an

incentive to learn from a programming experience and comparable to the "joys of the craft" as explained by Brooks. Education also has another perspective, when open source software is used for educational purposes. This provides developers with an incentive to develop open source software, as the efforts are rewarded with grades from the educational system. Ego gratification is the incentives inherent in the reputation game as discussed in Homesteading the Noosphere.

The Halloween Document dismisses the idea of firms having an incentive to participate in open source software development. This is largely based on the authors' opinion of the open source licenses, which require source code to be available and make it impossible to generate an income from developing open source software.

3.2.5 Cooking Pot Markets

The incentive for private developers in the Cooking Pot model is expected return and intrinsic motivation from developing open source software. A developer contributes to a project, which is used by many others, and he expects that at least one of the many others will make contribution that is useful to the developer.

The Cooking Pot Model does not offer any obvious incentives for firms to participate in open source software development Firms must have an incentive that goes beyond expecting a useful return once in a while, before wasting resources on open source software development.

3.2.6 The Simple Economics of Open Source Software

Private developers are motivated by a positive net benefit (net benefit = immediate payoff – immediate cost + delayed payoff – delayed cost) from developing open source software. And the developer's personal benefit is a key motivating factor. Need for a particular piece of software may motive a programmer to create the needed software. Career concern and ego gratification are delayed benefits providing incentives.

However, the explanation offered by Lerner & Tirole seems only to account for individual involvement in open source software development. The factors motivating the individual programmer do not provide incentives for firms. Ego gratification and career concern are not incentives for firms, however, signalling effects from participating may provide incentives for firms.

Firms may use involvement in open source software to signal a particular set of skills. Open source software has the advantage that the technical merit of a solution can be judged immediately by anyone interested. The skills can be viewed as a commodity, which a firm may sell, and from which a profit can be derived. In this perspective open source software development offers an opportunity for firms to enter a market that otherwise would be difficult and costly to enter.

3.2.7 Economics of Open Source Software

Again, personal interest and the "joys of the craft" play an important part in the incentives for private developers. The developer with the highest value-to-cost ration is the developer most likely to develop a program.

It is explained that features in open source software are often incomplete, because the developer looses interest, when the part of feature that he needs is working. This provides an opportunity for firms to complete the missing features. Firms could then derive a profit from completing features and selling these.

3.2.8 OSS: Free Provision of Complex Public Goods

In this model firms do not have an incentive to provide open source software, as it is assumed that there is no profit to be made. Rather the opposite view is promoted, firms do have an incentive to develop, and market software is protected by patents.

Private developers have incentives to develop open source software, if the cost of waiting is higher than the cost of developing the software. There are also reputation effects in open source software development that may offset the cost calculation and provide incentives for developing open source software. For complex software products, where the basic software exists, but where features are missing or incomplete, there are incentives for developing features. This type of software has highly individual usage patterns, and a person needing a feature cannot expect anyone else to have an incentive to develop the needed feature. This provides an incentive to develop the needed feature. Since there is basic software, which is open source, the GPL license will require that enhancements be returned to the project, if distributed.

3.2.9 Internet Entrepreneurship

Firms have an incentive to develop open source software or participate in open source software development. The incentive lies in attracting innovative capability, which the firm will need to make profitable innovations. Open source software represents a large amount of knowledge, which firms can use to develop products and services. Using this knowledge lowers research costs of entering a new field of research.

3.2.10 Concluding the Tentative Analysis

The analysis has demonstrated that the literature of open source software does not provide exhaustive answers to why open source software is being developed. The tentative analysis showed that different contributions had a common understanding of why private developers create open source software. The most common answer was personal interest and what Brooks called the joys of the craft – the intrinsically rewarding traits of developing software. It was also noted that reputation was a factor and reputation could have spill over effects into the real world and provide job opportunities. Interestingly none of the contributions discussed the increasing marginal cost associated with individuals developing open source software. It must be expected that individuals, which work on open source software in their spare time, actually have limited resources available and thus the marginal cost is rapidly raising.

The reason for firms' participation in open source software development, on the other hand, is not accounted for in the papers reviewed. It serves to be mentioned that none of the contributions was focusing on understanding the incentives for firms. However, some explanation was given by applying the theory of the articles to the question. It seems that some of the papers reviewed can be developed to provide an understanding of firms' involvement. In the tentative analysis of the reviewed papers it was demonstrated that firms could leverage of the signalling and reputation effects associated with contributing to open source software. Firms can also use existing open source software projects and modify them into a more usable product with a notably better finish.

It is my opinion that the contributions do not offer the full answer to the question of why open source software is being developed. The contributions tackle different aspects and special situations and do provide insight in these areas. The contributions demonstrate some interesting approaches to understanding open source software, and some of the ideas will find use in this thesis.

3.3 Developing Research Questions

The tentative analysis did not provide a satisfactory answer and revealed that the present body of literature only accounts for the involvement of private developers in terms of the joys of the craft and solving a personal need. The papers did not provide explicit explanations for firms being involved in open source software development. I find that these explanations are too narrow in their scope to provide an explanation. Personal need and fun is certainly easy to understand and a compelling reason for individual developers' involvement. Should this really be the only dynamic and the sole reason for the existence of open source software development?

The purpose of this section is to develop a set of research questions, which in combination will provide an answer the meta-question of why open source software is being developed. Developing research questions is a task that involves intuition, personal preference of theory, and empirical knowledge. The process will start by briefly stating what we know about open source software development, my perspective and what I find interesting about open source software. We then proceed to discuss different elements, which should be part of an explanation to the meta-question. In course of doing so, a model will be introduced to explain why open source is developed. The model will be used for analysing and understanding open source software development.

3.3.1 What We Know and the Interesting Aspects

We know that:

- Private developers create and use open source software
- Firms create and use open source software
- Some open source software has a dominating market share
- Individuals use open source software to satisfy computing needs
- Firms use open source software to satisfy computing needs
- Open source software allows free copying, distribution and modification
- Open source software developers are situated all over the globe
- The Internet provide central means of communications
- Open source software is still showing a growth in adoption[10]
- Software development is time consuming.

The list of what we know about open source software can be characterised as a simple form of stylised fact about open source software. It is simplistically because the list contains no answers, but merely reflects accepted empirical facts.

My perspective is a combination of engineering and economics, and this naturally influences what I find to be interesting aspects of open source software development. Open source software development is interesting, because both individuals and firms are contributing to development of the same software. As an engineer, I can easily relate to the notion of developing for fun, however, it seems that only private developers can afford such leisure. Firms must have some sort of incentive for developing open source software,

[10] Sourceforge.net a website specialised in hosting open source software projects continue to increase its user base.

perhaps they need the software for internal computing i.e. a file server, web server etc., or the firm wishes to profit otherwise.

However, open source software allows anyone to copy, modify, and distribute the software, and this makes it difficult to understand how open source software can be a revenue generating activity. It is very interesting that open source software is being developed in spite of doubtful perspectives of profiting from the efforts put into development. I venture that other forces besides fun or personal interest must be at work to account for the success of open source software. I believe that there are mechanisms in open source software development that can be uncovered and explained.

3.3.2 What Should be Researched?

Open source software is a fairly young area of research, and it seem that every aspect of open source software deserves to be researched. However, there are limits to the size and scope of this thesis, and boundaries must be set. I am interested in understanding why open source software is being developed. In itself this question is extremely broad and must be qualified further.

The question reflects a lack of understanding of the incentives to develop open source software, and it focuses on the current reasons for developing open source software. I am not interested in the historical explanation of the existence of open source software, which may be very different from the reasons, why firms and private developers contribute to open source software today. I am 'just' interested in understanding why open source software is being developed in light of the market driven society of the western world.

As illustrated by the reviewed papers, open source software can be analysed using different sorts of theory. The choice of theory to be used here is determined by my knowledge of available theories, personal preference, and intuition of how a theory might help understanding.

I am pursuing the incentives of firms and private developers to contribute to open source software development or creating new open source software projects. I believe that any reasonable explanation should account for incentives to develop open source software, and this account should be based on theoretical arguments. Further, this explanation should identify mechanisms related to open source software itself or aspects of its development.

3.3.3 A Model for Analysing and Explaining

The following model is proposed for analysing and understanding why open source software is being developed. It is not a model of open source software development It is a model that is intended to illustrate the elements that I believe should be a part of the explanation. This is an instrumental approach to identify and test elements responsible for open source software being developed. It is not suggested that the elements proposed present an exhaustive list, but it is believed that these elements are significant.

I believe that the incentives to develop open source software should be explained by two interacting elements: 1) Properties of open source software, 2) The type of agent involved. Properties of open source software are actually a subset of the general set of properties of software, in which case the first interacting element should instead be "Properties of software". This can be illustrated as follows:

```
Properties of software + Type of agent=> A certain behaviour
(incentives and mechanisms)
```

Properties of software are properties, which are tied to the actual open source software and affect behaviour. For instance, software of any kind can be reproduced at little or no cost at all, but open source software is explicit in allowing the software to be distributed. Commercial software uses copyright protection and technical measures to prohibit redistribution of the software. I propose technical properties of software and properties of software licenses as the two elements of 'Properties of software', which determine behaviour and create mechanisms.

Agents, as mentioned, consist of two different types based on their incentives: Individuals and firms. Firms must pay salaries to employees developing open source software, and this very fact influences the behaviour of firms. Individuals do not have to pay salaries and generate income from their involvement in open source software development, and this affects their behaviour. It must be taken into account that private developers incur an opportunity cost when spending time on developing open source software, time which could have been spent elsewhere generating income.

It is possible to further divide firms and private developers into subcategories, which are related to how the firm uses open source software. Firms may use open source software for all sorts of computing needs, for instance providing network services like file and print services. Firms may also market open source software or supply open source software as part of other sales like hardware or service (consulting). Incentives for participating in open source software development are different in the mentioned examples. Private developers, we have learned from the literature, may participate for fun, they may also participate, because recognised contributions to open sour software projects document a skill, which may be job qualifying.

The result of interaction between the two elements is a certain behaviour, which is characterised by incentives and mechanisms. Incentives are motivation to which agents respond in a certain way. Mechanisms are cause-and-effect relations induced by interaction between the two elements.

So far the analytical model has neglected to discuss the level of aggregation on which analysis is performed. Traditionally economics have defined a macro, meso, and micro level of aggregation. The macro level refers to the level of national states, meso, refers to the level of the industry, and micro level refers to the level of a single firm, individual or household. The level of analysis in this thesis is the agents participating in an open source software project. An open source software project consists of a number of agents, which may be firms and/or individuals interacting as users and/or developers of the project software.

A project is traditionally defined as the effort to produce a particular program. The definition does not contain a number of developers or a size of the program to be developed. Rather a project seems to be defined as a rational division between programs. Larger programs are split into smaller projects with a more narrow focus, as the original project becomes too large. When a programming effort is formed and available for participation on the Internet, it seems clear that a project has been formed.

Theoretical understanding and use of theory play a large part in this analytical model. The relation between elements should be understood using theoretical argument. The following section presents the theoretical tools, which are used for understanding the interaction between elements.

3.3.4 Theoretical Tools

The analytic model with its two elements does not pretend to explain the nature of the relation between the elements. Theory and theoretical argument should be the tool for explaining this relationship.

Again, personal preference plays a part in the choice of theory, and I believe that economics has a lot to offer our understanding of open source software development. Economics is a large and diverse theoretical body, and it is important to focus on select areas of economics, which I believe are well suited for analysing open source software development. The choice of theory also reflects my personal intuitive understanding of the important dynamics in open source software.

The first of the two elements is property of open source software and defined by technical properties of software and properties of the license. I propose theory of economic goods as a tool for understanding how properties of software influence behaviour. It has been mentioned in the literature reviewed earlier[11] that open source soft can be viewed as a public good. I, however, venture that is far more complex and depending on agents and the properties of software, the same software can be different types of good.

In open source software development, one agent contributing to a project can to some extent be viewed as adding value for all users of this very software, notably without being compensated. In general this phenomenon is referred to as externalities, and I believe that externalities are a key dynamic in for the existence of open source software. For this reason, theory of economic externalities will be employed to analyse and understand the existence of open source software.

Economic goods and externalities are not discrete elements; rather, they complement each other. Economic goods refer to properties related to consuming the goods, and externalities can exist in both consumption and production. Some types of goods show externalities and can be defined as special cases of externalities.

Software is notoriously a hardware-software system, as using a program always consists of at least two components: 1) A hardware platform and 2) A software component. Often the program, with which the user interacts, i.e. a word processor, is dependent on even more components such as operating systems and software libraries[12]. Because software is hardware-software systems, there is a tendency to create dominating standards. This effect is believed to also be present in open source software development and to be a factor determining behaviour. For this reason economic theory of competing technologies is included as the third theoretical tool for understanding open source software development.

3.3.5 Research Questions

The research questions in this thesis proceed down two different lines of inquiry: 1) The hows of open source software and 2) The whys of open source software.

The hows are focused on documenting the empirical aspects and understanding how open source software is used, developed, exchanged, and perhaps even sold. In this line of

[11] See the review of Justin Pappas Johnson and James Bessen in chapter 2.

[12] A software library is a set of functions residing outside the individual program, which the program may use. This is an economical and consistent way of programming as several program can use the same functions.

inquiry, the processes that lead to open source software must be described. Given the current state of affairs in the literature of open source software the hows are very important, and no serious analysis can be conducted without a solid empirical understanding.

The whys are focused on providing a scientific explanation of why open source software is being developed. The whys must ask questions that facilitate a theoretic explanation, which is consistent with the empirical facts.

To summarize: The meta-question is: ”Why is open source software being developed?” The model for analysing open source software and answering the meta-question is to propose that open source software is being developed as a consequence of a particular behaviour (incentives and mechanisms). This particular behaviour is induced by interaction of two elements each containing two sub categories: Properties of software and agents. Three types of theory are suggested as tools to explain the interaction between elements.

So far little attention has been devoted to the empirical aspects of answering the meta-question. Needles to say, empirical evidence is an important issue when trying to explain, why open source software is being developed. A substantial amount of empirical evidence will be collected and used in conjunction with the proposed theory. Open software is a broad theme, where private developers and firms develop and use open source software. Understanding how open source software is developed, used, and exchanged is important to the later analysis. The following two research questions (RQ1 and RQ2) are directed at gaining empirical understanding of open source software.

```
RQ1: How is open source software being used, exchanged, and
marketed?
```

The open source development process will be analysed using theory, but before this stage it is important to describe the actual development process. The following research question will do so:

```
RQ2: What characterizes the open source software development
process?
```

In order to answer the meta-question it must be divided down into separate elements. Properties of software were thought to be an important influence on behaviour.

```
RQ3: How do properties of software (license - and technical
properties) affect open source software as an economic good?
And how does the type of good affect behaviour?
```

It is anticipated that open source software in most situations will not be a private good even, although open source software sold in boxed set in shops could be characterised as such. It is also expected that to both types of agents will be present in most situations and open source software will have public good characteristics. It is my belief that economic externalities, in particular positive externalities, of open source software are important to understand.

```
RQ4: What are the externalities of open source software?
What is the source of these externalities?
And how do these externalities affect behaviour?
```

The last two research questions are answered as part of developing a qualitative model of open source software development. The model is the thrust of the thesis and single most important contribution to be made.

3.4 Research Methodology

This thesis uses a research methodology, which can be best described as a hybrid between an explorative approach and a hypothetical deductive approach. The aim of the thesis is to answer the meta-question: "Why is open source software being developed?" This is done by creating a model for open source software development, which describes the relationship between agents in a project. The model for open source software development takes into account properties of software and the type of agent.

Before creating a model, the properties of software must be understood. This is done by dividing properties of software into technical properties and license properties. For this reason a collection of open source software licenses are analysed, and that resulted in the choice of three different licenses. The analysis of the licenses is the first explorative study of this thesis, and precedes creation of the model software development and consumption. The explorative study of software licenses results in three licenses being selected for their distinctness, and behaviour is analysed using these licenses. The model of software development and consumption is then tested on a number of mini-cases.

Analysing a number of mini-cases tests the validity of the model software development and consumption. The mini-cases are development stories from in-depth interviews with seven open source software developers. The development stories are created by analysing the interviews and extracting all statements referring to the development of a particular program. The extracts are then transcribed into a mini-case.

The model of software development and consumption is then used to analyse the mini-cases. The model is considered valid if, indeed, the model is able to provide a rational explanation for the behaviour observed in the mini-cases.

3.5 Empirical Sources

When studying open source software, one quickly finds that good empirical descriptions are few and far between. The ones that have been made are often used in many different articles. For instance, the empirical insight from Eric S. Raymond's papers is used in most of the other papers referred.

Apart from this, the real advantage of studying open source software is the fact that developers communicate using mailing lists and newsgroups stored in accessible archives. The drawback to this is the amount of emails and news postings that pass back and forth on these mailing lists. The mailing list for Linux kernel developers has in excess of 1200 messages per week.

There is a number of websites, which publish articles and reports from various open source software projects. These are consistent weekly development reports following various projects. Three sites have been very important and followed intensely: 1) Kernel Traffic[13], 2) Linux Weekly News[14], and 3) Linux Today[15].

[13] Kernel Traffic presents a weekly summary of discussions on the Linux kernel mailing list produced by Zack Brown. Kernel Traffic can be accessed at http://kt.zork.net/kernel-traffic/

Kernel Traffic is a weekly digest of important events and discussions that have taken place on the Linux kernel mailing list. Kernel Traffic often includes lengthy discussions under the subject "Developer Interaction", and they present a snapshot of how open source software development is conducted.

Linux Weekly News covers nine different sections: Main page, security, kernel, distributions, development, commerce, Linux in the news, announcements, and letters. Linux Weekly News provides a good and extensive coverage of what happens in the Linux community. The main page covers headlines, security announces and discusses security related problems and how to fix them. Distributions cover news related to Linux distributions, and kernel serves as a supplement to Kernel Traffic. Linux Weekly News provides a through and broad coverage and is generally considered to be fair in their descriptions.

Linux Today is a site that serves as a news bulletin board. All sorts of open source software related news are mentioned, and links to the original articles are provided.

Three open source software projects have been monitored during the project: 1) The Linux kernel, 2) Amanda, and 3) Zinf (Previously FreeAmp). The Linux kernel mailing list has as mentioned in excess of 1200 messages per week and it is impossible to read all posts. The Linux kernel mailing list has been followed to keep track of major developments and follow selected discussions. Amanda is an open source backup tool capable of scheduling backups, keep track of tape archives and much, much more. Zinf is a free MP3 music player with an innovative way of keeping track of ones music collection. The latter two projects have been followed closely and I have been using the programs extensively.

To get to know Linux and open source software in general, I have installed, used and tested various Linux distributions and software packages. Using open source software has also meant involvement in the open source software community. I have also participated in various newsgroups related to using open source software. I have been engaged in the Danish Skåne Sjælland Linux User Group (SSLUG), and I have participated in meetings, conferences, and discussions in their news groups.

Interviews have been an important source of information, and the next section presents the interview guide that has been used during the interviews.

3.5.1 The Questionnaire

The empirical basis for the project is a number of in-depth interviews conducted with open source software developers in Denmark. The interviews were explorative but followed the same questionnaire. The respondents were allowed ample room to elaborate and become inspired by the questions and the interviews lasted between 2-4 hours each. The following are the questionnaire used during the interviews

Project Participation

[14] Linux Weekly News or LWN as it is called is a weekly Internet publication discussing open source software related issues. LWN sets it self apart from the rest by publishing informed commentaries on recent events. LWN can be accessed at http://lwn.net

[15] Linux Today is a general news site reporting headlines and links to other news sites. Linux Today can be accessed at http://linuxtoday.com

- Do you participate in the development of open source software?
- For how long time has you participated in open source software development?
- How many projects do you currently participate in?
  - Which projects?
  - Why these particular projects?
  - How many individuals participate or have participated in these projects?
- When did you start participating in the different projects?
- At what stage were these projects, when you began participating?
  - (Start-up, in the middle, or it was already a complete product)
- How much time do you spend on the various projects?
  - Free time or at work?
- Do you perceive your work on open source software as a hobby? (No expectations of any monetary outcome)
  - Or, is there a strategic perspective (expecting payoff like job or recognition etc.)?
- What influence does the particular license have on your participation?
- Would you participate, if it wasn't GPL, MPL, NPL, Artistic License?

Actual Development

- In general, how does the actual development happen in an open source software project?
- What are the technical demands for participating in open source software development?
- What infrastructure has been used in the projects?
- What was the programming languages used in the projects?
- Are different languages used in different projects?

The Development Process

- How is code developed in the projects?
- How do you choose, which part of the project to work on?
- How do you ensure that there are not two persons working on the same part of the code?
  - Is there a division of labour?
- Is there a kind of collaboration between participants about how to solve problems?
  - What kind of collaboration?
  - What media is used for communication between participants?
  - Pros and cons?

- Who judges what work gets included in the project?
- Do different types of software pose different programming challenges?
- Is it more difficult to develop the OS than a word processor?
- How does modularisation arise?
- Is open source software development different from commercial development?
- How are conflicts resolved?
  - (Is someone acting as a judge and jury?)
- How do you get participants for the different projects?
  - (Personal contacts, home pages, news groups etc..)
- How is source code distributed to interested parties?
  - (Email, CVS, newsgroups, web, ftp etc.)
- What type of programs is not suited for open source software development?
- Is open source software development a structured development process or pure ad hoc?

Maintainer/Leader

- Who owns/controls OSS projects?
- What is the role of the maintainer?
- Is the maintainer a motivator?
- Is the maintainer in a position to pass orders to participants?
- How do participants react to contributions being rejected?

OSS in General

- Is there anyone, who makes money on OSS development?
- What kind of people participates in OSS development?
  - Students, unemployed, employed, self-employed, what?
- Why does people participate in OSS development?
- Types of innovations
  - Does OSS development produce other than incremental development?
- Supply and demand for both developers and software
  - How is demand for 'not yet developed' software articulated?
  - How are developers "captured" to participate in projects? (There must be many with good ideas, and few with resources to carry them through)
- Bug fixing - is it something special?
  - Do OSS-projects have special advantages compared to commercial development?

- Is bug fixing in OSS-projects faster?
- Is the use of users as beta-tasters an advantage?
- What are the problems with this approach?

- Is security in OSS as high as claimed?
  - Bug fixing depend on people actually finding the error. Errors are rarely a consequence of a systematic bug hunt.

3.5.2 Documenting the Interviews

The interviews were taped for later analysis and transcribed. The complete transcriptions can be found in the separate volume III "Interviews". Please note that the interviews and transcriptions are in Danish. The interviews are confidential but can be obtained at request and are not freely distributable.

3.6 How this Thesis Differs

This thesis is different from the presented contributions in sheer size as well as in its perspective. This thesis tries to understand the existence of open source software from an economic perspective.

The review of the present literature revealed a lack of understanding of firms' involvement. In this thesis firms and private developers will receive equal attention.

The thesis continues a tradition of analysing open source software as economic goods, which have special properties and externalities. However, the treatment in this thesis is not based on mathematical modelling, but on a qualitative approach. It is my opinion that a qualitative approach is the better choice, given the current level of understanding.

Empirically this thesis makes use of personal interviews, news group discussions, and literature. Where the thesis differs, is in the amount of interviews. The interviews conducted have been intensive, explorative, and have tried to go into depth. Most interviews lasted between two and four hours. The choice of subjects was constricted by financial issues, which meant that they had to be found within the borders of Denmark. Luckily there are many fine and prominent open source software developers in Denmark, and also a very active open source movement consisting of many Unix and Linux user groups with many members. One of the largest Linux user groups in the world, SSLUG, is situated in Copenhagen. According to a recent study by Lancashire [2001], Denmark, Sweden, and Norway are the countries in the world with the highest number of open source software developers per capita.

3.7 Summary

This chapter has been rather lengthy and has covered a tentative analysis based on the papers reviewed in chapter 2. The main conclusion of the tentative analysis was that the current literature did not offer a substantial explanation of why open source software is being developed. In particular the tentative analysis found that incentives for personal involvement could be understood in terms of personal need, the fun factor, and strategic behaviour. Firms' involvement in open source software development was not explained.

Following the tentative analysis a set of research questions was developed. In course of this development it was proposed that the answer to the meta-question should be found

in incentives and mechanisms. These incentives and mechanisms were in turn a consequence of two interacting elements. The elements were defined as 1) Properties of software and 2) Agents.

The interaction of these elements should be understood using theory, and economic goods, externalities, and competing technologies were proposed.

The following research questions were stated:

1. How is open source software being used, exchanged, and marketed?

2. What characterizes the open source software development process?

3. How do properties of software (license - and technical properties) affect open source software as an economic good? And how does the type of good affect behaviour?

4. What are the externalities of open source software? What is the source of these externalities? And how do these externalities affect behaviour?

The answer to the latter two research questions (RQ3 og RQ4) will be combined when developing a qualitative model of open source software development and consumption. The model will be tested on a number of mini-cases collected during in-depth interviews with open source software developers.

4 Theory

This chapter begins by presenting basic economic goods and explains how different properties in consumption define the type of good. The different types of goods have certain properties and these are explained.

When an agent takes an action that affects others, and the agent does not incur the full cost or receive the full benefit of his action, there is an external effect – an externality. Externalities are introduced, and different types of externalities are presented. Unlike economic goods, externalities occur in both production and consumption.

Economic goods and externalities are closely related, and some types of goods are said to be special instances of externalities. It does, however, seem purposeful to keep the two separate, as the theories are to be used independently.

When externalities are discussed, they are always followed by a discussion of their implication on public policy. Open source software has yet to enter the realm of public policy, although there are signs that it will happen in the near future. The present thesis is not concerned with public policy implications on open source software, and they will not be treated here.

The last theoretical contribution to be presented and also used in this thesis is the theory of competing technologies. This theory also builds on externalities, but it also focuses on competition between technologies, when adoption of the technology is subject to either increasing, constant, or diminishing returns to adoption. For goods with networking properties like operating systems this is a very interesting theoretical perspective.

4.1 Economic Goods

Open source software is a good in the sense that it can be exchanged between agents and used for different purposes considered valuable to the agent. The general term, open source software, encompasses many individual programs performing different functions and should be considered different goods. The following analysis of open source software as an economic good is not specific to a particular type of open source software.

The framework of economic goods is interesting for understanding, how a good behaves in an exchange economy. An exchange economy does not have to be based on monetary exchange; other goods may be part of an exchange like a pure goods exchange, one good in exchange for another good like a barter economy.

The theory of goods provides a basic framework for analysing open source software. The theory of goods is directed at understanding, how the properties of a specific good influence its exchange.

The structure of this section is deductive in the sense that it begins with a definition of a few simple properties related to the exchange of goods. The simple definition is used to understand goods in general and their exchange.

It is assumed that goods are produced and exchanged on a voluntary basis, and thus no agent is performing against his own will. In the simplest form goods are privately produced and privately consumed. The price of the good determines what is consumed,

produced and in which quantities. The market is the place, where goods are exchanged and is considered a very efficient allocation mechanism. However, the market place is only efficient, when the goods are private, and ownership of the good can be transferred. In fact a key condition for a market transaction is that the ownership of a good can be transferred or denied, depending on context [Kaul et al. 2000:3]. If this is not possible, then the market, as a mechanism that determines the price of goods ceases to be an efficient mechanism for allocation. In this situation other mechanisms begin to influence on how goods are allocated and produced.

Goods are described by two properties: 1) Rivalry in consumption and 2) Excludability [Kaul et al. 2000:3].

4.1.1 Rivalry in Consumption

When one person's consumption of a good makes it impossible for another person to consume the same good, rivalry in consumption exists. The classic example is about a bakery and a loaf of bread. If one person buys a loaf of bread and consumes it, this particular loaf of bread is then consumed and cannot be enjoyed by others, hence the utility of others are zero. Rivalry in consumption has two extremes. In one end of the scale is the just mentioned rivalry in consumption. In the other end of the scale is non-rivalry in consumption. Non-rivalry exists, when one individual can consume the good without detracting, in the slightest, from the consumption opportunities still available [Cornes & Sandler 1996:8,145]. Looking at flowers is an example of a good, which has the property of non-rivalry in consumption. People looking at the same flower and thereby consuming the good, do not diminish the value of the good, and thus others may also consume the good.

4.1.2 Excludability from Consumption

Excludability refers to the ability to exclude people from consuming a particular good. In one end of the scale are goods with properties of excludability. As such it is possible to exclude people from consuming the good. The loaf of bread, again, provides a good example. The baker can easily prevent people from buying and consuming his bread, and thereby exclude potential consumers. This requires that the precondition of voluntary exchange is met. When a customer enters a bakery, the baker transfers ownership of the good at a given price. At the other end of the scale are goods, where excludability is not possible, or possible at prohibitive cost. Broadcast radio is an example of a good that is non-excludable. The radio station transmits radio signals, and has no means of excluding users from receiving their radio broadcast. Surely, the radio company could hire an army of inspectors to check, if people listened to their radio station, and collect fees. This would, however, be very expensive, the cost would exceed the potential profit, and exclusion would be done at prohibitive cost. The scale from excludability to non-excludability is closely associated with the cost of excluding. Thus excludability exists, when it is economically possible to exclude people from consuming a good. For example, a large natural reserve with lots of rare wildlife is interesting to visit, and visiting the reserve can be considered a good. Since we assume that this reserve is large, very large, it is not economically feasible to erect a high fence around the reserve, and make people pay an entrance fee to enter the reserve and enjoy the good. But, if our reserve was the last place on the planet with wildlife, it might indeed be feasible to erect the fence and charge a high entrance fee. Thus, excludability is tied to the economic consequences of excluding people from consuming a good.

Excludability and rivalry in consumption with their extremes and combinations can be illustrated in a 2x2 matrix, see Table 1. The four combinations make up a matrix, which shows four different types of goods.

	Rivalrous	NonRivalrous
Excludable	Private good (Loaf of bread)	Club good (Cable TV)
Nonexcludable	Commons (Fish in the ocean)	Pure public good (The ozone layer)

Table 1: The four types of goods.

4.1.3 Private Goods

Starting in the top left corner of the matrix goods of this type exhibit rivalry in consumption, and exclusion is possible without notable costs. Simple consumer goods like apples and oranges are examples of private goods, where exclusion is possible and economically feasible. Producers can then exclude consumers from enjoying the good, if they do not wish to pay the price demanded by the producer. The ability to exclude consumers relies on laws of private property, i.e. society by law recognises that people have ownership of their creations. This ownership could in turn be transferred at a certain price – often determined by the market.

4.1.4 Commons

Commons are characterised by non-excludability and rivalry in consumption. The name commons refer to the natural resources, which nature has provided, and man has been using. Commons have historically been a free good, owned by no one - like fish in the sea. No one owns the fish, until they are caught, and only then can ownership be transferred and profit made. However, once caught this fish is not available to others, and thus commons exhibit rivalry in consumption. Non-excludability arises from the fact that commons are all around us like the air, we breathe, or the ozone layer. Excluding people from consuming these goods would be impossible or at best extremely expensive, and thus not feasible.

4.1.5 Club Goods

The club good is characterised by excludability and non-rivalry in consumption. It is easy and economically feasible to exclude people from consuming a good, and one persons' use of a good does not reduce the value of the good for others. Cable TV is an example of a club good; one person's consumption of cable TV does not affect other people's ability to consume the same good, and neither does one person's use prohibit the

use of others. However, excludability is easy, and the producer can charge a fee for consuming the good.

In general commons and club goods are referred to as impure public goods as they posses part of the properties of the pure public good of non-rivalry in consumption.

4.1.6 Pure Public Goods

Last is the pure public good, which is non-rival in consumption, and where exclusion of consumers is not possible. The ozone layer is the perfect example of a pure public good, where every living being on the planet is consuming the good provided by the ozone layer. The good provided is the protection from the sun's dangerous ultraviolet rays. It is impossible to exclude a single person or a certain group of people from consuming the benefit of the ozone layer, and one person's consumption does not reduce the amount of the good available to other people.

4.1.7 The Type of Good can be Influenced by Use

The four types of goods exist only, when certain assumptions are met. A public freeway could initially be viewed as a pure public good. The freeway is public, and thus no one can be excluded from enjoying the good. Of course a car is required, and the public is limited to drivers travelling on the freeway. It is assumed that the utility of consuming the good is equal to the average speed when travelling on the freeway. When only one person consumes the freeway good, his average speed is equal to the speed limit of the law, or higher, and thus his utility is as high as can be. This is true, also when many others are consuming the freeway good at the same time.

But, when the number of cars on the freeway rises to a certain level, the average speed on the freeway drops. When this happens, the capacity level of the freeway has been reached. When the number of users exceeds the capacity limit, the freeway is no longer a public good, since the consumption of the good is no longer non-rival. One extra consumer of the freeway good results in a lower utility for the other users.

From this demonstration it is apparent that the properties of a good may change, depending on whether the total consumption is within capacity limit or not. Often a good exhibits changing properties and thus become a different type of good, when a capacity limit is reached.

4.1.8 Provision of Goods

Private goods can, as mentioned in section 4.1.3, be owned, and ownership may be transferred. When consuming or producing a private good, all costs are incurred by the individual, and the cost is equal to the cost incurred by society (assuming the goods does not exhibit externalities). Private goods are allocated through a market, where the price of a good reflects most of the properties of the good. Likewise club goods are allocated through a market, as consumers can be excluded, but unlike the private good, the club good is non-rival in consumption, and this has an impact on the producers of the good. Because producers of club-goods do not incur a cost when producing an extra unit of the good, there are huge benefits to be gained from this type of good. When unit costs per consumer are falling with each additional consumer, club goods are the source of natural monopolies.

Goods from which exclusion is not possible pose other problems to the allocation of the good. Obviously, if a good is free, and people cannot be excluded from consuming the good, the good is in high demand and in danger of depletion, if it exhibits rivalry in

consumption. Public goods like fish in the sea have within the past 20 years come under heavy pressure from over fishing. This has resulted in steps being taken by the political establishment to prevent depletion of this important natural resource. It is a vicious circle, where increasing scarcity results in higher prices allowing fishermen to make investments in more sophisticated equipment to scoop up the remaining few fish. Of course this is not an acceptable outcome, and it shows non-excludable goods to have provision problems.

Pure and impure public goods result in what is known as externalities (discussed in detail in section 4.2. Externalities arise, when a person or a company performs an action but do not incur the true cost. Pollution is the classic example, where a company pollutes a river, thereby saving the cost of cleaning its wastewater. In turn pollution reduces the value of the river by killing a percentage of the wildlife, and thereby reducing the nearby fishermen's income. In this example the fishermen incur a cost, which is related to the action of the company, and this is called a negative externality. Pure public goods are, as noted, non-rival in consumption and non-excludable, which makes them susceptible to overuse and supply problems. If a good does not have a cost, and people cannot be kept from using the good, it will be used in large amounts, sometimes resulting in depletion or extinction. If a producer is not able to profit from his efforts, the incentive to produce is likely to be very low, since the producer will incur a cost but no benefit. As noted by Stiglitz:

> "The central public policy implication of public goods is that the state must play some role in the provision of such good; otherwise they will be undersupplied"
> [Stiglitz 1999:311]

If a public good appears to be produced in sufficient quantity, we have a phenomenon that should be studied in details. Open source software is one such good, which appears to be a public good produced in sufficient quantity. Open source software will be analysed using the theory of economic goods in chapter 6.

4.2 Externalities

Externalities is a broad and diverse topic in economics, and it has been discussed since the days of Adam Smith and probably even earlier. In the previous section public goods were discussed, and it was seen that they would often be underprovided. Adam Smith saw the justice system and enforcement of laws as public goods that should be provided, if society were to function properly [Cornes & Sandler 1996:3]. However, it was also recognised that these goods could not be provided privately, as the provider would bare all the costs and receive but a fraction of the benefits. It was believed that it would be the duty of national states to provide such goods.

Whenever externalities are analysed and treated in the literature, the purpose is to determine whether Pareto equilibrium exists, the potential for market failure and possible policy implications. In this treatment it is not the purpose to determine Pareto equilibrium, market failure, and possible policy implications. The focus in this section is to describe different forms of externalities and their implications.

Economic theory of externalities and public goods will be reviewed in the following chapter with the intent of developing a model of open source software development and consumption, which provide theoretical answers to the research questions posed in the previous chapter. The previous section on economic goods was focused on how properties of goods, when consumed, affected the type of good and thus the incentives to provide

goods. This sections looks at externalities, which arise from production and consumption (development and use) of goods.

This section on externalities begins by defining them, and the main part of the next section is a description of different types of externalities. Lastly, the chapter describes externalities and public goods in combination with different institutional frameworks as incentive structures.

Most books and articles on externalities use a strict formal mathematical notation to describe externalities. This treatment will go a long way to use a qualitative language to present and describe externalities. However, on occasion it is necessary to use the mathematical notation in order to maintain precision, but the mathematics are kept at a minimum. Generally, the utility function is used to illustrate which variables are present in a utility function. Lastly, it must be noted that this section on externalities are highly inspired by Cornes & Sandler, 1996, "The Theory of Externalities, Public Goods and Club Goods".

4.2.1 Defining Externalities

Definitions of externalities are numerous and have been a source of controversy, but nonetheless a definition must be made, and it will serve to focus the understanding of externalities in this thesis. Stiglitz defines externalities as:

> "Externalities arise whenever an individual or firm undertakes an action that has an effect on another individual or firm, for which the latter does not pay or is not paid" [Stiglitz 2000:80].

There is an implicit understanding in the above definition that externalities are generated in a market setting. It is by no means certain that a market exists, and that it is the market, which facilitates externalities. A more general definition, proposed by Meade and adopted by Cornes & Sandler, is:

> "An external economy (diseconomy) is an event which confers an appreciable benefit (inflicts an appreciable damage) on some person or persons who were not fully consenting parties in reaching the decision or decisions which led directly or indirectly to the event in question" [Meade in Cornes & Sandler 1996:39]

The latter definition is somewhat broader than the former, and it does not define the institutional setting and therefore externalities can happen in absents of markets. For the purpose of this thesis the latter definition is better, as the institutional setting of development and exchange of open source software is not well defined. Open source software is exchanged under a number of institutional settings, some market-like and other resembling free exchange e.g. free download from the Internet.

A precondition for selling and, indeed, excluding people from enjoying a good is property rights. A popular but wrong belief is that property rights and open source software were incompatible, and that open source software was in the public domain. Nothing could be more wrong. Property right and in particular copyright is the cornerstone of open source software licensing. Copyright ensures that authors of open source software may have copyright for their work, and thereby authors are able to license their software under any condition they choose.

However, property rights for open source software are not as straightforwardly understood, as it may seem. From society's point of view, the intention of property rights is to make it possible to set a price for a good, and exclude people who do not wish to pay the price demanded. Copyright is a way of imposing rivalry in consumption and excludability on goods, which does not possess these properties. Copyright creates a temporary monopoly for authors to exploit, thereby providing an incentive for authors to develop and publish intellectual goods.

Open source software turns this intention upside down by using copyright and open source software licenses to exclude people from enjoying exclusive rights for the software. There are many different open source software licenses, and they have different implications for users and developers. Naturally the author may choose a different license and enjoy the exclusive rights, but then again this would not be open source software.

In this perspective it is the license of open source software, which governs the rules of exchange, and which can be said to be one defining factor of the institutional framework that governs exchange and development of open source software As will be shown in chapter 5 there is a variety of open source software licenses, and they all have their differences that influence exchange.

The purpose of this section is to prepare for a later analysis of properties of software, which are believed to be one of the two elements determining behaviour.

4.2.2 Describing Externalities

The previous definitions of externalities do not provide a good description of what externalities do, and how they work. Externalities are pervasive but indeed ambiguous if not discussed in contest of a something tangible. The most obvious example is pollution of natural resources.

Suppose a firm is producing a commodity, which requires use of toxic agents, and these agents are released into the environment upon production. When producing the commodity, the firm pollutes the environment and makes the surrounding rivers toxic and unfit for fishing. The value of the rivers for people using the rivers for fishing and recreation has been diminished as a consequence of pollution. The firm on the other hand is able to produce its product at a much lower price at the expense of the users of the river. In this case the firm is generating a negative externality from production. ´

The firm may also choose to educate their low skilled female workers with the purpose of increasing productivity. However, education of low skilled female workers has a positive effect on child survival. The company does not directly enjoy a benefit from reduced child mortality. Education does induce a positive externality from which society and the women in question enjoy a benefit.

When discussing externalities it is useful to distinguish between two types of goods. The first type are goods that are marketed, where individual consumers or producers may choose to consume or produce amounts only constrained by their budgets or production functions [Cornes & Sandler 1996:51]. The other type is goods that are not available on a market or environmental goods, which are supplied in quantities exogenous to the agents. Goods not available on the market and environmental goods are distinguished from marketed goods because the consumer's choice does not affect price or quantity available. Each consumer tries to maximise a utility function containing a number of marketed goods and a number of environmental goods.

This gives the following utility function for a consumer:

Eq. 1 $$U^h = U^h(y_1^h,...,y_m^h;e_1^h,...,e_n^h)$$

The first utility function illustrates utility as a function of two types of goods: 1) Marketed commodities *y* and 2) Environmental goods *e*. It is seen that the indirect utility does not place a price for the environmental good *e,* whose quantity is exogenously determined. The utility function is constrained by a budget constraint on the marketed commodities *y*, which naturally has a price. The budget constraint can be expressed as:

Eq. 2 $$I^h = p_y y^h$$

In the following pages various cases of externalities will be discussed. The purpose is to understand, what is the source of externalities, and how these externalities influence consumption and provision. This will be discussed in two situations, one where the good in question is marketed and subject to constraints in budgets and production functions. The other good is one labelled as environmental, where the quantities supplied are determined exogenous to the agents.

4.2.3 General Externality

In this situation a consumer, *h*, consumes two marketed goods *y* and *z*, and the consumer is constrained by a budget *I.* Other consumers, whose consumption affects the utility of consumer h, also consume the good z. This gives the following utility function for consumer *h*:

Eq. 3 $$U^h = U^h(y^h,z^h;z^1,z^2,...z^{h-1},z^{h+1},...,z^H)$$

Consumption performed by other individuals is considered environmental goods, whose quantities are exogenous to consumer *h*. The utility function illustrates that consumer *h* consumes two goods, *y* and *z,* and is able to choose the quantities to consume only constrained by a budget. The externality enters the utility function as a consumption of good *z* conducted by all other users. This consumption affects the utility of consumer *h*, but he is not in a position to influence their consumption.

This is the general externality, where one agent's consumption of a good affects the utility of other agents consuming the same good.

4.2.4 The Standard Pure Public Good

Public goods are subject to non-rivalry and non-excludability in consumption - no agent can be excluded from enjoying the benefits, and the benefits received by one agent do not detract from other agents' utility. The standard pure public good is a marketed commodity supplied by agents, who individually choose how much to contribute to providing the good. Determining the quantity provided is a simple case of summarising the contributions made be each agent $Z = (z^1 + z^2 +,...,+z^H)$. In this situation provision is anonymous, it does not matter whether an agent provides 1 or 1000 units of the good. As the good is non-rival in consumption, the full quantity provided can be made available to all agents, and it is assumed that each agent consumes the total good produced. The regular user's utility function becomes:

Eq. 4 $$U^h = U^h(y^h,Z)$$

In this situation agents receive a full share regardless of their contribution to providing the good. Depending on situation the contribution z^h can be viewed as consumer z's subscription or contribution to supplying the pure public good.

This leaves little incentive for agents to contribute to the provision of the good. The agents contributing to provision of the good will observe that non-contributing agents enjoy full benefit from participating. The only incentive to participate is one, where the providing agents enjoy a net benefit from providing the good (net benefit= benefit – cost). A large shipowner may provide a lighthouse to the benefit of all shipowners in the area. The large shipowner faces a huge potential cost from not having the lighthouse, thus providing an incentive to provide the lighthouse. This can be illustrated in the following utility function:

Eq. 5 $$U^h = U^h(y^h, z^h + \widetilde{Z}^h), \text{ where } \widetilde{Z}^h = Z - z^h$$

Again this is subject to a budget constraint. This utility function illustrates the problem of the consumer: How much of y^h should be consumed, and how much of y^h should be contributed in order to consume the optimal quantity of *Z*.

This type of problem generally has no Pareto optimal equilibrium, and it has stimulated a discussion of alternative allocation mechanisms and possible policy actions to ensure supply.

4.2.5 The General Public Good

The general public good is different, because unlike the standard pure public good no anonymity and contribution from individual agents matter. In the standard pure public good example we saw that it did not matter, who supplied the good and in what quantities, once supplied by an anonymous agent, the full quantity was available to everybody.

Consider the hypothetical example where farmers live on a flat island surrounded by a wall protecting the farmers from folding [Hiershlifer 1983 cited in Cornes & Sandler 1996:54]. Each farmer on our flat, round island owns a wedge-shaped piece of land, and the connecting piece of wall. Each framer is responsible for maintaining his part of the wall.

In the unfortunate event of storm and flooding that causes the wall to breach, all farmers will suffer a cost, as the circular land is flat, and all farmland will be flooded. In this situation the security of all farmers depend on the quality of the least well-maintained part of the wall. It does not matter, if all the other farmers maintain perfect walls, if just one farmer lets his part of the wall decay.

The public good, defined as security or protection against flooding, is determined by the minimum of all contributions – the poorest maintained part of the wall determines the level of the public good. Formally this is described as:

Eq. 6 $$Z \equiv Min\{z^1, z^2, ..., z^h, ..., z^H\}$$

The total amount of the public good Z is defined by the lowest value of the individually provided parts of public good of flood protection.

In the same line of reasoning we find a situation, where it is not the minimum contribution, which determines level of security, but rather the contribution from best individual. Imagine a city surrounded by defensive gun emplacements, protecting the city from incoming raids by bombing airplanes. In this situation it is the accuracy of the best

gun, which determines the level of the public good of protection against bombing airplanes. If just one gun aims with impeccable accuracy, the other guns would not be needed, and the average accuracy of these guns would not matter. This is formally described as:

Eq. 7 $$Z \equiv Max\{z^1, z^2, ..., z^h, ..., z^H\}$$

4.2.6 Price-excludable Public Good

A pure public good is characterised by non-rivalry in consumption, consumption by one individual does not detract from the amount of the good available to others. Pure public goods are also non-excludable in consumption, and it is not possible or costly at prohibitive prices to exclude people from enjoying the good.

If this is true, the only incentive, which individuals have to provide such a good, is to satisfy personal need. If others enjoy the benefit of this good, then it is only a positive accidental side effect [Cornes & Sandler 1996:55]. Should it for some reason become possible to exclude firms and individuals, who have no interest in consumption, but has an incentive to provide the good. The utility function becomes:

Eq. 8 $$U^h = U^h(y^h, Z^H)$$

The budget constraint *I* is interesting, as it reflects the price of excludable public good *Z*, meaning that a consumer must pay a price for all of the supplied goods, not just the fraction z^h supplied by the consumer:

Eq. 9 $$I^h = p_y y^h + p_z Z^h$$

Exclusion provides an incentive to provide a good and is prerequisite for operating a market. It is well known that efficiency problems are associated with public goods, and it is interesting to analyse, whether excludability is sufficient to restore efficient market provision. As these goods are non-rival in consumption, there seem to be a huge incentive to provide them, if it is possible to exclude people from consuming the good.

Various writers have investigated this problem, and no general conclusion can be presented. Some argue over-provision, and others under-provision, but all agree that Pareto efficiency is not guaranteed.

4.2.7 Impure Public Good or Bad

In this situation an individual commodity appears two times in the consumers' utility function, once on its own and once in combination with the quantities consumed by others. An example of this is driving, which is a private good, and however, driving also contributes to pollution and congestion, which is public bad.

The utility function can be described as:

Eq. 10 $$U^h = U^h(y^h, z^h, z^h + \widetilde{Z}^h), \text{ where } \widetilde{Z}^h = Z - z^h$$

y^h is consumer h's consumption of product y, z^h is consumer h's consumption of product z, and $\widetilde{Z}^h$ is the amount of z consumed by all other consumers. Each consumer must maximise his utility function subject to an income constraint:

Eq. 11 $$I^h = p_y y^h + p_z z^h$$

The impure public good or bad is interesting in the sense that the good consumed is non-excludable but the amount available is dependant of both my and other agents consumption. This model is often used to describe pollution, disease, and congestion problems. A person consuming a highway good is enjoying benefits but also leaving less of the good available to other consumers, which is also the case for all other consumers.

4.2.8 Open Access Resource

Goods like fish in the ocean, oil wells, and common grazing land can be accessed by anyone, and can be thought of as open access resources. The activities of each individual affect negatively on the amount of the good available to others i.e. a negative externality. Each agent has free access and free use of the resources, and they only pay costs directly associated with hired inputs such as man-hours.

Open access resources raise inter temporal issues, as the negative externalities affect the potential income of other agents. If agents decide to fish a lot of the available fish today, there will be less fish tomorrow. This has two implications: 1) Breeding stocks will be diminished, and 2) the potential income will be less tomorrow. In this scenario depletion is a likely outcome, as agents have an incentive to maximize their income by fishing everything today leaving nothing for others.

4.2.9 Common Property

In this situation a productive resource is made freely available for exploitation to the public. Under certain conditions this property may be exploited to a level, which imposes severe social costs. Grazing land may become desert, wells may be depleted, and whales hunted to extinction.

There are three general regimes of resource exploitation [Cornes & Sandler 1996:60]: 1) The open access regime, where there are no property rights, 2) The market regime where rights to exploitation are given to individuals or groups, and where users can pay a price for exploiting the resource, and 3) The agency regime where a central agency allocates rights to exploitation. All of these regimes have their problems. For instance, overexploitation of certain species of fish can be attributed to a, in hindsight, failed attempt to maintain an open access regime for fishing. In recognition of this fact, a quota system has been established, allocating fishing quotas between fishermen based on estimations made by biologists. Privatising resources may also have severe negative distributional effects, when a monopolist is able to control a market. There are many examples where it is the best solution to pass responsibility to an agency. Without proper control mechanisms or other mechanisms to intervene in agency decisions, corruption may be allowed to flourish.

Although these three forms of regimes represent the general forms, they do not present the complete list of possible institutions to govern resource exploitation. There are institutions, which seem to be able to harness the potential negative effects of exploiting open access resources. In many occasions communities have developed institutions, which have placed constraints on individual's consumption and exploitation of common property resources. The resulting regime has been a mix between the three, involving neither wholly open access to all comers to an unpriced resource, nor the assignment of exclusive rights to members of a certain community [Cornes & Sandler 1996:60]. These systems restrict entry to some degree and, often impose restrictions on the input levels of individual members. This also has implications on the total output to be divided between members. For instance some fishermen might have a strong tradition to divide the catch of the day between

members equally. The utility of the individual fisherman then becomes a function of the total production and his own production. The share of the open access resource is in turn equal to the total production of all individuals divided by the number of individuals.

By sharing the catch of the day the incentive structure is completely altered. The amount of time, which one fisher spends on fishing, becomes a positive externality for the other fishermen. Without the sharing rule, more time spent on fishing would have been a negative externality, where less fish would have been available to the other fishermen.

The scheme limits the amount of people, who have access to the resource, and modify the output sharing rules, but they also often limit the level of input that a member can apply. Such a system has been in effect in the Swiss Alps, where farmers had agreed to a rule, where farmers could send their cattle up to the grazing grounds in the summer. However, no farmer was allowed to send more cattle than he himself could feed during the winter. This system has been backed with fines, and was initiated as far back as in 1517.

4.2.10 Club Goods

So far the size of the community consuming a good has been exogenously given in the sense that excludability is not present. With club goods it is possible to exclude agents from consuming the good and the agents who control the club good will determine which agents are to access the good.

. Club goods present a general situation of public goods, where the size of the community is endogenous. A golf course is one such example, where members must pay a fee in order to use the facilities. Naturally this is no longer a pure public good, as there are effective means of excluding people from enjoying the benefits supplied by the club. Each individual, who decides to join the club and become a member, generates a benefit for the other members, as the total costs of supplying the good are distributed evenly across the members, given a fixed size of the good supplied by the club.

Models of club goods share a common feature in describing varying degrees of congestion, which influence the desirable size of the club. The utility of a swimming pool or golf course is greatly diminished, as the number of concurrent users reaches a certain limit determined by the good. In a simple form every single member is faced with a maximization problem where he must decide how much to consume, how many members the club should have and the size of the club good. Members can choose to enlarge the club thus allowing for a greater amount to consume. Naturally a better club is more expensive and users must provide a greater share. The number of members can be increased to provide more income for the club at the expense of the amount available to the other members.

where y^h is individual h's consumption of the club good, X is the club good, and s is the size of the member base. An increase in y^h and X results in an increased cost to all members, distributed evenly, and an increase in s lowers the cost per member, as costs are distributed to all members. *F* is budget or technological constraint like the size of a swimming pool. *F* has to equal zero, as the club provides the good *X* at the cost of the sum of the member fees, as a club it is not profit maximising body. Mathematically this argument can be expressed in the following equation:

Eq. 12 $MaxU^h(y^h, X, s)$, subject to $F^h(y^h, X, s) = 0$

4.2.11 Externalities and Public Goods as Incentive Structures

Particular goods and services are often seen as having particular externalities or characteristics of public goods as a consequence of their very nature. Lighthouses and defence, for instance, are often hailed as being examples of public goods.

Perceiving certain goods and services as intrinsically public may be a good start for understanding and analysing goods and services. However, one must not take lightly on the task of labelling goods and services as having particular characteristics [Cornes & Sandler 1996:63]. It was noted in the discussion of open access resources that a simple agreement between members could shift the incentive structure, and turn a negative externality into a positive externality for the members of the community. A society or community can choose to supply some goods and services in ways, which might create the incentive structures associated with public goods. It is important that the institutional framework, used to deliver particular goods and services, is considered in depth. Given the particular situation, the pros and cons of the available institutional frameworks must be judged.

Instead of viewing certain goods and services as being synonymous with the previously described situations, it is interesting to view them as different incentive structures [Cornes & Sandler 1996:64]. The incentive structures affect the production and consumption of a particular good or service. The incentive structures are determined not only by technological properties, but also by the institutional framework, which society or a community has chosen. The institutional framework need not be a result of a single decision, but may have evolved over long periods of time. Individual preference and the distribution of individual preference in society also influence the institutional framework.

Institutional frameworks are pervasive and go past the examples of open access resources presented here, such as fishing, oil fields, and grazing land. The following example will illustrate the importance of the institutional framework. A group of H individuals, we shall refer to them as a community, has decided to build something together, thus supplying their labour to the collective enterprise. The community has agreed to a sharing rule, and each member will receive a fraction of 1/H of the total output. It is also agreed that the members are allowed to choose his own labour supply, however, the labour supply of the rest of the community is exogenous to the individual.

For a community the production function is the sum of the members' individual labour contribution in producing the common good. The marginal product is positive but declining, illustrating a diminishing return to adding labour. Our little tightly nit community shares same preference regarding leisure and same consumption of the commonly produced goods. Leisure is defined as the part of total time, T, not spent working, hence $x + l = T$.

Each member of our community maximizes his utility function, U, which is a function of consumption, y, and leisure, x, thus $U(x,y)$. We know from the community agreement that each member consumes his even fraction of total production:

Eq. 13 $$y = 1/H * F(L)$$

It was assumed that H was exogenously fixed, and thus it makes sense to drop H from the explicit argument, leaving us with:

Eq. 14 $$Max\{\hat{U}(l + \tilde{L}, x \mid x + l = T\}$$

This has the same incentive structure as a pure public good. The individual only supplies a small part of the labour and has no influence on the total amount of labour

supplied, and thus the incentive to spend a larger amount of the total time T on leisure is very large.

Changing the rules of how to distribute the commonly produced good can change the incentive structures, and alter the public good's characteristic of the commonly supplied good. Having observed, how many members of the community choose to supply only a small fraction of their time as labour for producing the common good, our community decides to change the rules. It is decided that the amount of the good, which should be available to each member, should be proportional to the amount of labour, which the member contributes to producing the common good. The new system makes sure that if the community member, Brian, supplies twice as much labour as Dennis, Brian will also receive twice as much as Dennis. Under this new regime the utility function of the individual is

Eq. 15 $$U(1/L^{*}F(L), x+1) \mid x+l=T)$$

This utility function has the same first argument as a price-taking firm using an open access resource. The second argument is different, and reflects leisure in this example, whereas it would reflect wages paid under the open access regime.

By simply changing the rules in the community, the utility function of member was changed, and the incentive structures likewise changed. This illustrates the importance of considering the institutional framework, which induces incentives.

4.3 Competing Technologies

Some technologies have properties, which make their value increase with the number of agents, who adopt the technology. As the agents adopt the technology, experience is gained and the technology is improved, or perhaps there are more agents to share media with. When technologies exhibiting increasing returns are competing for potential adopters, events - that might seem to have no bearing on the adoption process - may influence the outcome of the adoption process. By chance, one such insignificant event may provide one of the competing technologies with an advantage, which makes this technology become more adopted than its competitor. The technology, which gets a head start, will improve faster and thus have a greater appeal to the potential adopters. This completes the circle: More adopters -> more experience -> better product ->more adopters and so forth. The model presented here has been developed by W. Brian Arthur [1991].

Therefore a technology, which - maybe by chance - gains an early lead, will through increasing returns have a significant chance of 'cornering the market' of potential adopters. The result may be that other technologies never get a chance to prove their potential, before their compared value has diminished so much that no potential adopter will choose this technology. Various 'insignificant events' might provide a technology with sufficient adoptions and learning by using to dominate the market. These events could be the success of a prototype[16], many early developers, different technology, or a different political climate.

The aforementioned 'insignificant events' introduce random events into the model, and allows for them to affect the adoption process and thus the technology, which is

[16] Development of the very successful Mustang P-51 fighter aircraft took only 101 day from start of development to at prototype was constructed.

adopted. This makes it possible to analyse and understand, how 'insignificant events' affect the adoption process, and how a sequence of random events may cumulate. The cumulative process may influence the market share outcome in favour of one or the other technology. The introduction of random or insignificant events may also provide insight into two increasing returns properties, that is non-predictability and potential inefficiency are generated. Increasing returns enhance the effect of chance events in the adoption process, and may allow for an inferior technology to become the dominating technology.

4.3.1 The Simple Model

This model will only explain the simple case of two new technologies competing for adoption. The new technologies A and B are competing for the adoption of a number of potential economic agents, and the number of agents is unspecified but large. The technologies are not sponsored, and open for adoption by all potential agents. The economic agents are consumers, who develop the technologies by adoption and by use of the technologies. The development happens as learning by using, as discussed in the previous paragraph.

When an agent, *i,* arrives in the market at time t_i, the agent has a choice of two different technologies, A or B. We assume that agent R and agent S are presented with the latest version of technology A and B. After making a choice, the agent will use the technology in a period of time.

In the simple model, only two types of agents exist (R and S), and they have different preferences regarding the technologies, agent R prefers technology A, and agent S prefers technology B. The technology, which the agents choose, will remain the same after the moment of choice. Thus the agent's choice and expected payoff are only affected by the past choices of other agents, and not by expectations of the future choice of other agents.

Some technologies do not exhibit increasing returns to adoption, for example a congested highway where each additional user negatively affects the utility of the highway good as average transportation time is lowered. Other technologies are unaffected by the number of adoptions, in the sense that they exhibit constant returns to adoptions. It is assumed that the returns to choosing technology A or B for any agent is dependent on the number of agents n_A and n_B,, who before the time of choice have adopted the technology (see Table 2). The factors *r* and *s* reflect a returns factor dependent on the number of past adoptions. If *r* and *s* are equal to zero, the returns function is not affected by the number of adopters. A positive number indicate positive returns and so forth. a_R, a_S, b_R , and b_S indicate R and S type agents' natural preference for either of technology A or B, for instance b_R is agent R's preference for technology b. By defining $a_R > b_R$ and $a_S < b_S$, R agents will have a natural preference for technology A, and the S agents will have a natural preference for technology B.

	Technology A	Technology B
R-agent	$a_R + rn_A$	$b_R + rn_B$
S-agent	$a_S + sn_A$	$b_S + sn_B$

Table 2: Returns to agents choosing either technology.

Chance is defined as those small historical events that are not included in the ex-ante knowledge of an agent. It is outside the discerning power of the agent to understand the implications of the small historical events.

An agent has full insight into all the returns functions, but not into the events that determine the time of entry to the market and the time of other agents' choice. Assuming that r and s are zero, other agents' adoption does not affect returns from choosing technology A or B. The agent, who is about to make a choice, sees a sequence of choices, where agents choose technology A or B according to their natural preference. Assuming that the number of agents with natural preference for technology A or B is equal, the probability of the next agent choosing technology A is 0,5.

This is a simple model of allocation in a neoclassical setting, where two types of agents choose between the two available technologies, and the returns to adoption, r equals zero. In this case the agent will choose the preferred technology. This is a trivial situation where the market share outcome is determined by the distribution of agents preferring one or the other technology. However, the situation becomes interesting, when r and s are positive or negative, indicating positive or negative returns to adoption. In this situation the past adoptions matter, and the value of a technology to the agent will depend on the number of past adoptions.

It is interesting to understand whether these small historical events will affect the market share outcome in an adoption process. To aid the understanding, some properties relating to the adoption process must be introduced. The adoption process is:

- Predictable, if a small degree of uncertainty averages away, and the agent have enough information to determine the market share outcome.
- Flexible, if it is possible for a subsidy to influence the market share outcome.
- Ergodic, if different sequences of adoption lead to the same market share outcome.
- Path-efficient, if the adoption process at all times, when choosing the more adopted technology, provides a better pay-off than the competing technology; assuming that the competing technology would have been developed to the same level as the preferred technology.

The three types of returns, r, define three regimes, a constant (r=0), an increasing (r>0) and a diminishing (r<0). In this case, a sequence of choices with two types of agents makes choices in an unknown order.

4.3.2 Allocation

Let $n_A(n)$ and $n_B(n)$ be the number of choices of A and B respectively, and n is the total number of choices. After n choices have taken place, it is possible to describe the adoption process by x_n as the market share of technology A, after a total n choices has been made. The difference in adoption can be expressed as:

Eq. 16 $$d_n = n_A(n) - n_B(n)$$

The market share of A can be expressed as:

Eq. 17 $$x_n = 0.5 + d_n/2n$$

In this constant returns, case R-agents will follow their natural preferences and choose technology A, and S-agents will likewise choose B. This is regardless the number of previous adoptions, since r=0 (refer to Table 2). Here the market share is determined by the way R and S agents line up before choice, that is, the way the choices cumulate. If an R agent is next in line and choose technology A $n_A(n)$ is increased by one unit.

In the increasing returns regime, r>0 and will affect the adoption process tremendously, and the simplicity is spoiled. Like before, R-agents have a natural preference for technology A, and vice versa for S-agents. The difference now is that since r>0, the number of adoptions $n_A(n)$ or $n_B(n)$ of the two technologies will affect the payoff of new agents. Agents are rational and will choose the technology that provides the highest payoff. If enough R agents have chosen technology A (their natural preference), it might happen that the value of technology A becomes so great that also S-agents will choose this technology. This situation can be expressed as follows:

S-agents will switch to technology A, if

Eq. 18 $a_{S+} sn_A > b_{S+} sn_B$

Eq. 19 $sn_A - sn_B > b_S - a_S$

Eq. 20 $n_A - n_B > (b_S - a_S)/s$

Eq. 21 $d_n > (b_S - a_S)/s$ using Eq. 19 for substitution.

Likewise R-agents will switch to technology B, if:

Eq. 22 $d_n < (b_R - a_R)/r$

Thus, as soon as adoption shares of technology A or B reach the boundaries defined by Eq. 24 and Eq. 25, the adoption process will become locked-in. The payoff for the not-preferred technology at these boundaries surpasses that of the naturally preferred technology, and both types of agents will choose the one technology. The consequence is that agents will have a choice of technology in the period of time, defined by:

Eq. 23 $(b_S - a_S)/s > d_n < (b_R - a_R)/r$

This can be described as an adoption process with absorbing barriers, when the barriers have been crossed, both type of agents choose the same technology.

4.3.3 Properties of the Regimes

As noted, the constant, increasing and decreasing returns to adoption can be defined as the technologies adhered to different regimes. By defining the regimes, it is emphasised that there must be certain properties that differ in the different regimes.

4.3.3.1 Properties in the Constant Returns Rregime

Under constant returns r=0 and s=0, hereby reducing the equation in Table 2 to the natural preference. Thus the number of adoptions does not affect the payoff to either of the technologies. This provides a predictable long-term adoption share not affected by number of adoptions. The share will settle at the same distribution as the agents R and S.

In the constant returns regime, flexibility is not achieved; unless a subsidy has such a considerable size that that agent's natural preference is overturned by the subsidy.

Ergodicity is achieved under the assumptions that agents do not line up in a extreme order, that is a pure R-agent lines up, when both preferences are presented in the group of agents. The probability of agents lining up in a way, which does not reflect the distribution of preference among agents, is zero, when viewed over a period of time.

The constant returns regime is path-efficient, because the number of adoptions does not affect payoff. No matter the number of adoptions, the payoff remains the same, and agents choose the preferred technology.

4.3.3.2 Properties in the increasing returns regime

Under increasing returns the situation becomes more interesting. In this case, the variables r>0 and s>0 and this result in the number of adoptions affecting payoff. Given that the increasing returns regime will favour one of the competing technologies, the agents' prediction about the long-term market share of one technology will be either 0 or 100%. Both predictions are true with the probability of 0.5, and thus the predictability is lost.

When the adoption process has passed the point of lock-in, the given by $d_n > (b_S - a_S)/s$ (Eq. 24). The payoff to both types of agents will have switched, and even agents with a natural preference for the loosing technology will receive a greater payoff and choose the winning technology. In this case no subsidy or tax adjustment will result in a change in payoff, and thus flexibility is not present.

Ergodicity, the effect of different line-ups of agents is interesting in the increasing returns regime. If agents line up in a R,S,R,S,R,S,R,S,R,S,R,S,R,S-sequence, both technologies get equally adopted, presuming r=s this situation could continue for ever. However, in the event that r≠s, the payoff to the competing technologies will rise at different speeds, even though equally many agents adopt the technologies. If agents line up in a different manner than the above, it might be possible for the payoff of one of the technologies to reach the lock-in value. Depending on the line-up of agents both of the technologies may win the market, and thus the increasing returns regime is non-ergodic.

If the returns to technologies change over time, so that the value of the variables r and s are not constant, the adoption will not be path-efficient. This can be illustrated by a technology that develops slowly in the first period of time, and then as a result of a break through goes through a period of rapid development. In this period of development, the payoff to adoption becomes significantly higher than the competing technology, and the situation becomes non-path efficient.

4.3.3.3 Properties in the Diminishing Returns Regime

The adoption process in the diminishing returns regime is predictable due to the reflecting barriers. When a technology has been adopted a number of times, the payoff to adoption diminishes, and at a certain point agents will choose the not-preferred competing technology. The adoption process will be a walk between the reflecting barriers, and the long-term market share will reflect the distribution of the two types of agents.

Flexibility also holds true. The diminishing returns make the payoff to the preferred technology decrease, and for each adoption, the subsidy to make an agent choose differently than the preferred choice is reduced. Thus any amount of subsidy will change the choice of agents, and the size of the subsidy determines how soon the effect will be apparent.

Adoption in the diminishing returns regime is ergodic. Unlike the constant returns regime, agents lining up in extraordinary ways do not affect ergodicity. The diminishing returns ensure that no matter the sequence of adoptions, at some point in time the payoff has diminished and agents will choose other than their preference.

Also this regime is path-efficient, given the diminishing returns it is not possible for the market to adopt an inferior technology. The diminishing payoff to adoption ensures that both technologies will get adopted and developed.

The properties of the three regimes can be summarised in the following table:

	Predictable	Flexible	Ergodic	Path-efficient
Constant	Yes	No	Yes	Yes
Increasing	No	No	No	No
Diminishing	Yes	Yes	Yes	Yes

Table 3: Properties of the three regimes.

4.3.4 Extending the Model

The above model is limited to two technologies and agents, who have no rational expectation. The model can be extended in various directions such as unlimited number of technologies, uneven distribution of agents, and agents having rational expectations or even sponsored technologies.

Of interest is the case of agents with rational expectations, where the number of agents choosing the technology in the future affects the value of a technology. The battle between the video systems VHS and Betamax provides an instructive case.

The value of having a VCR is greatly dependent on the size of the associated network [Øst and Edwards 1996] having a Betamax VCR, when everybody else has VHS. It is not possible to share tapes with others, and the selection of titles in the local movie rental store is likely to be minimal, if not non-existent. Thus, the rationally expecting agent will choose the technology that he thinks will be the dominating one, and thereby ensure the investment.

The effect of rational expectations on the simple model is that the time to lock-in is shorter. The choice of one agent is interpreted by other agents, who will consider all past adoptions a to be a trend, and on this basis predict the long run market share outcome. This could have a dramatic influence on the adoption process, if for example the first 10 agents, by chance, are adopting technology A; agents waiting to adopt will interpret this a an all A adoption process. All agents will thus choose technology A, and the adoption process is locked-in after a very shot adoption process.

4.4 Summary

This chapter has presented three theoretical perspectives, which are to be used to answer the meta-question of why open source software is being developed. The first section presented simple economic theory of economic goods. Simple economic goods have the advantage of quickly being able to characterise the incentives for a produce or consume a good.

The theory of externalities and public goods goes much further and describe in detail the utility function of different types of public goods. This provides a more precise means of establishing the incentives for agents to supply the public good.

Lastly the theory of competing technologies was presented, which gives a description of how a technology's returns regime affect properties of adoption. It was observed that technologies with increasing returns to adoption favoured a single winner who would capture the market. Interestingly, the theory pointed out that the outcome of such a competition was unpredictable and random historical events could indeed change the outcome of an adoption process.

The theory presented in this chapter will be put to use in part II: Developing a model.

PART II: Developing a Model

The second part of this thesis marks a change in focus. So far attention has been devoted to existing literature, propose research problems, and describe the theory that are to be used as a basis for proposing.

When the research questions were developed it was proposed that incentives to develop open source software should be explained by two elements.

```
Properties of software + Type of agent=> A certain behaviour
(incentives and mechanisms)
```

Properties of software were defined as a combination of technical properties and license properties. Agents could be either individuals or firms. Properties of software are thought to be very important for understanding why open source software is being developed. For this reason a deeper understanding of open source software licenses must be obtained before developing a model.

This understanding is gained in the chapter 5 on software, property rights, and licenses. The chapter presents a brief overview of the history of property rights and international conventions. Further differences between national states is discussed as software is developed in one nation and further developed and used in another nation. We then proceed into a detailed but brief description of the object, which is protected by law. The main part of the chapter is devoted to analysing a selection of 10 software licenses.

Chapter 6 will review a selection of open source software licenses. Following the license review software will be analysed as simple economic goods and the chapter contain three sections and the first section analyses properties of software as technical and license properties. It must be noted that an analysis based on the properties of software should be based on a selection of three software licenses. A table comparing the previously reviewed licenses is presented and serves as a basis for choosing three distinct licenses. The following second section analyses the two types of agents, firms and individuals, and discusses their differences. The third and main section of the chapter analyses software distributed under the three chosen licenses from a perspective of simple economic goods.

Chapter 7 develops a qualitative model of open source software development, which uses the three chosen licenses as arguments for developing a model explaining agents' behaviour. The chapter begin by outlining the general principles of the mode and then proceed to analyse each of the three licenses from a perspective of economic externalities and standard pure public goods. The qualitative model is focused at the level of the individual software project i.e. at the level of projects like the Linux kernel or the Apache web server, which was mentioned in section 1.1 and 1.2 in the introduction. The model is only focused at explaining the mechanisms and incentives, which arise when agents engage in source software development.

Chapter 8 brings the analysis to a higher level than the individual software development projects. This is the level where the analysis recognises that more than one software development project may exist and that these projects are competing for adoption. Elevating to this level enables the analysis to analyse what mechanisms are governing adoption of the programs by agents.

5 Software, Property Rights and Licenses

This chapter begins with a brief history of property rights and the problems associated with immaterial goods. Next, the major international conventions that govern immaterial property rights are mentioned. Naturally there are differences in national laws, and this is briefly discussed in the section 5.4 "The Object of Protection". The main section in this chapter is an analysis of a selection of ten software licenses, of which all but one are open source software licenses. The criteria for labelling a license as open source have been the open source definition, which is presented and discussed in a separate section.

5.1 Historical Perspective

Ownership of property is a central issue in society, and has been so since ancient time. Most people do not question property rights of material goods, and associated property rights seem natural. However, ownership of property has been a central theme in politics, and so strong that governments have been overturned. The debate is ongoing, and the political discussions of ownership, in particular ownership of the means of production, have been raging the past few centuries. The socialist wing argues that ownership of the means of production should belong to the government, and in the opposite end we find the liberalists arguing about private ownership of everything.

Property rights of immaterial goods such as creations of art, inventions, etc. are on the other hand not as old as material property rights. The reason for this is probably that the potential for making money from these creations have until recent times been of no economic consequence. Regardless of potential income, the ability to create art, invent things or advance understanding has always been a source of respect. Skilled artists have through time gained fame and have obtained respect and recognition for their works. Here we probably find the reason for some of the early laws for protecting the rights of creators of immaterial goods. These laws were based on exclusive rights, which typically also meant that imitation of the protected work of art was prohibited [Koktvedgaard 1994:21]. Imitators and people otherwise trying to pass on the work of others as the work of their own, have always been frowned upon. The roman poet Marcus Valerius Martialis, who lived approx. 40-100 A.C. was strongly displeased, when others took credit for his poems – his spiritual children. He compared this theft with the crime of kidnapping, 'plagium' [Koktvedgaard 1994:22]. Plagiarism has later become a very negative term describing particularly rude examples, where other peoples' work of art is copied, and the name of the creator is suppressed. If, one the other hand, the creator is asked, and subsequently he allows others to use his works, joy fills him as this shows respect and appreciation.

Intellectual property rights became an issue with the advent of the printing press. This made mass production of books possible, and a considerable economic interest was suddenly attached to printing rights. In those days (~1450Ad.) any producer of immaterial goods had to ask the sovereign for special permission to his own production. If granted, this was done through special privilege, was an exclusive right, and was usually formulated through prohibition.

5.2 International Conventions

Free travel of immaterial goods meant a demand for laws that were applicable on an international level. No state was interested in a situation, where the citizens of one state could exploit the works of another nation's citizens without consequence. This prompted creation of international conventions, which would make sure that national laws were alike and would ensure international protection of intellectual property. It is, however, not the purpose of this thesis to enter into a discussion of the particulars of the various conventions, but the most important ones deserve to be mentioned.

The most important global conventions is the Convention de Berne (Berne Convention) dating back to 1886, last revised in 1971 in Paris, and the Universal Copyright Convention. The Berne Convention for the Protection of Literary and Artistic Works defines a minimal set of rules that member states must incorporate in their national law. Countries, which have ratified these conventions, are part of a union to the protection of intellectual property rights. One of the most important principles in the Berne Convention is the rule that no nation can make protection of immaterial rights dependant on any formal requirements, i.e. authors do not have to deposit copies or register their creation in order to be protected [Koktvedgaard 1994:37]. Currently 147 states are members of the Berne Convention. Interestingly enough, it was this very requirement that kept the United States of America from ratifying the Berne Convention for many years. The United States of America has historically relied on a system that requires creators of artwork to register in order to receive copyright protection, which is incompatible with the Berne Convention. As of the first of March 1989, USA has ratified the Berne Convention and is now a member of the Unions. The absence of USA as a member of the Berne Convention lead to the creation of the Universal Copyright Convention in 1952, which is administered by UNESCO. The Universal Copyright Convention, in contrast to the Berne Convention, requires the applicant to insert in his work of art the international copyright mark ©, followed by the name of the copyright holder and year of publication in order to enjoy protection.

The difference between the two conventions may seem inconsequential, but nonetheless it is important. The formal requirement for enjoying copyright protection assumes that creators of art are knowledgeable of the law, which is not always the case. One effect of formal requirements could be exploitation of artists not familiar with the requirements. In a sense, formal requirements discriminate the less knowledgeable. The Berne Convention treats all creators of art equally, and all shall enjoy protection without any need for formalities.

5.3 Differences Between States

Despite international conventions there are differences between national states. The following discussion assumes a situation where the rules of the Berne Convention apply. The main principles of the Berne Convention are the following:

> "Authors of literary and artistic works protected by this Convention shall have the exclusive right to authorizing the reproduction of these works in any manner or form" [Berne Convention, Article 9]

There are a few exceptions regarding fair use, teaching, etc. This is, however, a matter of the individual countries to judge these special cases. The convention ensures some minimal requirements for national law, and these must be met prior to ratification. It is a fundamental demand that citizens from different union states receive the same protection as the state's own citizens[17]. The member states are free to provide stronger protection than specified in the Berne Convention.

There are differences between the intellectual property laws of the member states, and this thesis will use Danish law on intellectual property as the basis. The states in the European Union are somewhat harmonised, and differ in their protection from the USA. Many of the developers of open source software are situated in both the European Union and the USA, and thus a quick glance at the differences is warranted.

The difference between the intellectual property rights in the European Union and USA is most profound in the patent aspect. The USA allows software patents, which the European Union does not. There are currently discussions whether to adopt the American view on software patents in Europe. At present it is unclear, whether patents will be adopted or not. There is strong opposition from proponents of open source software against adopting software patents. Proponents of open source software fear software patents, and argue that patent protection of software and algorithms are too strong and likely to impose severe negative effects on open source software development.

The patent issue is very interesting and holds vast implications for the future of open source software development. The object of protection and the actual protection are very different in copyright and patent law. So far open source software has relied on copyright to ensure intellectual property protection, and has tried to avoid developing software that would infringe software patents.

The danger of software patents is the strong protection that they offer. This is best illustrated with the case of the gif-files. The gif-file format is a particular way of storing an image on a computer. Having the gif-algorithm translate a picture into a Gif-picture and then store the gif-file, generates gif-files. Retrieving the image from file also requires use of the Gif-algorithm. This gif-algorithm is patented, and Gif-files are widely used for storing images in web pages. For a long period of time the patent-holder did not enforce the exclusive rights granted by the patent. In 1997 Unisys bought the patent and announced that users of gif-files had to pay royalty for using their algorithm. Unfortunately, it is not possible to create other ways of retrieving the images from the gif-files. Unlike other areas of science and production, where patents inspire development of alternative solutions, the gif-example illustrates that computer patents effectively block others from a particular use, in this case gif-files.

We now turn away from the patent discussion and focus on copyright, which is central to open source software development. The possible implications that patents might have for the conclusions of this thesis will be discussed in the discussion chapter (Chapter 11), but a separate analysis is beyond the scope of this thesis.

5.4 The Object of Protection

The classic definition of the object of copyright is literary work (a book) and artistic work (a painting). The definition, however, is very broad and determining, what should be covered by copyright has in itself become a complex issue. Literary works are typically

[17] Koktvedgaard:36

distinguished by the use of letters, but also works in Morse code and Braille (letters for blind persons) are covered. Works based on musical tones are referred to as music and covered by copyright.

Source code for computers is considered a literary work of art. This definition is fairly new and was included in a directive from the European Union in 1991 [Koktvedgaard 1996:56], and this is the principle used by most countries. Any work of art is protected by copyright, if the work is the consequence of the independent and personal effort of a person. The ideas, procedures, algorithms, etc. expressed in the work are not covered by copyright but by patent law. Thus, it is the actual expression of the work, or expression used in the work, that is a matter of copyright. This means that two programs solving the same program are not copyright infringing, if the implementation (the expression) is clearly different.

Copyright grants the author of a given work of art the exclusive right to [Koktvedgaard 1996:100]:

1) Manufacture examples of the work of art.

2) Make the work of art available to the public.

These exclusive rights may seem almost identical, when one thinks of software that can be reproduced and made available for the public, but the difference is perhaps best illustrated using a picture as an example:

The creator of a picture receives copyright for his work upon creation. The creator can then sell, make copies or place his picture on display in a gallery. When the creator sells the picture, the buyer receives the picture or a copy hereof, but not the actual copyright. The buyer may look at his picture and show it to friends, and even take a few photographs to show off, all within fair use. However, the buyer may not invite a group of people with the intention of allowing them to look at the picture, as this would constitute making the work available for the public.

In software terms this means that the author of any software upon creation has the exclusive right to copy and distribute the software. Copyright grants authors a strong set of rights, which the author may appropriate in any way he pleases. These rights may be sold to third party, and the author may decide how much of these rights, he wishes to sell.

The duration of copyright is the lifetime of the author plus 70 years – in the software world this is eternity. Computers have yet to exist for 70 years, and in contrast the Linux operating system just turned 10 years.

In the event that a company should employ a developer, the general situation is that the developer surrenders copyright of all creations to the employing company. The duration is still the lifetime of the developer plus 70 years.

5.5 Software Licenses

The legal instrument that a copyright holder uses for transfer of a specific set of rights to third party is called a license. A license is a permission from the copyright holder to a licensee to exploit a specific set of rights at given conditions. A license can be seen as a contractual agreement between copyright holder and licensee, in which the licensee agrees to certain terms. These terms and conditions are our actual interest, as they define what the licensee may and may not do etc. etc. In other words, a license defines how the licensee may use licensed software. The license thus provides a strong indication of the incentives for using, modifying and contributing to a program. For example if a license

outlaws modification of the source code, the licensee is not allowed to change the program, and further development is greatly discouraged if not outlawed.

In this thesis it serves our purpose to define two general categories of licenses: 1) closed source software licenses and 2) open source software licenses. The defining difference is, whether a license complies with The Open Source Definition [Version 1.9].

In the following, the Open Source Definition and several licenses that meet the requirements of the Open Source Definition will be reviewed and discussed, and a table of some of their differences will be presented. A broad selection of licenses have been chosen for review, and starting with the Open Source Definition, the following licenses will be reviewed: The GNU General Public License (GPL), the GNU Lesser General Public License (LGPL), Mozilla Public License (MPL), the BSD License, the Artistic License, The IBM Public License, the Apache Software License, The Free Beer License and the Microsoft End-User License Agreement. The Open Source Definition will not in itself be discussed, as it is the particular licenses that influence incentives to use and develop open source software. Even so, the review will begin with the Open Source Definition. It is the purpose of the following review to translate the legal language used in the licenses into something more readable, with a consistent use of terms. However, this will lessen the precision of the statements, and I believe this to be a wise trade-off.

5.5.1 A few Definitions

Before embarking on a discussion of selected licenses, a few definitions are in order. The use of words and terms in the licenses vary and this is compensated by using a common terminology for all licenses. This is a trade-off between precision on the one hand and readability, comparability on the other hand - this review emphasizes the latter.

A licensee is a person, who is granted a license to a program. The copyright holder is the person who holds copyright to a program or parts hereof. Often this is the original author of the program source code, and the author may surrender or sell his copyright, as he pleases.

Program and software are used synonymously, and both refer to a program or software that is to be run on a computer.

A derived work denotes a work (program) that is formed on the basis of a copy of the version, which the copyright holder released, referred to as the original work (original program). Suppose developer X receives program A from his friend developer Y. The program is a nice text editor, but it is not able to save the edited text. Developer X decides to extend the functionality of program A, so that it will save text in files for later retrieval. Developer X modifies program A to save files for later retrieval. This program, which we will call program B, is a derived work of program A.

The terms work and derived work are broader in scope than the term program, and refers to both the program, accompanying documentation and other material, usually distributed with the program. In this sense a derived work could be the original program redistributed with detailed instructions of how to use the program in a setting, completely different from what was intended.

A contributor is a person or entity that contributes code to the original program. The original program is the program, on which the program is based. A copy refers to a copy of an original program. When forming a derived work or otherwise modifying a program, the contributor is modifying a copy of the original. The program refers to the original program plus contributions. In this terminology the

Code refers to one or more lines of actual source code that are part of a program.

5.5.2 The Open Source Definition

The open source definition a set of criteria defined by the Open Source Initiative (OSI), with which a software license must comply in order to be OSI certified. OSI attempted to register “Open Source Software” as a trademark, but the term was far too ambiguous to be a trademark. The OSI certification mark is used instead.

The very term “Open Source” and, of course, Open Source Software was created by the Open Source Initiative. In 1998, when Netscape expressed a desire to release the source code for the Netscape browser, a brainstorm at Netscape coined the term “Open Source Software”, and shortly after this Bruce Parens and Eric S. Raymond launched www.opensource.org [opensource.org 1999]. The launch prompted a debate in the open source community about the term open source software versus the old free software term. The essence of the discussion was the nature of the relationship between the free/open source software community and the business community. Proponents of open source software felt that free software sent a wrong message to business managers that prevented them from adopting open source software. In particular the political undertones of the Free Software Foundation was thought to discourage adoption from the business community. Later that same year Netscape released the source code for their browser as open source software. Late 1998 the term open source software has been widely adopted by the press, and is the preferred term compared to free software. Although several different open source licenses existed e.g. the GNU Public License and the BSD license, Netscape decided to create their own license to meet their specific requirements. Since then many firms, which have entered into open source software development, have created open source licenses that met their specific demands.

The open source states nine points, which a software license should comply with:

1. Free Redistribution

Any licensees must be allowed to freely redistribute the program, and must be allowed to charge money for the services related to redistribution. Should the licensee wish to include the program in their own distribution, they are free to do so.

2. Source Code

The licensee must be allowed to redistribute the source code for the program, and the program must include source code. If, for some reason the source code is not included, there must a well-publicized way of obtaining the source code. The cost of obtaining the source code must be no more than reasonable costs, associated with reproduction.
The licensee can make money from packing the binary compiled version, perhaps by making the binary version easily accessible, but the source code must be available at a cost no higher than reproduction cost.

3. Derived Works

The licensee must be allowed to make a work based on the original. This makes it possible to develop modifications and extend the use of an existing program, and make it available under the same license as the original program.

4. Integrity of The Author's Source Code

The licensee may be prohibited from distributing derived works, only if the licensee is allowed to distribute patches to the original work. The licensee may further be required

to distribute derived works under a different name than the original. The license must explicitly permit distribution of software built from modified source code.

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 Endeavour
The licensee must not be prohibited from using the program in a particular setting or context. This has lately come under attack by the political establishment [Corbet 2002] in a case, where a group of developers created a violent racist computer game called Ethnic Cleansing. The game was built on open source software. Open source software proponents, however, feel that licensing is not the way to solve the problem.

7. Distribution of License
The rights granted in the license must apply to all, to whom the program is distributed. There must be no need for additional licenses to cover other users, to whom the program is distributed.

8. License Must Not Be Specific to a Product
Should the program be part of a distribution or packaged in other ways, then the license must not depend on the distribution or package. The program must be allowed to be extracted from the distribution or package and redistributed to others, who shall have the same rights, as granted in the original distribution or package.

9. The 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.

Compliance with the open source definition ensures that source code for a program is available and may be copied, distributed, and modified. If modifications lead to a derived work, this is also allowed, and the derived work may – but is not required to carry the same license. It is not possible for a license to discriminate against people, groups or fields of endeavour. The license follows the program, and any user who receives the program, is covered by the same license. The license allows the program to be part of a commercial distribution, and this does not influence the other software in the distribution and does not depend on the distribution.

5.5.3 The GNU General Public License

The GNU General Public License (GPL) [Version 2] was created by the Free Software Foundation in the 1980'ies, and version 2 used here is dated 1991. The GPL is a classic open source software license, however the GLP does not use the wording open source software, but rather Free Software. The GPL is widely used and perhaps the most used OSS license, for example the Red Hat Linux Distribution contains 50.4% code licensed under the GPL license [Wheeler 2001a:12]

The GNU GPL [Version 2:1] starts off with a fairly large preamble, which states the purpose of the license and goes to great lengths to explain, why free software is good. The preamble mentions, "The licenses for most software are designed to take away your freedom to share and change it [the software]". Lastly the preamble mentions software

patents as something that threatens free software and declares that patents in free software must be free to everyone, or not used at all. The preamble is written in a form that resembles a manifesto of open source software, and the strong wording has probably alienated many firms in the early days of open source software.

Going through the license, we will not repeat every word of the GPL here. However, one can turn to appendix 1.2 for a full copy of the license. The GPL starts off by defining how to distinguish, if a program is being covered by the GPL. Basically any software, in which the copyright holder mentions that the software is covered by the GPL, is covered, unless of course the program infringes other rights. The scope of the license is copying, distribution and modification - no more.

The GPL permits the licensee to copy and distribute verbatim copies of the source code, as he pleases. The only requirement is that the copyright notice is published on each copy. It is allowed to charge a premium for the actual physical act of copying the source code. This is interesting, as it outlaws charging for the actual source code, and makes sure that the source code is available.

The licensee is allowed to modify his copy of the source code as he pleases, and even form a work based on the copy. However, when doing so, the licensee must make a notice stating when and what has been changed. If the licensee decides to distribute or publish a derived work (his own modified version) of the program, i.e. it contains at least parts of the original program, the derived work must also be licensed as the program at no charge.

So, the licensee may charge for the physical act of transferring a copy of the source code, but the licensee may not charge for the license. This means that the licensee cannot charge others for a license that permits modifying, copying and distribution of a program. This applies to a derived work as a whole, and not just the part that was part of the original GPL'ed program. If a person includes a microscopic part of a GPL'ed program in his own very large program, the very large program automatically becomes GPL'ed. This is the so-called viral effect of the GPL license.

The viral effect of the GPL is quite controversial and has been widely discussed. Proponents argue that this ensures that commercial companies do not exploit GPL'ed software. On the other hand it is also a deterrent for people and companies that might find the software useful, but wish to retain all rights for their software.

The GPL allows people to distribute executable versions of the program, but there are certain restrictions to this. The complete source code for the program must follow, and this should be in a form that can be edited. Or the executable program must come with a written offer valid for 3 years, which gives third party access to the source code for no more than shipping and handling. Or the executable program must be distributed with the original offer of how to obtain the source code.

This ensures that one can make money from selling executable versions of the program, but seller/distributor must observe the three conditions. An executable version can have a significant element of added value, for example if the executable version is better documented and easier to install. The conditions ensure that the source code for the program is available at reasonable cost, and that the source code is machine-readable (it can be edited on a computer). This clause also ensures that the source code is not provided on paper. 100.000 lines of source to a program on paper would be useless, if one should wish to modify the program.

Should the program be redistributed, the receiver is automatically granted the rights in the original license. It is not allowed to alter the license or impose any restrictions on the

rights granted by the license. In other words – it is not possible to change the license, unless it is to a more recent version of the GPL.

Should the program for some reason infringe other rights (patent, copyright etc.) and thus violate the conditions in the GPL license, the program must not be distributed.

GPL'ed software comes with no warranty what so ever, and the copyright holder cannot be held liable for any damage (broadly speaking) caused by the program in any way. This last paragraph is natural, since the program is free, and no one can expect the copyright holder to offer any warranty – unless the copyright holder or distributor explicitly claims to do so.

5.5.4 The GNU Lesser General Public License

There are situations, where the GPL might prove more troublesome and hinder the wider adoption of free software, which is the purpose of the Free Software Foundation. For example, programmer X has written the 'Muzz' library, a set of functions and small programs that makes it easy to process music on a computer, and he wants this library to have the widest possible adoption. A program that uses the Muzz library is dependent on this library to the point, where the library is part of the program. If the Muzz library is GPL, then it is required that the program also becomes GPL, as parts of or the entire Muzz library is used in the program. In technical terms the program is either linked at compile time or at runtime. If linked at compile time, the used parts of the Muzz library is included in the program. If inked at runtime, the program uses the Muzz library when executed, requiring the Muzz library to be installed. Either way the Muzz library is considered part of the program, and thus the program has to use the GPL (the viral effect). Recognising these problems, the Free Software Foundation has created the Lesser General Public License.

The GNU Lesser General Public License (LGPL) addresses the exact problem of licensing software libraries and using libraries in software that are not considered free or open source software. On the flipside the LGPL relaxes some of the terms of the GPL, which is why it is called the Lesser General Public License. In particular the viral effect of the GPL is relaxed, and LGPL may be mixed with non-free software.

From an open source software point of view there are two competing interests: 1) To ensure that all work based on free software is free and 2) To ensure the widest possible adoption. Point 1 promotes the viral effect, and ensures that all code that is mixed with GPL'ed code also becomes GPL'ed, and will remain open source. Point 2 is interesting: When we speak of libraries, this is in fact a standards/compatibility discussion. A library is as said a set of functions that programs can utilise. Wide adoption of a particular library does in fact mean that a standard is being formed on the basis of the particular library. For instance, the Muzz library could easily consist of algorithms for decoding digital music in various formats. This will make it possible for both open and closed source developers to base applications on the algorithms in the Muzz library. Wide adoption lowers the incentives for others to develop a competing library that might impose rude license fees for using the library. The LGPL is aimed at the situation, where a company might corner a market by having a particularly important library.

Actually, there was a situation, where library license issues had a profound impact on the direction of development. The K Desktop Environment (KDE) is a graphical user interface for Linux that makes extensive use of a particular library. The library was called Qt and developed by a company called Troll Tech. The license called for $1.500 developer license, and modifications to the library are not allowed [Parens 1999:175]. KDE could not function without the Qt library, which was a very important part of the system. Even

though KDE itself was open source and based on the GPL license, open source proponents argued that KDE could not be open source software, as it was dependent on non-open source software. The KDE development team tried to negotiate a separate license for KDE, but this was still perceived as not addressing the real problem of being non-open source. As a consequence, another group of developers began development of a competing graphical user interface, GNOME, which was to be completely free software. As GNOME began to show potential, Troll Tech realised that Qt would never stand a chance in the Linux market, if the license was not changed. Subsequently Troll Tech released Qt with an open source license. Today both GNOME and KDE are in development and widely used, the competition is friendly, and work is progressing to make integration better between them.

Returning to the license review and going through the LGPL, we observe a structure similar to the GPL, only modified to take into account the special issues of libraries. The LGPL begins with a prolonged preamble that discusses the problems of libraries and the reason for creating a license, which is less protective of free software than the GPL.

The first paragraph defines a library and a few other things, and states what is covered by the license.

Like the GPL, the LGPL permits the licensee to copy and distribute verbatim copies of the library's complete source code. Again it is noted that the license refers to the source code for the library. The licensee may charge a fee for the physical act of transferring a copy of the source code. Should one feel compelled, it is possible to offer warranty for the library and to charge for this service.

The licensee may modify his copy of the library to fit his needs and form a derived work (a work based on the library). This derived work may in turn be copied and distributed, but the following terms must be met:

a) The modified work must in itself be a library

b) All changed files must carry notice hereof.

c) The derived work must be licensed to all third parties at no charge

Section a) above is different from the GPL, and ensures that a LGPL'ed library does not turn into a program itself with less powerful LGPL license. Thus it is not possible to develop a library into a program, and use parts of this program in software with non-free licenses.

A library can be distributed with other types of software, and this software is not affected by the LGPL license. It is possible to distribute and copy the library in an executable form, but the complete machine-readable source code must be available. Again this ensures that the source code can be edited on a computer – the source code is no good on paper.

The LGPL distinguishes between programs that merely use the library, and programs that are statically linked. A program that uses the library is not within the scope of the LGPL, and any program may use the library, and will not be affected by the license. Programs that statically link to the library and incorporate parts of the library thereby creating an executable program, must meet the following conditions:

A program, which is statically linked to the library and contains parts of the library, is considered a derived work of the library. The derived work may be distributed under any terms that the author chooses, but the elements of the library included in the program

remain under the conditions of the LGPL. But, the terms must allow modification for the customer's own use and allow reverse engineering for making such modifications. Naturally, derived works must carry all sorts of notices, which state what library was used and the license of the library. Also one of the following things must be done:

a) The complete machine-readable source code must accompany the derived work.

b) There must be a written offer for three years for obtaining the complete source for the derived work.

There are three other options that one may choose from, and common for all is the intent to ensure access to the complete source code, while still making it possible to create and sell derived works.

This section shows some of the main differences between the GPL and LGPL. Unlike the GPL, the LGPL allows the licensee to distribute executables, and these executables may carry a license that does not allow redistribution. Considering potential longevity of the executables, the LGPL requires that anyone, who has acquired the executable, must be allowed to have the source code and modify it to fit his own needs.

The library files may be placed together with other library files, and these other library files are not covered by the LGPL. It is also allowed to distribute the aggregated library, provided that:

a) The library is accompanied with a copy of the same work based on the library, and

b) Notice is given that this is a work based on the library.

Like the GPL, the LGPL does not require anyone to accept the license, but note that there is no other way of being allowed to copy, modify and distribute the library. By installing, one also indicates acceptance of the license.

The last part of the LGPL contains the usual legal wording that the licensee can only do what the license states, and that receivers of derived work must automatically receive a license from the original licensor.

Like other open source software, the LGPL offers no warranty what so ever, and the author cannot be head liable for any damage caused by the program, of cause unless required by law or statute.

5.5.5 The Mozilla Public License

Mozilla is an open source browser based on the source code from the Netscape browser, which was released in spring 1998. The name Mozilla was the original code name for what later became Netscape Navigator and Communicator[18]. It then became the dinosaur-like mascot of Netscape:

[18] For a description of the mission and goals for the Mozilla development effort see [website] http://www.mozilla.org/mission.html

Picture 1: Source, http://home.snafu.de/tilman/mozilla/mozabout.gif

Rumour has it that Mozilla is a pun on Godzilla, a giant nasty reptile that attacks a big city, because the code for Netscape was a big nasty pile.

The Mozilla Public License (MPL), see appendix 1.4 for the full text, was created, when Netscape decided to release the source code for the Netscape Browser. Netscape actually created two licenses, the Mozilla Public License and the Netscape Public License (NPL), the latter being an amendment to the first. It seems clear from these two licenses that Netscape's strategy was to create a relationship with open source developers. In this relationship Netscape would release most of the source code for the Netscape browser, and in return receive improvements and bug-fixes to the code. The MPL is interesting, as it is the first open source license to come out of a company that changed the license of their product from closed source to open source.

The MPL is written with the intent that Netscape would release the source code under the MPL, and then open source developers could chime in and contribute code. Therefore the NPL defines an initial developer grant and a contributor grant. The initial developer grant is aimed at the person or entity that released the initial version of the program. The initial developer grants the licensee the right to copy, modify, and distribute the program in whole or in parts.

The contributor is the person or entity that contributes code to the original program. As the contributor holds copyright for the code that he writes and contributes to the original program, the contributor must also grant a license to copy, modify and distribute the modifications with or without the original program. The MPL anticipates that many developers contribute to the code and thus many license holders. The license accommodates this by explicitly mentioning that contributors grant the same rights to the licensee, as does the initial developer. This is to ensure that contributors do not include code, which has a license that favours the individual contributor, and not the project as a whole. It would be very unfortunate, if a developer could halt development of the larger project by retaining special rights to his parts of the code.

Modifications to the program are governed by the NPL, and the source code for the program can only be distributed under the NPL and may not be offered under terms that restrict the rights granted in the NPL. The NPL underlines that source code for any changes to the program must be available.

Like the GPL, the MPL the licensee is obliged to distribute a copy of the license along with any distributed source code. The source code for any modifications or contributions made must be available for 12 months, or 6 months after latest version has been released. And any changes made to the source code must be recorded in a document that states the nature and date of the change.

Also like the GPL, the NPL demands that all changes and modifications made to files are documented with the date of change. The MPL offers no warranty what so ever for the

covered code, and likewise the distributor and developers cannot be held liable for any damage caused by using the program.

The NPL allows for binary only distributions, as long as the requirements of the NPL are met. However, such a binary only distribution may carry a different license, if this license complies with the terms of the NPL, and explicitly states that the added terms are exclusively on behalf of the distributor, and not of the other contributors.

Should the source code become part of a larger program not covered by the NPL, the NPL does not require the same license to cover the larger program. This is unlike the GPL that would have required the larger program also to become GPL (the viral effect)

Like all other open source licenses, the NPL does not offer any warranty or liability for any damages.

5.5.6 The Netscape Public License

The Netscape Public License (NPL) is basically a series of amendments to the MPL. The amendments ensure that Netscape retains special rights to code released under the NPL. The NPL permits Netscape to use/release code, contributed under the terms of a different license, and to use the code in other products without having to release the modified source.

This allows Netscape to take modifications made by (unpaid) contributors and make them private, and to release the modifications in a separate product while choosing a license at their leisure.

5.5.7 The IBM Public License

The IBM Public License (IPL) was created to accommodate IBM's entry into open source software development. IBM is betting heavily on Linux as a platform, and has spent significant amounts of money modifying (porting) Linux to run on their mainframe hardware. IBM is offering a range of Linux distributions for their workstation and server systems. For instance companies, running IBM mainframes (very expensive), have the option to run several parallel sessions of Linux along with their other applications on the mainframe. The advantage of offering Linux for their hardware is the range of applications, which now run on Linux, and which can be ported to the mainframe hardware.

IBM is investing in development for Linux, and some of this development is released as open source software. Not knowing the actual incentives for IBM's venture into open source software, it seems plausible that IBM expects some kind of return from the open source community for releasing their software for free use. However, the IPL does seem to be a fair open source license.

The IPL starts off with a few definitions and by stating that anyone, who uses, reproduces or distributes the program, agrees to the terms of the license. The definitions do not contain anything controversial, and they just make sure that there are as few misunderstandings as possible in the license. Only thing remotely interesting is the definition of "contribution". Contributions are changes and additions made to the program (original program plus changes) by a contributor. The "contributor" is defined as IBM and anyone, who distributed the program. Thus the contributor does not have to contribute anything to the program, as long as he distributes the program. The definition of contributor makes the distinction between distributor and developer collapse.

The IPL grants the licensee the right to reproduce, prepare derivative works, public display, distribute and sublicense the contribution, and derivative works in source or object code form.

The IPL grants the licensee the right to make, sell, offer to sell, and import the contribution in source or object code form. Should the contribution make use of patented code, the licensee is granted a royalty-free license to use the patent, but only in the combination of the original program and the contribution. It is of-cause noted that there is no license for any hardware related patents.

The IPL makes an effort to ensure that not just the IBM grants these rights, but also that any contributor grants the rights. Naturally, this is done to ensure that a contributor does not incorporate code that has special license terms, and thus hinder development at a later stage. By agreeing to the license, contributors also state that they have sufficient copyright for the code, they donate, and that contributors themselves are responsible for infringement caused by code contributed to the program.

A contributor may distribute the program (copy of original program plus contributions) in object code form under its own license, provided that:

a) It complies with the terms in the IPL

b) Its license agreement

    a. disclaims all warranty and liability on behalf of all contributors

    b. states that any provisions that differ from the IPL are on behalf of the contributor alone

    c. states how to obtain source code for the program.

Should the licensee choose to make the program available as source code, it must be made available under the IPL, and a copy of the IPL must be included. The licensee is allowed to distribute the program under a different license, for instance offering additional services. The new license must, however, comply with the IPL.

The IPL allows the licensee to make commercial distribution and to further provide extra services and warranty at the licensee's choice. Any licensee, who does so must ensure that the extra service or warranty disclaims any liability for other contributors than the licensee offering service or warranty. This is quite important, as the licensee could otherwise offer all sorts of warranty, and then claim liability of other contributors.

Like all the other open source licenses, the IPL does not offer any warranty or liability what so ever for any damages resulting from use.

5.5.8 The Artistic License

The Artistic License was created for PERL (Practical Extension and Reporting Language), a powerful scripting language often referred to as the glue of the web. PERL is used for all sorts of automation tasks and as translator between applications, hence the glue analogy. PERL itself is an interpreter that is installed on a computer, and then users can pass PERL commands and PERL programs to the interpreter, which performs the execution. In addition to the features found in PERL itself, a vast number of extra functionality is found in modules that can be downloaded from the web. A module is a PERL program that is designed to perform a specific task. Modules are designed for incorporation into other programs, thus offering easy integration of functionality into a

new program. Most modules can be found at CPAN[19] (Comprehensive PERL Archive Network), a web site dedicated to making PERL modules available to the public.

The purpose of the artistic license is to define a set of conditions, under which PERL modules may be copied, and the copyright holder retains some control over the direction of development of the module. The artistic license explains that the licensee should ensure users the right to distribute in a customary fashion. A customary fashion refers to a machine-readable form of the source code, and as mentioned earlier source code is of little use, if thousands of lines of code have to be transcribed. PERL is an interpreted language, so there is no binary code, and the scripts are interpreted directly by the PERL interpreter, thus there is no way of hiding the source code for a module or a script.

The Artistic License grants the licensee the right to distribute verbatim copies of the version of the program, which the licensee received. The usual requirement of preservation of copyright notices applies like the other open source software licenses. The program may be patched and modified with code, which originates from the public domain or from the original author, and which will still be considered the original version. Source code in the public domain is not copyrighted in any way, and when included in the program, public domain code will become copyright by the original author.

The licensee may modify the program, provided that notice is given about how and when any given file was changed. It is also required that the licensee does one of the following:

a) Places the modifications in the public domain or makes them freely available, or allows the copyright holder to include the changes in the original version.

b) Uses the modifications within the organisation of the licensee only.

c) Rename any executables, which have a name that conflict with the original version of the module.

d) Makes an arrangement with the copyright holder of the original package.

The restrictions on modifications are very interesting and different from the other open source licenses reviewed. Basically, a developer who has made modifications to a module must either surrender his copyright for the modifications (paragraph a) or keep them to himself (paragraph b). Placing modifications in the public domain seems a very poor choice, as the author of the modifications thereby forfeit the only option to protect his intellectual property. It is noteworthy that GPL uses copyright to ensure that software remains available. Placing anything in the public domain makes it possible for anyone to claim copyright for a derived work of the modification. If changes or a derived work are kept private, it is not possible for others to elaborate on the derived work. Paragraph c makes sure that the files in the original and the derived work share no resemblance, thus making it difficult for third party to identify the derived work. The last paragraph requires arrangements to be made with the copyright holder of the original code, and how this affects the legal status of the licensee's new code, will be a matter for the copyright holder

[19] CPAN is network of websites dedicated to distributing modules, code, and updates for PERL. A module is self-contained function allowing other PERL programs to easily use an advanced feature without having to implement the complete feature. See www.cpan.org for more information.

and the licensee to decide. In any case, the licensee would have to discuss matters with the copyright holder, which in itself would be a hindrance to a creative development process.

The GPL license does not require the licensee to surrender the rights for code that he has contributed. This makes it easier to extend existing software, and it is probably also the reason why almost all PERL modules are dual licensed, and carry both the Artistic license and the GPL [Parens 1999:183].

So far the Artistic License has not mentioned distribution of modules, which is an important issue in the open source philosophy. The Artistic License allows the licensee to distribute modules covered by the license, provided that the licensee does one of the following:

a) Distributes the original version of the module along with any required library files together with instructions of where to get the original version.

b) Accompanies the distribution with the machine-readable source for the modified version of the package.

c) Accompanies any modified executables with copies of the original version, and names the modified versions differently. Also any differences must be documented clearly in the manual pages.

d) Makes arrangements with copyright holder.

Like the requirements for modifying a module, the requirements for distribution ensure that the original copyright holder retains creative (artistic) control over the module. A licensee, who creates a derived work by modifying the original code, must do one of the above points, which in all cases places an extra burden of work on the licensee.

The licensee is allowed to charge the cost of media plus labour plus shipping and handling for distributing the module. The licensee is allowed to charge any fee, he may choose for support of the module. Like the other open source licenses, the module may be distributed with other software, as an aggregated compilation of software and modules, and the license of this module will not affect the other software. The other software may be commercial, and the compiled distribution may be commercial as well.

The license does not offer any warranty at all, and the module or software covered by the artistic license is provided as is.

There has been some discussion regarding the wording of the Artistic License. Bruce Parens, the author of the open source definition, regards it as a sloppy, written license with loopholes [Parens 1999:184]. In particular, the Artistic License does not allow the licensee to sell the module or software for a price higher than media + labour + shipping. But the same paragraph states that software with Artistic License can be aggregated with other software and sold. Thus compiling a lot of software with Artistic License and selling it for a huge profit would be allowed.

The Artistic license allows C or PERL subroutines, supplied by the licensee and linked into the module, to be considered as not being part of the module. In reality this means that when a module is linked, the license no longer applies. When linking the module to some other software, the licensee is able to take the module private [Parens 1999:184] and apply a different license.

The Artistic License seems to over emphasize the artistic control, which the copyright holder of a module should enjoy. This makes the license a hindrance for others, who wish to use modules and extend to their own use.

5.5.9 The BSD License

The BSD license originates from the early Berkeley Software Distributions, dating back to the 1970'ies. The version used in this review is dated in 1999, where the license was changed in order to remove a much-discussed advertising clause. Basically the clause stated that any programs, incorporating code covered by the BSD license, must include the sentence "This product includes software developed by the University of California, Berkeley and its contributors." in any advertising material.

The BSD license is the shortest license of the ones reviewed, approx. one page long. The BSD license grants the licensee the right to redistribute, modify, and use the program in source or binary form. The licensee is also allowed to redistribute derived works in source and binary form. These rights are granted, provided that the following conditions are met:

Redistribution of source code must include the original copyright notice, the list of conditions, and a disclaimer of warranty and liability. If the program is redistributed in binary form, the binary must display the original copyright notice. These conditions and a disclaimer of warranty and liability must also be available in the documentation provided with the program. Lastly, no name of any organisation or person, who has contributed to the program, must be used to promote programs derived from the original program.

Unlike the GPL, it is not required that the license of derived works is the same as the one of the original program. This means that a derived work can change license to a closed source license, which does not allow free redistribution or availability of source code. Programs based on the BSD license can be taken private, and turned into commercial products with very strict license terms.

It was not by accident that the BSD license permitted anyone to take the software private and to distribute it under different terms. Historically, the BSD license was applied to software developed under US government funding [Parens 1999:183]. The logic being that, since the citizens had already paid for the software, they should be granted the rights to use in any way they please.

At first sight this may seem like a license, where the first person to get a copy and change the license gets all the benefits. This is not the case. License changes can only happen 'down stream'. Hacker X receives a copy of the program, and must comply with the conditions of the license. When Hacker X redistributes the program, either in the original version or modified version, Hacker X may choose a different license. All Hacker X must do is to retain the copyright notice, the warranty disclaimer, and the no promotion clause from the original developer and contributor.

Suppose Hacker X redistributes a version with a strict license that does not permit redistribution, and starts charging for the program. Hacker X is within his legal right to do so and the one affected by the license change are the ones, who receive a copy of the program from Hacker X. The version of the program, on which Hacker X based his version, is still based on the BSD license, and it can be changed and modified as possible under the BSD license.

5.5.10 The Apache License

The Apache license (see appendix 1.9) is much like the BSD license, but has been adapted specifically for the Apache web-server. The Apache license grants the licensee the exact same rights to use, distribute, modify, create, and distribute derived works like the BSD license. The conditions for these rights are, however, different from the BSD license.

In general, the changes reflect that the Apache web-server is well respected, and that the Apache Software Foundation does not want to see the reputation of their software spoiled by program versions not approved by the Apache Software Foundation.

The conditions, like the conditions in the BSD license, require a redistribution of the program to carry a copyright notice, and a disclaimer of warranty and liability. Likewise redistributions of binary versions must also carry the copyright notice, and disclaimer, and the conditions must accompany the program in a file.

Different from the BSD license is the condition that any redistribution must include an acknowledgement of the program containing software from the Apache Software Foundation.

The name Apache must not be used to promote sales, and any software derived from the program must not be called Apache. The name Apache must not appear in the name of the derived work, unless of course the Apache Software Foundation has given their written permission.

Like other open source software the Apache license is very explicit when it comes to warranty, and the license states that the program is provided with no warranty whatsoever, and the developers cannot be held liable for any damage caused by the software.

5.5.11 The Free Beer License

The Free Beer License (FBL) has an amusing ring to it, but it is actually a license that has been applied to programs in the real world. In short, the FBL states that the licensee can use the program and source code, as he sees fit. The licensee is allowed to copy, distribute, modify, and create derived works.

As with other open source software, the license explicitly states that there is no warranty, and the licensee retains the full responsibility for any undesired results caused by the program. The licensee is not allowed to charge money for redistributing the program, and if the program is redistributed, the complete source code must follow with notice of the original author and where to get the original source. The license requires that further distribution of the program - be it in original or modified form - must carry the same license i.e. The Free Beer License.

The FBL has some amusing points. If the licensee uses the program frequently, and thinks it is worth a few dollars, then “you [the licensee] are not allowed to send the author anything else than beer, or means that provide facilities to get beer in me [copyright holder] (i.e. openers, glasses)” [Free Beer License, Version 1.0].

This clause caused a few lifted eyebrows and perhaps headaches, when IBM wanted to use parts of a program developed by FreeBSD developer Poul-Henning Kamp. IBM contacted Poul-Henning Kamp and asked, how much beer they would have to supply in order to comply with the license. Poul-Henning Kamp answered that since they had so many doubts, the program obviously wasn’t good enough, and thus IBM did not have to provide any beer at all.

The FBL is actually a very strict license, and unlike the GPL, it is not possible to make money from redistributing FBL'ed programs. It is allowed to charge a fee for the services required to distribute the program. The license does not contain any clause about how and with what the program may be distributed. Therefore FBL'ed programs may be distributed with other programs on, for instance, a CD with a compilation of programs. Charging money for the distribution is only allowed for the services related to distributing the software. For this reason it seems impossible to bundle FBL'ed programs with programs, for which no money is charged, as the distinction between what is actually paid for, and what is not, becomes blurred. It could be argued that any FBL'ed programs are free and bundled as an extra service. However, anyone buying the compilation in order to obtain the FBL'ed programs would be paying for more than just services being part of distributing the programs.

5.5.12 The Microsoft End User License Agreement

For the sake of contrast and to understand the implications on license types on goods and software, the Microsoft End User License Agreement (EULA) is included. The version used here is from Microsoft Windows2000 operating system. The EULA is by far the most elaborate license in this review and counts for seven pages with minimal line space; please see appendix 1.11 for a copy of the EULA.

The EULA describes a legal agreement between Microsoft and the end user, who may install, use, or copy the product. Microsoft refers to their software as a product – very unlike the previous licenses.

The EULA grants the licensee the right to install, use, access, and display only one copy of the product on a single computer, unless a license pack is acquired. It must be noted that laws in individual states may view this paragraph different from what is Microsoft's intention. In Denmark fair use of the product ensures that a licensee may install a copy on more than one computer, provided that the product is not used at the same time on these computers. For instance, if a person has both a laptop computer and a desktop workstation, this individual would be allowed to install the same program on both computers.

The licensee is only allowed to use the product on two processors; so using Windows2000 on a computer with four processors would not be allowed. The Windows2000 operating system is designed for use in networks and file- / printer sharing, and other networking features are possible. However, the license only allows for ten simultaneous connections to the same computer, and it is not allowed to run software residing on the licensee's computer.

The licensee is allowed to store a copy of the product on a network and install the copy on the licensee's computer. Should other computers install from this network, they would be required to purchase a license per computer. Microsoft naturally reserves all rights not explicitly granted in the license.

The EULA mentions upgrades as a separate issue, where the licensee looses his rights to the original product once upgraded. An upgrade would be a user with Windows95 on his computer that upgrades to Windows98. The upgrade itself will only function, if a prior version exists on the computer.

The EULA allows for some restricted transfer of license. The licensee may move the product from one computer to another. This makes sense, as users do from time to time acquire new computers. The original licensee is further allowed to make one, and only one,

transfer of the license to another end-user, provided that all the original material, i.e. manual, is passed on to the next licensee.

It is explicitly not permitted to reverse engineer, de-compile, and disassemble. However, some nations allow reverse engineering as a natural part of the flow of knowledge in society. Like patents that grant exclusive rights, but in return require that the knowledge contained in the patent is fully disclosed and available to the public. This is recognised by Microsoft, and the license therefore includes an amendment to the paragraph stating that applicable law may state differently.

The EULA has a limited warranty, which is very unlike open source software that has no warranty whatsoever. Microsoft warrants that the software performs substantially in accordance with the material that accompanies the product. The period, in which the product must be in substantial accordance, is ninety (90) days. The end-user must discover any defects within this period of time, or there will be no warranty of any kind. The end-user may get a refund for the product, if defects are discovered, or get the product fixed, which Microsoft will decide. The warranty, however, explicitly states that the end-user is not entitled to any damages resulting from errors covered by the warranty.

The warranty is followed by a disclaimer of warranty explaining that Microsoft provides the product as is, and with all its faults, and thereby disclaims any warranty for the product's fitness for use etc. etc. Microsoft is not liable in any way for damages caused by the product, and if Microsoft should for some reason be held liable, even though a whole paragraph is dedicated to exclusions from damages, the warrant can never exceed the price of the product or $5,00 (five). The limited warranty versus the disclaimer of warranty is interesting, as the limited warranty only covers things, which are explicitly stated in the documentation accompanying the product. Further, the capability described in the accompanying documentation only covers a small subset of the capabilities of the product. One gets the impression that the limited warranty is a marketing ploy to establish that Microsoft does indeed offer warranty for their products.

In other words, the EULA grants the licensee the right to use the product, and no more. The licensee is not allowed to copy, modify or reverse engineer the product. The warrant on the EULA is akin to the disclaimer, found in the open source software licenses, where the user can expect no warranty at all. It should be pointed out that Microsoft is not obliged to fix problems in the product, and the licensee is not allowed to fix bugs, as this would be in violation to the reverse engineering clause.

Unlike open source software, the source code is never available for Microsoft products. Source code is treated like trade secrets, and guarded closely. The source code for one Microsoft product would reveal much about the way that Microsoft implements and designs software. But, and more important, the source code would show the specification of the different network protocols that Microsoft uses. It must be remembered that Microsoft dominates personal computing and is in a position to dictate, how the different protocols should be implemented. If Microsoft was able to modify protocols like the HTTP (web surfing) protocol, users of other software would be unable to use the Microsoft controlled part of the web.

5.6 Summary

This chapter has covered software, property, and licenses. The chapter began by providing a historical perspective for understanding the laws of immaterial goods. The two

most important international conventions were presented and the importance of common international law highlighted.

While international conventions exist there are important differences between the laws governing immaterial rights. This aspect was discussed and it was noted that while the licenses originate in the USA they must be analysed from a Danish perspective.

A broad selection of 10 licenses was reviewed and analysed. It was attempted to use a common vocabulary to analyse the individual licenses in order to overcome their different wording.

6 Software as Simple Economic Goods

Four different research questions were proposed for offering an answer to the meta-question: "Why is open source software being developed?" Two of these research questions were directed at the empirical understanding of how open source software is being developed. The last two research questions were directed at answering the meta-question.

This chapter consists of three sections, which follow the analytical model for understanding why open source software is developed:

```
Properties of software + Type of agent=> A certain behaviour
(incentives and mechanisms)
```

The first section analyses properties of software and concludes that properties of software should be divided in two: 1) Technical properties and 2) License properties. The technical properties are discussed first, and then license properties are discussed in depth based on the review and analysis of various software licenses in chapter 5. A simple comparison between the reviewed licenses is made and three licenses are chosen as arguments for the further analysis of behaviour.

The second section analyses the two types of agents, firms and individuals, and discusses their differences.

The third section uses the simple theory of economic goods to analyse software distributed under the three licenses chosen in the first section. The analysis determines the type of economic good, which the three licenses create. The incentives for choosing a license are analysed under the assumption of a simple situation, where two agents: a maintainer and user distribute and obtain a program. The maintainer develops and distributes a program, and the user obtains the program. The simple model is used to propose a hypothesis regarding the behaviour of the maintainer and the user. The hypothesis is a general statement about agents' behaviour under the three license regimes.

6.1 Properties of software

Properties of software refer to properties that are embedded in the actual software. As noted in section 3.3, when developing the research questions, the properties of software are expected to influence and determine the incentive structure surrounding open source software. Properties of open source consist of technical properties and license properties. Technical properties are general to all software and license properties are specific to the license of the given software. In order to clarify properties of open source software, they are contrasted with a closed source software license. The technical properties and the license properties will be discussed separately, beginning with the technical properties.

6.1.1 Technical Properties

Open source software is software like all other software: A sequence of instructions to be interpreted by a computer, which then perform the desired actions accordingly. The software in itself is stored on a computer or some sort of digital medium, which a computer is able to read. Software being a digital manifestation may be copied without loss of quality, and infinite copies identical to the original can be made.

If the software is stored on a computer with no connection to the outside world, it is clearly possible to exclude people from obtaining the software and enjoy the benefits, which it provides. However, should the software be made available on the Internet, it is possible for anyone interested to obtain (download) and enjoy the benefits of the software. Once made available, it is not possible to control access to the software. If only one person has downloaded the software, it is made available to anybody. Once software has been released to the public, the software becomes non-excludable. The original developer is no longer able to exclude people from enjoying the benefits of the software good, and identical copies can be made at non-prohibitive cost.

When a program is executed on a computer, the computer reads the program files from storage, usually a disk or a network share. Regardless of the type of storage, information in the files is *read* from storage and subsequently executed by the computer. Before, during, and after a computer has performed the reading action, the files remain the same. The program files themselves are not altered or changed in any way, and therefore the software remains the same in the process of using the software. Files containing user data or other kinds of variables are likely to be manipulated by the software. These files are not considered part of the software, but are results from using the software.

6.1.2 License Properties

Licenses for open source software were discussed in depth in chapter 5, and the results will be used in this chapter. It must, however, be underlined that a license is an agreement between the rights holder and the licensee. The license defines the terms and conditions, under which the licensee is allowed to use the software. The licensing scheme used in open source software is dependent on copyright laws to enforce the license conditions. An often-encountered misunderstanding is that there is no copyright for open source software – nothing could be further from the truth. Without copyright there would be no open source software licenses, as we know them, and they would have to rely on a different regime.

It is the purpose of this section to determine the important license properties that are to be used as arguments when later analysing the impact, which license properties have on behaviour. It is evident from the review of various software licenses that indeed many licenses exist, and that each of them provides the licensee with different rights. It is not possible to analyse all licenses and determine their impact on behaviour. Thus the license properties to be used in the remaining part of the analysis must be a subset selected from the licenses reviewed. Two strategies for choosing license properties exist: 1) Choose individual license properties, and 2) Choose a selection of complete licenses.

Choosing individual license properties is the process, where the licenses reviewed in chapter 5 are discussed individually, and common and distinguishing license properties are identified. After identifying individual properties, the importance of the properties must be evaluated according to their expected effect on behaviour. This is a process, where the theory presented in chapter 4 is used to discuss the importance of the individual properties.

Choosing a selection of complete licenses is the process, where the differences of the reviewed licenses are discussed, and three complete licenses are chosen. The three licenses should be been chosen according to their relative differences. This has the advantage that the chosen licenses can be compared

The latter strategy has been chosen as the possibility of comparing complete licenses is judged to be very important. The strategy of choosing individual license properties suffers from various weaknesses, which begin with choosing the most important license

properties. This process would, as noted, require use of the theory presented in chapter 4. Thus the theory would be used in both the process of evaluating license properties and analysing the impact of license properties on behaviour. The danger lies in a premature analysis of license properties, whereas the choice of properties might be premature, as the full consequences of a license property is not known. This approach also has the disadvantage of splitting up the licenses, which are documents, which themselves cannot be atomised, and should be analysed as whole.

6.1.3 Choosing Licenses

The choice of licenses to be used, when analysing the impact of licenses on behaviour, should be based on their comparative difference. It seems appropriate to choose three licenses of which two represent opposites of a closed source-open source spectrum, and the third somewhere in the middle. For the purpose of choosing three licenses the licenses reviewed in chapter 5 are compared in table 4.

In a sense table 4 anticipates the analysis, as certain elements of the licenses are chosen and thought to be more important than others. The license properties chosen, are those expected to influence behaviour of the two types of agents. It is important to underline that table 4 is a simple vehicle for choosing three licenses according to their comparative difference.

The following properties were chosen for the comparison of licenses:

- Properties of distribution, copying and modification to binary and source code must be analysed.
- Binary distribution is important for selling to potential end-consumers. The terms of binary distribution note whether the licensee is allowed to distribute and sell binary versions of the program. It is also noted, if a binary distribution requires the source code to be available.
- The table notes whether modifications require the same license as the original program from which it was derived, and whether the license requires modification to be identifiable.
- The conditions for derived works are: Is it allowed to make a derived work? Must the derived work carry the same license as the original program, and lastly if the derived work must carry a different name.
- Conditions for mixing the source code or parts hereof with other programs are noted. Does the license demand that the program, into which the code is inserted, adopt the license of the inserted code (the viral effect).
- For the individual developers, who contribute code to projects, copyright is important, and the table notes whether a license requires a contributor to surrender his copyright for the code, which he has contributed to the program.
- For adoption by end-users, warranty can be important, and the table notes whether a license offers warranty.
- The table notes whether some party mentioned in the license retains a special privilege.

- Sublicensing is also important, and to the end-user it may be significant, if a licensee has the right to charge for sublicensing.
- The last three rows are about special restrictions in profit, price of sale to 3'rd party, and the cost of source code.

X marks explicitly positive mention in the license, - marks explicitly negative mention in the license, and empty cells mark positive answer, but not positively mentioned in the license.

	GPL	BSD	MS	LGPL	MPL	NPL	IPL	Artistic	Apache	FBL
Distribute source code	x	x		x	x	x	x	x	x	x
Copy source code	x	x		x	x	x	x	x	x	x
Modify source	x	x		x	x	x	x	x	x	x
Distribute binary	x	x		x	x	x	x	x	x	x
Sell Binary	x	x	x6	x	x	x	x	x	x	x
Binary-only distribution requires source to be avaialable	x			x	x	x	x	x4		x
Modifications must carry the original license	x			x						x
Modifications must be identifiable	x	x		x	x	x		x		
Create derived work	x	x		x1	x	x		x	x	x
Derived work must cary same license	x			x						x
Derived work must carry a different name								x2	x	
Code can be mixed with code carrying a defferent license		x		x	x	x			x	x4
License affect other software in aggregated distribution										
Contributor retain copyright to modifications	x	x		x	x	x		x5		
License offer warranty			x7							
Original copyright holder has special privileges to modifcations						x3		x2		
May charge for sub-licenseing the program or derived work		x							x	
Restrict copyright-holders profit										x
Restrict price of sale to 3'rd party										x
Restrict cost of source code	x			x				x		x

Table 4: The table shows a comparison of the software licenses reviewed in chapter 5.

1 Derived work of the LGPL must be a library.
2 Modifications are allowed if a) they are in the public domain or b) the derived work is not distributed or c) names are changed or d) arrangements are made with original copyright holder.
3 Netscape retains the right to use NPL'ed code in other products under different licenses.
4 Only if the original copyright-holder agrees.
5 The license only refers to the program and not parts of the source, but modified works must carry same license. Incorporating parts may constitute a derived work, in which case it is viral like the GPL
6 The complete software package may be sold once and only by the original buyer.
7 Microsoft offers a very limited warranty.

The obvious choices of the two extremes are the closed source Microsoft EULA and the open source GNU GPL license. The Microsoft EULA is a typical closed source software license offering only the right to use the software, and a very limited and perhaps inconsequential warranty. There is no right to copy, distribute, or modify the software, and only a limited right to sell the software to third party.

The GNU GPL on the other hand requires the source code to be available, and allows people to use, copy, distribute, and modify the source code. It is, however, required that all

changes, which are distributed, are made publicly available. The GNU GPL also includes the so-called viral effect, which requires that a program, incorporating code from a GPL'ed program, becomes GPL as well.

The choice of the third license is less straightforward. It must, however, be an open source software license, and it must be less strict than the GNU GPL. The BSD license is proposed as the third choice, and unlike the GPL, the BSD license

- Does not require source code to be available, when making a binary distribution
- Does not require modifications to carry the original license
- Requires that derived works must carry a different name
- Can be mixed with code carrying a different license
- Does not restrict costs of supplying source code.

Although both licenses are open source software licenses following the Open Source Definition, they are different beasts altogether. The BSD license allows far greater degrees of freedom in the sense that individuals or firms can use the licensed code in closed source software. In contrast, the GPL license provides far more freedom in terms of ensuring that the program remains free and open source. The BSD has no requirements about returning modifications to the project, and the BSD-license places no restrictions on using the code in other projects.

6.2 Two Types of Agents

In section 3.1 it was explained that there was a problem with understanding, why open source software is being developed. The problem was related to the fact that both firms and individuals participate in open source software development, and that they could assume the role of both developer and user. Firms and individuals are expected to have different incentives, as firms pay salaries and must consequently generate profits. Individuals do not rely on open source software development as a source of income.

The key theoretical difference between the two is that firms are profit maximising, and individuals are utility maximising. When individuals' utility maximise, they seek to strike a balance between work and leisure. When working too much, there is no time for leisure, but when working too little, there is no money for leisure. The difference between the two agents is most obvious, when the two types are placed in a situation, where choices have to be made. The firm is assumed to always make a choice based on profit expectations, whereas the individual will also value other benefits that might be gained, such as the joy of working with something interesting.

The two types of agents, firms and individuals, can assume two roles: 1) Developers and 2) Users. Developers are firms or individuals, who contribute actual source code to an open source software project. Users are firms or individuals, who use the software developed by open source software projects. It is imperative to understand that agents can assume the role of both user and developer, and they often combine both roles. In some situations the agent is only a user configuring an Apache web server. In other situations the agent is only a developer, who perhaps finds the technical aspects of web servers interesting and spends time helping with the development of the Apache web server. The agents can also assume the roles of both user and developer, as in configuring an Apache web server, the agent discovers some missing functionality and decides to implement it.

6.2.1 Resources and Incentives

Firms and individuals differ in available resources, in incentives for using and/or developing, and in the goals for using and/or developing open source software.

Firms are able to commit the amount of resources necessary to complete the desired task. Needless to say, firms have to act within the boundaries of economic capability and expected return from investment. Open source software may be free of charge, but implementing a solution to a problem certainly requires an investment in terms of time and training of users.

In contrast, an individual can only commit his personal time. If unemployed, the individual can spend all his time developing open source software, but only his time. Individuals cannot allocate other individuals' time to complete a desired task.

It is assumed that individuals do not have a direct profit incentive, and that they rely on other activities i.e. a job to generate income. Thus the resource available to an individual can be no more than his personal spare time. For this reason we use opportunity costs to measure the cost of an agent developing a program or helping to develop the program. Opportunity costs are defined at the cost of a resource, measured by the value of the next best, alternative use of that resource [Stiglitz 1996:a16]. For a person developing a program in his spare time the opportunity cost is equal to the income, which might have been generated in the time spend. Should the individual be hired by an employer to develop open source software, this very individual would then be considered a part of a firm and therefore adhere to the logic of a firm.

Incentives for using and developing open source software differ between firms and individuals. Lerner & Tirole [2000] pointed out that signalling incentives could be part of the incentive for individuals developing open source software, and that this would serve them to acquire a future job. Firms might also have signalling incentives, however, these are small compared to the individual. Eric S. Raymond [1999ab???] pointed to reputation effects as the incentives for participating in open source software development. Reputation effects was of value to individuals, who sought to gather a pile of reputation, which could be used to attract other developers to the individuals' own projects. Firms were not mentioned to be part of the reputation game. Firms might have a cost minimizing incentive to develop open source software. If a program developed by a firm, perhaps for internal use, were interesting to developers who were not employed by the firm, then the firm would gain a productive resource for free. One could also imagine a political incentive to use open source software i.e. discontent with a monopolist software vendor. Firms are not able to afford such leisure, unless political consumers pressure the firm, in which case there is a market incentive. Both firms and individuals have a cost saving incentive to use open source software. Open source can be copied freely, and there is no licensing cost for using it. Access to source code and the possibility of being able to modify programs to suite particular needs, are also incentives, which relate to both firms and individuals.

6.3 The Economic Good Perspective

The first research question subject to theoretical treatment is research question 3:

> RQ3: How do properties of software (license - and technical properties) effect open source software as an economic good? And how does the type of good effect behaviour?

This research question is divided into a main and a secondary part. The main part will motivate an analysis of how software licenses affect properties in consumption

(excludability and rivalry in consumption). The second part uses the analysis of open source software as an economic good to determine effects on behaviour.

This research question uses the theory of economic goods, which has been discussed in section 4.1. This research question is directly aimed at understanding the consequences of the technical properties of open source software in combination with license properties. The license properties are exemplified by the three licenses chosen in section 6.1.2, the GPL, BSD, and Microsoft EULA. The first part of this research question is solely focused on the effect of the licenses on consumption.

The three different licenses and the technical properties of software must be analysed in order to determine, how rivalry and excludability in consumption are affected. Rivalry [see chapter 4] in consumption refers to whether one person's consumption of a good affects the future consumption of the same good by others. Excludability refers to whether or not it is possible to exclude people from consuming the good.

The term consumption is central to this analysis and therefore important to discuss. It must be established, exactly what the economic term consumption means in conjunction with software. A good's properties in consumption describe the incentives for the consumer to consume the good. From the incentives of the consumer it is possible to derive the incentives, which the producer might have for supplying the good. In this framework a good is produced and then transferred to the consumer and often some sort of payment in involved. However, the discussion of payment and how prices are established are not part of the properties in consumption. We shall refer to the producer as the maintainer since this is the term used within the open source software community when referring to a person or firm, who is responsible for releasing new versions of a program.

The consumer receives a good that is consumed, like a consumer who buys a loaf of bread and eats it. We shall use the term user synonymously with the consumer in order to maintain a consistent use of terms. It is not the purpose of this theory to depict a situation, where consumer uses the good to create a new good, or to copy and redistribute the good. And right here lie the implications for this analysis - the implicit understanding of open source software is that people contribute to the code. Open source software grows, because those, who enjoy the benefit of the software good, develop the code further, and hence the good. Even though this is undisputedly a key element in open source software development, this very element is disregarded, when analysing properties in consumption. We only focus on the situation, where software has been produced and is transferred to a user. It is a static analysis, which ends when the transaction between the maintainer and the user has ended.

The user who receives the software is a simple end-user who does not further develop the software. This does not limit the discussion of excludability, rather it focuses the discussion to the situation for which it was intended, the question of whether consumers can be excluded from enjoying the good. While it is entirely true that consumers in some occasions do redistribute the software goods, which they possess, it must be observed that the very action of copying and redistributing software, transforms the user into a producer. This analysis is not concerned with the situation, where the software good is redistributed.

Within the confinements of this simple theory, we imagine a situation of two agents, where the maintainer is the producer, and the user is the consumer. The maintainer develops a program, distributes the program, and the user obtains the program and uses the program, see Figure 2. The maintainer may chose to develop a program, and further the

maintainer may chose the license, under which he wishes to distribute the program. The user only has a choice of whether or not to obtain and use the program.

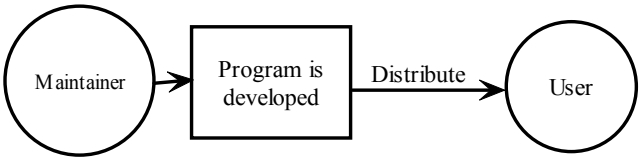


Figure 2: The simple situation, where the maintainer develops a program, and the User consumes the good contained in the program.

6.3.1 Technical Properties of Software

The technical properties of software are properties, which software exhibits as a result of its very nature. Software is a digital being that is stored on computers and like other things digital, software can be copied in unlimited numbers and in unlimited generations without loss of quality. One person can consume a software good, and this same software good is still available to other individuals to enjoy. Different individuals can use the exact same software at the same time or at different points in time. One person's use of the software does not detract in the slightest, nor does it cause wear or tear to the software good. It can be argued that software must be maintained in order to preserve its value due to change in user needs or bugs discovered. However, technically, software does not decay like a loaf of bread getting too old, and software is not worn by usage. The technical properties of software result in software goods being non-rival in consumption.

The non-rival nature of software has implications of excludability, but two situations exist, which must be distinguished. The first situation is one, where the producer and only the producer has possession of the program. In this situation, the producer is in a position to exclude people from obtaining the program and enjoying the benefits of his program good. Imagine a situation, where the program is contained in the computer of a producer, who can easily control access to the computer by simply locking the door and preventing unauthorized access. This program is excludable in consumption, as the producer can exclude other people from obtaining the program. The program is non-rival in consumption, because one persons' use of the program does not detract from the value for others, who wish to enjoy the benefits of the software. This program is a club good.

However, in the second situation, the producer distributes the program on the Internet. Once available on the Internet, the producer looses the ability to control access to the software. When the first user has obtained the program, he can in turn make the program available to other users. In this situation it is not possible for the producer to exclude other people from obtaining the program and enjoying the benefits of the program.

Being non-rival and non-excludable in consumption places software in the category of pure public goods. Theoretically, we must expect that such goods suffer from under-provision [Stiglitz 2000:132]. Under-provision is a consequence of not being able to exclude agents from obtaining the software, and thus agents, who are not willing to pay the demanded price, cannot be excluded. The result is that all agents prefer to obtain the good at zero cost, which leaves little incentive for the producer to provide the good.

It is the second situation, in which the program is distributed, that is of interest here. If the program just resides on the computer of the producer, and is never distributed, it cannot be consumed by the user. However, the technical properties of software do not tell the complete story about rivalry and excludability, as licenses influence this. It is well

known that property rights are the basis for voluntary exchange of goods. If a firm or an individual is unable to protect his property and exclude the ones, who do not wish to pay the price, the market will exhibit various types of market failure [Cones and Sandler 1996:43]. Software licensing is a way of extending property rights into software goods, and thus licenses affect excludability and rivalry in consumption. Analysis of the different licenses will begin with the Microsoft EULA.

6.3.2 The Microsoft EULA

The Microsoft End User License Agreement (MS EULA) used here is from the Microsoft Windows2000 operating system. The main characteristics of the MS EULA can be summed in the following short description. The program is personal and may only be installed on one computer, and this computer may only allow 10 concurrent users to access services (for instance files) on the computer. It is not allowed to disassemble, de-compile, or otherwise reverse engineer, unless applicable law permits so. The last clause "unless applicable law permits so" refers to the fact that some countries like Denmark allow reverse engineering. In the USA reverse engineering is felony if the person performing reverse engineering is trying to circumvent a security measure [The Digital Millennium Copyright Act]. In other countries the law might not decide whether or not reverse engineering is allowed and this leaves it for the license to define the terms of use. The product may only be sold to one other end-user. Microsoft offers a limited warranty discussed in section 5.5.12. The product may be copied to network storage, but may only be installed on a computer, for which a license has been purchased.

The characteristics of the license affect excludability and rivalry in consumption in several ways. Even though it is technically possible to make unlimited copies of the Windows2000 operating system, the license outlaws this. As the software is covered by copyright, and the license only permits copying to a network share with the intent to install on a computer with a legal license, it is not permitted to make and distribute copies. In this thesis it is assumed that transactions are mutually beneficial, meaning that agents only sell or buy voluntarily. It is recognised that widespread illegal copying of software exists. However, this is illegal the transaction constitutes an involuntary transaction on the part of the copyright holder. While such transaction is no longer mutually beneficial and also illegal, it must be recognised that widespread pirating of software exists. The Business Software Alliance (BSA), which is a coalition of software vendors, estimates that one third of all downloaded software is not covered by valid licenses [BSA Press release, 29. May 2002]. The BSA is usually alerted by former, disgruntled employees, who report their former employer. Former employees are aware of illegal licenses being used. Even though it is illegal to copy and distribute Windows2000, it is evident that a fraction of those enjoying the benefit of the Windows2000 good, do so.

This raises questions of whether the maintainer (the producer) is able to exclude other agents from enjoying the good. While some are excluded and pay to enjoy the benefits of the good, others are obviously not intimidated by copyright protection, and enjoy the benefits of the good without paying.

The EULA does not allow the same copy of a program to be installed on more than the one computer, for which the license was intended. Clearly, the intention of the license is to create a situation of rivalry in consumption. Assuming the user obeys the law, the MS EULA changes the properties of software from being a pure public good to a good, which is excludable due to the license. Such a good is a club good.

The MS EULA makes the program excludable because the license prohibits two persons to use the same version at the same time. If one person uses the program and another person wishes to enjoy the benefits of the good, then the second person is required to purchase a license and thus rivalry in consumption exist. The MS EULA is excludable because agents are excluded from using the program without a license through copyright and the terms of the license. The rights are enforced through law and the cost of excluding (assuming that agents obey the law) is not prohibitive. It is noteworthy that club goods often lead to partial rivalry in consumption as agents crowd to consume the good [Cornes & Sandler 1996:348]. Partial rivalry can easily be confused with the good exhibiting rivalry in consumption but the fact of the matter are that the consumption value of the good are not diminished and agents will just have to wait their turn. This is the exact situation of the MS EULA as agents are lot allowed to use the same license on two computers they will have to wait their turn. No matter how much the program is used the value of the program good available to the next agent is the same. Therefore the MS EULA creates a club good.

In the case that the user does decide to break the law by using, making, or distributing illegal copies, the illegal good shows signs of being a pure public good. The dominating properties of the software are now the technical properties allowing unlimited copies to be made. However, being illegal, a large group of users do not wish to use the software, and are thus excluded from enjoying the good. As the outlaws are able to copy and distribute the software at non-prohibitive cost, the good is non-rival in consumption. The illegal software goods then becomes a club good, non-rival but excludable, as legal users are excluded from enjoying the benefits of the good. Like other clubs, there is a cost associated with being a member, and in this case the price is the risk associated with criminal activity.

6.3.3 The GNU GPL License

The GNU GPL makes a sharp distinction between a source code and binary forms of a program. A further distinction is made, as to whether the source code or the binary is the original, or has been modified. Source code may be copied and distributed verbatim, as long as the copyright notice and disclaimer are produced. The GNU GPL license has some of the same properties as the BSD license. However, the GPL license demands more in return from the licensee. The GNU GPL allows the licensed software to be modified, copied, and redistributed, as long as the conditions of the license are met, see section 5.5.3 for a through description of the GPL.

This research question (Rq. 3) is concerned with the way, in which the GPL affects consumption properties of software licensed under the GPL. For this reason the implications of modifying and redistributing the modified software are disregarded for now. In this situation we are interested in understanding the situation, where a program with the GPL license is developed by the maintainer, and where the benefits of the software good are consumed by the user.

Like the above discussion of the technical properties of software, it is evident that software not released to the public is a club good. The maintainer can easily exclude people from obtaining the program, and there is no rivalry in consumption.

When the program is released to the public, the maintainer looses his ability to exclude people from obtaining the program and enjoying the benefits of the program good. In contrast to the MS EULA, the GPL license allows everybody to make copies and distribute the program. The GPL demands that the original copyright notices are kept intact. Keeping these copyright notices intact does not require intervention by the user

copying and distributing the program, in fact user intervention is required to remove or edit the files containing copyright notices. When released to the public, the program carrying the GPL license is non-excludable in consumption. The GPL license allows copying of the program, and since one consumers' use of the software does not detract from the value available to other consumers, GPL software is non-rival in consumption. As a program carrying the GPL license is non-rival and non-excludable, it can be categorised as a pure public good.

It does not matter whether the user is a firm or an individual. Once the maintainer has released the program, it is available to any agent at the cost of obtaining the program.

6.3.4 The BSD License

The BSD license is the shortest license of the ones reviewed, and the properties of the BSD license can be summarised as: Redistribution in source and binary forms with or with out modifications are permitted, providing that the following criteria are met: 1) The copyright notice is reproduced, 2) The name of the copyright holder must not be used for promotion. The software comes with no warranty whatsoever. See section 5.5.9 for a revive of the BSD license.

When only focusing on properties in consumption, the properties of the BSD license are identical to the properties and arguments made for the GPL license. It can thus be concluded that software, carrying the BSD license, is both non-rival and non-excludable in consumption. The BSD license then turns software in to a public good.

6.3.5 Effects on Behaviour

The analysis of the three licenses concluded that they are indeed different, and this effects the type of economic good, which is created. The MS EULA creates a club good, and the GPL and BSD license both create a public good.

There are two sides that must be taken into account when discussing effects on behaviour: 1) Consumption, represented by the user and 2) Production, represented by the maintainer. We are analysing development of software, and although the cost of creating copies is negligent, the cost of development is not. Software development is time consuming, and there are production and opportunity costs for firms, and only opportunity costs for private developers. Firms must pay salaries to developers, and private developers cold have spent their development time generating an income. Firms could also have chosen to have their software developed by another firm, which is the typical make/buy discussion. Whatever the firm chooses, there is always an opportunity cost represented by potential income from alternate investment. It must be expected that developers, be it firms or private, seek to cover the costs associated with development. Private developers incur an opportunity cost, as the cost of their time is not zero, i.e. development is done in spare time. Private developers could have spent their time differently earning an income, and therefore there is an opportunity cost associated with development.

6.3.5.1 Considering the User

The user is faced with the question of whether to consume the software good developed by the maintainer, or not. Whether the user is a firm or an individual, who enjoys the benefits of a program good, they are expected to behave in the same way. Both must analyse the costs and benefits of the available software, and choose the ones that satisfy their needs.

Given two identical programs, one proprietary and one open source, it is obvious that the user will prefer the open source program, simply because it is freely available. The open source software product can be used and copied freely, while the proprietary software must be expected to have a cost as the maintainer, through the license, is able to exclude agents, who do not wish to pay the price demanded.

The situation is rarely as simple as comparing two programs with identical functionality. While there are many programs, which exist in a proprietary and an open source incarnation, the comparison goes far beyond just comparing raw functionality. Firms often emphasize support from the vendor of the program. In this situation the analysis was restricted to production and consumption of a program, and this does not include maintenance and support of the program.

6.3.5.2 Considering the Maintainer

The maintainer must first choose whether to develop the program or not, and if the program is developed, decision must be made whether to distribute the program, and under which of the three license. The maintainer is aware of the behaviour of the user under different licenses, i.e. under a MS EULA license the user will pay for the program, and if a large population of users is considered, it is not possible to completely control excludability, and it must be expected that a percentage of users will use the program illegally. If the program is distributed under one of the open source software licenses (GPL or BSD), the agents will use the program, as it is a freely available public good. The maintainer also knows that, if a competing open source software product exists, it is likely that the user will choose the free open source version.

The Maintainer as a Firm

Assuming the maintainer is a profit maximising firm the first choice is whether to develop the program. This is a classic make-buy decision, where the maintainer must evaluate the costs and advantages associated with developing the program. We do not consider a situation, where an open source version of the program needed is available. The maintainer can only buy the program form another firm, or make the program. When buying the program, there is a direct cost, which is equal to the price, and when developing the program the cost is equal to the opportunity cost and direct cost of employees working on the program.

The reason why a firm acquires a program, either by self-development or by buying it, is use of the program for productive purposes or sale of the program for a profit. When using the program for productive purposes, the program becomes part of the firms' production system. If the program is the source of a competitive advantage, it is unlikely that the firm will choose to distribute the program. Distributing the program would undermine the competitive advantage achieved by using the program.

If the program is not a source of competitive advantage, the firm can distribute the program and seek to profit from doing so. The only license ensuring revenue from distribution is the MS EULA license. This license turns the program into an excludable club good, and thereby a firm can profit from the investment in the program.

The open source software licenses do not provide an opportunity to exclude people from enjoying the benefits of software. An open source program might be stuffed in a drawer, and thereby excludable. But once distributed, the program is freely available. Theory of public goods predicts that firms have no incentive to provide a public good, unless the firm cannot benefit without providing the good. One situation, where there is an

incentive to provide a public good, is where the cost of excluding others from enjoying the benefits of the good is too high, and other mechanisms ensure profits. One such situation is where the benefits from providing the good outweigh the costs of providing the good, which is available for other agent. Consider, for instance, a large ship owner in a port surrounded by reefs. There is a substantial risk for the ships to crash into the reefs at huge costs for the ship owners. The large ship owner builds a lighthouse to make sure that his ships can pass the reefs safely. The other ship owners can also use the lighthouse good provided by the large ship owner. The other ship owners do not pay for enjoying the benefit of the lighthouse good. Since the benefit of providing the lighthouse good is very high to the large ship owner, it does not matter that others enjoy the benefit of the good without paying.

It is not possible to imagine a situation analogous to the lighthouse example for the maintainer developing a program. The problem of the lighthouse is the fact that the maintainer benefits from the lighthouse, but it is not economically feasible to exclude other agents from enjoying the benefits. It is always possible to exclude agents from enjoying the benefits of a program through licensing, unless the agents are using the program illegally.

In this simple static situation, where the maintainer distributes a program to the user, there are no gains from distributing the program under an open source license. The maintainer only stands to loose from distributing the program under an open source license. The MS EULA on the other hand ensures a profit from selling the program to the user. The maintainer, as a firm, thus only has an incentive to distribute a program under a MS EULA style license.

The maintainer as an Individual

As an individual the maintainer does not depend on generating an income from selling software. The maintainer must first choose whether to develop the program or not, and this is the make/buy decision. However, the make/buy decision for individuals is somewhat different from that of firms, which consider costs and competitive advantage. Individuals are utility maximizing rather than profit maximizing, and their utility may consist of the joy of programming as mentioned by Brooks [1995].

Individuals, like firms, might face a situation, where the desired program is not available, and this induces an incentive to develop the program. Individuals might need the program as a productive resource used for personal activities. The productive resource is not perceived as a source of competitive advantage compared to other individuals, as they do not compete like firms compete for customers.

The cost of developing the program is equal to the opportunity cost associated with the time spent on developing the program. The maintainer will develop a desired program, if the opportunity cost of developing the program is less or equal to the benefit of using the finished program.

The maintainer must then decide whether to distribute the program or not. If the maintainer decides to profit from the developed program, the individual is transformed into a firm, which will distribute under a MS EULA style license. This license turns the program into a club good, where the maintainer, to some extent, is able to exclude non-paying users. As an individual not wanting to profit from the program, there is no incentive to even distribute the program. The maintainer will not benefit from distributing his program under an open source software license. Because the transaction between the maintainer and the user is static and only happens once, there is no incentive for the maintainer to distribute the program. Assuming the program is distributed via the Internet

as a free download like other open source software, the maintainer will not be compensated, when the user obtains the program.

If the maintainer is motivated by what Brooks [1995] calls the joy of having users, the maintainer has an incentive to distribute the program under one of the open source software licenses. However, this type of transaction goes beyond the simple situation analysed here.

6.4 Summary

This chapter began by analysing properties of software and defined this as a combination of technical properties and license properties. Technical properties were the same for all types of software, and license properties imposed special properties. The previous chapter reviewed a number of different licenses, which revealed that the licenses were very different and allowed for different kinds of use of the software.

It was stated from the beginning that properties of software are very important for the behaviour of agents, and for this reason the reviewed licenses were compared and three licenses selected as basis for further analysis. The three licenses were: The GPL, The BSD, and the MS EULA

Following the choice of licenses the three licenses were analysed as simple economic goods with the intention of revealing whether this theoretical framework would be sufficient for answering the meta-question. Indeed the theoretical framework was insufficient but did however contribute to our understanding of licenses by illustrating thatdifferent licenses create different economic goods. The MS EULA uses intellectual property rights to create a club good, the GPL on the other hand uses the same intellectual property laws to create a public good.

The analysis further revealed that a static analysis of production and consumption of software does not explain, why open source software is being developed. It is an empirical fact that open source software is being produced by both firms and individuals, however this analysis failed, and apparently the static approach to the phenomenon is not sufficient to understand, why open source software is being developed.

7 A Model of Software Development and Consumption

This chapter goes further in trying to explain why open source software is being developed. The chapter builds on elements of the previous chapter and uses the three licenses selected in the previous chapter (The MS EULA, GPL and BSD licenses) as the defining characteristics of three types of software to be analysed. The chapter develops a model of software production and consumption capable of explaining why open source software is being developed.

The chapter begins where the previous chapter ended with a simple relation between the maintainer and the user for software released under a MS EULA type of license. Next the GPL and lastly the BSD license is analysed and discussed. The chapter ends with synthesizing a model covering all three licenses.

7.1 Methodology and Theoretical Foundation

The methodology in this chapter is to analyse the relationship between the maintainer and the user, which was defined in the previous chapter. The theoretical foundation of this chapter is the theory of externalities and competing technologies as found in chapter 4. Thus the relationship between the maintainer and the user will be analysed from the perspective of economic externalities (as well as personal incentives). This differs from the perspective presented in the previous chapter, which focused only on the maintainer and the user as producer and consumer respectively.

From the literature of open source software in chapter 2 two terms will be extensively used in this chapter: 1) User-developers and 2) Use-products.

A user-developer is an agent who is a user and potentially a developer [Johnson 2001]. In this view a user-developer always begins as a user and for reasons, which will be explored here, may become a developer who contributes to developing a program. The user-developer term is tied to open source software development by Johnson [2001]. However, this restriction is relaxed here, and user-developers may exist for software licensed under each of the three licenses. The reason for relaxing the restriction is to allow for a consistent analysis of the three licenses without having to separate the analysis of the MS EULA into a different type of analysis.

A use-product is an agent's particular use of a combination of features in a program. Software is not a simple good like grain. Software is a complex good in the sense that a software good consists of many different features working together [Bessen 2001:1]. Agents have different preferences and will desire different features, and in turn agents will use different combinations of features in a program. By containing many different features in one single program, the supplier tries to satisfy the aggregated demand while actually serving many discrete markets. A program containing a number of different features represent a number of use-products equal to the number of possible combinations of features. Given the individual preferences, each agent will demand only a single use-product i.e. only a single combination of features. It is important to understand that the notion of a use-product does not constitute a neither/nor disposition, where a program is rejected because the exact number of features desired is not present. A user may desire a use-product consisting of n features of which only n-1 features are present in the program.

Given no alternative program containing the desired use-product, the user will have to make do with a use-product consisting of n-1 features. While an agent desires the complete use-product, the agent will maximize his benefits by choosing a good containing the most important features. A user will not choose not to use a program at all and the features it may offer, just because one or more of a large number of features is missing.

The term "user" used in the previous chapter were sufficient for illustrating the simple situation analysed. However, in this chapter a model is developed, which allows users to make and distribute modifications to some of their software. For this reason we shall use the term user-developers, which describe an agent who may choose to use or make modifications to a program (assuming the license permits).

The maintainer is still the agent who is distributing the program. Within the open source software community the agent, who is responsible for releasing new versions of a program, is referred to as the maintainer, and we shall continue this tradition. Lerner & Tirole [2000] has referred to this person as the leader of a software project, but in the eyes of this author a maintainer is unable to lead in a traditional sense. The maintainer in an open source software project has no means available to direct the efforts of user-developers. See Edwards [2001] for an in depth discussion of this issue.

These terms, use-product, user-developer, and maintainer, will be used extensively when analysing the costs and benefits in the following pages.

We begin by analysing the MS EULA, as this is the most restrictive license and thus the simplest to understand. Next the GPL is analysed, and the section ends with the BSD, as this is the most permissive license. This is an approach, where the simple relationship between the maintainer, which was used in the previous chapter (Chapter 6, see Figure 2), is extended. The relationship between maintainer and user-developer is not just a simple flow from maintainer to user-developer and feedback loops are added between the user-developer and the maintainer. Where the previous chapter divided the discussion of the maintainer and the user, this chapter discusses both agents at the same time to emphasize the importance of the relationship.

The analysis of the three licenses begins by a brief presentation of the most important characteristics of the license. Based on the characteristics of the license the model of software production and consumption is expanded with the relevant feedback loops and other agents who might participate in production and consumption. This is done in a sequence of steps, where the appropriate elements are identified and then added graphically to the model. Having developed the model, the incentives for the maintainer and user-developer and their relationship are analysed. Lastly, the dynamics generated by imagining the model being an image of a continuing process is analysed.

7.2 The MS EULA license

The MS EULA does not permit reverse engineering, and software distributed under this kind of license does not come with source code. This type of software is only distributed as binary programs ready to run on a computer. The license only allows use of the software, and use is restricted to installation on a single computer, the implication being that only one person can use the software at the same time. The license explicitly prohibits copying and distribution of the program covered by the license.

We now consider a situation as depicted in Figure 3, where the maintainer develops a program, which is distributed under a MS EULA type of license. The maintainer incurs a cost from developing the program that is equal to the opportunity cost of the time spent on

developing the program. The maintainer may be either a firm or an individual, and regardless of the type we can assume that the maintainer has a profit incentive, as the license turns the program into a club good. The maintainer can then distribute the program at a price, which is determined through monopolistic competition.

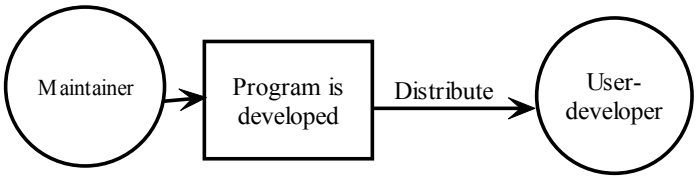


Figure 3: The simple situation, where the maintainer develops a program, which is distirbuted to the user-developer.

The user-developer can now obtain the program at the price demanded by the maintainer. Given that the program is distributed under an MS EULA type of license, it is not possible for the user-developer to make modifications to the program. Thus the user-developer under a MS EULA type of license cannot fulfil the role of the developer and can only act as a user. During the analysis of the MS EULA we shall refer to the user-developer as just a "user".

The user has a choice of whether or not to purchase the program offered by the maintainer. If the user does not purchase the program, the situation ends. If the user does purchase the program he may use it. The user is trying to obtain his desired use-product, which is his personally preferred combination of features in the program. If the desired use-product is contained in the program, the user will be content with the program and will have no reason to further pursue his desired use-product.

This situation is similar to the discussed relationship between the maintainer and the user in the previous chapter. The situation resembles a simple market where a seller and a buyer meet, exchange goods for money and then leave. If the desired use-product is contained in the program, the user will be content with the program and will have no incentive to further interact with the maintainer.

However, if the desired use-product is not completely contained in the program, the user has an incentive to further pursue the desired use-product. We can imagine two general types of problems with obtaining the desired use-product from the program: 1) Missing features and 2) Faulty features.

A missing feature is a feature not present in the program, and no combination of features or other manipulation of the program will produce the desired use-product. An example of this is a simple word processor not capable of performing a spell check. No matter what the user does, or how he combines other features, the desired use-product cannot be obtained.

A faulty feature, on the other hand, means that a feature is present but not functioning correctly, and the desired use-product cannot readily be obtained. The interesting property of faulty features is that they can be made to function, although it does require a workaround. A faulty feature can be made to function given the user knows how to manipulate the program i.e. the user must know how to work around the problem. The consequence of this definition is that a faulty feature, which cannot be made to function, is a missing feature. A feature will also be classified as faulty, if the feature is present but the documentation does not describe how to obtain the functionality. Most computer users will recognise the situation, where one struggles to obtain a particular functionality but is

unable to do so, although the prescribed procedure is followed exactly. When either experimenting or asking other users, a different and undocumented procedure for obtaining the functionality is discovered.

Obtaining the desired use-product in the two situations requires the user to engage in further efforts to obtain the desired use-product.

If the feature is missing the user must seek to influence the maintainer in such a way that a new version of the program is sure to contain the feature so that the desired use-product is now contained in the program. Thus we extend the simple relationship between the user and the maintainer shown in Figure 3 by a feedback loop from user to maintainer and derive Figure 4, where "private use" is a separate action performed by the user:

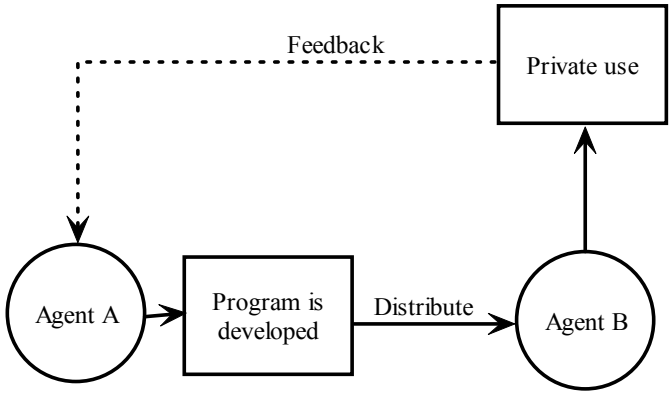


Figure 4: When a user decides to use a program he may provide feedback to the maintainer, if there are features missing or not functioning.

If the feature is not functioning correctly, the user can use the same feedback loop as described for missing features and hope that the maintainer will release a version that has been repaired. The defect may also be of such a nature that is only present under special conditions, and a workaround may exist to work around the problem and use the desired feature. The maintainer may know of this workaround and divulge the information to the user.

However, other users may also use the same program, and some of those other users may experience the same problem as our user. Each of the other users is trying to obtain his personally desired use-product from the combination of features contained in the program. While the desired use-product differs among users, there will be some degree of overlap. The more users the larger the probability of more than one user desiring the same use-product. When more users desire identical or overlapping use-products, the users will have complementary knowledge of the program and how to obtain their desired use-product. It is known from many programs, for instance MS Word 2000, that the same feature can be obtained in many different ways. For instance: One user may use keyboard shortcuts to execute macros performing many formatting operations at the same time. Another user may perform the same formatting by hand using the mouse for each and every formatting operation. The two users are obtaining the same use-product in two different ways. The first user is using the scripting capabilities of MS Word to obtain his use-product, and the other user is using the graphical user interface to obtain the same use-product. Imagine the graphical user interface is broken, and the second user is unable to obtain his use-product because of this.

This information is useful to the second user struggling to obtain his desired use-product. The first user possesses knowledge, which represents a workaround for the second

user's problem, and by gaining the knowledge of the workaround the second user becomes capable of obtaining his use-product.

Users can then interact and exchange knowledge about the program, which may help the users obtain their desired use-product without involving the maintainer. Thus we add another feedback loop between the user, on which we focus, and other users of the program, and we shall refer to this as user-to-user assistance. User-to-user assistance can be organised in many ways, and the simplest way is two people using the same program and sitting next to each other. These two users are likely to discuss their problems using the program and exchange tips and tricks. User-to-user assistance may also be organised as newsgroups on the Internet, where people discuss various topics. We know from research conducted by Lakhani & Von Hippel [2002] that such newsgroups are continuously formed and used as a means of solving usage problems with a particular program. All sorts of users from the most knowledgeable to beginners use such forums, and the more knowledgeable answer question from the beginners.

These two feedback loops to the maintainer and agents who use respectively expand the model depicted in Figure 5 to the following model of the relationship between the maintainer and the users:

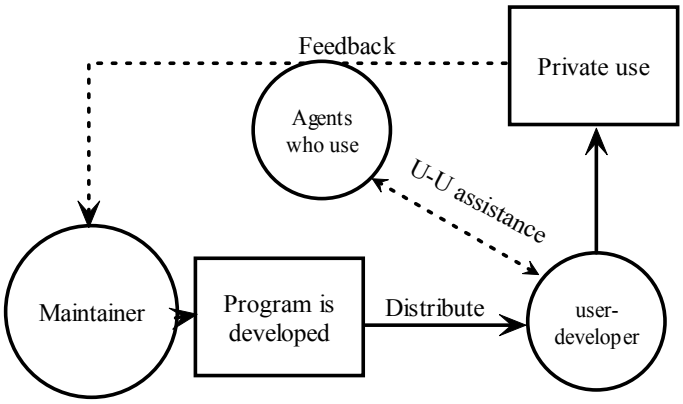


Figure 5: Programs distributed under a MS EULA style license restrict modifications and only allow a simple feedback mechanism to the maintainer of the progam and other users who may engage in user-to-user assistance.

In light of the above figure we now turn to discuss the economics involved then the actions available to the maintainer and the users. We are interested in understanding why and how the maintainer and the users interact, and analysing costs and benefits from the different actions depicted in Figure 5 will establish this.

7.2.1 Analysing the Maintainer

The maintainer has a choice between developing a program or not, and if the program is developed, the maintainer must decide whether to distribute the program. For simplicity we assume that there are no costs associated with distributing the program. Under this condition the choice of whether to distribute the program or not is a non-issue, as the maintainer will distribute, if the program is developed.

The situation we are analysing is a simple market situation, where a maintainer must decide whether or not to offer a product on the market. The maintainer will offer the product for sale, if he estimates that the cost of production including opportunity cost can be covered.

We know that software is a special good with no physical existence, which only has an existence on a computer, and the cost of production i.e. the act of reproducing the program is negligent compared to the cost of development. When developed, a program can be reproduced in countless copies at insignificant cost. The cost of developing a program is then equal to the opportunity cost associated with the time spent on development. The maintainer will choose to develop the program, if he is convinced that future sale of the program may cover his opportunity costs.

The maintainer will benefit from feedback externalities, if users decide to provide the feedback. The feedback represents information about missing or not functioning features, which users desire. If the maintainer implements the missing feature or repairs the defect feature, there is a possibility of enjoying an increase in sales. However, the maintainer will also actively develop the program to increase sales. The maintainer is offering the program for sale in competition with other programs, which induce an incentive to improve the program to satisfy both existing and new users (customers).

The maintainer will suffer a cost, if he decides to implement the desired features. The maintainer will implement the features, if the cost is expected to be less than the additional income generated by offering the features.

Feedback from users is welcome, as it reduces the cost of developing new features and also ensures that the features implemented in the program are actually wanted by users. While there is a feedback externality present in the relationship between maintainer and user, it is not the driving factor for distributing new versions and motivating the maintainer. The maintainer has chosen to create a club good and profit from selling this club good, and it is the profit from selling the program, which is the driving factor. The feedback externality is merely an added bonus, which the maintainer may decide to use.

7.2.2 Analysing the User

The user interacts with both the maintainer and perhaps also other agents who use the program. The two types of interaction are different and must be treated separately, and we shall begin by analysing the relationship between the user and the maintainer.

The user has a choice between buying the program distributed by the maintainer or not. If the program is bought, the user can then further choose between engaging in one of the two feedback loops or not. This being a standard market situation, the user will buy the program if the expected utility is equal to or higher than the price. Assuming the user has bought the program, we now analyse the economics of the two feedback mechanisms.

7.2.2.1 The User-Maintainer Relationship

If the user is unable to obtain his desired use-product either because features are missing or not functioning, the user must decide whether to provide feedback to the maintainer and/or enter into user-to-user assistance.

The feedback may be directed at a public forum or a closed forum. The maintainer has an obvious interest in controlling the information contained in these feedback messages given they will often express negative user experiences. If the information is directed at a public forum, competitors of the maintainer are likely to use this information to improve their own program and create a negative marketing image of the maintainer.

If the user provides feedback to the maintainer, the user suffers an opportunity cost equal to the time spent. The user knows that he will only be compensated if the maintainer

releases a program containing the desired use-product or provide a workaround for the problem.

The maintainer, on the other hand, may use the feedback to improve the program making it more attractive to purchase for new users. Thus the feedback represents an externality, as the user is not compensated. The maintainer receives a benefit from the users action, but the action is performed without consent of the maintainer. The maintainer will only modify the program if the cost is lower that the expected benefit i.e. more sales. While the program tries to cover an aggregated demand, the maintainer must be sure that the missing feature is sufficiently general to warrant the extra cost.

The user will provide feedback if the cost of doing so is lower than the expected benefit. The user valuates the expected benefit based on past experience, other information regarding the maintainer, or written material from the maintainer supplied with the program.

7.2.2.2 User-to-User Assistance

The user may also engage in user-to-user assistance in which case the feedback is directed at a public forum such as a mailing list. In this particular situation the feedback takes the form of a question to the other users regarding a problem with the users personal use-product.

When engaging in user-to-user assistance, there is a cost associated with posing a question as well as providing an answer. Other users of the program have their own different preferences, and thus consume different use-products; however, there is a probability of some of the other users having experienced a similar problem and thus can provide help.

The user will pose a question to the forum, if the cost of stating the question is lower than the expected benefit from obtaining an answer. Per definition a user-to-user assistance forum is a place, where users help each other by asking questions and providing answers. For this reason users will perceive the probability of obtaining an answer from such a forum as very high. Lakhani & Von Hippel [2000:24] found that users frequently asking questions had their problem solved in 92% of the times. A user struggling to obtain a desired use-product is likely to have spent a significant amount of time trying to obtain that very use-product. The additional cost of posing a question is very small, as the user is well aware of the problem and is able to state his problem in a few minutes. Lakhani and Von Hippel [2000:24] found that users posting a question spend on average 11,5 minutes preparing and posting a question.

At first sight it appears puzzling that some of the other users will provide answers to a question for free. However, the users providing the answer to a question knows the answer, and therefore the cost of providing the answer is equal to the opportunity cost of time spent on writing it. The user providing the answer has had previous experience with a similar or identical use-product, and therefore the knowledge is present in his mind. By providing an answer the information provider illustrates a willingness to help, which can be perceived as goodwill. The information provider is aware that he might himself need help at a later stage, and this goodwill helps to ensure that others will provide the needed information. Framed differently, information providers will only spend time answering question from the same person so many times before feeling misused. Lakhani & Von Hippel [2000:35] argue that users provide answers to questions because the value of the

information is low and because they believe that other users also posses the same knowledge. Lastly the cost of providing is very low with a typical time cost of 1-5 minutes.

A similar effect was described in Gosh' [1998] "Cooking Pot Markets", where the nature of digital beings and the searchable nature of the Internet (Google and other search machines) provide easy access to historical data. Possession of historical data and access to previous answers of other agents enhance the effect of user-to-user assistance, because the answer is already provided, and other users do not have to respond. However, answers to questions only have a certain lifespan, which is not recognised by Gosh. As the program grows and thus changes, the problems once encountered by agents using the program no longer exist, but new problems have arisen.

User-to-user assistance is an externality, as the agents posing questions and providing answers do not compensate each other for doing so. The externality is quite powerful as one answer to a question can be reused many times, like the effect described by Gosh [1998].

There is a significant difference between the two feedback mechanisms, as the maintainer is a firm. User-to-user assistance is performed in a public forum outside the maintainer's control, and consequently it is not possible to hide information. Feedback to the maintainer is not public, and neither does a firm have any incentives to divulge information about problems with their programs. Potential buyers will take such information into account when choosing between competing products.

7.2.3 Dynamics

In general the relationship between maintainer and user is that of a market with a feedback loop. Users buy programs developed by the maintainer at a price, and users may offer feedback to the maintainer thus creating an externality. The maintainer is driven by a profit incentive and will use the feedback to improve the product and thereby increase sales. In this relationship the dynamics, which drive development and consumption, is driven by an invisible hand. The feedback provided by users is technically an externality, as the users are not compensated nor encouraged to do so. While the externality is welcome, it is none the less a secondary effect. If the maintainer does not improve the product, another maintainer would satisfy user's demand by providing a competing program.

User-to-user assistance is on the other hand driven by externalities. Each user answering a question is generating an externality, the value of which is further sustained by the searchability of Internet based forums. Users offer answers to question in the anticipation that they may need help later, and offering help is a way of signalling that it is worthwhile to help this user. It can also be argued, as done by Lakhani and Von Hippel [2000:35], that users providing answers to questions perceive that value of their answers as close to zero and thus not worth protecting. However, Lerner and Tirole [2000] argue that open source software developers are motivated by signalling incentives, which may also help explain why users provide help to other users. By doing so a user signals a strong competence in solving a certain type of problems. Offering help also signals competence, which is the desired good in these forums, and hence users gain social status. This assumes that some degree of stability exists in the forums, as newcomers will be unaware of who has knowledge and who does not. However, the status derived when providing help is likely to be contained in the narrow forum defined by the community of users of the program.

The conclusion is that for the MS EULA feedback externalities have little impact on the dynamics. The behaviour of the maintainer and the user is by and large directed by simple market mechanisms. However, as we shall see in the next chapter (Chapter 8), the dynamics may change significantly, if the program is developed as part of a compatibility regime.

7.3 The GPL License

The GPL license is far more permissive than is the MS EULA, and this has major implications for the model of software production and consumption. Unlike the MS EULA, the GPL allows the licensee to copy, modify and distribute a program licensed under the GPL. It goes without saying that the GPL license allows for unrestricted use, and the user may use the program on as many computers as the licensee may please. The GPL license requires the source code to be available, and thus users are able to make modifications to the source code, and for this reason users become user-developers.

Looking at this description it immediately becomes apparent that programs licensed under the GPL must resemble a pure public good. The license allows for copying and distribution, which essentially ensures that programs are non-excludable. It is already known that the technical properties of software ensure that a program is non-rival in consumption. A program licensed under the GPL license can thus be defined as a pure public good. From the theoretical chapter (Chapter 4) it is known that pure public goods suffer from provision problems, and this chapter will use the theory of externalities to analyse the incentives for providing the good

Following this brief recap of the GPL license and public goods, we will now discuss the implications of the GPL licenses in respect to the model of software production and consumption, and again we consider the maintainer and the user-developer separately. We first modify the model in accordance with the additional options that a user developer has under the GPL license and add them to the model just developed for the MS EULA. Having modified the model, we analyse the new model from the perspective of economics of externalities and determine incentives for agents in the model.

When a maintainer has released[20] a program under the GPL license, the program is available to anyone interested. Any user-developer may now obtain the program and use, copy, modify, and distribute the program. When engaging in private use, the GPL license offers the same feedback mechanisms as the MS EULA, and user-developers may contact the maintainer and/or other user-developers. While the possible feedback mechanisms are the same, the incentives are not quite so, and we shall soon address this.

The GPL license further allows user-developers to modify, copy, and distribute the program. In this analysis we are only concerned with the right to modify and distribute the modified version. The right to copy is a necessary prerequisite for modifying and distributing a program, and the analysis will not treat copying separately. A user-developer may very well choose to distribute identical copies of the program, which he received and is thereby re-distributing the program.

We are not concerned with the possible choice of re-distributing an unmodified program, as the program is already assumed to be available to user-developers as a download from the Internet. We are, however, interested in user-developers, who spend

[20] It is assumed that a release does in reality mean making the program available for download from the Internet.

resources modifying a program. If a user-developer makes modifications to a program, it triggers a clause in the GPL license, which makes a sharp distinction between whether the user-developer keeps the modifications private or decides to distribute the modifications.

If the modifications are kept private, the GPL license does not require the source code for the modifications to be available for other agents. A user-developer can obtain a program distributed by the maintainer, modify the program, and keep the modified program private. These two choices: 1) Modify the program and 2) Keep the modifications private are added to the model of the MS EULA in Figure 5. The two choices must be seen as closely connected, as a user-developer upon decision on making modifications to a program must further decide whether to keep or distribute the modifications. Addition of the two further choices results in the following figure (Figure 6):

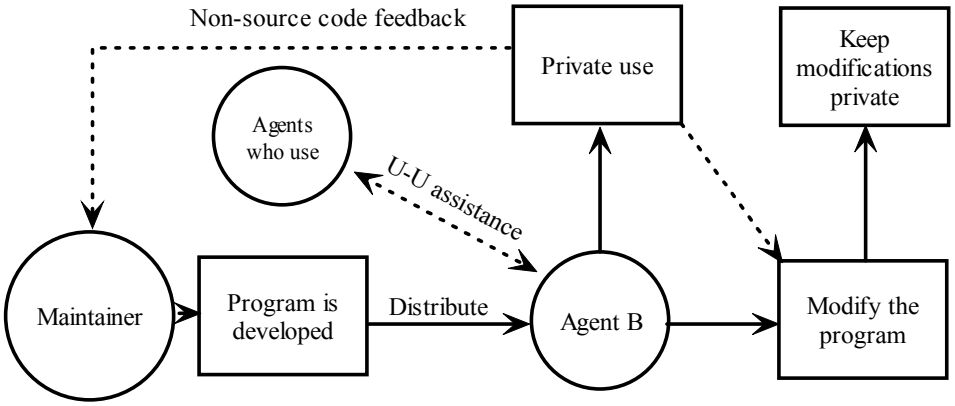


Figure 6: Under the GPL license the user-developer is allowed to make modifications to the program and keep them private. The dottet line from Private use to Modify the program indicates that a user-developer may use the program before deciding to make modifications.

However, as just noted the GPL license further allows a user-developer to distribute the modified program. Technically, the result of modifying a program can be either a modified program or a set of modifications (a patch, see section 9.2.1 for a description), which can easily be applied to the original source code. We shall refer to both instances as “the modifications”.

When modifications are distributed, the GPL license requires that: 1) Modified files must carry a notice stating which modifications were made, and the date of these, and 2) Any work that is distributed and derived from the original program must be licensed to any third party at no charge under the GPL license, and 3) The source code for the modifications is available.

The first requirement ensures that the various copyright holders can always be identified. The second requirement is quite important, as it ensures that modifications cannot change license. A user-developer cannot modify a program and distribute the program as a product under a more restrictive license like for instance the MS EULA. The third and last clause ensures that if a user-developer decides to distribute the modified program as a binary program, the source code must be available to users of the binary program.

While a user-developer may distribute the program to whomever, he may desire, it is assumed that modifications distributed are returned to the maintainer. This is an important assumption, as it adds a feedback loop from the user-developer to the maintainer. Unlike the feedback provided from private use to the maintainer and/or other agents who use, this

feedback loop contains modifications to the program. Such modifications may be added features, a new use-product, which a user-developer has desired, or a modification, which fixes a defect use-product. illustrates the addition of an additional choice for the user-developer, who modifies the program to the model of software production and consumption.

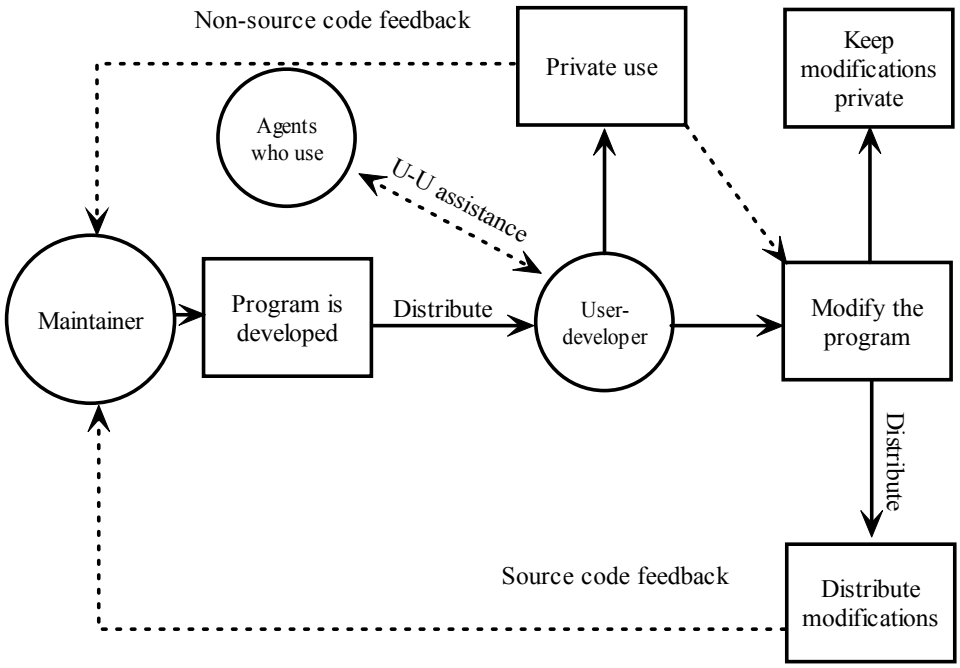


Figure 7: When modifying the program the user-developer has an additional choice of distributing the modifications, which creates a feedback loop to the maintainer.

We now have a situation, where the maintainer distributes a program, and a user-developer decides to modify the program and distribute the modifications. This situation only contains one maintainer and one user-developer, but like the situation of private use other user-developers are likely to make modifications to the program. If more user-developers are modifying the program it will be beneficial for the user-developers to discuss their modifications.

Like user-developers, who just use the program and engage in user-to-user assistance to solve problems with their desired use-products, a user-developer who modifies the program may also engage in similar activities. For ease of understanding, a user-developer who has modified a program shall now be referred to as a developer, and user-developers who only use shall be known as users. Developers like users have a desire to discuss the problems and ideas they encounter while modifying a program. However, as users provide non-source code feedback, and developers provide source code feedback, they are unlikely to find help for their problems in the same forums. Developers and users need separate forums to discuss their problems, and thus we add another forum to Figure 7, which shows a forum where developers may discuss ideas and problems related to making modifications. This forum has the special property of the maintainer participating. The maintainer is participating in this forum because the other developers are making modifications to the program originally released by the maintainer. The maintainer being responsible for releasing new versions of the program will be interested in discussing modifications to “his” program. If the maintainer is not interested in discussing

modifications and releasing new versions containing modifications from developers, the developers may very well release a new program containing the modifications.

We therefore add an additional forum for agents who modify, a forum which also serves to include the maintainer in the discussions. This addition to the model marks the final element in the model, which now contains the possible choices available to the maintainer and user-developers under the GPL license. This results in the final figure (Figure 8) of the model of software development and consumption under the conditions of the GPL-license.

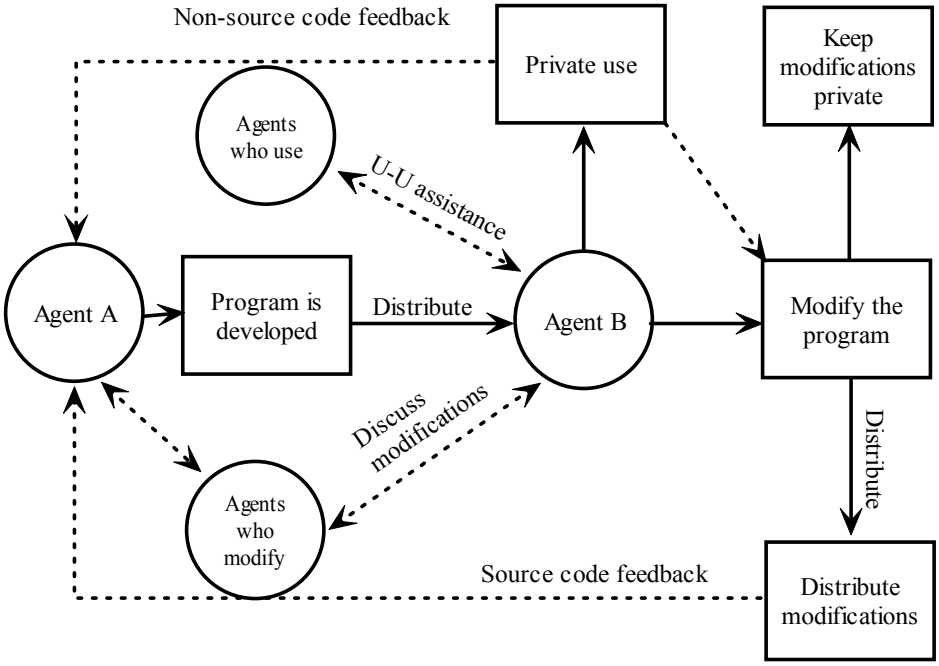


Figure 8: When other user-developers are making modificaitons to the program user-developers will discuss their modifications in the forum for Agents who modify.

Given the complete model for software development and consumption under the GPL license, we now turn to discuss the incentives for user-developers to modify the program, keep modifications private, distribute modifications, and discuss modifications. The conditions for providing non-source code feedback is almost identical to the situation under the MS EULA, and the dissimilarities will be discussed first. Again we divide the analysis into two major parts, one focusing on the maintainer and one focusing on the user-developer. However, before doing so we characterise the good developed.

7.3.1 Characterising the Good

A program licensed under the GPL can immediately be characterised as a public good based on the properties of the license, and we shall now further characterise the good developed.

A program can be viewed from many perspectives each defining a good with particular properties. The choice of perspective is important for the subsequent analysis of the good and will largely determine the properties, which are to be analysed. One perspective might be the good of security provided by the program. Imagine a program with two features each providing a distinct functionality but also exposing the user to a potential security risk. Defining the economic good as security, we find security to be a

general public good with no anonymity, and the characteristic of each security related feature is important. The size of the good of security becomes equal to the lowest level of security in each of the features in the program, as a security breach in just one feature causes all to be compromised. Formally this can be expressed as $Z = Min\{z^1, z^2\}$, where Z is the size of the public good of security in the program with two features z^1, z^2.

This little example highlights the importance of carefully characterising the good, which is to be analysed. In this thesis we are, however, not interested in security as a good. Here we focus on a good defined as the sum of features present in a program. For this reason we define the size of the public good, Z, as a function of number of features z present in the program, and thus $Z = F(z^1, z^2, ..., z^n)$, where n is the number of features.

In this perspective we view each of the features in a program as input to an aggregator function. The function ensures that the value of the public good increases with the number of features i.e. the function captures the number of use-products present and returns the size of the public good Z.

Each of $z^1 ... z^n$ is a feature present in the program and developed by either the maintainer or the user-developers. Each user-developer and the maintainer desire a particular use-product defined as a particular combination of features $z^1 ... z^n$. Although each of the user-developers or the maintainer does not desire all of the features for their particular use-product, the size of the public good, Z, is still defined as $Z = F(z^1, z^2, ..., z^n)$, as this is the size of the public good available to all agents.

It is tempting to define the size of the public good as the sum $Z = (z^1 + z^2 +, ..., +z^n)$ of features. In this treatment there is a noteworthy difference, as the individual contributions are not homogenous. Each contribution is different from the other and adds a feature to the program, and this difference has implications for using the theory of general pure public goods.

As the contributions are not homogenous, we cannot expect agents to consume the full quantity of the good available, given the premise that agents pursue a particular use-product. Agents, however, may choose from the full quantity of available features, and thus the full quantity is available to all agents. But, as the contributions are not homogenous they cannot be interpreted as a subscription. A program good, which is distributed under either the GPL or the BSD license, is available for all agents in full quantity, given the fact that a program good can be copied without loss of quality at insignificant cost.

Theoretically, a general pure public good suffers from provision problems, as agents only have an incentive to provide, where the agents enjoy a net benefit from doing so. As the full share of the standard pure public good is available to all agents, the incentive to free ride is high. However, as we shall see, incentives do exist for agents to provide the goods $z^1 ... z^n$.

It is arguably problematic to use the number of features as an approximation of the size of the good available. Another approach would be to evaluate the alternate cost of obtaining the same use-product ready-to-use software package. Still, the number of features is an indication of the usability and value of a program. In this situation the feature approximation seems appropriate, as it captures the essence of the behaviour in the model:

Agents desire a particular use-product, which is a combination of features present in a program.

7.3.2 Analysing the Maintainer

The maintainer has a choice between developing a program or not, and if the program is developed, the maintainer must decide whether to distribute the program. The maintainer will be distributing the program under the GPL license and is thereby creating a public good. As a public good the maintainer cannot control access to the program, and it is not possible for the maintainer to charge user-developers for obtaining the program. The license permits redistribution and copying, and the cost of doing so is negligent, thus the maintainer cannot expect to control distribution of the program, once the first copy has been distributed. We must therefore assume that the motivation of the maintainer is not a direct profit incentive like maintainers of MS EULA programs, as he could have just chosen a different license. However, the maintainer must have some sort of incentive to warrant his decision to distribute the program under the GPL license.

When developing a program the maintainer suffers an opportunity cost equal to the time spent on developing the program. Empirically it is undeniable that maintainers develop programs and distribute them under the GPL license. The question, however, remains why the program is being developed and distributed under such a permissive license.

The maintainer has an incentive to develop a program, if a desired use-product is not available. The desired use-product may not exist or may not be available to the maintainer for other reasons. Regardless of the reason, the maintainer desires a use-product not available, which leaves the choice of developing the program himself or paying someone else to create the program. If the maintainer decides to pay someone else to create the program, he will suffer a direct cost equal to the price demanded by the developer. While this is possible, it is a course of action, which places the maintainer outside this model, as the person developing the program becomes the actual maintainer, and the person ordering the program becomes a user-developer. This situation introduces a special relationship between the maintainer and the user, who employs the maintainer, which is outside the scope of this model.

Having developed a program the maintainer faces the decision of whether to distribute the program or not. If the maintainer keeps the program to himself, he will not enjoy any benefits from distributing the program, and the analysis ends here, as there will be no further actions related to the model. If the maintainer decides to distribute the program it must be expected that there are valid incentives for doing so.

When distributing a program, the maintainer creates an externality, as the program distributed has an effect on other agents, i.e. the user-developers who obtain the program. The maintainer is in no way directly compensated for his action, and user-developers may freely obtain the program.

Lerner & Tirole [2000] suggest that programmers are motivated by personal need and immediate plus delayed payoff. Personal need is a person's need for a particular program. Lerner & Tirole [2000] do not use the term use-product, albeit this appears to be what they actually refer to. A person has an incentive for creating a program containing a desired use-product, if this is not available through other programs. Immediate payoff is the performance improvement from using the desired use-product. Delayed rewards consist of both career concern incentives and ego gratification incentives, which are both grouped into signalling incentives.

Signalling incentives have special meaning for programs distributed under the GPL license. The source code for programs distributed under the GPL is available and this enhances the signalling incentives. Other developers and future employers may use the source code to ascertain the qualifications of a particular developer. This is, naturally, not possible for programs distributed under the MS EULA, where resumé and recommendations are the only means available to illustrate programming competence.

If the maintainer creates a program, which is never distributed, he will never receive any benefit from signalling effects. The program may solve a personal programming problem and be fun and educating to create, but no one will ever know, if the program is not distributed.

The maintainer will also derive a benefit from what Brooks [1995] refer to as the joy of having users. When a program is distributed and gains a user base, the maintainer will experience the joy of having users. The joy of having users is a secondary incentive, as the developer a priori cannot know, if any user-developer will find the program interesting.

The maintainer will benefit from feedback by users, should they decide to provide so. The feedback represents information about missing or not functioning features, which users desire. If the maintainer implements the missing or defect features, there is a possibility of enjoying an increase in the number of users. In the case of the MS EULA, the maintainer had an obvious incentive to make modifications to the program in accordance with the feedback provided by users. This would increase the sales of the program and the profit of the maintainer.

A program distributed under the GPL is not a private good, and while the maintainer may enjoy additional users of his program when implementing suggestions from feedback, he will not profit from this. This changes the incentives for the maintainer to use feedback to implement modifications. According to the three incentives mentioned above, the maintainer will implement modifications from feedback if 1) It creates a use-product, which the maintainer personally desires, 2) Implementing the feature creates significant signalling effects, or 3) It significantly increases the joy of having users i.e. expands the user base.

A maintainer may, unlike the MS EULA, participate in the user-to-user assistance forum. Feedback to the maintainer will often be an email from the user to the maintainer stating a problem or suggestions for future improvements. Such feedback is private and not available to the public, much like feedback under the MS EULA. It requires personal response from the maintainer, unless the maintainer decides not to respond to feedback at all. It is, however, unlikely that a maintainer will choose to ignore feedback about the program that he maintains and has a personal interest in, as well as it is likely that the maintainer will receive more than one message about the same issue.

The maintainer may therefore decide to create a public forum (a newsgroup or mailing list) in order to reduce his workload from feedback. Users can then be directed to the public forum to find answers for their questions. The maintainer has an incentive to participate, just to make sure that he only answers the same question once. The maintainer will be very knowledgeable about the program but only to a limited extent. Users will use use-products in the program, which the maintainer does not desire, and these users will be more knowledgeable about the particular use-product. The users form the basis for user-to-user assistance, which was also the case for the MS EULA.

The maintainer has no incentive to control feedback from users, as this will only increase his workload and thus his opportunity cost. The more information available to

users, the less is the need for the maintainer to spend time helping users obtaining their desired use-product.

7.3.3 Analysing the User-developer

When a user-developer obtains a GPL program, he can choose to use the program privately or to make modifications to the program. As illustrated in Figure 8 a user-developer may initially decide to just use the program and then later choose to make modifications to the program. When using the program privately, the user-developer may further choose to enter into user-to-user assistance and/or provide feedback to the maintainer. If the user-developer decides to make modifications to the program, he may further decide to keep the modifications private or to distribute the modifications. A user-developer may also start with private use and at later decide to make modifications to the program.

7.3.3.1 Private Use

Private use of a program distributed under the GPL license is much like private use under the MS EULA. The maintainer distributes a program, and the user-developer decides to use the program. The basic incentive for user-developers downloading a program is that they desire a particular use-product. When obtaining a program, the user-developer examines if the use-products offered by the program matches the use-products desired.

If there is a sufficient match between the offered and desired use-products, the user-developer has an incentive to adopt the program. As the program is licensed under the GPL, there is no cost associated with obtaining the program other than the opportunity cost of finding, downloading and installing the program. Maintainers of GPL programs are, unlike maintainers of a MS EULA program, creating a public good, and thus there is no market related profit incentive. The monetary reward from having one or a hundred user-developers is the same. A maintainer cannot be expected to distribute a program with the same kind of finish, ease of use, and documentation. This has consequences for the opportunity cost of user-developers trying to adopt the program. The cost of finding, downloading, and installing a program, which offers a particular use-product, must be expected to be higher for GPL programs.

When using the program, the user-developer may experience that the desired use-product is not completely contained in the program. Like the program licensed under the MS EULA, the desired use-product may be either missing or not functioning. If the user-developer experiences such problems in obtaining his desired use-product, he may, like the MS EULA, consult other user-developers for help, provide feedback to the maintainer or lastly use the source code and make modifications to the program.

In general it must be expected that programs distributed under the GPL are poorer documentation than are programs licensed under the MS EULA. Writing documentation does not add to the value of the personal use-product of the maintainer. Documentation serves the purpose of helping user-developers install and use the program and thereby obtain their desired use-product. This will generate, ceteris paribus, a higher pressure on the feedback loops to the maintainer and user-to-user assistance forums. In a sense this induces an incentive for the maintainer to write documentation in order to avoid acting as support for users of his program.

7.3.3.2 Feedback from User-Developers

As under the MS EULA, the user-developer provides feedback to the maintainer in anticipation of the maintainer fixing the particular use-product. Under the GPL license the maintainer has an incentive to keep all feedback in public forums and not try to control information. If a user-developer has problems obtaining a use-product in which one of the features is not functioning and a workaround exists, this information is better spread in a public forum. If the maintainer does not use public forums, the workload from processing incoming requests is higher than if user-developers can find help in a public forum. For this reason we discuss both feedback between user-developer and maintainer and user-to-user assistance under the same heading.

User-developers have incentive to provide feedback about missing features in the program, as the cost of providing feedback is negligent compared to the cost of actually implementing the missing feature. As the program is distributed under the GPL, user-developers know that the new versions will also be distributed under the GPL license. The GPL license requires derived works to be distributed under the same license, and a new version containing new features is exactly a derived work. User-developers then know that new versions of the program can also be obtained for free, and this raises the incentive to request new features in the program. At the same time the maintainer has little incentive to implement new features, which are not part of his personal use-product. Consequently, user-developers cannot expect the same responsiveness as from maintainers of MS EULA programs.

User-developers, who are providing feedback about a not functioning use-product, can expect the maintainer to provide a fix for the problem as either a new version of the program or a workaround.

It is known from Lerner & Tirole [2000] that there are significant signalling effects associated with open source software like programs distributed under the GPL. A program with defect features reflects negatively on the maintainer and therefore provides an incentive to fix the defective feature and distribute a new version.

Given the existence of a workaround, the maintainer or other user-developers may answer and provide the workaround. Whether the maintainer chooses to provide a fix or a workaround will depend on how the defect is perceived. If the defect is perceived as an obvious programming fault, there is a strong incentive to fix the problem. However, if the problem is perceived to be related to the particular user-developers' environment there will be little incentive to provide a fix.

7.3.3.3 Modifying the Program

Under the GPL license user-developers has the option of modifying the program, as the source code is available. A user-developer has an incentive to modify a program, if the desired use-product is not available in the program. Compared to the maintainer, the user-developer faces a significant lower barrier to obtaining a desired use-product, as there is an existing structure to work from. Following this logic a user-developer only has an incentive to modify a program, if that very program promises a significantly lower cost for obtaining the desired use-product. Otherwise the user-developer might as well obtain another program or create a new program and act as maintainer for this program. The cost of obtaining the desired use-product is equal to the opportunity cost associated with the time spent on implementing the missing or defect features.

Here we touch upon the essential problem of standard pure public goods: Provision. Modification of the program must be interpreted as provision, as the user-developer adds to the number of features in the product instead of just consuming the quantity available. The theory makes it clear that user-developers would prefer to free ride and enjoy the benefits of the program good without contributing to its provision.

A program good was characterised as a combination of features, and this definition makes it possible to understand just why a user-developer has an incentive to provide the good within the theory of standard pure public goods. Theory states that agents have an incentive to provide a standard pure public good, if the agents stand to enjoy a significant benefit from doing so. A user-developer, whose use-product is not contained in the program, indeed stands to enjoy a significant benefit, if his modifications makes it possible to obtain his desired use-product. The cost of making the required modification must match the benefit (net benefit = benefit – cost), which we shall define as a significant benefit. The theory of standard pure public goods argues a bit differently, although the essence remains the same. According to Cones & Sandler [1996] a user-developer would have to decide on the optimal quantity to use and produce of the good, given the user-developers' budget constraint. In this situation the budget constraint is given by the amount of time available the opportunity cost of this time versus the benefit from obtaining the good.

For example, a user-developer obtains a program containing the public good Z, the size of which is the sum of the available features $Z = z^1 + z^2 + ,...,+z^{10}$ (assuming the program contains only 10 features). This particular user-developer, however, desires a feature not contained in the program, which can be added by modifying the program. Depending on the cost of modifying the program to contain the desired feature z^{11}, the user-developer may have an incentive to add the 11^{th} feature.

Technically the desired feature z^{11} is a club good, as long as the modifications have not been distributed. This observation is most important, as the trick of defining features as agents' contribution to the public good Z only explains the incentive for making modifications to the program. Having made modifications to the program and implemented the desired features, the user-developer must decide whether to 1) Keep the modifications private, or 2) Distribute the modifications. The theory of standard pure public goods does not take into account a situation where agents have a choice of whether to contribute to provision or not notably after the contribution has actually been made. The two possible choices will now be discussed.

1) Keep the Modifications Private.

When keeping modifications private, the good (feature) created actually becomes an exclusive good (club good), as the modifications are kept within the confinements of the user-developer who made them. Unlike a normal club good the license does not allow the user-developer to distribute the club good. If the modifications are distributed in any form, it constitutes a distribution, which turns the modification into a standard pure public good.

There are several situations in which user-developers may have an incentive to keep their modifications private. If the use-product desired is a source of profit generating activity i.e. the program is used for some sort of business purpose, the user-developer can shield his business from competitors by keeping the modifications private.

If the user-developer is a poorly skilled programmer, there may be negative signalling effects associated with distributing the modifications, and he will choose to keep the modifications private instead.

It may be easy to create the desired use-product by creating a quick modification, which is working within the confinements of the particular user-developer, but not necessarily for other user-developers. Distributing such a modification will result in negative signalling effects, and the maintainer is likely to require the modification improved, if it is to be included in the next version of the program. While negative signalling effects might not be significant for this user-developer, there will be an additional cost associated with improving the modification. Obviously most programmers are interested in obtaining their desired use-product by creating a correct modification i.e. programming the modification in a generally accepted manner. However, if the correct way to obtain the desired use-product is significantly more costly, there is an incentive to make the quick and dirty fix and keep the modifications private.

2) Redistribute the Modifications

When the modifications are distributed, they become a standard pure public good and a potential addition to the program. It is a potential addition, as the maintainer has to approve the modifications before they are included and distributed in a new version of the program. The theory of standard pure public goods does not provide any incentives for any reason, why user-developers should distribute their modifications.

However, it is possible to imagine several situations where user-developers have an incentive to distribute their modifications. A modification included in the program provides signalling effects for the user-developer who made the modification. The GPL requires all modifications to be dated and to carry the name of the person who made the modification. Thus anybody may review the source code for the program and ascertain the quality of a modification and the identity of the person who made it. This provides a signalling incentive for user-developers to distribute their modifications and have them included in the next version of the program.

If the desired use-product can be obtained with a modification for which the correct solution is not significantly more costly than is a quick and dirty modification, the user-developer will not suffer negative signalling effects or requests for an improved modification.

User-developers may also distribute modifications in order to avoid having to maintain a separate feature. If a modification is kept private, the user-developer will have to implement his desired feature, whenever the maintainer releases a new version of the program. If the modification is distributed and integrated into the next version of the program, the user-developer can obtain future versions of the program and rest assured that "his" feature will be present in the program. This perspective will be extended and discussed further in the following section on the dynamics in the GPL license.

7.3.4 Dynamics Induced by the Cost of Being too Late

The analysis of the maintainer revealed that there are incentives for the maintainer to develop and distribute a program under the GPL. There are also incentives for the user-developer to make modifications to the program and redistribute these. However, the incentives are mostly derived from the maintainers and user-developer's private need, and the focus has so far neglected to take dynamic effects into account. A brief insight into the dynamics was seen in the last situation, where the user-developer distributed his modifications in order to avoid having to implement his modification each time a new version of the program was distributed by the maintainer

Dynamic effects arise, when the process of software development and consumption is repeated many times and becomes a continuing process, where the maintainer and user-developers continuously distribute their modifications. This principle can be illustrated using the concept of goods, where the size of the available good (Z) is equal to the sum of features (z) i.e. $Z = F(z^0, z^1, ..., z^n)$ where n is the total number of features in the program.

We now analyse a situation, where a maintainer releases a program containing a single feature z^0. The situation also involves two user-developers, who obtain the program. Both user-developers desire a use-product not contained in the program, and this provides an incentive for making modifications to the program. For the sake of clarity, the two user-developers will be named UD1 and UD2. UD1 desires a use-product requiring the feature z^1, and UD2 desires a use-product requiring the feature z^2. In order to obtain the desired use-product both UD1 and UD2 must make modifications to the original program. However, both desired use-product require modifications made, which are in the same area of the source code. This is an important premise, as the two modifications mutually exclude the other from being integrated into the original program without significant additional costs from integrating the two modifications.

UD1 is the agent who is the first to distribute his modifications to the maintainer. The maintainer finds the modifications interesting and good enough for inclusion in the program, and subsequently a new version of the program containing the modifications are distributed. The size of the available good in this new and modified version of the program is $Z = F(z^0 + z^1)$.

At the same time the maintainer distributes a new version containing z^1, UD2 has finished creating z^2 and distributes his modification to the maintainer. The maintainer receives the modification from UD2 and observes that the modifications are incompatible with the new version of the program, and implementing the modification from UD2 will break the modifications made by UD1.

This leaves the maintainer with a choice between 1) Reject the modification from UD2 or 2) Rework the modification and integrate it into the program. The difference between the two choices is a matter of who will have to cover the cost of integrating the two modifications. From the maintainer's perspective, the first choice has no cost other than the cost of answering "No" to UD2. The second choice results in the maintainer becoming responsible for integrating the modifications, and he must then cover the opportunity costs associated with the time spent on making the integration. The maintainer has an obvious incentive to reject the modifications from UD2 in order to minimize his own costs from maintaining the program.

UD2 is now faced with a potentially sunk cost, as the modification will not be integrated in the program by the maintainer. If UD2 is still interested in using the particular use-product, which he has developed, he must continue to use his personally modified version of the program, which in the model is equivalent to keep modifications private. However, doing so has consequences and UD2 cannot enjoy the benefits of new versions of the program, as they are incompatible with the particular use-product that he desires. This leaves UD2 with a program, where the size of the good is $Z = F(z^0 + z^2)$. Note that the feature z^1 is not present.

UD2 may choose to make modifications to the new versions of the program and integrate his personal use-product. This would create a program, where the size of the good is $Z = F(z^0 + z^1 + z^2)$. Naturally, UD2 suffers a cost from doing so, and each time a new version of the program is released, UD2 must decide if the new version of the program offers advantages, which justify implementing his personal use-product in the new version of the program.

UD1 on the other hand is in a position, where he gets the best of both worlds. His modifications are integrated in future versions of the program, and he can anticipate the benefits of new versions of the program. As new versions of the program contain UD1's particular use-product, he does not have to make modifications to the program.

A rejection signals not only to UD2 but also to other user-developers that they will have to distribute modifications, which are based on the latest version of the program. In this perspective, a maintainer's rejection of a modification becomes the source of dynamics in the model. Rejection of modifications imposes costs on the user-developer who has implemented the modifications.

The unfortunate situation between UD1 and UD2 could perhaps have been resolved though coordination. Given that both user-developers were seeking to obtain similar but not identical use-products, it is likely that a solution satisfying both could have been created. UD1 and UD2 should then have signalled their intentions to make the modifications in the forum for agents, who make modifications. While UD1 and UD2 may have signalled their intentions, this does not ensure coordination. Signalling is costless and making the actual implementation is not, and since agents have no way of guaranteeing their commitment, such signals may not be taken seriously. UD1 and UD2 cannot be sure when either of them will deliver, and therefore there is a high degree of uncertainty regarding the two agents' intentions.

If the two agents were to present a preliminary version of their modifications in the forum for agents who modify, they would both be able to signal their intentions by presenting their modifications. This would also allow the two user-developers to offer comments and suggestions for changes, before the cost of changing the modification becomes too high. Over time UD1 and UD2 will learn from observing each other's behaviour and whether to trust promises about future modifications.

A program rarely has only two user-developers, and the more user-developers the greater the possibility that someone is making modifications to the program. The more use-products present in the program, the more appealing will the program be. This has the additional effect of increasing the likelihood of user-developers making modifications to the program. With each new version of the program more use-products are contained in the program, and the more appealing it will be. When a user-developer obtains the program and discovers that 99% of his use-product is contained, he may be willing to modify the program, adding the last 1% so that 100% of his use-product is contained. Whether he makes the modification or not will depend on the cost of making the modification compared to the cost of engaging in strategic waiting for someone else to make the modification.

A user-developer, who engages in strategic waiting, cannot know when another user-developer will implement the last 1% of his desired use-product and thus the user-developer cannot predict the cost of waiting.

This process of improving the program and releasing new versions gradually lowers the barriers for modifying the program. The lower the cost of obtaining the desired use-

product, the lower the barrier to making modifications. However, here we begin to touch on the technical issues of programming and making modifications to a program, which are not readily contained in the model of software production and consumption. The essence of the argument is that the source code for a program can be structured in a variety of ways, of which some are easier to comprehend than others. As program grows, it suffers the danger of becoming a large kludge, which only the maintainer can understand. If this happens, the barriers to modifying the program raise significantly, as user-developers will have to invest significant amounts of time in just learning how the program functions. This time is essentially unproductive and little fun, as the user-developer during this period is unproductive and is not modifying the program to obtain his desired use-product.

7.3.5 Low Profit Potential Inspire the GPL license

If a maintainer could a priori know that his program would eventually dominate a particular market segment, he would choose a different license allowing the option to create a club good. And here we touch on a central issue in the production and consumption of open source software.

One reason for distributing a program under the GPL license is because the profit potential, when releasing the first version of the program is close to zero. The first version of a program is usually very rudimentary and only containing the very core feature, which the maintainer desires. Such program resembles what Brooks' refer to as a little program, which has difficulties functioning outside the confinements of the development environment where it was conceived. Such a little program as a profit potential close to zero simply because of the problems users will experience trying to obtain the desired use-product.

The program has been developed to provide the maintainer with a particular use-product. At this point in time the cost of the program is equal to the opportunity cost of the time spent. It is assumed that these costs are covered if the maintainer can obtain his desired use-product. Depending on the characteristics of the maintainer the cost may vary. The opportunity cost of a student is quite small compared to an experienced professional. Regardless, the maintainer will only continue to develop the initial version of the program if the cost of doing so is equal or less than the expected outcome (his particular use-product).

It is beneficial to use the classification of programs by Brooks [1995] to understand the costs and benefits faced by a maintainer. In opposite ends of the scale of total cost of developing a program we find the little program and the programming systems product. The little program is a small idiosyncratic program, and the programming systems product is the complete program, which is well documented and tested. The two programs contain the same number of features and use-products for the maintainer, but the cost of the programming systems product is nine times that of a little program.

When a maintainer creates a program to solve personal need, it will be a little program given the cost of expanding the program into a programming systems product. The maintainer will have to invest nine times the effort in order to transform his little program into a programming systems product, which he may sell and profit from. A maintainer then faces a situation prior to distributing a program, where he must ask himself if he is willing to make a further investment in the program before distributing the program. If he is willing to make the investment and believes the program has sufficient sales potential, also considering the competition, he must wait and distribute under the MS EULA.

However, as the program is developed to solve a particular problem and provide a particular use-product for the maintainer, the profit foregone by distributing the program is close to zero when distributed as a little program. As a little program, it has no profit potential and must turned into a programming systems product in order to be sold.

The maintainer can on the other hand expect that if he distributes the program and it is adopted by user-developers who make modifications, he will receive a benefit. User-developers will enhance their personal use-product, and if it is overlapping the personal use-product of the maintainer, he will receive a benefit, the value of which is equal to the opportunity cost of the maintainer having implemented the use-product himself.

7.4 The BSD License

The BSD license is the last and the most permissive of the three licenses allowing even greater degrees of freedom than does the GPL. The BSD license distinguishes itself from the GPL by allowing the user-developer to distribute his modifications and a modified program under a different license altogether.

In terms of the model of software development and consumption, the BSD is similar to the GPL license with the addition of one choice for user-developers, who have decided to modify the program. Besides keeping the modifications private and distributing the modifications, the BSD license introduces a third choice: Distributing modifications as closed source. This additional choice of being able to distribute the modified program under a different license has implications for our understanding of the maintainer and user-developers. The additional choice can be seen in Figure 9

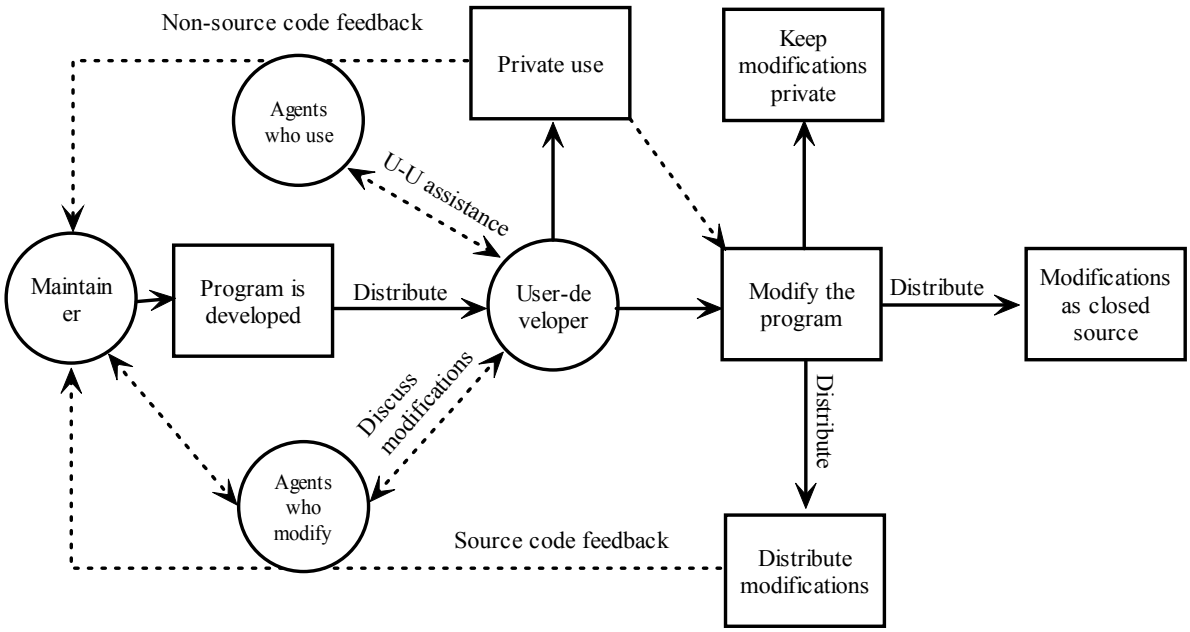


Figure 9: The BSD license introduces a third choice for user-developers, who make modifications to the progam: Distribute modificaions as closed source.

Given the complete model for software development and consumption under the BSD license we now turn to discuss the incentives for user-developers distributing modifications as closed source. The conditions for providing non-source code feedback is almost identical to the situation under the MS EULA, and the dissimilarities will be discussed first.

The analysis will be restricted to analysing the difference between the GPL and the BSD license. The two licenses are similar in all aspects besides the choice of distributing modifications as closed source. The following analysis will focus on this particular choice, and how it affects the maintainer, the user-developers and the dynamics.

7.4.1 Analysing the Maintainer

When a maintainer develops and distributes a program licensed under the BSD license as depicted in Figure 9, he does not immediately receive more choices than available under the GPL license.

The maintainer has the same incentives to participate in user-to-user assistance and to discuss modifications with agents who modify. The maintainer also incurs the same costs, when developing his program, as does the maintainer of a GPL licensed program.

The real difference is the ever-present opportunity of distributing the program as closed source. As the program is distributed under the GPL, the maintainer, as are the user-developers, is always in a position where he may choose to distribute the program as closed source.

However, the user-developers who have distributed modifications still retain copyright for their modifications and do not forfeit their opportunity to distribute their modifications in other ways.

7.4.2 Analysing the User-Developer

The user-developers receive an additional choice when obtaining a program distributed under the BSD license. User-developers may choose to distribute the program under a completely different license such as the closed source MS EULA license.

When doing so the user-developer changing the license also changes role and becomes the maintainer of the program, which he just distributed. The user-developers of this program in turn become user-developers of his program. The relationship between the maintainer of the original program, the maintainer of the distributed program, and its user-developers can be illustrated as in the following Figure 10:

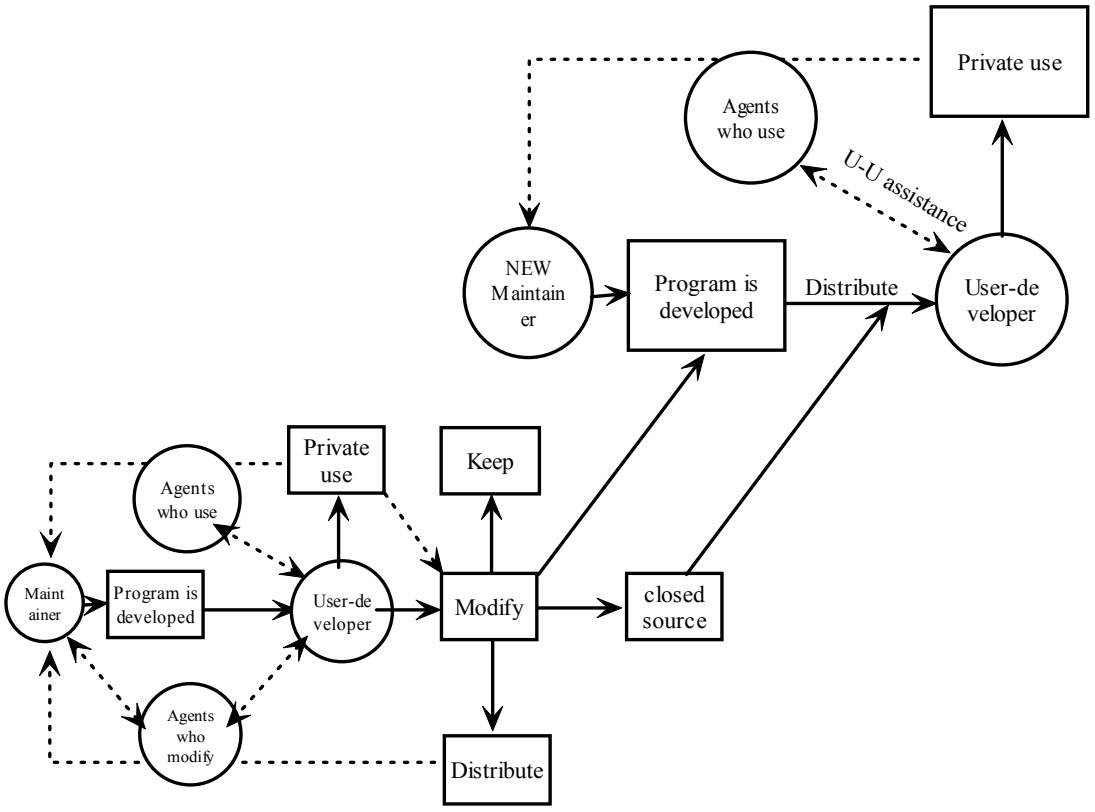


Figure 10: When a user-developer modifies a program and distributes the progam as closed source, he creates a separate instance of the model for software developerment and consumption. The two arrows between the two models illustrate how they are interconnected.

When the user-developer modifies the program to be distributed as closed source, he is entering the MS EULA model and becomes the “NEW maintainer”. Modifying the program compares to the “Program is developed” situation in the MS EULA model, and “Distribute as closed source” compares to “Distribute” in the MS EULA model.

After having distributed the new version, the relationship between user-developers of the new program is disconnected, as the source code is not available and the license prohibits modifications. The NEW maintainer may, however, continue to import modifications from the original program into his program.

The program distributed by the original maintainer begins as a public good, which is an externality. The original maintainer is in no way compensated, when the user-developer distributes the program as closed source and thus transforming the program into a club good. Indeed it is fascinating that the BSD license allows the one public good to be transformed into a club good without any compensation. The NEW maintainer may even continue to use modifications from new versions of the original program to improve his own program. It deserves to be mentioned that the GPL does not allow such behaviour. The viral effect of the GPL ensures that a modification from a GPL program inserted into a differently licensed program changes the license of the combined program to the GPL.

7.4.3 Dynamics – The BSD License

The BSD license has implications on the dynamics in the model. The dynamics of the GPL were tied to expectations formed between maintainer and user-developers. These expectations were tied to the fact that the GPL did not allow distribution of closed source

modifications. The maintainer and user-developers of a GPL program could rest assured that if someone made modifications worth distributing, the modifications had to be publicly available. Indeed the GPL license allowed user-developers to keep their modifications private, however the cost of maintaining them ensured that only few would do so.

The BSD license on the other hand allows user-developers to obtain a program, which is a public good and turn it into a club good without incurring any cost. This means that any user-developer may build a business on distributing closed source programs.

This changes the expectations between user-developers and the maintainer as a whole. User-developers may suspect that the maintainer or other user-developers are just waiting to capitalize from their work. It is easy to imagine the reaction from the maintainer and user-developers, who have distributed modifications, if a user-developer form a successful company based on the program. They are very likely to feel exploited.

Thus it must be expected that the BSD license, compared to the GPL, has lower incentives for user-developers to make and distribute modifications to the program. It is actually possible to imagine a situation, were user-developers make and distribute modifications to the program in anticipation of distributing the program as closed source when sufficiently matured. This would be a situation, where user-developers compete to be the first to market a closed source version of the program.

If a user-developer modifies and distributes a closed source version and becomes maintainer, he still has incentives to make contributions to the original program. It is known that maintaining a closed source program requires a significant amount of resources, and the maintainer will not receive modifications from user-developers. The maintainer may, however, still include modifications from the original program into his closed source version.

The maintainer of the new version still has an incentive to distribute some of his modifications back to the original program. For understanding why, it serves our purpose to distinguish between two types of modifications: 1) Modifications which adds a competitive advantage to the program and 2) General improvements.

Modifications, which add a competitive advantage to the new program could be an improved installation process that would let customers obtain their desired use-product without problems. Another example would be an improved user-interface, which would let customers obtain their desired use-product easily. In general it is only possible to identify modifications, which add a competitive advantage, by analysing the original program with the intended customer segment.

General improvements, on the other hand, do not add new features but will often entail improvements to existing features and fixes to defect features.

The maintainer will generally distribute modifications, which improve the infrastructure on which the new and added features rely. Having the modifications integrated in the original program lowers the cost of having to maintain these features personally.

This highlights the difference between the GPL and the BSD license. The GPL does not allow this kind of commercial exploitation of the original program. On the other hand the GPL thereby excludes itself from being used in places, where there are incentives to keep parts of the modifications private and distribute others.

The maintainer of a program faces a trade off when choosing the license, under which he wishes to distribute his program. The GPL ensures that modifications, which are worth distributing, have their source code publicly available. The BSD ensures that anybody may use the program for whatever they want, and they may (but do not have to) return their modifications to the maintainer.

7.5 Extending the Model

So far the model has had a simplified perception of the maintainer and the user-developers. In this section we extend the properties of the maintainer and the user-developer to briefly analyse, how this affects behaviour in the model of software development and consumption. It is further possible to distinguish between different types of programs depending on their degree of finish. When a maintainer has developed a program, properties of the maintainer and the program will influence the choice of license.

7.5.1 Two Types of Agents – Implications for Choice of License

In the analyses of the three licenses we neglected to take into account that the maintainer and user-developers cold both be either firms or individuals. It was explained in the previous chapter 6 that these two types of agents have different incentives and resources. Basically firms are profit maximising, and individuals are utility maximising. Given that firms are pursuing activities, which generate a profit, firms are also able to focus resources on obtaining these goals. Individuals need not generate a profit from their activities, and consequently individuals have limited resources, as generating income will consume a large portion of their productive time.

The following two sections will briefly discuss the implications of the agents in the model being either firm or individual and how this affects the choice of license.

7.5.1.1 The Maintainer as a Firm

Firms in general have no incentives to distribute programs, which they have developed under either the GPL or the BSD. Such a license would allow user-developers (customers) to use the program without paying anything to the firm. Naturally the firm has costs, which must be covered, and giving away presents does not cover them.

However, there are special situations where both the BSD and the GPL license might find use. The BSD license has its strengths in adoption, since anybody may use a BSD program, and there are no restrictions for future licenses and no obligations to return modifications. If the program implements a standard, it is beneficial to have the standard adopted by as many as possible, and the BSD makes this possible. (See the discussion of the increasing returns).

7.5.1.2 The Maintainer as an Individual

Individuals do not have a profit incentive but a utility maximising incentive, and thus the BSD and the GPL licenses are the most obvious choices. Individuals will benefit from releasing their program under one of these licenses. Both the GPL and the BSD provide the possibility of user-developers returning their modifications to the maintainer. These modifications contain added or improved features, which will benefit the maintainer. The maintainer must consider the choice of the GPL or the BSD against his wishes as discussed in the previous section.

7.5.1.3 User-Developers

In general user-developers, be it firms or individuals, face a cost when making modifications to a program. When the modification is integrated into the program by the maintainer, the user-developer incur no further cost from his modification. If, on the other hand, the modification is not integrated, the user-developer faces further costs when having to integrate his modifications into new versions of the program.

This provides a general pressure for user-developers to return their modifications to the maintainer. However, firms may derive a benefit in the shape of competitive advantage from their modifications to a program.

For instance, a firm is offering a web server service who develops a modification for the open source Apache web server doubling the capacity. This firm can now host twice as many customers on the same hardware because of the modification they have made. This modification effectively lowers the operating costs of the firm. This firm has an obvious incentive to keep the modification private in order to keep their competitors from gaining the same advantage.

Individuals, on the other hand, have an incentive to distribute their modifications, as this is the only way to enjoy some of the signalling effects from their work. Individuals are also more susceptible to costs arising from having to re-implement their desired feature and are thus interested in having their modifications integrated into the next version of the program.

7.5.2 A Little Program / The Programming Systems Product

It is illustrated in the model of software development and consumption that a maintainer develops a program, which is then released under a license of his choice. The type and size of the program has been touched upon in section 7.3.5, where the analysis noted that the development of a program should be seen as a process beginning with the maintainer distributing a program. Because the program, at that time, holds no promises for future sales there is no potential profit loss.

The developed program and the investment put in developing the program has implications for the choice of license. Brooks [1995] distinguishes between four types of programs 1) A little program, 2) A programming system, 3) A programming product, and 4) A programming systems product. Here we restrict our focus to type 1) A little program and type 4) A programming systems product.

The difference between the little program and the programming systems product is not the functionality, i.e. the number of features, but the degree of finish. A little program is a program that functions well within the confinements of the developer. The little program runs in an environment designed to support the program, and the little program is difficult to use outside the special environment.

The programming systems product is a well-structured program, which is designed to run in a variety of different environments. The maintainer can easily extend the programming systems product, and user-developers have no problems obtaining their desired use-product. The programming systems product on average requires nine times the development effort of the little program [Brooks 1995]. The programming systems product is essentially a full-fledged packaged program, which can be sold, and customers will find their desired use-product readily contained in the program.

We now imagine two different situations: 1) the maintainer has developed a little program and 2) the maintainer has developed a programming systems product. Both maintainers must to decide under which license their program should be released.

A maintainer with a little program has created a program that is functioning in his environment, and it is providing a desired use-product. The cost of the program has been the opportunity cost of the time spent developing the program. The benefit from the cost, which he has incurred, is the desired use-product. The program has little or no value as a product to be sold. If a customer were to buy a little program he would be faced with solving all sorts of problems simple because their environment (i.e. computer set up) being different from the maintainers.

Therefore the maintainer will not forego a large potential income by not distributing the program as a club good, i.e. under the MS EULA. If released under either the BSD or GPL license, the maintainer might receive improvements to his program from user-developers. This is a potential benefit from distributing the program under one of the open source licenses. Of the two open source licenses the GPL provides a higher probability of user-developers returning their modifications. The GPL further provide insurance against other user-developers redistributing the program under a different license and profiting from his work. The GPL ensures that the public good remains a public good and cannot be turned into a club good. In this situation the GPL license will be the license, which ensures the highest benefit for the maintainer.

A maintainer with a programming systems product has a program, where user-developers can expect to easily obtain their desired use-product. This provides an incentive to distribute the program under the MS EULA to make sure that user-developers pay for the program.

Interestingly, if the maintainer releases the program under the GPL, he must expect a lower degree of returned modifications than for the little program. If the program is functioning well, and user-developers can obtain their desired use-product, there is no incentive to make modifications to the program.

The BSD license is the worst choice, as other user-developers will immediately observe the quality of the program and redistribute the program under a closed source license.

7.6 Summary

This chapter has developed a model for understanding software development and consumption and the relation between three software licenses. The model makes a distinction between the maintainer and the user-developers. The maintainer is the firm or individual, who is responsible for releasing new versions of the program. User-developers are firms or individuals, who obtain the program in order to obtain a desired use-product. User-developers have the special characteristic that they may be users, but they may also decide to make modifications to a program in order to add or improve features so that they may obtain their desired use-product.

The model illustrates how the maintainer distributes a program under either the MS EULA, the GPL or the BSD license. Depending on the license, user-developers have different options. The MS EULA does not allow user-developers to make modifications to the program, which the GPL and the BSD do.

The model builds on the assumption that the maintainer and user-developers have a reason to interact because they each desire a particular use-product, which are contained in

the same program. If a desired use-product is not available in the program, user-developers will either contact the maintainer or other user-developers.

A use-product is the individual user's specific combination of features in the program. If a use-product cannot be obtained, the reason could be that a feature is not present, or the feature is not functioning correctly. Depending on the license, user-developers may be able to add features, fix features, or use a workaround to obtain the desired use-product.

The difference between licenses can be observed in the following figure, which uses colour coding to illustrate the possible behaviour depending on the license.

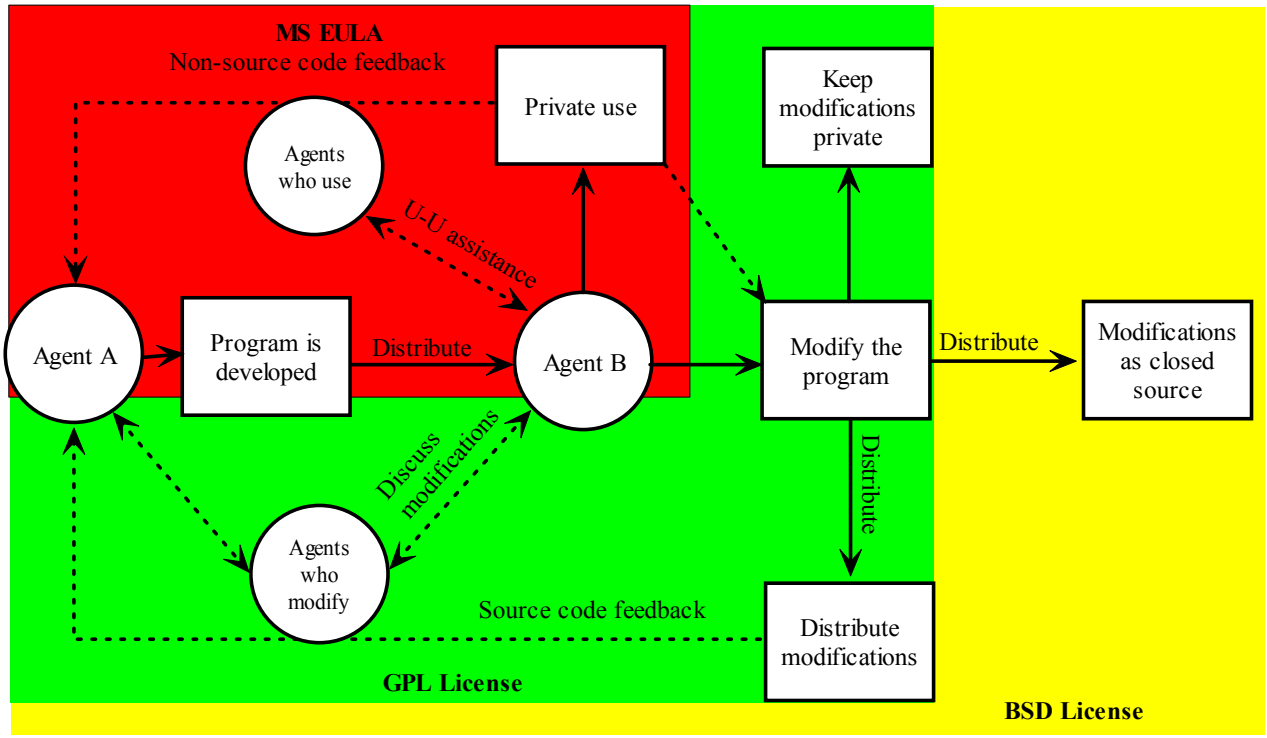


Figure 11: The Model for software development and consumption with colour coded licenses.

The model illustrates how different licenses provide different incentives for the maintainer and user-developers of a program. In general the maintainer and user-developers incur a cost when making modifications to a program. Both the maintainer and user-developers have incentives to make modifications, if they cannot obtain a desired use-product. Besides the personal incentives, maintainer and user-developers may derive a signalling benefit from making and distributing modifications.

The model accounts for the incentives, which lead maintainer and user-developers to distribute modifications as public goods under either the GPL or the BSD license. Further the model must be viewed as a process starting with the maintainer distributing the first version of his program. By distributing the program the maintainer creates an externality for all the user-developers, who later obtain the program.

The program is a little program, which provides the maintainer with a desired use-product. As a little program, is has no significant economic value, and thus the maintainer does not face a potential revenue loss by distributing the program under either the GPL or

the BSD license. In the same vein there are no incentives to distribute the little program under the MS EULA. It is more than likely that the maintainer at the time of distribution does not even perceive the little program as having any potential commercial value at all.

User-developers, who obtain the program, desire a use-product contained in the program If the desired use-product is not obtainable, the user-developer has an incentive to modify the program and implement the missing feature or fix the faulty feature. Having done so, the user-developer can choose, depending on license, between 1) keep modifications private, 2) distribute modifications or 3) distribute modifications as closed source.

For our purpose the most interesting choice is the second, distribute modifications, as this is central to open source software development. There are incentives to distribute the modifications, the first of which are signalling effects. However, signalling effects only account for a diminishing part of the motivation to distribute modifications. More important is the cost of making modifications, which will have to be paid again and again for each new version of the program that does not contain the desired use-product. With more than one user-developer, the pressure for distributing modifications before new features are added raises.

8 Increasing Returns to Adoption - Implications for Agents

The model for software development and consumption is capable of explaining the relationship between a maintainer and the user-developers using and developing a program. The model explains how different licenses create different incentive structures, and how these provide an incentive for distributing modifications thereby explaining, why open source software is being developed.

However, while powerful, the model of software development and consumption is merely capable of explaining the incentive structure for agents involved in using and developing a single program. The model does not explain the incentives, which makes a user-developer prefer one program to another.

This chapter elevates the analysis to a level capable of analysing the incentives for user-developers, who must choose between a number of other programs to use and/or develop. We imagine a situation where user-developers enter a market, where a number of maintainers offer their program. Some maintainers offer their program for sale under a MS EULA type of license, and other maintainers offer their program under either the GPL or the BSD license. In the latter two instances the programs are not sold, and user-developers can freely obtain and use the programs. We are interested in analysing and understanding what influences a user-developers choice of program.

The theoretical approach used for this is the theory of competing technologies developed by Arthur [1991], in which he analyses how properties of a technology affect its adoption (see chapter 4 for a description of the theory). Here we are interested in understanding, how the license properties of a program influence adoption.

The reason for choosing Arthur [1991] for this approach is that programs exhibit network effects i.e. the value of a program is tied to the number of users. One can be persuaded hereof by thinking of Microsoft's omnipresent Word program. Users of MS Word can easily exchange documents, while users of other word processing programs are incompatible and have problems using files generated by Word. These users are put to considerable disadvantages and expense, as they cannot access the content of MS Word files. This is what Arthur [1991] refers to as increasing returns to adoption, i.e. the observation that the value of a good increases proportionally more with the number of agents adopting the program.

When using Arthur [1991] to analyse one or more technologies, the most important analysis is to determine the technologies belonging to one of three returns regimes. The first section discusses and extends the criteria, which make a technology belong to one or the other returns regime. The following three sections analyses each of the three licenses and their belonging to a returns regime, and how this is expected to influence adoption. Lastly the main findings are summarised.

8.1 Criteria for Belonging to a Returns Regime

The central argument when using Arthur [1991] to analyse the adoption process of competing technologies is to establish the returns regime of a technology. In this chapter it is assumed that a program, in which this thesis is interested, is comparable to Arthur's

notion of a technology. Arthur [1991] is not specific about what causes a technology to belong to one or the other returns regime, and he restricts himself to saying that technologies exhibit this behaviour. This forces us to analyse a technology i.e. a program and evaluate if adoption causes the value of the technology to increase for the next adopter.

Arthur [1991] argues that a technology will adhere to one of three returns regimes, depending on how adoption of the technology affects the value of the technology for the next agent adopting the technology. The three regimes are 1) Constant, 2) Increasing, and 3) Decreasing (See chapter 4.3 for a thorough description). Constant returns mean that the value to the next adopter is unaffected by previous choices. Increasing returns mean that the value increases with each agent adopting the technology. Decreasing means that the value decreases with each adoption. It is imperative to understand that Arthur's [1991] argument is directed at the user and adopter of a technology. For instance, for an increasing returns regime the value of a technology increases with the number of other users and has nothing to do with the price of the technology. Some authors have argues that this is the fundamental problem of Arthur's [1991] concept of increasing returns to adoption. Liebowitz, & Margolis [1994] argue that there is no such thing as increasing returns to adoption and the effects observed is actually economics of scale where production prices go down as demand goes up. I, however, believe that the truth lie somewhere in between and in this particular case the perspective presented by Arthur [1991] appear to offer greater explanatory power and I will continue to use this perspective.

Programs and software in general represent what Katz & Shapiro [1994:94] refer to as hardware/software systems. Hardware/software describe a situation, where only some software can be used with a particular hardware. For instance, the two video systems VHS and Betamax each represented a hardware/software system, and software for the VHS system (video cassettes with content) could not be used with the Betamax system. This led consumers to consider availability of software as an integral part of the choice of hardware system.

Katz & Shapiro's [1994] hardware/software systems are by no means limited to just hardware and software. Hardware/software systems describe a much more general phenomenon, where dependencies exist between the elements consumed. For instance, a person, who possesses a number of data files in a particular format, is dependent on a particular program to access the data. This is referred to as compatibility regimes, where compatibility among elements exists within a regime. Dependencies can and do exist between programs, for instance, a program designed for Microsoft Windows cannot readily be executed on a computer running the Linux operating system. Dependencies in a compatibility regime exist on many levels, and three major levels exist for computers and programs: 1) Hardware, 2) Operating systems, and 3) Applications. The operating system must be designed to use the underlying hardware, and likewise applications (programs) must be designed to use the underlying operating system.

Whereas hardware/software systems refer to the technical dimension of a compatibility regime, communications networks refer to the social dimension of a compatibility regime. Communications networks describe a situation, where a technology is used in the process of communication, and this influences the value of the system. For instance, the telephone network is an example of a hardware/software system, where the communication is all-important. The value of the telephone system is tied closely to the number of users who are part of the system.

Programs may also possess the social dimension of a compatibility regime, if they are used as a part of a communication process. A word processor used for writing of texts,

stores the resulting text in a file, which has a particular file format. When a user decides to communicate the file to a friend, the word processor becomes part of a hardware/software system and a communications network. If the friend is not using a word processor capable of reading the file, communication fails, because the two agents are not part of the same compatibility regime. The receiver i.e. the friend must posses a program capable of reading the particular file format in order the display the data contained in the file.

Needless to say, such experience causes grief, and the agents incur additional costs. Either communication must be terminated, in which case the miscommunication becomes a sunk cost, or the two agents must find a common compatibility regime that permits the communication. In the latter case both agents incur an opportunity cost from having to negotiate a common compatibility regime. The effects from communications network are severe, and the reason easy to understand: Being part of the same communications network substantially lowers the cost of communication. Most computer users have had this kind of experiences where the sender uses e.g. the latest version of Microsoft Word and the receiver uses a version a couple of generations older, which cannot understand the files produced with the latest version.

Arthur's [1991] understanding of adoption of technologies under various returns regimes is well suited to analyse the considerations, which a user-developer must take into account when choosing which program to adopt. We have just learned that programs belonged to a compatibility regime, and that there were cost advantages for agents to belong to the same compatibility regime.

In the following sections we analyse the sort of returns regime produced by each of the three licenses. In the analysis only programs, which form a hardware/software system at the application level is considered. Thus we analyse a program, which is able to store data in a particular file format. But, before we turn to the analysis we briefly discuss what a file format is.

8.1.1 What is in a File Format

In this thesis the term "file format" denotes the particular way in which data, e.g. text, is stored in a file on a computer. A few examples will illustrate the meaning without having to become too technical. A text file is a file whose contents is not encoded and can be read using the host operating system without use of intermediate programs. A text file is stored on the computer in plain ASCII, which is a standard describing how to translate 7 or 8 bit characters into a letters.

The problem of text stored in plain text i.e. ASCII is the fact that 7 bit ASCII only provides 2^7 combinations, which is equal to 128 combinations each representing a single character. 7 bit ASCII is barely enough for the Danish letters and symbols. Such a file format does not allow for storing of additional information about the text formatting. Text formatting is the way in which the text represented in a document with bullet points, headings, italics etc. The problem of storing formatting information can be solved in two ways: 1) Mark up language or 2) Binary file formats.

Mark up languages, such as HTML, store data in plain text and additional information about formatting is placed within special tags. A program can then interpret these tags en display the text with the proper formatting. For instance, if a word is to formatted with a **bold** typeface in a HTML file it would be written as follows: <B>bold</B>. An abundance of tags exist for formatting HTML documents in many different ways.

Binary file formats encode the text data using special characters rendering the files unreadable by other than compatible programs. Microsoft Word is and example of such a program where a file stored in Word cannot be read as plain text. If one was to attempt reading a word document as plain text the data contained in such a file would appear garbled and incomprehensible.

A simple file containing the word text stored in plain text has a size of 4 bytes equal to the four characters used. A file containing the same text stored in Microsoft Word has a size of 27.648 bytes. The difference is testimony to the additional information stored in the word format, which cannot be part of the plain text file. The Microsoft Word file contain all sorts of information about the formatting of the document, version number, etc., which is hidden from the user.

The essence of this section is that file formats have great importance for the availability of data. If one does not posses the proper program, one cannot open a file and access the data contained.

8.1.2 The MS EULA License

It is evident that a program that is able to store files containing data immediately forms a compatibility regime between the program and the file. If the files are used for communications purposes, the program also forms a communications network.

The value of the hardware/software system is tied to the value of data, which is stored in a particular format. This can also be measured as the cost of converting the files to a different format. The value of being able to communicate increases with the number of users of the program. This is equal to telephone networks, where the value of being the only one with a phone is limited; however, as more and more agents get a phone, the value of the phone increases. The program, unlike the phone, has a value in itself, as the user is able to process and store data.

These properties lead to increasing returns to adoption and user-developers becoming locked-in to the particular program. User-developers cannot use their data files with a different program, as other programs are not compatible with the file format. This leaves user-developers with high switching costs, as the data files will have to be recreated in a new program. Such costs can be prohibitly high and prevent user-developer from switching programs.

It is the closed file format that ties a user-developer to a particular program. Other user-developers will experience a similar lock-in, and this is further re-enforced as user-developers communicate using the file format. This has the effect of creating inertia in the group of user-developers communicating. If one user-developer switches program and accepts the cost of converting or loosing his data files, he must further face the cost of having problems exchanging files with his peers.

We can then conclude that a program licensed under the MS EULA using a particular and closed file format exhibit increasing returns to adoption, in particular if the files are used for communication purposes. Given two comparable programs offering similar use-products and licensed under the MS EULA, user-developers will choose to adopt the program with the largest installed base, as this offers the largest communications network.

8.1.3 The GPL License

In this section the GPL license is analysed, the GPL license allows user-developers access to the source code, and this has major implications for the mechanisms, which induce the returns regime formed by the program. A program licensed under the GPL may use a special file format to store data generated when using the program. However, as the source code is available, it is always possible to inspect the source code and see the layout of the file format. The GPL and the BSD license are identical in respect to the returns regime formed and all conclusions regarding the GPL license are identical for the BSD license. For this reason a separate analyses of the BSD license will not be undertaken.

This makes it impossible for user-developers of GPL licensed to be locked in by a particular file format. Under the GPL license it will always be possible to obtain the full specification for the file format by reading the source code. This information can be used if a user-developer wants to switch to another program. Naturally, it is not possible to obtain the specifications for the file format of a program licensed under the MS EULA as the source code is not available.

The number of user-developers making modifications are important and a large number of user-developers making modifications is a clear indication of the program being continuously developed with new features added. This in turn allows user-developers to expect the number of use-products to raise and increase the benefit of the program.

The number of user-developers participating in user-to-user assistance is also important, as this provides an indication of the level of support, which a new user-developer can expect. If there is a very active discussion forum, a new user-developer may expect that it is easy to get help, and that he can obtain his desired use-product at a low cost. This is contrasted to a program, where there is no active discussion forum, and a new user-developer may expect to pay a significant learning cost to obtain his desired use-product.

Thus given two GPL licensed programs competing for adoption, we will observe increasing returns to adoption. However, the mechanism is different from the MS EULA, as this type of license cannot form closed compatibility regimes. The mechanism is created from new adopter (user-developer), who desires a particular use-product. If the use-product is contained in the program, a new user-developer will prefer a program with a forum, which is large and can offer help with obtaining the desired use-product. When the new adopter approaches the forum, it generates more discussion, and potential adopters will be more attracted.

If a desired use-product is not contained, a new adopter will look for a program with a large number of user-developers, who make modifications to the program. The adopter will choose the program, because he may expect his desired use-product to be available soon, based on the pace of development.

The searchable nature of information stored in web-based forums further enhances the impression of a program being more used and thus more appealing then another. Discussion forums are searchable, and therefore the question frequency will fall, as user-developers get their questions answered. When new user-developers search the forum for similar problems, most of the common answers will be given without having to post a question to the forum. When new features are added to the program, problems will arise again, and questions related to new use-products must be answered. This has a positive effect on the increasing returns regime created because user-developers will be able to

instantly access answers to questions already answered without having to wait for other user-developers to answer their question.

The analysis of the increasing returns regime might give the impression that when lock-in is achieved more and more feature are added and the program continues to distance itself from the competition. There are one property of open source software development that counters this effect and ensure that a single program does not become the omnipotent dominator.

There are limits to the number of use-products in a program! A small chat program is expected to provide features and use-products allowing a user-developer to chat easily. A small chat program is not expected to act as a large spreadsheet. Consequently, there are limits to the features, which user-developers will expect to be present in a program. This has implications for user-developers' motivation for making modifications to a program. As a program is gradually considered more and more feature complete (for the given type of program), the motivation for making modifications declines. This mechanism is particularly expressed in programs licensed under the GPL, as the programs are freely available, and as they are freely available, there is no incentive to add features that provide no personal benefit for the user-developer, who makes the modification.

In contrast, profit maximising firms have an obvious incentive to invent new features, regardless of their relevance, in order to persuade users to buy the upgraded version. As programs do not wear out, firms will have to sell upgrades or new products in order to keep generating revenue.

We can then expect programs distributed under the GPL to reach a level of maturity, where the program performs as expected, and where the expected use-products can be obtained.

8.2 Competing Licenses

We have now established how the individual licenses affects the returns regime formed by a program. In this section we ponder the interesting question of what happens, when two programs competing for adopting are licensed differently. The one program is licensed under the MS EULA and the other under the GPL or BSD (we assume a GPL program although the choice in this context has no consequences). We imagine a situation where two programs are distributed on the market, and user-developers line up for adoption. The two programs are not equal in terms of price. The MS EULA program costs money, and the GPL program is free. Over time both programs will evolve features and accumulate use-products with the distinct difference that in the GPL program it is the maintainer and user-developers that makes modifications and add features.

The outcome of an adoption process for technologies (in this case programs), which exhibit increasing returns to adoption, is unpredictable and depend on the adoption sequence [Arthur 1991]. We can imagine two distinctly different adoption sequences: 1) A significant majority of GPL adoptions in the beginning and 2) A significant majority of MS EULA program adoptions in the beginning.

From the Table 2 in the theoretical chapter (Chapter 4) we now rewrite the matrix and replace technology A and B with program A and B. Program A is licensed under the program and program B is licensed under the MS EULA. R-agents have a natural preference for program A (GPL) and S-agents have a natural preference for program B (MS EULA).

	Program A, GPL	Program B, MS EULA
R-agent, GPL	$a_R + rn_A$	$b_R + rn_B$
S-agent, MS EULA	$a_S + sn_A$	$b_S + sn_B$

Table 5: Arthurs' [1991] table modified with programs in stead of technologies.

If agents line op for adoption in such a way that all agents have a personal preference for the GPL program i.e. R-agents, then some of the user-developers will add features thus making the program more appealing to the following agents. This gives the following adoption sequence: R,R,R,R,R,R,R,R,R,R. The sequence is random and adopters could have lined up in all sorts of other combinations. However, this particular line up is interesting as it illustrated a particular property of the increasing returns regime: the element of chance. Arthur [1991] is very explicit when discussing the unpredictable nature of increasing returns regimes. A particular line up of adopters, like the one above, may posses the power to tip the scale in favour of the one technology, which will become the dominating technology.

Following the first 10 adoptions of program A an S-agent lines up and is faced with deciding between program A and B. The agents utility from choosing program A is $a_S + s \cdot 10$ and his utility from program B is $b_S + s \cdot 0 = b_S$, which is his natural preference for program B. We know that his natural preference for program A is lower than his natural preference for program B and the question becomes whether the positive returns factor, s, multiplied with the number of adoptions will make the utility of program A larger than program B. If this is the case a lock-in has been archived.

At this point in time the value gained through increasing returns to adoption will outweigh personal preference and all agents will choose GPL. The price of the programs must also be taken into account, and given the fact that the GPL program is free there will be a personal preference for adopting the GPL program.

In this situation there is a dominating system, but agents are not locked-in to a particular file format, as they could use the source code to learn the nature of the file format. The maintainer of a competing program could use this information to add a feature, which allows the data files to be used on the competing program.

However, a program licensed under the MS EULA has a cost, which user-developers must pay for adoption. Given two programs equal in features and use-products, user-developers will naturally choose the one costing the least. To compete for adoption, a MS EULA program will have to offer significantly more features and use-products in order to persuade user-developers to adopt the program. Given a significant head start, it becomes unlikely that a MS EULA program will be able to change the adoption situation.

If agents on the other hand line up in a sequence, where agents prefer the MS EULA program, there will also be a lock-in. This lock-in is different and relies on the file format, which agents are locked-in to. Agents will also be tied to a communications network, which will be made compatible with the GPL network. A competing program will not be able to read the source code and implement a feature allowing the competing program to use the files from the MS EULA program.

8.2.1 Overturning a Dominating System

We have just made two observations:

- A GPL program is capable of creating a lock-in due to expectations of support and new features, which is directly related to the number of user-developers. The price mechanism also plays a role because it ensures that user-developers will adopt the cheaper of two comparable programs, i.e. the GPL program. In order to compete a MS EULA program will have to offer substantially more features to warrant adoption. It is not possible to create a lock-in based on file formats due to the availability of the source code.
- A MS EULA program is capable of creating a lock-in due to hardware/software systems and communications networks based on the closed file formats. A competing GPL program cannot become part of the hardware/software system or the communications network because of the closed file format.

It is evident from these two observations that the GPL creates what might be described as a soft lock-in. It is soft in the sense that given enough resources a competing program can be developed containing enough features that it is more appealing than the GPL program. The GPL program is vulnerable because the file format can be implemented in the MS EULA program. Thus the MS EULA program can become a part of the hardware/software system and communications network originally formed by the GPL program. The MS EULA program can, however, not become part of the network of user-developers who make modifications.

The MS EULA creates what can be described as hard lock-in based on the file format, which cannot be implemented in the GPL program. However, as we shall now see there is a way in which a GPL program can grow into becoming a competitor to a dominating system.

It has been stated over and over again that a program consists of a number of features, which in combination define a number of use-products. We now imagine a situation where lock-in has been achieved for a MS EULA program and this program contain a number of use-products. But, a user-developer only desires a small fraction of the features for his personal use-product and will not pay the demanded price for the MS EULA program. This user-developer has an incentive to obtain his desired use-product from another program or create a program with the desired features by him self.

If he creates a little program licensed under the GPL it is aimed at providing the functionality, which is present in the MS EULA program and as such the little program constitutes a competing program. When the first version is developed the barriers to enter the process of adding features to the little program has been lowered significantly. Other user-developers who are not interested in paying the MS EULA price will be able to add their desired features in order to obtain the desired use-product.

This is a process of two competing systems where the inferior system competes on price and user-developers are adding features. To the user-developers of the little program the two programs provide the same use-products, as they only desire a subset of the features present in the MS EULA program.

The major problem for the little GPL program is the fact that it is not part of the hardware/software system or communications network of the MS EULA program. In order to become part of this, the little GPL program must become compatible with the file format of the MS EULA program. Compatibility can be archived in the same way as GPL programs are normally developed: User-developers and the maintainer gradually adding features to the program. While developing the compatibility feature might be a different

process as it relies on reverse engineering of the desired file format, it is still just more features added to the program.

The process of reverse engineering is the same from a development standpoint because compatibility with a file format is not an atomic feature. Compatibility can be implemented gradually as the need arises. A user-developer might desire the ability to just read the text in files from the MS EULA program and not care about any formatting. Other user-developers might desire more features such as being able to read footnotes and thus experiment and implement this feature.

The model of software development and consumption for the GPL license adequately explains the process of catching up with a dominating system. This leads to the conclusion that a competing GPL program will always be able to compete on distinct use-products. Given the fact the GPL licensed program is free, there is an incentive for user-developers to use such a program, if the desired use-product is available. This ignites the process of modifications, where the competing GPL licensed program is gradually improved. The competing program will not be able to completely implement the dominating file format, although reverse engineering will allow for some functionality to be achieved.

The conclusion to be drawn is that MS EULA programs will be able to dominate a market consisting of only MS EULA licensed programs. If GPL licensed programs exist on the market, a dominating MS EULA program must expect to loose adoption shares to the GPL licensed programs.

This analysis has only illustrated the rough characteristics of the adoption processes, and further research is needed to estimate adoption shares in the two described adoption sequences. It is expected that natural preferences have a large impact on the adoption of GPL licensed programs. However, it would be interesting, if the adoption of such programs would show signs of being able to catch up with a dominating MS EULA program. As mentioned, this is pure speculation and questions for further research.

8.3 Summary

Chapter 7 explained how the maintainer and user-developers of a program interacted and the incentives for making modifications and distributing the modifications. The explanation highlighted two types of dynamics arising from the interaction between a maintainer and several user-developers leading to new features added to a program.

This chapter has focused on explaining the behaviour of agents at a level above the individual programs. Before user-developers obtain a program and either use or modify a program, user-developers engage in a selection process. This chapter has argued that this selection process resembles an adoption process of competing technologies, and therefore that theory may be applied. The selection process resembles competing technologies, because both the MS EULA and the GPL/BSD licenses create compatibility regimes based on hardware/software systems and communications network, which in turn generate increasing returns to adoption.

This is basically a market perspective, where programs offering the similar use-products compete for adoption by user-developers. User-developers do not randomly choose which program to adopt but are subject to these selection processes. The key understanding of the selection processes is that user-developers seek to maximize their utility from adoption. The utility function in this perspective is composed by natural preference, and the number of users has previously adopted the program. Simply put:

Given two programs exhibiting increasing returns to adoption, the program with the largest number of users will be adopted.

The MS EULA and the GPL/BSD have, however, not found their compatibility regimes on the basis on the same mechanisms. The MS EULA is based on hardware/software systems and communications networks. The file formats of these programs are closed, and the user-developers become locked-in to the program, as data stored in the files become a sunk cost, if the user-developer changes to an incompatible program. If the files are used as part of communication, a communications network is formed, which is a powerful increasing returns generator. A user-developer, who is unable to communicate, is willing to pay a high price to become part of the communications network.

The GPL/BSD licenses cannot lock user-developers into a closed file format, as the source code is available. Maintainers (firms as well as individuals) of competing programs will always be able to add a feature to their competing program, so that it reads the files of the GPL/BSD licensed program. But increasing returns to adoption exist, as each new user-developer will potentially add features to the program. The more user-developers adding features, the higher the probability of a desired use-product being added. User-to-user assistance is also important, as an active forum increases the likelihood of a user-developer being able to obtain his desired use-product without having to invest significant learning costs. Thus it is the expectation of added use-products and ease of obtaining a desired use-product, which form the basis of increasing returns to adoption.

The outcome of an adoption process for technologies, which exhibit increasing returns to adoption, is unpredictable and depend on the adoption sequence. Still, we can imagine two distinct sequences, one where there is a significant majority of MS EULA program adoptions in the beginning and one with a significant majority of GPL/BSD adoptions in the beginning.

In case of the MS EULA the program will naturally dominate the market, and user-developers will become locked-in. However, if one or more use-products associated with such a program is not present or the cost is too high, it is very likely that a maintainer will emerge and create a little program offering such a use-product. Over time a program containing such a program may grow as user-developers adopt and add features to the program. Consequently the program grows into becoming a competing technology despite starting off far behind a dominating standard.

If, on the other hand, a GPL/BSD program becomes dominating in the beginning, it is unlikely that a competing program based on the MS EULA license will emerge. If use-products are missing, user-developers may develop the missing features, which will be less costly than developing a new program. A competitor will have to offer significantly more use-products in order to catch up with a dominating GPL/BSD licensed program. The MS EULA program has the advantage of being able to implement the file format of the GPL/BSD program and thereby become part of the hardware/software system and communications network formed by the GPL/BSD program. This is naturally a significant advantage for the MS EULA program. However it must be remembered that the MS EULA program must represent a significant advantage in terms of features and use-products in order to justify the higher price compared to the GPL/BSD program.

9 Empirical Evidence

This chapter presents the empirical evidence gathered in course of this project. The purpose of the empirical chapter is twofold: 1) Answer to the empirical research questions and 2) Presentation of empirical evidence to test the model developed in chapter 7. The chapter is divided into several sections, which gradually move from a historic general level down to a detailed level, ending with the results of the interviews conducted.

Linux and open source software as analysed in this thesis are the consequence of a lot of history, the specific details of which are irrelevant. However, the general history of Unix, and how it came to be, is interesting, and it illustrates a case of software development, which shares resemblance to open source software development. However, early development of Unix has the noteworthy difference of source code not being open source. Open source software is a descendent of the phrase Free Software coined by Richard M. Stallman and the Free Software Foundation (FSF) in 1982. Linux was the software project that came to symbolise Free Software and later open source software.

The chapter begins by describing the history of Unix and Linux from year 1968 to 2002, which is divided into five periods. The history of Unix is interesting and portrays open source software development as a consequence of a particular legal climate within AT&T (the American Telephone and Telegraph Company). It also shows, how Unix grew into a dominating operating system and later fragmented into incompatible Unix clones. The history of Linux is then presented. The history of Linux takes its beginning in 1991, when Linus Torvalds released the first version of Linux. The section presents historical highlights from Linux - a little more than 10 years of history. The history of Unix and the history of Linux are told chronologically.

From the historic perspective, we move a step further down and present a description of what open source software is, and how it is used, and obtained. This is a description, which contrasts usage of an open source software system compared to a Microsoft based system. It is the intention of this section to provide the reader with first hand impressions of how to use an open source software system. This section will use Linux to exemplify, as this is what the author is comfortable with. However, it should be noted that several other open source operating systems like FreeBSD, NetBSD, and OpenBSD exist, and could have been used as examples as well.

Knowing about the history and using open source software in general, we proceed yet further into the details of open source software development. The next section describes open source software development based on the conducted interviews. Different organisational forms of open source software development are described and exemplified with the words of the respondents of the interviews.

The next section presents a brief overview of some of the available statistics about open source software. The section refers some of the quantitative studies of open source software.

The chapter ends by presenting seven mini cases each containing one or two development stories. The mini cases refer to the individual interviews, and the development stories are the personal open source development experiences told by the respondents.

9.1 The History of Unix and Linux

The history of open source software is indeed interesting and has been researched extensively in this project. It is the purpose of this chapter to provide a basic understanding of the history of Unix and Linux. A detailed timeline of the history of Unix and open source software can be found in appendix 2.

The brief historical description of Unix, Linux, and open source software presented here is divided into five distinct periods. The first period is the early AT&T Unix Area from 1969 to 1976. The beginning of the second period is marked by the release of the first Berkeley Software Distribution (BSD). This distribution was free for those, who had an original AT&T Unix license. The third period marks its beginning with the birth of the Free Software Foundation and shows the fragmentation of the original Unix into incompatible operating systems. The forth period begins in 1991 with the release of the first version of Linux. The fifth and last period begins by making of the new "open source software" term and the subsequent release of the source code for the Netscape browser.

9.1.1 – 1976 Early Unix and AT&T – The Sharing of Software

In 1949 the Antitrust Division of the US department of Justice under the Truman administration filed a complaint against Western Electric Company, Inc. and the American Telephone and Telegraph Company. The claim was that the two companies were acting together and this was a violation of the Sherman Antitrust Act.

In 1956, after extensive negotiation, Judge Tomas F. Meaney decreed that AT&T and Western Electric were not allowed to engage in manufacture, sale, or lease of any equipment other than telephone or telegraph equipment. There were some exemptions to this decree, and an important one was that AT&T and Western Electric was allowed to perform experiments for the purpose of testing or developing new common carrier communications services [Salus 1994:56-57]. This exception made the development of the Unix operating system possible and is the reason behind the early licensing of AT&T Unix.

In the 60ties computers were beginning to develop. Computers of those days did not offer interactive computing like today, where we are accustomed to writing emails and browsing the web using a computer. But interactive computing was envisioned, and a project was formed to create an operating system that could accommodate these visions. The MULTICS project (Multiplexed Information and Computing System) was a joint research project between General Electric (GE), AT&T Bell Telephone Labs (BTL), and Massachusetts Institute of Technology (MIT). The project tried to create a multi user operating system, which could accommodate a thousand users at the same time. [Salus 1994:5].

The MULTICS system was eventually abandoned, because it failed to progress into the promised operating system. In 1969 AT&T withdrew from the project, and consequently the staff that had been working on the project was returned to AT&T, one of which staff was Ken Thompson. Although the MULTICS project failed to deliver an operating system, it succeeded in forming many valuable ideas.

Ken Thompson had developed a game called space travel for the MULTICS system, and he was looking to implement the same game on one of the computers at AT&T. Together with Dennis Ritchie they implemented space travel on first a GECOS and then on a used and available PDP-7 computer. Creating the space travel game required programming of an underlying operating system for the two computers.

Within about 6-9 month the operating system, which started out to support a game, was now showing a promising file system and some interesting tools [Ritchie 1984:8]. The operating system was called Unix as a reaction to MULTICS, and used many of the ideas invented as part of the MULTICS project. What was once a game had by 1970 evolved into a working operating system supporting three typists using the *roff* text editor. Although simple by today's standards, the editor did a god job supporting the typist job of writing and editing documents, and indeed it was a vast improvement on use of a typewriter.

Within AT&T, Unix had obtained a reputation for supplying interesting services, such as the *roff* text editor using only modest hardware. By 1971, Unix was growing fast, and the first version of the Unix programmers' manual was created. The programmers' manual documented all system calls, file formats, and everything about the Unix operating system. The release of the programmers' manual marked the release of what became known as the first version of Unix.

In the period from 1971 to 1976, the Unix operating system grew and was adopted in different parts of AT&T, universities, and also some firms were using it. Six versions of Unix were released in the period, and with each release the operating system grew with new features and improvements, which also extended into accommodating new hardware. The antitrust decree meant that AT&T could not charge for the use of the operating system, as Unix had to remain a research activity. Although AT&T could not profit Unix was distributed to a number of institutions, predominantly universities. The Unix versions distributed from AT&T were complete with source code that could be modified.

The various institutions using Unix were often modifying and adding features to Unix. Variations in hardware and needs meant that the underlying operating system and supporting software had to be modified. Many of the modifications were returned to AT&T and added to subsequent releases of the Unix operating system.

9.1.2 1977-1983 The First Free Unix Distribution and Fragmentation Begins

In 1977, the Computer Systems Research Group (CSRG) at the University of California at Berkeley (UCB) released the first version of the Berkeley Software Distribution, 1BSD. 1BSD was free to anyone, who owned a Unix license. This distribution was meant for use with Unix V6 on a PDP-11 computer. It was not a complete operating system, but mostly application programs. One of the programs was *vi*, the first full screen text editor [Quarterman & Wilhelm 1993:33], another was a Unix Pascal System (programming language) [Salus 1994:143]. About 30 copies were distributed of the first BSD release. By 1978 the second BSD was released and was still freely available to anyone with an AT&T Unix license, and this version vas also a series of add-on's for the AT&T Unix.

When AT&T released the 7th version of Unix in 1979, an announcement was made that had serious consequences for the fate of Unix. AT&T announced their intention to commercialise Unix. This meant that Unix was no longer to be distributed freely, and universities were not allowed to study the Unix source code. So far the source code for previous Unix versions had been studied carefully at universities, which in return had produced new features, programs, and had bug-fixes.

Berkeley, the developer of the BSD, reacted strongly by making the third release of the BSD a complete operating system, and thus forking the development of Unix. The fork meant that the two Unix versions were to follow different development paths and perhaps grow incompatible. Due to licensing issues, many other forks in the development of Unix

were to follow. The following figure (Figure 12) illustrates how the a variety of Unix operating systems with varying degrees of compatibility were developed following the announcement to change the licensing terms. The figure is very simplistic, and many other Unix-like operating systems have been developed but not included in the figure.

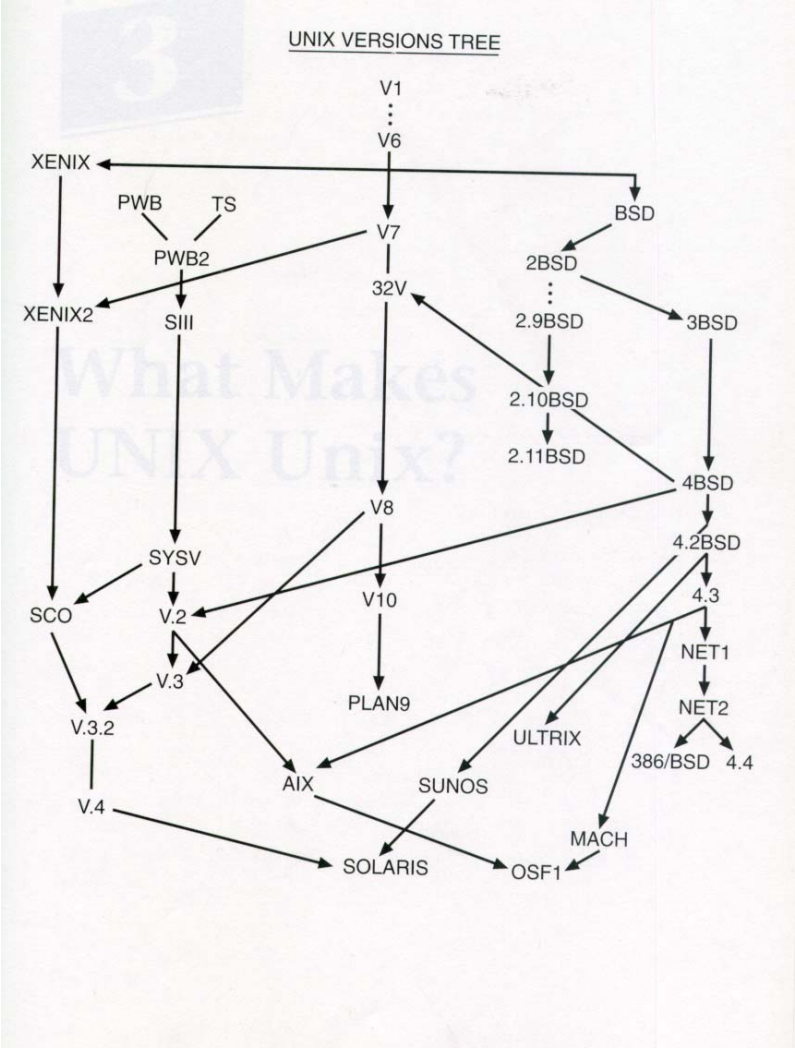


Figure 12: The fracmentation of Unix followed after Unix version 6 was released. This is a simplified figure only showing a selection of Unix versions [Salus 1995].

By 1983, the antitrust case against the AT&T was ended with a decree to split up AT&T into several smaller companies. Consequently, the restriction on selling other than phone equipment was lifted, and AT&T could now capitalize from selling Unix licenses. This was marked by releasing the commercial Unix System V that was developed by the Unix Systems Labs at AT&T. Same year saw the release of version 4.2 of the Berkeley Software Distribution (BSD).

9.1.3 1984-1990 The Free Software Foundation and More Fragmentation of Unix

The fragmentation of Unix continued, as many hardware vendors created their own proprietary version of Unix. IBM developed AIX, HP developed HPUX, SUN developed IRIX, and many other Unix-like operating systems were created, some of which were

based on the AT&T Unix. In this respect the period is no different from the previous period, although the number of Unix versions were growing much faster.

The important event this year (1984) was the creation of the Free Software Foundation by Richard M. Stallman [Stallman 1999]. Richard M. Stallman created the free software in frustration of the licensing terms of some of the software, which he was using. Stallman argued that software must be free and possible to extend and share among users. Richard M. Stallman began to create the GNU system. GNU is a recursive acronym for "GNU is Not Unix". The vision was to create a Unix operating system free of any of the codes, developed by AT&T or BSD, which had licensing issues not acceptable to Richard M. Stallman.

The Free Software Foundation had limited impact in its early years, but came later to be an important factor in shaping our understanding of what is free software. In their effort to create a free Unix, Richard M. Stallman and the Free Software Foundation created a large collection of tools, usually present in a Unix operating system, and made them available under the GPL license. The tools consisted among other things of the Emacs editor and the Gnu C Compiler elements, crucial to open source software development today. The Free Software Foundation and Richard M. Stallman managed to create almost all of the parts of a Unix operating system, but the kernel. Apart from the tools, the single most important product of the Free Software Foundation was the GPL License, which is the backbone of open source software licenses.

By the end of the period, Unix had fragmented into many different versions running on a variety of hardware but not always compatible. Most hardware producing companies had developed their own version of Unix based on the original AT&T or the BSD Unix. Companies such as Apollo, Digital Equipment Corporation, IBM, Compaq, HP, Intel, Microsoft, Motorola, and several others offered proprietary versions of Unix, but all of the offered Unixes required license from AT&T, as they contained parts of the original code copyrighted by AT&T or BSD.

9.1.4 1991-1997 Early Linux and the Spread of Free Software

In 1991 Linus Torvalds released the first version of Linux, and this becomes the first free Unix clone for PC class hardware. The only existing Unix-like operating system for PCs is the Minix operating system, developed by Andrew Tannembaum, who used it as a tool for teaching operating systems design. Minix was inexpensive compared to commercial Unix licenses, and even came with the source code. Linus Torvalds used the Minix operating system, but the educational status of the operating system meant that Andrew Tannembaum was not interested in developing Minix into a real operating system. A large community of users was actively developing Minix, and the result of their efforts was freely shared.

Linus Torvalds began developing Linux as just a hobby project to get to know the Intel 386 processor. Minix and the tools provided by the Free Software Foundation served as the basis for doing so. After months of very intensive programming, Linus Torvalds developed a functioning kernel. Being part of the Minix community, Linus Torvalds released the kernel and the accompanying source code by announcing the availability of his hobby project in the Minix news group. To Linus Torvalds' great surprise, many users began to take interest in Linux, and some of them even proposed changes and gave comments. Within a short period of time (6 month) Linux had an active user and developer base testing and helping development.

The development of Linux was done by Linus Torvalds, often releasing new versions of Linux on the University FTP[21]. Other users and developers could then download the latest version, as they pleased. Discussions regarding Linux were in a matter of month moved from the Minix newsgroup to a distinct Linux newsgroup. People, who had questions, or who tried to add features to Linux, discussed this on the newsgroup. Changes to the source code for Linux was also sent to the news group, and if Linus Torvalds approved the changes, they were incorporated in the next release of Linux.

This process happened quickly, and the evolution of Linux was fast paced. At one point in time the Linux newsgroup was one of the most active on the Internet. Linus Torvalds released new versions in an incredible pace, sometimes two new versions were released a day. Importantly enough, Linus Torvalds decided to release Linux under the GPL license ensuring that it was not possible for others to make changes to Linux and distribute them without making the changes public.

By 1993, Linux had reached such a level of usability that the first commercial attempt of distributing Linux was attempted. Patrick Volkerding marketed Slackware, which quickly became popular [Linux Journal 2002]. The Slackware distribution was a collection of disks containing a Linux kernel and supporting software such as compiler, editor, and other useful software.

Other signs of the raising popularity were the appearance of the first edition of the book “Linux installation and getting started”. The newsgroup, comp.os.linux, where all Linux related subjects are discussed, was now one of the top 5 most read newsgroups.

The interest in Linux was growing, and in 1994 Linus Torvalds was invited by Digital Equipment Corporation to New Orleans for a conference regarding Linux. During the conference Linus Torvalds was persuaded to rewrite (port) Linux to run on Digital’s Alpha processor.

Commercial interest in Linux was emerging, and the first Linux Expo was held in 1995. Same year a group of webmasters, who were tired of patching the NCSA httpd web server software, forms the Apache group and develops the Apache web server. The Apache web server is also freely available, but not distributed under the GPL license. The Apache license is very permissive and allows anyone to redistribute and modify Apache and even keep the modifications private. The Apache web server became the most used web server on the net.

While the period was very productive for Linux and other software, commercial interest was minor, and public knowledge of Linux in particular and of free software in general was almost nonexistent. The media coverage of the phenomenon was sparse, to say the least of it. It seemed as if Linux and Free Software lived a high profile life in closed technical communities, while the general public was completely unaware of the merits of the emerging competitor to Microsoft.

9.1.5 1998-now The Open Source Software Era

Almost over night Linux and open source software became household names. What had been known as free software changed name to open source software, and the event, which made the computing population aware of open source software, was the collapse of Netscape and subsequent release of the source code for their browser.

[21] FTP stands for File Transfer Protocol, which is a common and simple way of making files available on the Internet.

Eric S. Raymond met with Bruce Parens, Linus Torvalds, and a few others, and decided to adopt the term Open Source Software instead of Free Software. It was believed that the ambiguity of the term Free Software and the political rhetoric of the GNU manifesto was damaging to the corporate adoption of Linux.

Eric S. Raymond published "the Cathedral and the Bazaar", which became widely read, perhaps because the paper was instrumental in Netscape's decision to release the source code for the Netscape Communicator Browser. In March 1998 Netscape released the source code for the Communicator browser, and this draw attention to the term open source software to the mainstream press. Netscape and the release was discussed in the press and seen as yet another proof of the viability of the "new economy" as portrayed by Kevin Kelly.

Linux and open source software began to appear in the mainstream media, and the industry was beginning to take an interest in the phenomenon. IBM was one of the first large corporations to embrace Linux and open source software. In June 1998 IBM announced full support for Linux and Apache running on the IBM Netfinity servers. The large database vendors such as Oracle and Informix announced support for Linux.

The Halloween documents were released in November 1998 and caused even more focus on Linux and open source software and reached headlines in the press.

The stock market boom in 1999-2000 was very friendly with Linux and open source companies. In December 1999, the initial public offering of VA Linux took place and the price was $30 a share. The price reached $300/share, and the day closed at $250/share – the biggest IPO rise in the history of NASDAQ – Linux and open source software had achieved complete attention of everybody.

Linux was gaining momentum in the industry, and IBM announced plans to spend $1 billion on Linux. Other hardware vendors such as HP also announced support to Linux.

About a year after the VA, Linux stocks had fallen to $6,62/share [Torvalds and Diamond 2001:204], and the burst of the bubble economy was approaching

Although the bust of the bubble economy had put a damper on investments, open source software continued to be adopted. Wide adoption was happening in the server space, where Linux was second to only Microsoft Windows2000/XP, while desktop adoption was sparse. Linux distributors continued to make an effort to position Linux as a desktop operating system, however Windows users found lack of applications an obstacle for adoption. The only place, where Linux has been widely adopted on the desktop in the special effects movie industry. Most Hollywood companies have replaced their windows desktops with Linux desktops, mostly running Red Hat Linux.

9.1.6 Concluding the History of Unix

The history of Unix and Linux shares some resemblance in the way they were developed. Both started out as hobby projects, and both received contributions from other than the original group of developers. However, the change in licensing for Unix in the late 1970ties made it impossible for universities and other institutions to continue contributing to the development of AT&T Unix. The result was the fragmentation of Unix. Linux, on the other hand, has from the beginning been distributed under the GPL license, which demands that changes, made to the code and distributed, are made public under the same license. Thus even if one decides to make a competing version of Linux, one would be forced to make the changes available. Should these changes be of high quality, they could be integrated in the original version and thereby prevent fragmentation.

The pace with which Linux has been developed and adopted is amazing, and Linux and open source software still continues both adopting and development.

9.2 What is Open Source Software? – A Users' Perspective

This section will present open source software from the perspective of the user and illustrates some obvious differences between using a system (a computer with software) based on open source software and a system based on the offerings of Microsoft. Linux will be used as the open source software system, because this is what I am used to. Other open source operating systems like FreeBSD, NetBSD, or OpenBSD could also have been used, and the impression would remain the same.

Readers are expected to be unfamiliar with open source software, and therefore various pictures have been included to illustrate, how some open source software looks. This section discusses how to obtain open source software, what distribution is, and mentions a few words about firms and individuals involved in open source software. This section begins by presenting the basics of a Linux based operating system, which also offers a few notes on installation.

9.2.1 Technical Basics

Technically Linux refers to the actual kernel of the operating system. The kernel is the inner most central part of the operating system. The kernel is responsible for memory management, process administration, control of hardware and still more [Kofler 1997]. The kernel itself is supported by a vast number of other programs, which together form a Linux operating system. The Linux kernel is designed from a principle of being as small as possible and leave non-essential functionality to other programs. The general understanding is that what could function outside the kernel, should stay there.

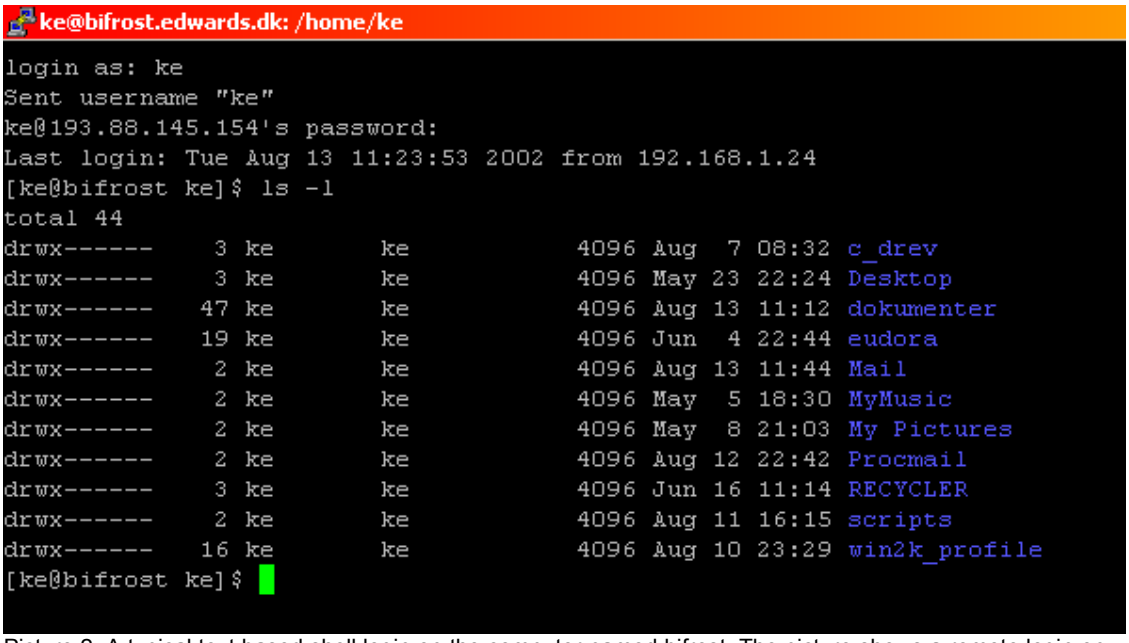
Linux is often referred to as an operating system. However, what parts should be present for a system to be called an operating system is somewhat blurred. Most will agree that an operating system consists of a kernel and a collection of programs for daily maintenance, file manipulation, and the ability to boot the operating system, when the computer is turned on. The operating system makes it possible to use the features provided by a computer, no program will run, if an underlying operating system is not present. In other words, an operating system provides a platform for running software. The operating system also facilitates a user interface, which might be text based or graphical. Most Linux and Unix enthusiasts claim that an operating system is not complete, if it does not come with a fully functional compiler[22]. This is where Linux and Windows differ greatly. Where a Linux operating system comes with both compiler and complete source, Windows includes none whatsoever.

The presence of open source software compilers for various programming languages makes it possible for Linux and other open source software users like FreeBSD, NetBSD, and OpenBSD to start programming and create programs. It often happens that some programs are only available as source code, in which case a compiler becomes a prerequisite for using the program, as it must be compiled before use. In contrast users of the Windows family of operating systems are forced to buy a compiler. A recent discussion

[22] A compiler is a program, which translates the human readable source code into a binary program a computer can execute.

on the ZINF[23] development mailing list addressed this very topic. A potential developer was interested in knowing whether his dated Visual Studio 4.2 was good enough for ZINF development on Windows, as he would not spend money to update his compiler. People on the list felt that it would do fine.

A Linux operating system can be used through a text based interface or graphical user interface. When using Linux text based interface, user interaction is done through a command prompt, much like old MS-DOS, only more powerful. The text based user interface lets the user type in commands and have the computer return results on the display. This is referred to as the shell, as it represents a shell around the kernel. The shell is responsible for passing commands to the kernel and messages back to the user. The shell also allows the user to edit commands, scroll through, and search in previously used commands. See Picture 2 for a picture of using the text based interface in Linux.



Picture 2: A typical text based shell login on the computer named bifrost. The picture shows a remote login on bifrost as the user "ke" and a directory listing with the command "ls –l", which displays a detailed listing of the contents of a dirctory. The login is done using a Wwindows program called puTTY, which is free software.

A variety of shells exist, which all perform the same basic function of passing messages to and form the kernel. However, the different shells use different syntax, for instance the C shell uses a syntax, which resembles the syntax used in C programs. The most widely used shell in Linux is the BASH shell. BASH is an acronym for Bourne Again Shell - a pun on the original Bourne shell developed for early versions of Unix.

The advantage of the shell, compared to graphical interfaces, is the complete access to all directories, configuration files, and programs without having to fiddle with a mouse and click through many layers of menus. In particular, the shell is convenient for system administration, which mainly consists of editing configuration files and starting and stopping programs. On the other hand, the text shell is not intuitive, and users have to learn before doing, while the graphical interface allows some degree of learning by doing. A main advantage of Linux compared to Windows, is the possibility of remotely administering a Linux computer using a remote shell like the one shown in Picture 2. This

[23] ZINF is an open source mp3 player, which comes in both Linux and Windows versions.

means that the computer can be tugged away and administered from anywhere. The remote text based shell also has the advantage of requiring only limited network bandwidth to function. Graphical user interfaces usually require physical access to the computer or good network connection to function properly, a lot more information must travel over the network to display a graphical desktop remotely. While the text-based approach has its advantages for administrators, desktop users find it impossible to work with.

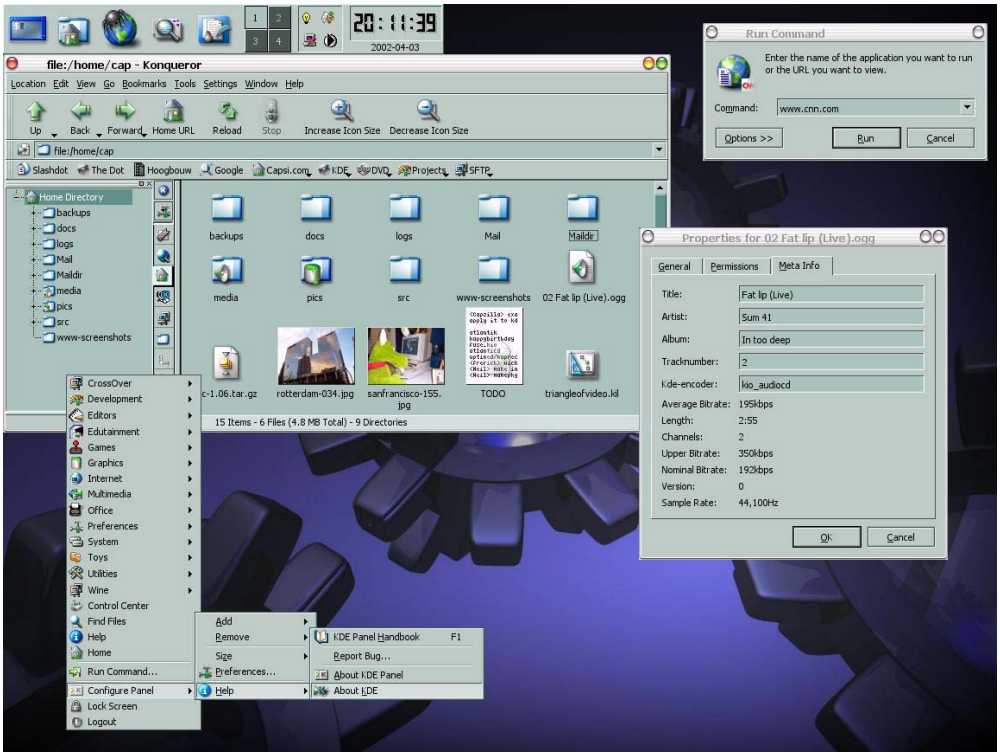
Naturally, a Linux operating system offers a graphical user interface, so that users do not have to fiddle with arcane commands. This is done through the X Window System, X for short, which offers a layer on op of the shell. Unlike the Windows operating system, the graphical user interface on Linux is not mandatory, and all features can be accessed through the shell, although some programs cannot run without the graphical user interface.

The X Window System is a collection of functions, which makes it possible to draw rectangles and other elements on the screen. Unlike the Windows graphical user interface (GUI), X also supports a network protocol, so that the image of one computer can be projected via network to another computer. The core of the X window system is the X server that serves as an interface between hardware i.e. mouse, keyboard and graphics adapter, and the X window system. Being the interface between hardware pieces, the X server must support each individual piece of hardware in order for them to function correctly. A brand new powerful graphics adapter is of little use, if the X server does not support it. The most used X server is XFree86[24], which is a free, open source implementation of an X window system. Commercial X servers like Metro-X and XInside also exist, and have other offerings such as professional support.

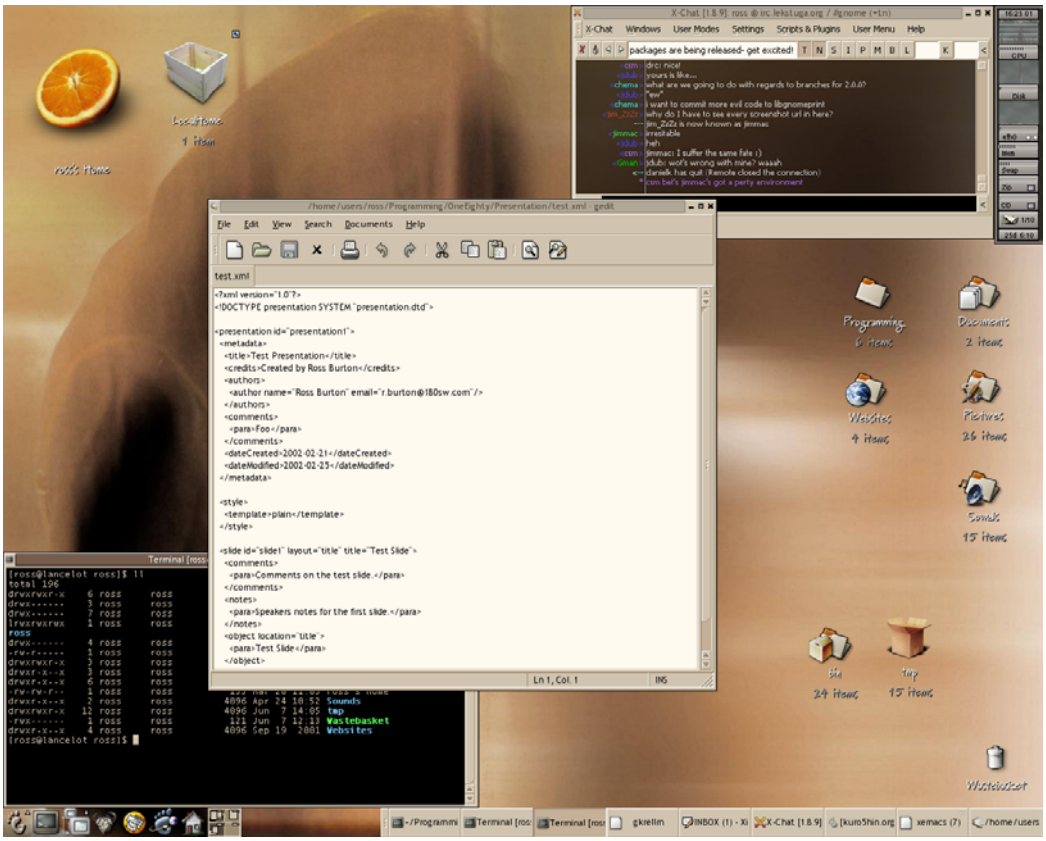
A window manager can connect to the X server and control the X window system. A window manager is basically an X program that controls the behaviour of windows and menus. Programs can also be programmed to behave in a particular way within the window manager. A variety of window managers for Linux exist, each of which look different and offer different functionality. The basic functionality is somewhat identical but menus and how to interact with the desktop is different.

KDE and GNOME are two projects developing complete window environments with applications and advanced integration between applications. KDE and GNOME use their own window manager, which support the features, and other window managers are free to use the applications from KDE and GNOME. Applications must, however, be programmed specifically to take advantage of any special features provided by the window manager. KDE and GNOME each have a separate set of libraries and widgets that programmers can use, when developing program for either of the window managers. A library is a set of routines and functions that a program can use, for instance, and all programs using the same library will use the same file open dialog. Widgets are all sorts of additional elements like buttons and shapes of windows that can be used in programs to provide a consistent user interface. In Microsoft Windows there is only one GUI, which is made for a very consistent user interface.

[24] The Xfree86 software can be obtained from [Website] http://www.xfree86.org



Picture 3: The KDE 3.0 desktop with the start menu, Konquorer (browser/filemanager) and properties dialog box.



Picture 4: The GNOME2.0 desktop showing an xterm, tekstteditor, and Xchat in the upper right corner, [http://vhost.dulug.duke.edu/~louie/screenshots/ross-desktop.png].

A Linux based operating system consists of several layers, and most of the components can be exchanged with others, should the user not be satisfied. In contrast, the Windows operating system integrates everything – even the browser[25]- and it is not possible to exchange any of the parts.

9.2.2 Acquisition and Installation

A Windows installation is obtained either preinstalled on a new computer or purchased as a package containing a CD with the operating system ready to be installed and a manual. A Linux operating system is obtained as what is known as a distribution. A distribution is a combination of the operating system (kernel, boot loader and supporting programs) and additional programs compiled by a Linux distributor.

Some distributions are commercial and can be bought in boxed sets containing several CD's and manuals. Examples include Red Hat Linux, SuSE Linux, Mandrake, Caldera, Turbolinux, and many others. As Linux and many of the supporting programs forming a Linux operating system are open source software, it is possible for anyone interested to compile these elements and create a distribution. And many also do, Linux Weekly News [accessed 8. August 2002] maintains a list of current distributions, which are actively maintained, i.e. they receive regular updates. The list contained 167 active distributions, and 27 no longer maintained. In comparison, there is only one distributor of Windows, which is Microsoft.

The advantage of distributions is that they are ready to use and easy to install. Most major distributions present the user with a graphical menu from which to choose what to install. The installation proceeds without user intervention, and following a reboot the use can boot into a complete Linux operating system with a graphical user interface as illustrated in Picture 3 and Picture 4. Usually a truckload of applications for all sorts of purposes like email, text editing, picture editing, music, and games are installed, most of which are open source. Programs, which are not open source, are additional offerings made by the distributor to attract customers. It is not uncommon to have several programs installed, which perform identical tasks, for instance Kspread and Gnumeric that are both spreadsheet applications optimised for KDE and GNOME respectively. If both the KDE and GNOME desktop environments are installed, almost all the installed programs appear in two versions, one for KDE and one for GNOME. This does seem like an impressive amount of programs, but usually only one version of each program is used, and the rest just clutters the menus.

Most distributions include roughly the same open source programs, as they are free, and the cost of including them is negligent. Distributions do, however, have differences, and in particular the installer (the program managing installation) and program for maintaining the system differ. For instance, it is this author's own experience from having tried Red Hat, Mandrake, Debian and SuSE, that not all distributions cope equally well with laptops. The commercial distributions also include proprietary programs either as trial versions or complete packages. For instance, the more expensive Mandrake Power Pack edition includes StarOffice 6.0, which is a Microsoft Office look-alike.

The differences between distributions are easily spotted in both the installation programs and on the desktop. Distributors are essentially offering the same ting: A Linux kernel, supporting programs and some additional programs. In this situation branding is

[25] It is not possible to remove the Internet Explorer from later versions of Microsoft Windows.

essential, and distributors make sure that individuals installing the distribution always know the name of the company providing the distribution. During the installation of a Red Hat system, a series of slides with Red Hat trivia and other information of the support offerings from Red Hat are shown.

The major distributors ship with custom kernels, which are modified versions of the standard kernels. Often distributors back-port features from development versions or unstable versions and stabilize them to offer their customers support for special hardware. For instance, the 2.2.x series kernel did not offer support for USB devices, something that many users wanted for their USB mouse or other hardware. Back in year 2000 – 2001, when the 2.2.x series kernels were the backbone of every distribution, most distributors back-ported USB. SuSE back ported USB from the 2.4.x series kernel to the 2.2.x series and could then offer support for USB [Interview, Axboe 2000:63]. In a similar vein, SuSE is sponsoring the ReiserFS (Reiser File System) and offers support for this as well as proposing it to be the default file system for their distribution. Red Hat, on the contrary, is sponsoring the EXT3 file system and uses this as the default file system in their distribution. Although both distributions offer both file systems, as they are licensed under the GPL, the default installation chooses the distributor's preferred file system.

Another difference between distributions is the file system layout, i.e. the ways, in which files have been organised in various directories, have also been different across distributions, and mainly for historic reasons. This has implications for compatibility, as developers of programs and software packages could not presume a particular file to be present in the assumed directory. As a consequence, certain programs could not run on some distributions without special configuration, something that the average user would be unable to provide. Lately efforts have been made to lessen incompatibility between distributions. Incompatibility between distributions is seen as an obstacle for further adoption of Linux and open source software. It makes development costs larger when developing for Linux, as the program has to be tested on all major distributions. The efforts resulted in the LSB (the Linux Standard Base) and FHS (File Hierarchy Standard). Both help to ensure that programs can find the needed files in correct locations [Interview, Makholm 2000].

9.2.3 It is all Available from the Internet

Many distributions can be downloaded from the Internet, although their size may be very large, ranging from a couple of hundred megabytes to the contents of many CDs. Several distributors like Red Hat, Debian, and Mandrake offer their distributions as a free download from the Internet. These distributors offer their distributions as a set of CD-images, which are a complete copy of the information contained on the CDs in their distributions. The images can be burned on to a CD and used for installation, and they can be obtained from the distributor or some of the popular ftp mirror sites like Sunsite. Many Internet Service Providers have a local mirror of these sites to keep the traffic local (and save money).

The latest Red Hat Linux edition (version 8.0 as of writing) requires download of 5 CD images, equivalent to 3200 Mega Bytes. Downloading such abundance of data requires a fast Internet connection, and this is as a natural deterrent for doing so. Assuming a 256Kbit ADLS (in reality 26 Kbytes/second) Internet connection, which is very common in Denmark, it would take approximately 35 hours[26] to complete, assuming nothing went

[26] 3200Mb = 3246899Kb, (3246899Kb /26kb/s)/3.600s/hr) = 34,7 hrs ~ 35 hrs

wrong and 100% utilization of the available bandwidth. Having downloaded the images, they must be burned on to a CDROM to be used. Whether to download or to buy a boxed set of CDs is a matter of personal preference. It seems, however, that the growing size of Linux distributions will drive many to buy the boxed sets.

A few years ago most Linux distributions could fit on to a single CD, and many computer magazines used the opportunity to include a CD with a Linux distribution when featuring articles on Linux. This probably helped the diffusion of Linux to a large extent, and ensured that many were given an opportunity to try out Linux (no doubt some were scared off immediately). Some distributors such as Red Hat have long approved of this praxis, while other distributors like SuSE do not allow it. Interestingly Red Hat supplies the distribution with a free download of CD images ready to be burned on to CDs, while SuSE does not. SuSE does offer the complete collection of software for download, but this must be prepared for installation before use, whereas a CD image allows the user to conveniently boot directly from the freshly burned CD.

The Debian distribution, which is a distribution created and maintained by a group of voluntaries, encourages the installation to be done via Internet to limit the download size and to only install the most needed programs.

9.2.4 Working with Linux

The daily work with any system be it Windows, Linux, or others, involves maintenance of the system, installation of new programs, and use of various programs useful to the user. This section will focus on the issues of installing programs and maintaining the systems (including programs).

When obtaining a program for a Linux system, it comes in two basic forms: 1) Source, and 2) Packaged. The packaged form resembles the programs ready for installation on Microsoft Windows. In source form the program is obtained as source code, and the user will only have to compile the program before installation and use. The procedure is not advanced and requires the source to be downloaded, which usually comes packed in a compressed TAR archive[27] to save time during download. The user must then unpack the archive using his favourite program; many users just invoke the *tar* command from the command line. Once unpacked, the user must run three consecutive commands: 1) *Configure, 2) Make*, and 3) *Make install*.

Configure is a script that configures the files used to compile (make) the program, special parameters such as where to install the program may be passed to the configure script. *Make* compiles the source into executable files, which is the actual program. *Make install* installs the program on the chosen location, and the program is ready to run.

Depending on the size of the program and source code, making the program will take some time. For instance, compiling a new kernel from 30 Mbytes source take well over 40 minutes on a 300 MHz PentiumII. Larger programs will naturally need much more time to compile. For this reason and to make it easier for people, who are not used to compiling their own programs to use a Linux system, packages have been introduced.

A package is a file, which contains a binary program, configuration files, manual pages, and a list of dependencies that must be satisfied for the program to run. When installing the packages, a program capable of reading the package, deciphers the contents

[27] A single file containing all the files organised in directories.

first, and checks that all dependencies are satisfied, and only then the program can be installed. Using packages are much simpler than compiling from source. Users can install packages without having to worry about unpacking TAR archives, compiling, or solving subtle dependencies. Dependencies are an important issue and refer to the fact that most programs require a minimum functionality of existing programs or libraries to run. Using a packaged program, the package will determine version numbers of required programs and libraries (dependences), and it issues a warning or simply refuses to install, if version numbers are below required minimum. The programs for handling packaged programs exist in both text versions and graphical versions. Most users are only familiar with the graphical version, which is activated when double clicking on the new program.

Packages make it easier to maintain a system, as package programs are able to make upgrades as well. The user can either download the latest versions of updates to his system, ask the package program to update packages, or subscribe to an update service. The latter method is a service offered by most distributors, and often this is a service paid for on a monthly basis.

However, when dealing with time critical security issues, packages have the drawback of being available later than the source code versions. On a daily basis, vulnerabilities are found in various programs, and occasionally the vulnerabilities may compromise the integrity of a system. In this situation it is urgent that the affected programs are fixed. This can be done by using the package format, or by patching the source version and compiling a new fixed version. Source patches are always available sooner than the packaged versions, as it is the maintainer of the affected program, who usually publishes a patch, and the various distributors, who later publish a package containing the patch. The difference may be critical and an argument for using source only programs. To the average user, who does not run servers, the use of packaged versions is a huge advantage.

There are two different package formats: Deb and rpm. The deb package format is used and invented by the Debian distribution, and Red Hat invented the rpm (Red Hat Package Manager) package format. The vast majority of distributions such as Red Hat, SuSE and Mandrake uses the rpm system. There is some dispute regarding what package format is better, and no judgement shall be passed in this matter. Both formats offer checks to establish, whether dependencies are satisfied, before a program is installed. The major difference between the two is that the Deb format is able to also satisfy dependencies, if the computer is connected to the Internet. The rpm format will complain of unsatisfied dependencies but does not offer to fetch and upgrade the packages.

Use of a common package format gives the impression that an rpm package for Red Hat could easily be used on a SuSE system. The differences in distributions will in some occasions make an rpm package for SuSE incompatible with Red Hat to the user's great annoyance. While standards are being made to accommodate this problem, the present reality is that incompatibility exists between distributions using the same package format.
Getting Help

An important element in using open source software, or any other software for that matter, is to be able to get help when needed. Often open source software is documented with only a manual page, which is the built-in manual system for Unix systems. For instance, if one wishes to know how to use the *ls* command, one merely has to enter *man ls* at the command prompt for a complete description of the command and a list of all the arguments. The drawback is that manual pages are terse, and that they do not offer many

examples or go into details of getting the program to work in combination with other programs.

If one has bought a Linux distribution, the distribution often includes a limited support period, where one may either call or email the support staff. Outside the limited period, support can be bought from either the distributor or from a dedicated support company. A few years ago, dedicated support companies did not exist, and they are still few and far between.

It is also possible to try to get help by asking questions in a public news group or mailing list. News groups are self-organised forums, where anyone interested may pose a question for others to answer. We shall not dwell on the reasons why people answer questions, but the fact remains that news groups function very well as a way of getting support, when open source software is used. It functions so well that Apache has been awarded Linux the “Best Technical Support Award“ by Infoworld [Foster 1997].

When obtaining help via news groups, a user seeking information must first decide in which forum to ask the question. A user can approach either a general user forum, like one belonging to a local user group or the specific mailing list for the particular program causing problems. Most open source software projects maintain two different newsgroups, one for development, and one for support. The development newsgroup contains all discussion related to development of the program and fixing of bugs, and it is to this list that people send their patches. The support list is for users, who have problems using the program. The following example illustrates the workings of a mailing list, where a user asks a question and gets a reply from more experienced users. The example is taken from the Amanda-users’ list; Amanda is an impressive open source backup program:

```
“Date: Thu, 16 May 2002 04:09:43 -0700 (PDT)
From: adi adi
Subject: Tape and dump cycles
To: amanda-users@amanda.org

Hello list, i'm a lil confused abt the dump and tape cycles
to put in amanda.conf. This is my situation. I am dumping
everyday, 7 days a week. The tapes are labelled for week 1
as monday1, tuesday2, wednesday3 and for the second week
would be monday2, tuesday2 and so on. Would it be right to
put  "dumpcycle 4 weeks" "runspercycle 7"  "tapecycle 8
tapes" in amanda.conf. Could anyone explain this for me,
thanks much. Btw, I have 31 tapes for amanda which i will be
overwriting every month.”
```

A few hours later, adi adi receives an answer:

```
“Date: Thu, 16 May 2002 07:41:28 -0400 (EDT)
[headers removed]
A "dumpcycle" is the maximum amount of time between
successive full dumps (level 0s) of any particular disklist
entry.  runspercycle is how many times you are going to run
amdump per dumpcycle.  And tapecycle is how many tapes you
are going to use.
From the numbers above, the only thing I can say for sure is
that your tapecycle should be 31.  You'll have to set
dumpcycle depending on how much data you have and how big
your tapes are.  For example, if all your data can fit on 7
tapes, then you can use "dumpcycle 1 week" and "runspercycle
7".  That way you'll have 4 dumpcycles per tapecycle, which
offers good redundancy.
Finally, you may want to rethink your tape labels.  Amanda
will tell you which tapes it wants when, and it may not
```

```
always "match" the day label on the tape, which might
confuse somebody."
```

It is also possible to get help from mailing lists, which are not specific to the actual program. For instance, a simple question about configuring the Apache web server is equally likely to be answered by the local Linux user group as by the Apache mailing list. This is because the Apache web server is a very commonly used program. For programs used by many persons, the local Linux user groups are often sufficient and ease communication by using the local language. However, more esoteric programs and questions must be directed at specialised forums.

The quality of service as defined by speed and usability of reply is very high for news groups. Lakhani and Von Hippel [2000] have conducted a survey of user-to-user assistance in the Apache project. The study is interesting, as it quantifies the number of questions and answers in the Apache news group forum, and as it has further tried to pursue the motivation for information providers to answer questions. The Apache news forum was very efficient, and frequently information seekers obtained help, and in 92% of the instances their problem was solved completely. Other seekers, so-called information seekers, who asked less than 4 questions in the period of the survey, only obtained help or solved their problem completely in 61% of the instances [Lakhani and von Hippel 2000]. 52% of the information seekers obtained a reply within the same day, 9 % within 2 days, 16% got private email reply, and 23 % got no reply at all.

Newsgroups and mailing lists will for many users be the preferred way of getting support, as both are easy to obtain, and only cost the time associated with asking the question.

9.2.5 A Simple Comparison of Three Major Distributions

Three distributions have been the subject of a simple comparison with the intention of observing and illustrating some of their differences. The thee distributions compared were SuSE 8.0, Red Hat 7.3, and Mandrake 8.2, which were all published in the spring of 2002. These three distributions are all mainstream distributions, targeted at the average consumer and supposed to offer a ready and easy-to-use Linux system, with all the software needed for basic computing needs. All distributions offer basic office applications, sketching and photo editing, multimedia applications for viewing video, listening and recording music, internet tools for browsing the Internet, news group reading, email, and collaboration a la Outlook, and much more. Each of the three Linux distributions should make the average computer user happy, as they offer all the needed software for every day use.

The purpose of the test installation was to go through the installation procedure with the intent of obtaining a usable Linux system. All hardware, i.e. CDROM drives, pointing devices, monitor, printer, and others, should be configured and usable. In this particular situation, the printer is an HP970Cxi configured as a network printer through a HP JetEx direct device, not unusual but hardly standard. Following the installation, the system would connect to NFS[28] network shares and install the ZINF MP3 player and use a network share containing a music collection. Since the systems are part of a heterogeneous network, i.e. different types of computers running on the network, the systems must be made able to access Windows shares on a Windows2000 system. This requires the LAN browser to

[28] NFS is the Network File System, a protocol for sharing files among Unix and Linux computers.

function and provide easy graphical access to available shares. Open source software is often updated with new features added and bugs fixed, and therefore a system upgrade is essential, which was also tested. The print system is tested by configuring the printer and sending a print job to the printer. The X system is tested by observing the functioning of the graphical display, mouse, keyboard, and functioning of special Danish characters. The test emphasized using the graphical tools provided by the distributor and not the general Linux tools.

Ideally the installation program should perform all these things, and a novice user should be able to perform the installation without having to consult manuals of neither the operating system nor the hardware. All three distributions should install and function equally well on the test computer. Configuration tasks other than the mentioned, for instance email, will not be performed.

The installations were performed on a semi old (3,5 years) Compaq Armada 6500 notebook with a PentiumII 300MHz, 192 MB RAM, and a pre-existing Windows2000 installation on two FAT32 partitions on one IDE hard disk. The system is fairly old, which should make it supported, and prior to installation it was affirmed that standard kernels would support the hardware. Having the Linux system live together with the Windows system and allow for dual boot into either of the systems is also to be expected for many users. The installation program should recognise this and seamlessly provide access for the Windows partitions, which often contain valuable data. Migration from Windows to Linux is deemed troublesome, if data on the windows partitions are not accessible.

Red Hat and Mandrake make their distributions available for download from the Internet as ISO images, which can be burned directly on to a CD that is bootable. The price of the Red Hat and the Mandrake adds to the cost of Internet time, plus three empty CD's. SuSE does not offer their distribution for download as ISO images, and instead they offer all the programs, referred to as packages, for download. SuSE does offer the complete collection of programs found in their distribution for download. However, it is not possible to easily obtain a bootable installation medium and packages for SuSE. The only way to obtain an easily installable SuSE distribution is to buy the boxed set, which includes 7 CD's and manuals.

The size of the three distributions differ somewhat, mainly because the available Red Hat and Mandrake distributions are the basic editions. Both Red Hat and Mandrake offer "pro" versions of their distributions, which contain additional software offers, some of which are proprietary. An example is the Mandrake PowerPack bundles StarOffice 6.0, and thus the distribution is quite a bargain, as the distribution costs \$139[29], while StarOffice 6.0 bought separately costs \$55.95 at Amazon com. The pro editions of Red Hat and Mandrake are of a size equivalent to the SuSE distribution.

Red Hat and Mandrake comes in 5 ISO images equal to 5 CDs including source. Only three CDs are required for a complete binary installation, as the last two images contain source code for most of the applications. SuSE comes on seven CD's, or one DVD. Needles to say, the DVDs are a nice addition, which saves a lot of disk swapping during installation, but requires a DVD drive. The ISO images for Red Hat and Mandrake was downloaded on a Windows2000 system and burned onto CDs, and this did not pose any problems apart from the many hours of downloading. The SuSE distribution was bought

[29] The price is from www.mandrakesoft.com accessed 28th August 2002.

via the Internet and arrived in a nice large box containing the 7 CDs, one DVD and 3 manuals, which detailed the installation and use of some of the featured software.

All of the distributions used a bootable CD to initiate the installation, and upon booting from the CD the user is presented with a brief overview of the installation. This is similar to the installation of a Windows system. The installation log of the three distributions can be found in appendix 3.

The installation of the three distributions follows the same general schematics: Boot the computer from CD, answer some questions about the hardware, choose language, enter username and passwords, and tell the installer where to place Linux on the hard disk. However, noteworthy exceptions were the addition of printer configuration in Mandrake and SuSE. The Red Hat installer is simpler and asks far less questions, than does the other two distributions.

The same partitioning scheme was chosen for all installations, in the following hda5 means partition 5 on the first hard drive on the primary controller (the "a"): /boot in hda5, swap on hda7 and / on hda8, the root partition was 8Gb and was not partitioned further[30]. The Windows boot loader was maintained to boot both the Linux and the Windows system. By installing the LILO boot loader on the first sector of hda5 and copy this to the primary Windows partition, it is possible to use the Windows Boot loader to choose what operating system to load, when turning on the system. Other solutions exist, but this is a robust solution, and I am comfortable with configuring and maintaining it.

9.2.5.1 Results from theTtest

Following installation, all distributions were able to boot and load the graphical user interface, KDE in this case. SuSE was the only distribution not capable of detecting the network interface, but manually choosing a PCMCIA network interface solved the problem. All of the distributions were able to browse the Internet after the initial boot, and required no further configuration. Note that SuSE required manual configuration of the network before browsing the Internet.

The computer was installed on a local network, where all files were stored on an NFS server. Therefore connecting to the NFS server had high priority. The Mandrake and SuSE distributions both offer an easy-to-use configuration tool for mounting NFS shares and Mandrake were even able to display a list of available shares on the network and access these by a simple mouse click. Red Hat did not include such tool, and the configuration had to be done manually issuing the following command:

```
$ mount –t nfs 192.168.1.1:/home/musik /mnt/musik
```

This mounts the NFS share /home/musik on the server at the ip address 192.168.1.1, and it is possible to use the name of the computer to connect to instead of the IP address. However, I found that browsing NFS shares was unstable using computer name, and the file browser would crash with an error. The graphical tools used by mandrake and SuSE did not mention that computer names might be a problem, and that they had to be solved by time-consuming trial and error.

After mounting the NFS shares, they were browsed looking for the ZINF installation files, which were in RPM format. The Mandrake and the Red Hat distribution had no problems with this, but SuSE kept getting errors, and it was impossible to get anything

[30] This is unwise for server installations but suiting for this test installation.

from the NFS shares. Despite this problem, the Internet connection functioned perfectly and allowed the source code for the latest Linux kernel (version 2.4.19) to be downloaded. Following the download, a new kernel was configured and compiled, which took a little more an hour. This helped the NFS problems, as it was now possible to assess the wanted files. Finally the ZINF installation could commence on the SuSE installation.

The ZINF installation files were contained on one of the NFS shares as an RPM file. The ZINF player installed without problems on the Red Hat distribution, whereas the other distribution showed problems. Mandrake reported an unsatisfied dependency on a library, but ignoring this and forcing the installation resulted in a functioning ZINF player. The SuSE distribution installed the ZINF player without problems, but it was not possible to use ZINF. The ZINF player failed, every time the program was executed. Downloading and compiling from source did not resolve the problem, and further investigation indicated that the install scripts assumed the presence of some specific directories, which were not present on SuSE.

When being on a network with Windows computers, it is important that the Linux computer is able to access any Windows network shares containing useful data. The Konqueror file browser also contains a LAN (Local Area Network) browser for just this purpose. However, the default installation of all the systems reported: “Unable to connect to Localhost”, when trying to browse the network. Investigation of the problem revealed that the program responsible for this functionality (referred to as a daemon) LISa, was not configured. Opening the Control Panel provided easy access to configuring the LAN browser, which was done with the help of a configuration wizard asking questions. Still it was not possible to browse the network, and further investigation revealed that the LISa daemon was not started, and apparently the configuration tool in the Control Panel did not recognise this. The LISa daemon was installed on both the Mandrake and Red Hat and could be started from the command line. The LISa daemon was not even installed on the SuSE system, and had to be installed before starting the daemon and browsing the network. Following the installation, the daemon started and worked without problems, although the installation erased the configuration already made, and required the configuration to be repeated. It was not possible to browse the network and access windows shares.

Some differences between the three distributions were noticed when accessing the local network. It was not possible to browse Windows2000 shares on the Mandrake system, which was quite annoying, as the network contained a few Windows2000 computers. The problem was not investigated further, as the LISa problem had already taken well over an hour to solve.

Having accessed the needed windows network shares, the time had come to access some of the data contained on the Windows partitions on the computer. SuSE and Mandrake had automatically mounted the partitions, and all data could be accessed without problems. Red Hat did not mount the Windows partitions, and manual mount of the partitions was required, before any files could be accessed. In order to do so, the following command was issued at the command line:

```
$ mount –t vfat /dev/hda1 /mnt/win_c
```

The above command mounts a partition of type vfat (fat32 is the file system used by windows) on the first partition of the hard drive and mounts in the directory /mnt/win_c. There were no problems accessing and using the files on any of the distributions.

All distributions offered a wizard for configuring a printer, which guided the user through a series of questions, resulting in a working printer. The wizards were somewhat different in appearance, but offered similar options.

The distributions were installed on a laptop with no floppy drive. The Red Hat and SuSE systems configured all the hardware correctly, and noticed, equally correctly, that there was no floppy drive. The Mandrake system on the other hand configured a floppy drive and persistently locked up file managers and other programs, when accessing the /mnt directory containing the non-existing floppy drive. Investigation into the problem indicated that a solution could be obtained by compiling a new kernel without floppy support, or make Mandrake ignore the floppy drive. The latter solution was chosen, and the configuration panel offered a hardware applet, which listed the components of the computer. It was possible to remove the falsely detected floppy drive and thus solving the problem.

Maintenance of an installed system is very important, and naturally this was also tested on the three systems. The Mandrake and the SuSE system allowed for easy update of the system. Their respective control panels contained an icon for just this, and both distributions worked without problems. Red Hat also provides an easy-to-use interface for updating the system, however, this requires registration, and only one system per site can be maintained free of charge, whereas other systems come at a monthly fee. The only complaint, which goes for all distributions, is the amount of time required to perform the updates. When updating the systems, it would take hours to download the hundreds of megabytes of updates, meanwhile the computer would be left practically unusable. In general the installation of each of the distributions would take roughly an hour from the initial boot to a complete system including a broad selection of application was installed.

In conclusion, it appeared that the three distributions briefly tested, although apparently similar, behaved differently. On the tested computer it was only Red Hat that was able to provide a functioning system, without having to fix serious problems (lockups and missing NFS network). However, the post install configuration[31] of network shares and update of the system was not easily accessible on the Red Hat system, and a user with no prior knowledge of Linux would have been in trouble. Installing and maintaining a Linux system is not for those, who dislike fiddling with computers, or who have no knowledge or ambition to learn. The process of getting the network browser to function correctly was troublesome, and newcomers to Linux would have had problems getting the desired functionality.

Installing and using the ZINF player was chosen as an example of installing a music program, which is considered to be one of the best. Installing ZINF illustrated that getting your favourite music player to work on any distribution can be a daunting task, and it was never succeeded on the SuSE.

In contrast a Windows2000 was installed on the same computer, and there were never any issues; All hardware was recognised and configured correctly. The Windows version of the ZINF player was also installed and worked perfectly. It must be noted that the manufacture of the computer support was Windows and not Linux. Prior to the installation it was assumed that, given this to be open source software, the three distributions would faire evenly and produce similar results when installed. This assumption proved to be wrong, and severe differences were discovered. Users buying a

[31] Post install configuration is the configuration, which are done when the initial installation is complete and the system is installed.

distribution must choose their distribution carefully, and a wrong choice might lead to many hours of solving annoying problems.

9.3 Open Source Software Development

In the following pages, two examples of open source software development are presented: 1) The Linux kernel and 2) FreeBSD development. The Linux kernel is a well-known example of open source software development representing a typical hierarchical organisation structure. Written sources and the Linux kernel mailing list will be used to describe the Linux kernel development. FreeBSD is a Unix operating system, but developed differently using a core team as the central authority. Poul-Henning Kamp is a FreeBSD core team member, and the interview will be used as a basis for describing how FreeBSD is developed.

In the following pages, the distinction between maintainer and user-developer known from the model of software development and consumption in chapter 7 is upheld. The maintainer is the original developer taking responsibility for releasing new versions, be it individual or firm. A user-developer is an individual or firm, making modifications to a program maintained by the maintainer.

9.3.1 Linux Kernel Development

The Linux kernel is the central component in a Linux distribution, where it is the core element of the operating system. The Linux kernel was originally developed by Linus Torvalds, who developed Linux as a hobby in 1991. In September 1991 Linus Torvalds released the source code for Linux on the Internet under the GPL license. Immediately Linux was adopted by a number of users, who found it interesting to the point, where some of the users began to modify the source code and add missing features.

Linus Torvalds was actively developing Linux and often released new versions of the source code, and sometimes two versions a day were released. Some of the persons, who modified the source code, returned the modifications to Linus Torvalds, who could then decide, if the modifications should be included in the next version of Linux. In time, the Linux kernel has grown, and so has the number of people developing and working on the kernel. The kernel is no longer developed by just a few people in their spare time, many kernel developers are now paid to do the work by companies relying on Linux.

The general forum for Linux kernel development is the mailing list "linux-kernel@vger.kernel.org", where anybody interested in kernel development may discuss relevant matters and exchange modifications. The Linux kernel must be viewed as work in progress from which different versions branch out and stabilize. The Linux kernel exists in several versions living a parallel life, and as of writing four official kernels are maintained: 1) The 2.0 series, 2) The 2.2 series, 3) The 2.4 series, and 4) The 2.5 series. The 2.0 kernels are extremely stable and have no known bugs, they are, however, also very old, and all the features of the late kernels are not included. The 2.2 series is also very stable, and like the 2.0 it does not include many of the features for the late kernels. For instance, the 2.2 series does not include support for USB devices. The 2.4 series is the latest, stable kernel, and as of writing, Linux-2.4.20 is the latest release in this line of kernels.

The kernel naming scheme tells, if the kernel is considered to be stable or in development. An uneven number in the second digit like kernel 2.5 indicates that this is a development kernel. Users of development kernels must expect the kernel to behave unpredictably, as it is a development version.

Once the current 2.5 development kernel has been stabilized and deemed not dangerous i.e. there is very little chance the average user accidentally loosing data, the version number will change to 2.6.0. This will be the first version of a new line of kernels. Linus Torvalds is currently in charge of the development kernels, and he must approve all changes to the 2.5 series. Linus Torvalds does not accept patches from everybody, but only from a few trusted maintainers.

Each trusted maintainer is responsible for a particular area, and anybody, who has a modification for that particular area, is advised to send the modifications to the maintainer. The 2.4.19 version Linux kernel currently contains a list of approximately 300 different subsystems, and the corresponding maintainer.

To complicate matters further, there are also individual mailing lists for some subsystems. For instance, the Reiser file system, among others, has a separate mailing list called reiserfs-dev@namesys.com. Whether a subsystem should have a separate mailing list depends on the amount of discussion, generated by the subsystem developers. Conclusions and information of general interest to other kernel developers are often propagated to the general kernel development mailing list. For instance, if a subsystem changes design or introduces a new interface, other developers, writing program using the subsystem will need to know. Developers using the subsystem are also likely to have suggestions as to how the desired changes could be implements. For all other kernel related items, which are not mentioned in the Maintainers[32] file, Linus Torvalds is responsible. Or, as it says in the bottom of the Maintainers file:

```
"THE REST
P:      Linus Torvalds"[33]
```

Thus the work process of the kernel is: The maintainer releases a new version, and a user-developer downloads the new version and begins to make modifications. These modifications are then submitted to the maintainer of the subsystem in question, and the message is cc'ed to the proper mailing list. Not only is the maintainer interested in the modifications, but other developers working on the same subsystem are likely to be interested as well. The submitted modifications are then discussed on the proper mailing list, where anyone is free to comment and make suggestions for improvement. Often maintainers require the proposed modifications to be tested by a few other developers and in different environments (different hardware and configuration), before accepting a proposed modification. This is to ensure at least some level of quality assurance.

Modifications to programs have historically been distributed in the patch format, but recently a new way, based on the proprietary Bitkeeper program, has emerged. A patch is a modification to existing work distributed in a special format, making it possible to easily apply modifications to existing files. An example:

Imagine a file distributed to a lot of people. The file "original" contains the following lines:

```
"This is a test
for illust
diff and patch"
```

[32] The file is placed in the root directory of the Linux kernel source in the file "MAINTAINERS"

[33] Bottom of the MAINTAINERS file in the Linux kernel 2.4.19 source.

Obviously there is a bug in line 2, and thus a user decides to correct the spelling bug by creating a modified file called “modified”:

```
“This is a test
for illustrating
diff and patch”
```

Now, running the command:

```
“diff -u original modified > changes”
```

produces a file called “changes”, containing the difference between the two files. The –u switch specifies the unified output format. Such a file is called a patch. The changes file contains the following:

```
“--- original    Fri Dec  6 11:35:12 2002
+++ modified    Fri Dec  6 11:35:35 2002
@@ -1,3 +1,3 @@
This is a test
-for illust
+for illustrating”
```

The patch file contains the changes between the original and the modified file. This is very useful, because the diff file can be used in conjunction with the patch program doing the exact opposite of diff. Using patch, the original file can be turned into the modified file by using the changes file.

Using diff and patch has two advantages over just sending the complete files: 1) It is possible to clearly identify the difference between two files and 2) It is much easier to send small patches than large files.

A developer, who knows his code, is able to quickly gain complete overview of a modification by examining a patch file. When files are very large, containing thousands of lines of code, a change in a few lines of code are easily distributed compared to the complete file. Patch and diff are not confined to just comparing single files, and they easily handle complete directories, which is necessary for comparing larger projects like the Linux kernel.

Patch and diff highlight the details of kernel development, and a closer look at the process is warranted: When a user-developer agent decides to make modifications to a program, he will download the latest version of the source code. Instead of just jumping directly in and start editing files, the user-developer must first make a copy of the source code, so that he now has two versions: 1) The original and 2) A copy for personal modifications. Each version of the source code is referred to as a tree, because a source code directory has a tree-like structure. The reason for copying the original is that the user-developer must have the original version to compare his personal modifications. When the user-developer has finished his modifications, he compares the original and the modified tree and generates a patch file containing the differences.

Unfortunately the world has not been standing still, while the user-developer made his patch. The maintainer and other user-developers have been making improvements to the program, and it has changed. When the user-developer submits his patch to the

maintainer, the patch will not apply cleanly[34], and either of the two will have to modify the patch, if it is to be included in the program.

Given coordination of open source software development, this type of situation is neither avoidable nor uncommon. The more developers working on a program, the more likely it is that the program has changed, when the modifications are finally finished. Mailing lists are used to announce work going on in various areas of the kernel, but the situation of two or more persons editing the same file in the same kernel version cannot be avoided.

To avoid submitting patches that do not apply, it is strongly recommended that the user-developer downloads the latest version and re-diff his modified tree against this. Then the user-developer can solve all the merging problems by hand before resending, and even then the user-developer is not guaranteed that the patch will apply cleanly.

Patch and diff is not the only way of exchanging modifications on the Linux kernel mailing list. Bitkeeper, a program designed for supporting the development of the Linux kernel, has recently been adopted by Linus Torvalds and by other but not all kernel developers. Bitkeeper is a proprietary program licensed under several licenses, one of which is an open source license. When using Bitkeeper as open source software, the licensee is not allowed to use Bitkeeper for developing other software than open source software, and the program must not compete with Bitkeeper – even if it is open source.

Bitkeeper solves some of the problems not handled very well by using patch and diff. The biggest problem is, when the personal development tree has diverted from the original, and the original has changed as well. In this situation developers must re-diff and solve all the merging problems by hand, which is often very time consuming, for instance, if the user-developer has piled 2MB of patches, and tries to apply these to the latest version. This is done using patch, as shown above. However, where patch fails, a .rej file is generated, and the developer has to search for all these files and solve the problems [Henson & Garzik 2002].

Bitkeeper is different altogether, using what is known as repositories and change sets (cset). A repository is a collection of source-controlled files, which may have a parent and a child. The user-developer clones the repository of the maintainer, thereby creating a child of the parent tree. Apparently this is similar to the process using patch and diff, and however, while cloning from the parent, Bitkeeper creates a lot of meta-information about the individual files. This enables Bitkeeper to identify and distinguish files, even though they have changed names. Patch and diff compares files on a directory/filename basis and will generate errors, if files have changed names.

The just cloned tree becomes the local mirror of the maintainer's original and the clone may live for the project life and be updated when needed. Bitkeeper uses commands like *bk push* and *bk pull* to push and pull changes to and from the parent repository. The user-developer now clones his own tree, and this becomes the working tree. The user-developer can create as many clones as desired, depending on the modifications desired. It is preferable to keep development themes separated in different trees, and clone for each theme.

Now, the user-developer has made some modifications that he wishes to distribute, and he generates a change set. Using *bk push* will push changes from child to parent, while

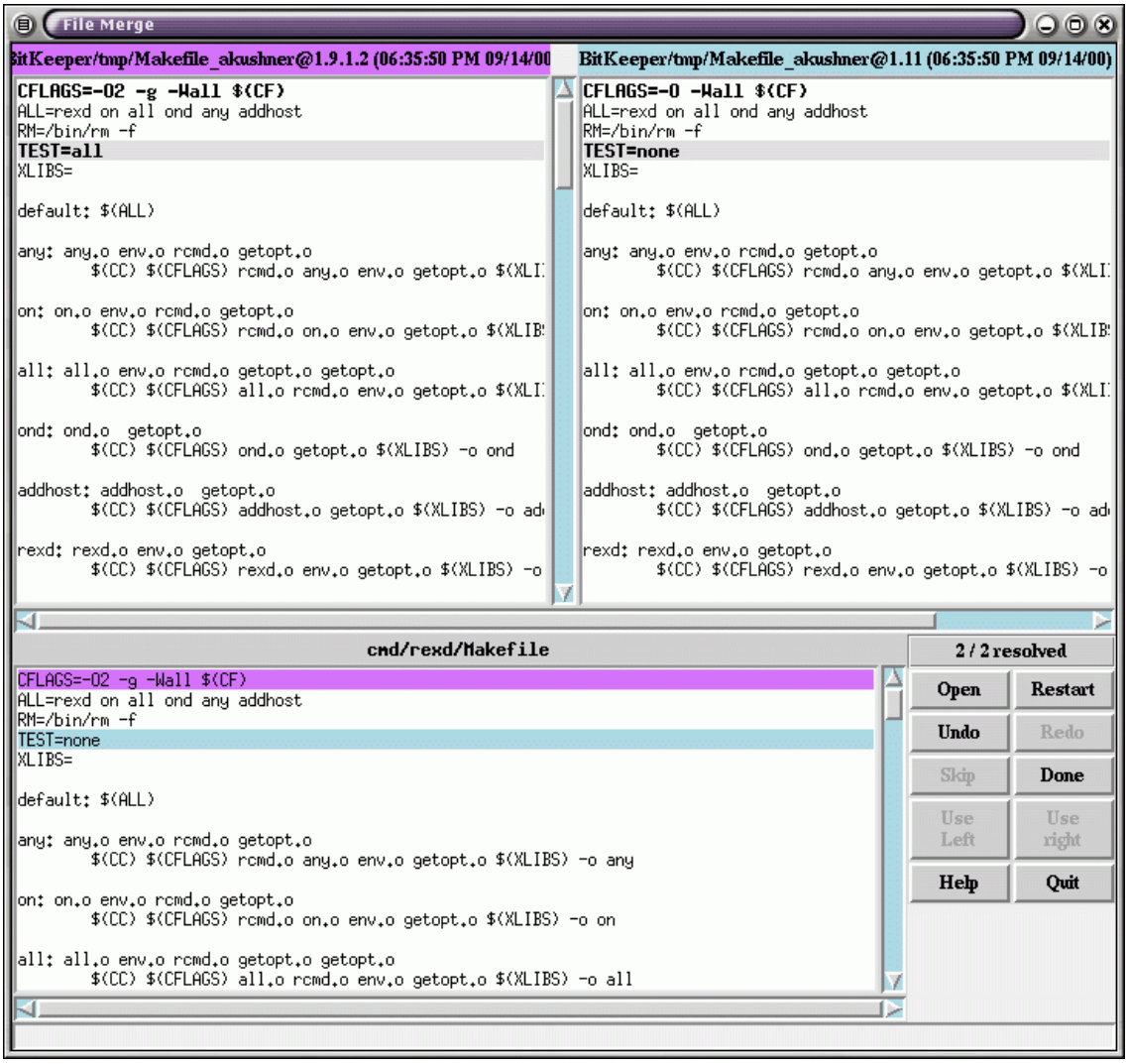
[34] A patch is said to apply clean when no errors are generated when a file is patched.

bk pull will retrieve change sets from the parent. In Linux kernel development, a change set is distributed by user-developers sending an email to the proper maintainer and a mailing list, asking the maintainer to pull from the user-developer's repository. In the following example, kernel developer James Simmons submits a patch for the frame buffer layer. The patch is submittet to Linus Torvalds, and the kernel mailing list, the frame buffer development mailing list, and the Linux console project development mailing list are CC'ed:

```
Delivered-To: ke@edwards.dk
Date:   Fri, 6 Dec 2002 21:25:02 +0000 (GMT)
From: James Simmons jsimmons@infradead.org
To: Linus Torvalds torvalds@transmeta.com
Cc: Linux Fbdev development list <linux-fbdev-
devel@lists.sourceforge.net>, Linux Kernel Mailing List
<linux-kernel@vger.kernel.org>, Linux console project
linuxconsole-dev@lists.sourceforge.net
Subject: [BK updates] fbdev updates.
X-Mailing-List: linux-kernel@vger.kernel.org


Hi!!!
  After much work and many fixes the final api for the
framebuffer layer is complete and alot of new functionality
has been added. Several drivers have been ported. Still
several more to go. Could you grab the latest changes from
bk://fbdev.bkbits.net/fbdev-2.5
Please sync it up to your latest tree. Thank you.
```

Like with diff and patch there will be conflicts when merging, if the change set does not correspond with the parent. Bitkeeper eases conflict resolution by using a special algorithm to make a suggestion of how to solve a program, and Bitkeeper also has a graphical tool to easily solve conflicts:



Picture 5: Merging files in Bitkeeper using the three way merge tool. The top two windows shows the child and parent, while the bottom is the merged file [http://www.bitkeeper.com/Products.Graphical.Screen.html].

Using Bitkeeper greatly reduces the time required to solve merging conflicts [Henson & Garzik 2002]. Bitkeeper makes it possible for developers to make their modifications publicly available on the Internet. Developers can either use a special program provided with Bitkeeper make their Bitkeeper tree available on their personal website or they can use a free hosting facility provided by Bitkeeper (www.bitmoover.com). Being publicly available has another advantage: The time lag between the maintainer's development version and the latest official release can be greatly reduced, if the maintainer pushes his modifications to his public repository, from which user-developer can synchronize their child repositories. The maintainer can decide to publish the latest changes at any given interval to minimize merging conflicts. The local repositories and easy merging between versions are particularly interesting for open source developers. Unlike employees in a firm working at the same time at the same location, open source developers are far more distributed. Open source developers are scattered all over the planet working at different hours and not constantly online. By cloning a parent tree, a user-developer can work from the child snapshot and later synchronise with the parent.

Bitkeeper supports the highly hierarchical organisation of the Linux kernel by allowing only higher-level maintainers of a parent repository to push to child repositories. Lower level developers must ask maintainers of parent repositories to pull and can only themselves pull from the parents.

9.3.2 FreeBSD Development

FreeBSD is an open source operating system, which like Linux is a Unix clone. FreeBSD is developed and organised differently and rely on a core team and CVS (Concurrent Versioning System) for source code control.

In Linux a single person is the top maintainer, who can single-handed decide what goes in and what does not. Projects using a core team rely on a group of people with no single authority to decide, what goes in and what does not. In FreeBSD, the core team acts as the project's Board of Directors [Kamp 2000] and consists of nine persons.

FreeBSD rely on the CVS source control system for keeping track of modifications. CVS functions as a central repository, where anybody is permitted to obtain the latest version, which is called "checkout"). However, only people with permission are allowed to upload (called "commit") modifications to CVS. The core team is responsible for handing out commit rights to persons, whom they see fit for the task.

When using CVS, the first thing to do is to retrieve a copy of the project source code by using the following command:

```
cvs -d ":pserver:anoncvs@some.project.org:/home/cvspublic"
login
```

This will download a copy of the project source code to the local computer. CVS is designed to use individual files and does not generate meta-data, which makes it difficult for CVS to track files with changed names. Developers using CVS develop in the same way as developers using patch/diff and Bitkeeper. Once the latest source code has been downloaded, a private development tree is created as a copy of the original, and development commences using the copied tree. The original serves as a local reference to which modifications in the copied tree can be compared.

Once the developer is satisfied with the changes, it is FreeBSD policy that, the developer must submit the patch to the development mailing list for review. It is further required that a patch remains on the development mailing list for 72 hours, without receiving significant criticism, before committing the modification to CVS [Kamp 2000].

In this respect CVS, unlike Bitkeeper, provides all developers with commit right and equal opportunity to upload changes. Developers are only restricted by an agreed set of norms of how to make developments in the FreeBSD project. Developers, who do not adhere to the agreed norms, are likely to have their commit privileges revoked sooner rather than later.

When a patch has survived on the development mailing list for 72 hours, the developer must re-sync with CVS before committing his modifications. If anyone has been working in the same area, it is likely that the patch does not apply, and the modifications will have to be merged by hand. Having resolved any conflicts, the developer can commit his modifications to CVS. Should any changes have been committed since last sync, CVS will give an error message and specify where the problem occurred, but no special tool is available to ease the merge process.

CVS does offer a number of features for helping developers keep track of versions by allowing different branches of the source code to bare different names. For instance, the project might want to have at least two versions available, 1) A stable version and 2) A development version. The stable version is for users, who just want to use the program without having to care about new features breaking all the time. In this branch only bug-fixes are made, and no new experimental features are added. The development version, on the other hand, is for developers trying out new features and code.

It is an ongoing process, where the infant project reaches a level of maturity, where developers decide to call the code stable. At this point a new development branch is created to allow development to continue without breaking the stable version. All sorts of branches can be created to accommodate various desires.

9.4 Data on Open Source Software

This section makes an attempt to present a combined overview of the some of the quantitative studies of open source software that have been published. The studies have focused on four areas 1) The size of a Linux distribution, 2) Who are the developers of open source software, and why do they develop, 3) Demographic composition, and 4) Market share.

9.4.1 The Size of a Linux Distribution and the Cost of Adoption

Studies of size of the code in Linux and distributions are interesting and reveal a great deal about the effort that has gone into creating Linux and a Linux distribution. David Wheeler has made a considerable effort to measure and compare Red Hat 6.2 and 7.1. The source code for all the packages offered in the distribution, including the kernel, has been measured and fed to the COCOMO cost model [Wheeler 2001b]. The COCOMO cost model is a model for estimating time from product design to actual implementation and testing. COCOMO can then be used to estimate the cost of developing a given amount of lines of source code based on the average cost of a programmer plus overhead. Wheeler [2001b] also analysed the code to identify the licenses used.

The Linux kernel used in Red Hat 7.1 is kernel-2.4.2 and consisted of little more than 2.4 million SLOC (source lines of code). What is interesting is the fact that 1.400.000 SLOC (57%) are drivers for all sorts of different peripherals (scanners, mice, video adapters, sound cards etc. etc.). The most used license in the Linux kernel is the GPL license.

When using the COCOMO model for estimating costs, Wheeler [2001b] discovers that commercial development of all the software contained in the Red Hat 7.1 distribution would cost little over $1 billion in year 2001 dollars, and would require 8.000 person years to develop. The cost of the one-year younger Red Hat 6.2 distribution is $600 million and requires 4.500 person years. This is a difference of $400 million and 3.500 person years in just one year. The difference is attributed to the increasing maturity of open source software, which could be included in the Red Hat distribution. Linux distributors are interested in including as much software as possible, without compromising the overall feeling of quality. Thus, programs, which are tested to be unstable or to otherwise cause problems, are not included in the distribution.

Analysing the distribution as a whole, a variety of licenses are used, but the most dominant licenses GPL and LGP with respectively 55,3% and 10,1% of the lines of code are closely followed by the MIT license. A very small portion of the code ~0,5% did not

have any copyright and in the public domain (abbreviated to PD in Table 6). The license types noted as Dist in Table 6 are distributable licenses. Distributable license are licenses that permit redistribution of the original code but not of derived works, as exemplified by the license for the PINE mail- and newsreader. Distribution of derived works is only allowed, if the copyright holder grants special permission. Although the data are obtained from the Red Hat 7.1 distribution, they are not specific to the Red Hat distribution. Most distributions include roughly the same applications, which is natural, as they are open source and available to anyone.

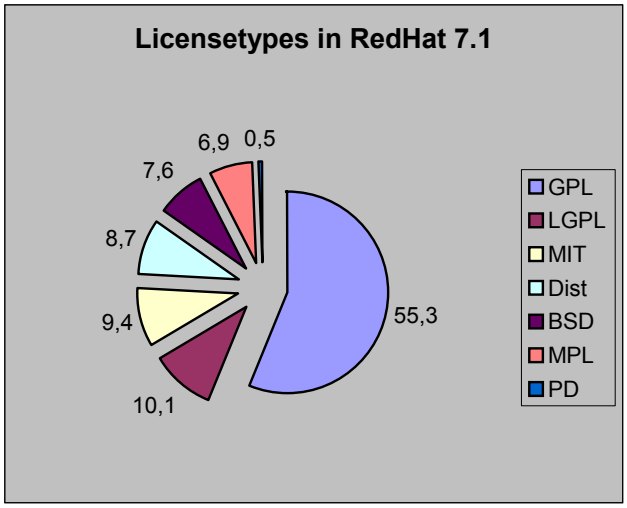


Table 6: Distribution of licenses used in Red Hat 7.1 based on SLOC. PD is puplic domain and Dist is distributable [Wheeler 2001b].

The total number of lines of code in Red Hat 7.1 is 30.152.114 and seems staggering. For comparison the following table shows the number of SLOC in other operating systems.

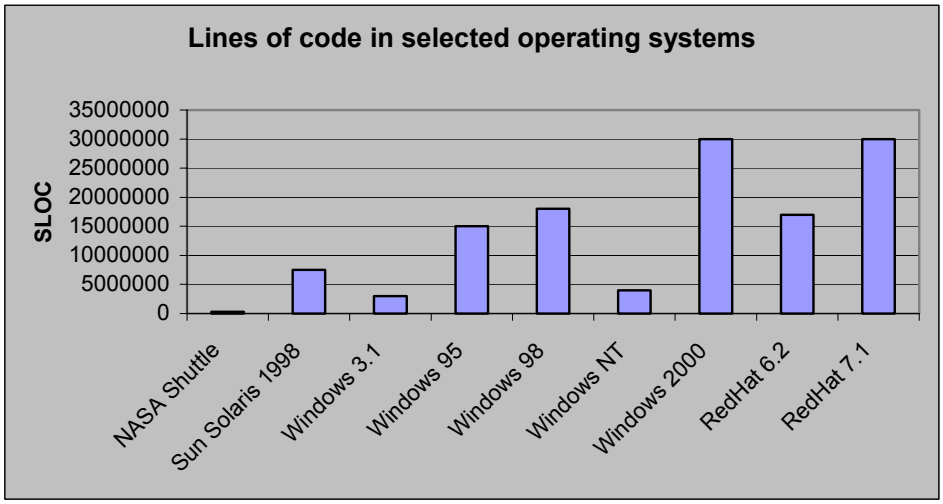


Table 7: SLOC in selected operating systems [Wheeler 2001b], the number for Windows 2000 was obtained from http://www.osopinion.com/Opinions/RussellBishop/RussellBishop1.html.

However the numbers must be taken with a pinch of salt, as Linux distributions such as Red Hat 7.1 contain far more applications than does Windows or Sun Solaris. Many of the counted lines of code stem from the programs, which are contained in the Red Hat

distribution. For instance, the version of the Mozilla browser shipping with Red Hat 7.1 is made from 2.065.224 SLOC, which is 6,7% of all SLOC. It should be noted that the windows operating systems also ship with the Internet Explorer browser included. Apart from Mozilla and Internet Explorer, a Linux distribution such as Red Hat also contains a vast number of other programs such as compilers, a verity of window managers, and often several choices of programs that solve the same problem. For instance, a distribution as Red Hat includes at least 5 different browsers: Mozilla, Konqueror (spelled correctly with a K), Nautilus, Lynx, and Netscape.

The latest version of the Red Hat distribution, version 7.3 as of writing is roughly 50% larger than Red Hat 7.1. This is based on the amount of megabytes of data contained in the distribution CDs.

Wheeler [2002] compares the price between Microsoft Windows 2000 and Red Hat Linux 7.2. The comparison identifies the costs associated with setting up a public web server or intranet file server and email server, which use a relational database and C++.

	Microsoft Windows 2000	**Red Hat Linux**
Operating system	$1510 for 25 clients	$29 standard, $76 deluxe, and $156 professional. Unlimited clients.
Email server	$1300 for 10 clients	Included, unlimited clients
RDBMS server	$2100 for 10 clients	Included, unlimited clients
C++ development	$500 pr. Person	Included

Table 8: Price comparison between Microsoft Windows 2000 and Red Hat Linux, [Wheeler 2001a].

A public web server with a RDBMS (relational database management system) totals at $3.610 for a Microsoft Solution and $156 for a Red Hat solution. A file server with email server comes at $2.810 for the Microsoft solution, and $76 for the Red Hat solution. Bear in mind that unlimited clients can connect to the Red Hat solution, while Microsoft like other software vendors charge pr. client connecting to the server.

Another survey compared the initial costs of setting op a computing solution for companies of varying sizes in Table 9.

	Microsoft solution	**Linux solution**
50 user company	$69.987	$80
100 user company	$136.734	$80
150 user company	$282.974	$80

Table 9: Comparing cost of a complete software solution base on either Microsoft or Linux for different size companes [Cybersource Pty. Ltd. 2001].

From both tables it is obvious that there are significant amounts of money to be saved by using Linux and open source software based solutions. An open source software license allows a company to freely use the same copy of a program on all computers in the company. Microsoft licences do not allow this practice and require all installed programs to be licensed. Other studies have shown similar results of open source software solutions are cheaper than similar products from Microsoft [Harris 2001, Jimmo.com 2001].

Upgrade costs for the two solutions show a similar pattern, as there is significant cost associated with upgrading Microsoft solutions. In contrast, open source software is virtually free to upgrade by either downloading upgrades or purchasing the latest version of the operating system [Wheeler 2002]

Looking at the numbers in the Table 8 and Table 9, it makes perfect economic sense to choose open source software solutions. However, the above numbers do not take into account the cost of compatibility, re-training of staff not used to work with Linux based systems, and the possibility of open source software not being able to deliver the desired functionality. In particular the Microsoft Office application has become a de facto file standard and competitors must be compatible. Presently only Windows and Mac users have access to native[35] versions of Microsoft Office, and Linux users may buy the CorssOver Office product from CodeWeavers[36] and run the Windows version through a compatibility layer. The download edition of Crossover Office is $54.99 and the CD edition is $64.95[37]. The solution works, but 100% functionality is not guaranteed, and this solution also adds a further cost to Microsoft Office, although costs are saved by not having to buy a Microsoft Windows2000 operating system at the cost of $262,99 at Amazon.com[38]. Thus using Microsoft Office on Linux through Crossover Office would save $202 from not having to buy a Windows2000 license. It must be observed that most end users obtain their Windows2000 license as part of buying a computer at a much lower price; and often it is not possible to separate the license from the computer. However, system builders would be able to save a lot of money by installing a downloadable Linux distribution and Table 9 is testimony to this, illustrating the saving obtainable for companies of varying size.

Microsoft Office being the lingua franca of the Internet makes it important that Linux office applications are able to use and export these file formats. Not being able to do this will severely hamper adoption of Linux on the desktop. Although there exist file formats supported by both Linux and Windows applications such as the RTF file format. However using RTF or any other file format, which is not default, requires user intervention and thus not a solution.

Most people are used to work with Microsoft products, and implementing an open source software solution in a company will require extensive re-training, and lower productivity must be expected for extended periods of time.

For server purposes it is fairly easy to replace Microsoft with open source software solutions, and the cost of re-training the computing staff is easily covered by reduced licensing costs. However, some functionality is not yet available as open source software, and in this situation a change is impossible. For instance, the Microsoft Exchange server integrates perfectly with the Outlook client and offers some very nice collaborative features. While the Outlook client can be replaced with Ximian Evolution, and if added features are bought from Ximian, it is possible to have the same functionality as Outlook.

[35] A native version is a version of a program written for the same platform as it is used.

[36] CodeWeavers has created a commercial product based on the open source WINE project, which tries to implement Windows infrastructure on Linux so that Windows software will run on Linux. See www.codeweavers.com

[37] Prices obtained from www.codeweavers.com the 28th August 2002.

[38] Price obtained 28th August 2002.

For many purposes it is entirely possible to substitute a Windows desktop with a Linux based one. For most purposes such as email, word processing, and browsing, open source software programs are of comparable quality. In particular the openoffice.org office suit has reached a level of quality and functionality, making it a candidate for replacing Microsoft office.

During this project I have attempted several times to convert to Linux and have failed every time. The section 9.2.5 illustrated that Linux distributions are easily installed and come with a complete set of applications. However, from personal experience I have found the one thing missing that has prevented me from switching. The screen appearance of fonts on laptop displays is horrible. It is not possible to use openoffice.org for continued periods of time without getting sore eyes. The problem is twofold: 1) available fonts and 2) the FreeType library responsible for rendering fonts on the screen. Only a few fonts, which are displayed in high quality on the screen, are freely available. Microsoft has inadvertently helped open source users by releasing a handful of fonts on the Internet. Still, freely available, high quality fonts are few and far between. FreeType is open source, and naturally only limited resources are available for the project. Achieving good-looking screen fonts are indeed a difficult endeavour, and while Linux has come a long way in the past few years, it is far from good enough for me as a desktop replacement.

9.4.2 Who are the Open Source Developers and where are they?

While most studies of open source include some element of empirical research, only a few larger surveys has been conducted. Some of them will be referred in the following pages.

David Lancashire [2001] analyses the nationality and number of developers in the Linux kernel and the GNOME projects. The conclusion from the survey finds that the United States rank number one in terms of total number of developers for both the Linux kernel and the GNOME project. Naturally, absolute numbers are seductive and do not tell anything about the relative merit of the countries. The United States is by far the single most software developing and exporting nation in the world and thus the number of programmers contributing to Linux kernel or the Gnome project will be the highest in absolute numbers.

When analysing the number of developers per capita, the United States are no longer dominating, in fact it falls to an average 10^{th} place [Lancashire 2001:14], and the European countries rise to the top. In particular the Nordic countries (Denmark, Sweden and Finland) rise to top of the list of the most contributing nations per capital. In general most developers come from Canada, United States, Mexico, Brazil, Australia, and the European countries. Sub-Saharan countries do not appear in the survey.

Lakhani et al. [2002] analyses open source development and has analysed a 10 % sample of OSDN (Open Source Development Network). OSDN is a web site, which offers hosting of open source software projects, and it is the single most used facility for this purpose. OSDN offers a web-based interface to the projects, a repository for the source code, easy set up of mailing lists with extensive search facilities. In short OSDN offers most, if not the entire, infrastructure required to run an open source software project and it is free. OSDN has in excess of 337.000 registered users and over 33.000 hosted projects. Lakhani et al. [2002] selected a 10 % sample of the developers, which were sent an email containing a link to the web-based survey, and 34,2% responded.

The survey tried to identify the motivation for participating in open source software development. The motivation for participating was segmented into four categories: 1)

Believer (33%), 2) Fun Seekers (25%), 3) Professionals (21%), and 4) Skill enhances (21%).

All Believers thought that code should be open. Fun Seekers answered that the code they developed were used in a non-work functionality and that they found open source development intellectually stimulating. Professionals used open source software as part of their work functionality and derived status from their work on open source software and also status within the open source community. Skill enhancers were those who participated, because it made them better programmers.

Lakhani et al. [2002] also identified the nationality of the respondents in the survey, and came to similar conclusions as did Lancashire, although only absolute numbers were used. The Americas (Brazil, Canada and United Stated) contributed 48% of the responses, Europe 42,2%, Australia 5,2% and the rest of the world 4,6%.

A part of the survey focused on the amount of time, which developers spent on open source development. Respondents were asked to rate the amount of time spent on "this project" and "all projects". "This project" was the particular project, on which the respondent was working. Over 35% of the respondents spent 1-5 hours a week, 20% spent 6-12 hours a week, and about 40% spent more than 12 hours a week on open source development of a selected project. The mean for "this project" was 7,8 hours a week, and the mean for "all projects" was 14,4 hours a week. 30% of the respondents contributed to only one project while 25% contributed to two projects, and more than 40% contributed to more than three projects.

Predominantly, the respondents in the Lakhani et al. [2002] survey were experienced IT professionals. Almost 45% worked as programmers, 20% were students, 5% system administrators, 5% IT managers, 5% from academia and the remaining 20% were grouped as other. On average the respondents had 11 years of programming experience.

Koch and Schnieder [2000] provide data on the GNOME project and analyse development intensity among the GNOME projects members. By scanning the CVS repositories, Koch and Schnieder [2000] are able to obtain precise numbers of how much the individual developer contributed to the project. The study measured lines of code (LOC) added or deleted per individual programmer, and the striking observation is that a very small number of developers are responsible for adding the vast majority of code. The mean LOC added by all programmers was 21.000, and the maximum added by an individual programmer was 931.000.

The Linux kernel and the GNOME are both projects that attract many developers. However, many of the open source software projects in the world live a much more quite life and never reach the high numbers of more than 300 developers as reported in the Linux kernel or the GNOME project.

Krishnamurthy [2002] has examined a random selection of 100 open source projects hosted on the OSDN. OSDN groups the projects after their stage: Planning, Pre-Alpha, Alpha, Beta, Production/Stable, and Mature. At the time of survey, the number of projects marked at mature were 480, and out of these 100 were selected for further study. The total number of projects was 29.274. Mature projects were chosen, as they had existed for some time (on average 1,5 years before the study), and mature projects are also expected to have generated a stable user and developer base, which uses and maintains the project software.

The main finding from the survey was that the majority of the projects were developed by a small number of programmers. On average the number of developers for a

project was 6,61 with a minimum of 1 and a maximum of 42. 19% had more than 10 developers, and 22% only had one developer.

9.5 Seven Mini Cases

This section presents a series of mini cases describing open source software development efforts undertaken by the respondents of the interviews. All of the respondents have been involved in a variety of open source efforts ranging from software development to participating in standard bodies and all Danish residents.

The cases have been extracted from the raw interview transcriptions by identifying particular development histories matching the general model presented in chapter 7. The interviews, which can be found in the separate interviews volume, contain much information about open source software development and being part of the open source community. Interesting as the interviews may be, much of the information falls outside the scope of the model, and in general they are far too comprehensive to be included.

Extracting a series of mini cases allow for subsequent analysis based on a general model. The analysis will be conducted in the following chapter (Chapter 10), where the maintainer and user-developers will be identified in the mini cases. Identification of agents makes it possible to use the model to understand the actions of the agents. The results of the analysis will then be used to further analyse the interviews in search of support of the proposed rationale.

The mini cases are structured in the following way: A brief mentioning of the programming background and of some of the projects of the respondent, and afterwards the actual mini case is described.

9.5.1 Henrik Størner

Henrik Størner had been using Linux since 1995, when he became interested in development of the Linux kernel. Henrik Størner (HS) has participated in several open source software projects including the kernel, Fetchmail, and a driver for reading Windows FAT partitions. HS holds a M.Sc. in computer science and has worked as a programmer and network security consultant. In the following we will focus on HS' involvement in two projects: 1) the Linux kernel and 2) Windows95 long file names.

9.5.1.1 The Linux Kernel

HS became interested in Linux and in particular the Linux kernel in 1995, and he subsequently subscribed to the Linux Kernel mailing list (LKML). LKML is the mailing list, which anyone wishing to work on the kernel has to subscribe to. When subscribed, all messages sent to the LKML is automatically forwarded to the subscriber, who can then follow and participate in the discussions.

In early 1995 Linux had reached version 1.2, and development was proceeding rapidly with several new versions released a week. Following the Linux kernel, development meant downloading the latest version of the kernel and trying it. The procedure for testing a kernel consists of downloading it, configuring the new kernel, compiling the new kernel, and rebooting to test it.

When downloading, one could choose to download either the complete kernel source, the size of which was around 1.8 to 2.2 megabytes[39]. Another alternative was to download a patch containing just the difference between the latest and the previous version. The sizes of the patches are significantly smaller ranging from a few kilobytes to under a hundred kilobytes. In 1995 when the Internet connection offered a maximum of 28.8 kilobit per second patches was preferred way of upgrading a kernel. After obtaining the latest version of the kernel, it had to be configured in order to compile the correct drivers matching the hardware. The next step was to compile the kernel, and here HS began to stumble on some problems. The process of compiling a kernel took more than half an hour on the hardware that was available to HS. This was a significant amount of time, and the process often had to be repeated because of errors in the new source code.

Around the time of the 1.2 series kernel, the Linux kernel had been redesigned to support loadable modules. Loadable modules mean that additional functionality can be inserted into the running kernel, e.g. if different hardware is connected to a computer, it is possible to load a driver (module) without having to reboot. Modules also have the advantage of keeping the memory footprint of the kernel as low as possible, as unused modules are not loaded. Linux distributors are particularly interested in modules, as they allow distributors to create a common kernel and to support modules that will run on a wide variety of hardware.

When configuring the kernel, the user could choose between compiling some functionality as modules or directly into the kernel. But, no matter how the kernel was configured, the complete source code had to be compiled every time, and that would take at least half an hour.

HS observed that when the compile process failed, it often happened in a module that HS did not use. It was fine to fix problems with modules for which he had a personal need. However, waiting for all the modules to compile and on top of this being annoyed with unneeded modules that didn't compile – that was unbearable.

HS began to investigate the problem, and found that it would be relatively easy to modify the way in which the Linux kernel compiled. The change involved adding a new choice when configuring the kernel, and the choice was not to compile the selected functionality. The result would be that only the required functionality would be compiled, saving a lot of time during compilation. It also meant that functionality that did not compile could be left out, and the compilation could be completed.

Implementing the change was not difficult, but it required that all the scripts controlling compilation had to be modified. The changes were made over a weekend and submitted to Linus Torvalds for inclusion in the kernel. They were accepted and included in the subsequent kernel version.

Some discussion erupted on LKML following the implementation of HS' changes, and some argued that they had discussed the idea earlier. No matter the complaints, HS was the first to actually implement the changes, and thus his name entered the credits file.

9.5.1.2 Windows95 Long File Names

HS also participated in a project about Windows95 and its long file names. In the old days of DOS and Windows3.11 a filename could only be 8 characters long and have an

[39] Size estimates obtained from www.kernel.org kernel archives [accessed 7. October 2002]

extension of three characters. But with the advent of Windows98, Microsoft changed the file system to support filenames of 32 characters, an improvement as meaningful filenames could now be used.

Many users of Linux configure their computer to dual boot into both Linux and Windows, and in this way it is possible to use both operating systems. Windows saves its files on a windows partition on the hard disk, and is unable to read the partitions used by Linux. Linux, on the other hand, is able to access data on the Windows partition and this worked for regular DOS file systems. But Linux was unable to use the long filenames of Windows95.

Like many others, HS had his computer configured to dual boot, and he also used the long file names of Windows95. However, to his dissatisfaction, the standard Linux kernel did not support windows' long filenames. Incidentally, an unofficial driver (a patch) for the kernel existed and provided the needed functionality. A US citizen named 'Gordon' maintained the patch. Gordon had created the driver for Windows95 by copying the functionality of the original DOS drivers and extending it. And the source code for the driver was "a complete mess".

HS was often testing new kernels, and obtaining the required functionality meant patching the new kernel with the patch from Gordon. Gordon was actively maintaining the patch and was modifying the patch to fit the new versions of the kernel. To compile a new kernel, HS had to obtain the patch from an annoyingly slow ftp server in Berkeley, USA.

The only way to avoid having to patch each and every new version of the kernel was to have the long file name patch included in the official kernels. HS contacted Gordon and proposed that they together cleaned up the patch, and submitted it for inclusion in the official kernel. Gordon agreed that it was a great idea, but he was preoccupied and would not have the time, so he was more than happy to let HS do the work. HS accepted, and this was HS' first attempt at real kernel development, and he notes:

> "There is a certain responsibility associated with kernel development. If you make a mistake it might result in the operating system crashing and the user loosing his data!" [Størner 2000]

HS did not assume maintainership of the driver, and it was agreed that Gordon would continue as maintainer and be responsible for the core logic, while HS restructured the code. The core logic, containing the additional functionality required to read Windows95 long filenames, was separated. Code that was reused from other parts of the kernel was identified and included by liking directly to the original code.This would make the driver much easier to maintain, and the added bonus was no code reuse. The long filename driver would automatically use updates to the other drives, as they shared the same code.

The modifications were submitted to Linus Torvalds for inclusion in the mainline kernel, but were rejected. It so happened that Linus Torvalds did not approve of the C coding style of HS, and changes had to be made. HS had about three iterations with Linus Torvalds, before the driver was accepted.

All the projects, in which HS has been involved, have used the GPL license. Although HS is pragmatic in his choice of the open source software licenses, he does prefer the GPL, and he notes:

> "… although the BSD license is a bit antisocial as you can take some code and turn into a proprietary program." [Størner 2000]

9.5.2 Keld Jørn Simonsen

Keld Jørn Simonsen (KJS) has been an active programmer, has developed various programs, and has also contributed to the sendmail project. Presently KJL is actively involved in the standardisation of the C programming language, POSIX standardisation, and localisation both by acting as a translator and by proposing standards for character sets. Character sets refer to the mapping between an underlying symbol and a specific character. When a computer sends an email, this email is encoded using the character set used on the computer, and if the receiver uses a different character set, some characters will be displayed wrong. This issues is of particular importance to nations, whose alphabet contains special characters like the Danish alphabet containing æøå.

The case represented here will focus on the Danish localisation of Linux, i.e. the effort to provide a Danish version of Linux for people, who only understand Danish.

KJS' interest in localisation began in the early 1980ies, where he was a systems administrator at DIKU (Department of Computer Science University of Copenhagen). DIKU served as an email hub in Denmark in the early 1980ies, which received emails from Europe and re-distributed the emails to University Departments and firms using modems. Many of the sites, from which the emails were received, did not use the same character set, and at one point about 60 different character sets were in use. The problems with character sets has made Danish localisation of Linux and is very important to KJS:

> "I have been into Unix and Linux a number of years and consider myself a bit of a missionary in the area. It was so that I took the initiative to make Linux Danish in February or March (2000)" [Simonsen 2000]

A complete localisation of Linux requires all programs to be translated into Danish. It is not the actual program source code that has to be translated, but the messages that the program presents for the user. Some programs have these messages hard coded in the source code making them difficult to translate. The larger projects store their messages in separate files, and the translation process requires the Danish translations to be inserted in the files, or a new version of the file with a name reflecting the Danish content is saved. This makes it easy to maintain several language versions of a program.

KJS is involved in translating some of the major distributions like Red Hat, Mandrake, and SuSE. But since the distributions share many of the same programs, the effort to do so just requires translation of the distribution centric items. For instance, all of the mentioned distributions contain both the GNOME and KDE desktop systems. The translation of these is not done within the framework of the distributors but contained in the projects.

As some of the work is done for commercial Linux distributors, there are deadlines to be met, for instance Red Hat is committed to release the next version of Red Hat Linux at a given date. The translators must have their work completed on time to be included in the next version. The projects stimulate the translators by setting up web pages detailing the progress of each of the national translations efforts:

> "There is an internal competition going on and there is a certain amount of prestige associated with being the leading team. From the Danish side, we have been leading for the Mandrake distribution for a long time. Yesterday, there were none other which had 100% (completion rate)" [Simonsen 2000]

It might not be the same team working on all the translation projects, as there are many teams working on different projects. One team might be working on the GNOME project, while another is working on the KDE project.

The Danish translation teams from a variety of projects holds monthly meetings. The purpose of the meetings is, besides the social nature, to coordinate use of wording and to ensure that words are translated in the same way in the different projects. It is important for the translators that users are presented with a consistent Danish user interface using the same terms and phrases.

The translation efforts are organised with one person responsible for translation of the next version. It might be the same person, who is responsible for other versions, but only one can manage a particular version. Some projects use a website listing all that has to be done before the translation is complete. People interested in participating can obtain a work package from the website. The website will then mark the work package as work in progress for others to know.

Each of the translation teams have their separate email list that serves as the primary means of communication. For instance, there is a list for the translation of Mandrake, and this list also receives notification from Mandrake that new packages have arrived, and that they need translation.

Within translation projects, there is a danger that two persons are working on the same translation and thus doing redundant work. The texts for translation are kept in a CVS (Concurrent Versioning System) repository, which ensures that all changes can be clearly identified and removed, if they are wrong. When the new text is in CVS, the translator retrieves a copy of the text and begins to work on the translation. When the text is translated, it is sent back to the CVS repository, and the next person retrieving a copy will get the latest version containing the changes.

However, if it happens that two persons unawares work on the same text, the first person to return the changes to CVS will be the one, whose translation is used. The other person will receive an error message from the CVS system, and his work might end up not being used. This serves as an incentive to finish a translation once pulled from CVS:

> "Ones work might be wasted if the other upload (to CVS)
> before one self. Sometimes I wonder that I have to hurry
> before the others start to do too much work" [Simonsen 2000]

Although it happens, and serves as an incentive to complete a work package, it is rarely a problem, as the translation community is small enough for most translators to know what the others are working on.

9.5.3 Jens Axboe

Jens Axboe (JA) is a Linux kernel developer, who is currently employed by the German Linux distributor SuSE. JA has worked on many aspects of Linux, and besides his work on the kernel he is the CD-ROM maintainer. The job of the CD-ROM maintainer is to keep the CD-ROM drivers current and make sure that it supports as many as possible of the available CD-ROM drives. We will focus on JA's work as a CD-ROM maintainer and as the developer of his program "packet-cd", a program for writing CD-RW's (re-writable CD's)

9.5.3.1 The CD-ROM Driver

JA began working with the CD-ROM driver, before he became CD-ROM maintainer. JA was experiencing problems with his CD-ROM drive and began to investigate, and at some point JA had to contact the maintainer of the CD-ROM driver. Together they made fixes for some of the problems. The maintainer did not have the time to maintain the code and asked JA to become maintainer of the CD_ROM driver. As JA expressed:

> "This was my luck, cause he asked if I would (become maintainer) and I would." [Axboe 2000]

From that moment JA kept getting more and more involved in kernel development. Although JA became the official CD-ROM maintainer, he still had to submit his changes to Linus Torvalds to get the changes into the kernel. The CD-ROM drivers are just one of the all the drivers in the Linux kernel, and Linus Torvalds has the overall responsibility for the Linux kernel. While maintainers assume personal responsibility for the code, they maintain, they do not automatically get code included in the kernel. Linus Torvalds in his capacity of maintainer of the kernel must approve modifications. In general the kernel development process is tough on developers:

> "It's an incredible tough way to develop (code). If you present something that is poorly designed or poorly coded, then it can't be used for shit. Then it's back to the drawing board" [Axboe 2000]

Most of the work was done at home involving little collaboration with the others. Usually the process is that someone proposes a solution to a problem to another developer on the mailing list. Some of the developers will go through the code to see, if the problem is solved correctly. Since anyone can look at the code, there is a greater chance of getting problems fixed instead of buried, which was JA's experience from working in a firm producing closed source software.

The CD-ROM driver project turned out to be terrible, as there are 50 CD-ROM vendors all twisting the CD-ROM specifications to their personal need. It is a dreadful task to manage, which JA did not anticipate when becoming maintainer of the project. For other driver projects, like, for instance, network cards there is a specific driver and perhaps a few workarounds.

Currently, (at the time of interview) JA does not spend a lot of time maintaining the CD-ROM project, actually he spends very little. However, the advent of DVD-drives has resulted in the CD-ROM driver having to be updated, as it is important that Linux supports DVD-drives. Although all the code developed for the project is tested before submitted to Linus Torvalds, there are bugs, but some of the bugs cannot be corrected, before they are identified, and this usually happens on a user's computer.

Although the amount of feedback is limited to mostly bug reports, it is only in the rarest occasions that a patch is included in a bug report. When submitting bug reports there is a standard procedure for users to follow, and this usually makes it easy to identify the problem and solve it. On some occasions JA has asked the bug submitter to try out a few test cases, or JA has written a patch for the user to test. In a few extreme cases JA has had to ask the troubled user to let him log into his computer in order to examine the problem. The number of bug reports greatly depends on whether a new kernel has been released. Following a new release more than 10 bug reports can arrive in a day, and in other periods

there can be weeks between them. JA feels that in the long run it pays of to help users solve their problems:

> "When there is one reporting a problem and he has to lead me to the problem. Usually it is something like holding his hand through 10 emails until the problem is solved. But it pays off cause the chance of him sending me a patch the next time is much better. A very small percentage include patches, this is because it is the kernel, people are intimidated by getting into it just because it is the kernel – although it's just a program like all other" [Axboe 2000]

Sometimes it is not possible to get the documentation needed to create a driver, and in this case JA has to disassemble the Windows driver and see which registers are used. From there on it is a matter of trial and error.

9.5.3.2 Packet CD

Packet CD is a kernel patch developed by JA to allow packet writing to re-writeable CD-ROM disks. The idea is to be able to merely write to the CD-ROM drive without having to use a special application. The CD-ROM behaves like any other hard drive and saves the data on the CD-ROM.

The project began in 1999, when JA needed a feature that was not present in Linux, and he just wanted to:

> "Throw it (a CD-ROM) into a drive, mount it, throw some files on it, and just take it out (the CD-ROM disk). Ehhh it was just like Adaptec DirectCD under Windows - and I wanted it on Linux" [Axboe 2000]

JA started coding on the project, but he soon lost interest, and the project lay dormant for some time. Later the project caught interest again, and JA resumed his work. The project was rewritten from scratch four times, before a good design emerged. When developing a program or feature, an overall structure of the program must be made, before actual coding begins. However, during the process of writing the code, the weakness of the initial design became apparent, and sometimes assumptions did not hold true and:

> "There is actually a lot of them but some of them is so unfortunate that you have to say: 'This sucks!' You actually have to rewrite the whole thing" [Axboe 2000]

The project was hosted on sourceForge.net[40], which also managed the mailing lists that served project communication. When packet-CD was first released, there was a lot of response, and the patch was downloaded by a plethora of persons. As of writing packet-CD has been downloaded 9000 times[41]. When new a version is released, there is usually a lot of feedback and:

> "´when nothing happens it's just a few emails a month saying 'I cant compile this'. If I'm in a bad mood I just erase them" [Axboe 2000]

[40] SourceForge.net can be accessed on the Internet at [Website] http://sourceforge.net/projects/packet-cd

[41] SourceForge.net collect statistics for all projects and the packet-CD stats was obtained freom [Website] Statistics http://sourceforge.net/project/stats/?group_id=7091 accessed 8. October 2002

It is a problem attracting developers for the project. Many persons are interested in the functionality but not many have submitted patches. People have to be very interested in the project before they make modifications as submit patches and for this reason JA has done most of the work the packet-CD project. However, some collaboration has taken place between JA and the person, who is responsible for the UDF-file system, on which the packet-CD is depending. The collaboration between the two was good, and they complemented each other, as JA knew about CD-ROM drives, and the other developer knew about the UDF-file system, and they shared the interest in getting packet-CD up and running.

The packet-CD patch is still available and applies to the latest 2.4.19 kernel, although no changes have been made since July 2001.

9.5.4 Peter Makholm

Peter Makholm (PM) is a Debian maintainer, and he maintains several packets that are part of the Debian project. Debian, as discussed in section 9.2.4, uses a special packet format making it easy for Debian users to maintain their system and install programs.

Developers of a program rarely provide more than the source code for the program, i.e. packaged versions are not supplied. This is where the Debian maintainers fit in, as a Debian maintainer is responsible for packaging programs into easily installable Debian packages. The packages must be maintained by updating the package to reflect the latest version of the program. The Debian maintainer performs a service to all Debian uses by providing an easily installable program, where users do not even have to compile the program in order to use it. This mini case will focus on Peter Makholm (PM) as a Debian maintainer of the game Slash'em[42].

PM began his work as a Debian maintainer in 1997, mainly because there were some interesting programs, which did not exist as Debian packages. Turning these interesting programs into Debian packages and make them part of Debian provided a sense of giving something back. PM was no longer 'just' a user.

A few of the packages, which PM currently maintains, came to his attention, as they were not actively maintained. The original maintainers of the packages had lost interest, and PM took over maintainership. PM was using the packages and found it annoying that they were no longer maintained.

Slash'Em is an adventure role-playing game descending from the role-playing game Nethack. The development of Nethack was going too slow, and there was some discontent with the Nethack maintainers, the Slash'Em mission statement reads:

> Provide the Nethack community with a more open, collaborative and responsive model for the game's development. To this aim, the Slash'EM developers make ongoing source changes publically available, incorporate patches in a timely fashion, and are active members of the newsgroup." [http://www.slashem.org/ assessed 12 0ctober 2002]

[42] The slash'em home page can e found at http://www.slashem.org/

The discontent sparked the development of Slash'Em[43], which was based on release 3.3.1 of Nethack. Naturally Nethack and Slash'Em are both open source software, and this allowed the development of Slash'Em to be based on the source code of Nethack.

As a Slash'Em maintainer PM is placed between the end user and the developers of Slash'Em. When a new version of Slash'Em is released, PM obtains the source code for the new version and creates a Debian package, which is subsequently released to the public. This in-between role causes users to turn to PM for advice or simply to complain if the package or program does not function. Most users have no clue whether it is the actual package or the program contained in the package that is malfunctioning:

> "If there are errors in the package, then they (the Debian users) email me and we have to find out if it was my way of packaging the program that is the problem or if I have to send the problem further upstream (to the Slash'Em developers)" [Makholm 2000]

The package maintainer adds a layer between the program developer and the user, and ideally the maintainers should provide the program developers with the fixes they develop. However, PM notes that this collaboration is not perfect in all situations. In some situations package maintainers fail to return their modifications upstream to the developers of the program.

The source code for Slash'Em is contained in a CVS repository where anyone can access the latest version of the program. Although the source code is contained in CVS, the development team has a release policy, which ensures that only well tested versions are released as stable version. People wishing to just play the game are advised to use only the stable version, whereas the more experimenting types wishing to test upcoming features should try the development versions.

It is only the stable version that will be packaged by the Debian maintainers. Development versions are better described as being in a constant flux, whereas the stable versions freeze for long periods of time (from one to several months).

In course of maintaining the Debian versions of Slash'Em, PM has been involved in making changes to the source code, which have been returned to the Slash'Em maintainers. Although Slash'Em is contained in CVS, PM does not have access to committing to the CVS repository, and instead PM has to return his changes to one of the Slash'Em maintainers, who then commit the changes to CVS after reviewing these.

The actual work of creating a Debian package is a process, which may require some changes to the source code of the program. The Debian maintainer too must package the program in such a way that all files when installed wind up in the correct locations, notably in the locations preferred by the distribution. Some programs place their configuration files in the directory */usr/etc*, while it is Debian policy to place configuration files in the directory */etc*. This may require some tinkering with the program to ensure that distribution policy is upheld, but once done, and being aware of what changes to make, the process of making new releases becomes quite easy. The Debian maintainer then knows what files to modify in order to make a Debian package.

Making Debian packages is more than just un-packing files into the preferred location on the hard drive. If the program to be packaged needs some networking services,

[43] The Shash'Em software and information about the project can be found at: http://www.slashem.org/#whatis

the package must check, if the service is installed and running. If this is not the case, the service must be installed and started.

If the program has to make changes to existing configuration files, things become a bit trickier:

> "… I mustn't touch other people's configuration files. And those configuration files that I MUST make changes to I have to do using the scripts, which the owner of the configuration file have provided." [Makholm 2000]

The Debian maintainer also makes sure that the package can be uninstalled in case the user dislikes the program. The uninstall process should remove any files installed by the package. A bit more complicated are those packages, which make changes to existing configuration files in order to function. When such packages are uninstalled, the package must contain such information that the modified files can be returned to their original state. Again, and this can only be done using the scripts provided for the job.

9.5.5 Claus Sørensen

Claus Sørensen (CS) was at the time of interview chairman of KLID (Commercial Linux Interest in Denmark), and an active member of SSLUG, where he is engaged in discussions and book writing. CS also teaches Linux to students and gives talks about Linux and open source software as part of his private company PLOMUS.

The interview with CS covered many areas, and we shall take a brief look at CS' work in KLID and a closer look at CS as a part of the SSLUG book writing effort. SSLUG is a Linux user group in Copenhagen and has consistently maintained an effort to write and publish high quality books and articles about how to use Linux. The main effort has been a series of book, which roughly translates into "Linux – The Freedom to XXX". Where XXX can be substituted with "choosing office applications", "System administration", and 10 other titles. The books are distributed under a free publication license, which as open source, allows anyone to use, modify, and distribute the books. The books can be downloaded from the homepage of SSLUG[44].

KLID is an interest group under DKUUG (the Danish Unix User Group) seeking to promote Linux on the commercial arena, and KLID has about 40 member firms. The main activities of KLID are to hold regular meetings with talks targeted at commercial interests and to write articles in Danish computer magazines. The effort of writing articles in KLID resembles the process of developing open source software. CS authored an article in Computer World and used the KLID network to circulate the article before sending it to Computer World.

The book writing effort of SSLUG involves the work of many individuals, who write sections, chapters, or edit what is already written and check for errors. The Linux books are contained in a set of files available for all interested. Lately, CS has corrected a number of errors in "Linux – The Freedom of Security":

> "What I did was to read the book at say: 'No, What is this they have written'. And then I ended up, instead of reading the book, then I sad down and corrected the book – I always get the source files for the books. So there was just 100 corrections that made it into the book that night. That is

[44] See [Website] http://www.sslug.dk/linuxbog/ for a complete list of available books.

> the way I do it, if I have to read it, that I might just as well correct it" [Sørensen 2000]

The Linux books are formatted using the free publishing tool Docbook, which formats the text using a mark-up language. The Docbook format makes it very simple to publish the books in a variety of formats including PDF, PostScript, HTML, and others. The Docbook format makes it easy to distribute chapters among writers, as each chapter is a file. The writers of the Linux books have agreed on a common layout. Having many people working on the same book at the same time may create problems:

> "I know I have work to do because I actually made a new division of the books. […] But five iterations have happened since then so I have to do a new division and I haven't taken the time to do it. […] And therefore it (the division) has become too old. If you don't have the time don't even start. As soon as you reach a point where you don't want to do any more, just send it off, even if it's not finished. Just say it's not finished and it might be used instead of just hiding it" [Sørensen 2000]

9.5.6 Kenneth Rhode Christiansen

Kenneth Rhode Christiansen (KC) is a GNOME developer, who has been deeply involved in development of the Gnumeric[45] spreadsheet. KC is also involved in a variety of translation projects and was instrumental in organising the year 2001 GNOME conference held in Copenhagen. KC has been involved in many projects, since he began using Linux many years ago, when Debian version 1.1 was the latest approximately 1996. KC is currently involved with the graphical design of GNOME and has contributed both icons and widgets KC has coordinated many bug-fixes in GNOME, before version 1.0 was released. Apart from the GNOME work, KC has contributed to the GIMP photo-editing program. The interview with KC was very long and the amount of detail staggering. This section will focus on KSs' work with GNOME in general.

GNOME is a project that aims to provide a free desktop environment including applications such as office tools. GNOME provides an overall framework that serves to provide means of integration between individual GNOME programs. GNOME is shipped as a single large entity containing the desktop environment, and all the individual programs that are part of GNOME. The GNOME project consists of a host of different programs fitting neatly into the GNOME framework. The individual GNOME programs are by a host of different people maintain each their preferred GNOME program.

KC has been working on different aspects of the GNOME project for a long time and began developing an interest for the GNOME project, when he started using Linux. KC was interested in having a nice, clean, and consistent desktop, and much to his dismay this was not available under Linux….

> "When you started a program which used one toolkit then the buttons would be displayed in one way. Starting a program, which used a different toolkit the buttons, would be displayed in another way. Nothing fit together and I just wanted something or someone that tried to do to so (create a consistent desktop)" [Christiansen 2000]

[45] Gnumeric is an open source spread sheet which can be obtained at [Website] http://www.gnome.org/projects/gnumeric/

GNOME was the desktop environment that looked the best, and consequently KC got involved in the GNOME project, and the first effort concentrated on getting the GNOMElibs[46] to function correctly. Many programs depend on GNOMElibs to function, and modifying GNOMElibs often resulted in the failure of many other programs. Each time GNOMElibs was modified, other programs also had to be slightly modified, and that was a time consuming and iterative process. But it was important to bring the infrastructure (GNOMElibs) up to a certain level of quality allowing it to be used by all the other programs without crashing all the time. The process started with GNOME version 0.13 and continued for a long time adding new components to the basic GNOME infrastructure. It was a very exiting time, where each day meant improvements to some programs or libraries. At approximately version 0.30 GNOME began to almost function and it was almost possible to compile and install it.

The improvements went on for a long time, and at one point it was decided that GNOME was to release version 1.0, which was a psychologically important release indicating the GNOME was now ready for day-to-day use…

> "I compiled a huge list of all the bugs – I think I spend a week just finding the bugs. Ad when the list came, it sort of became the list, to use when fixing bugs. More and more bugs was listed and the list grew bigger and bigger and then a bit smaller. The list was then distributed at regular intervals" [Christiansen 2000]

The bug fixing effort culminated with a Linux Expo somewhere, and Miguel Icazza, the leader of the GNOME project, wanted version 1.0 to be ready by then. The timing was bad as there were only about 50-100 core developers plus the testers working on it. Version 1.0 was released, and people booed GNOME off the SlashDot stage because of all the bugs still present. One of the results from this experience was the opening of separate branches in the CVS repository. A head branch was created, where stable versions would reside, and only tested modifications would be applied. Then a few development branches were created, in which all sorts of new things could be tested.

This has now become a standard procedure, when new versions are to be released. A few months before the release date, a feature, and API[47] freeze are issued. Beyond the freeze it is not possible to add new features or changes the API, and the time until release must be spent correcting bugs and making the programs stable. The advantage of this approach is to avoid the situation, when a change to the API (GNOMElibs) resulted in all sorts of changes that had to be made to the programs using GNOMElibs. Program developers can develop their program and rest assured that it will remain compatible at least until the next major release.

All the individual elements of GNOME, be it libraries or programs are assigned to a maintainer, who is responsible for the element. The complete source for GNOME is maintained in CVS allowing everybody to get hold of the latest source. However, not everybody is allowed to upload changes to the CVS repository. The process of getting permission to upload changes to the repository is based on peer review. If a person without CVS access has been making changes, he must go through the maintainer of the modified programs to get them into CVS. The maintainer serves as quality control and only allows approved changes into CVS.

[46] GNOMElibs provide common functionality for all GNOME programs.

[47] API (Application Programming Interface) The interface other programs use to interact with the program.

When a developer has consistently been making changes to programs and the maintainers have approved them, time has come to let the developer have access to the CVS tree:

> "It is not something you just get (CVS access) you have to get around and become accepted. It could be if I for instance had developed a patch and the maintainer for the program says 'Write Miguel and tell him from me that you should have CVS access" [Christiansen 2000]

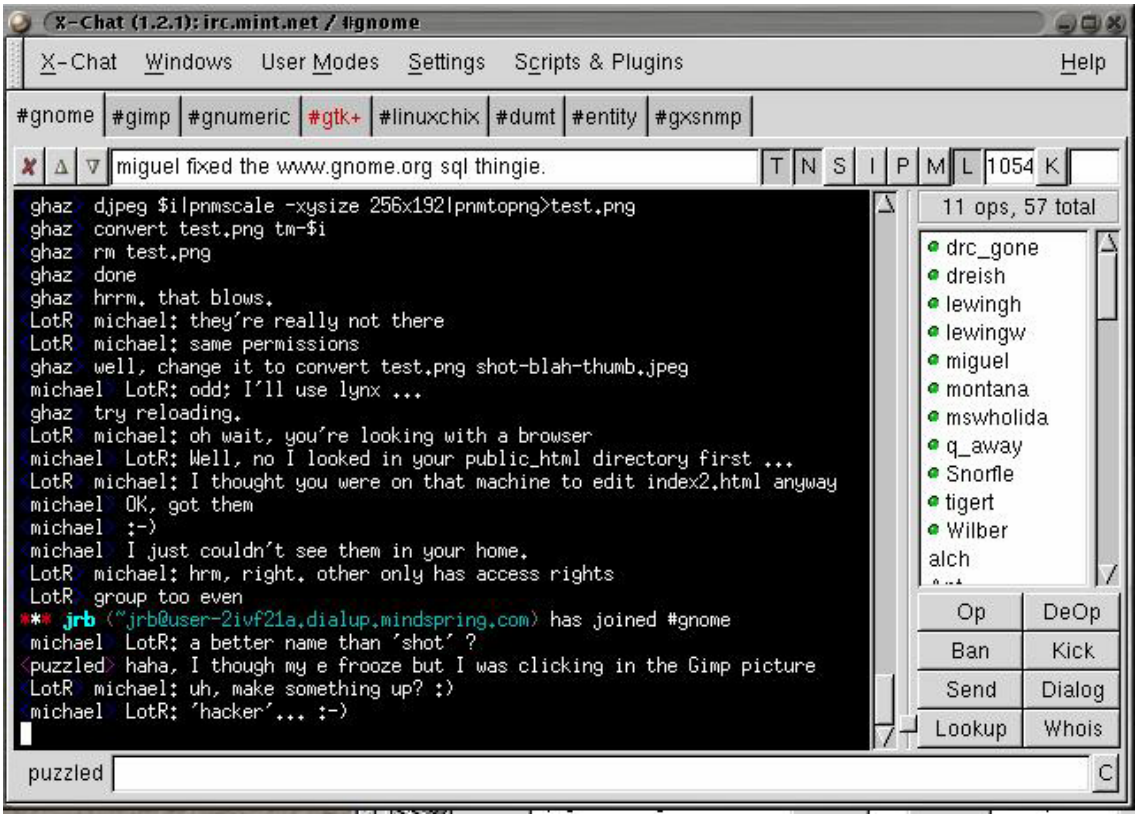
This is the usual process, when people wish to obtain CVS access. It happens that some people do not know how to use the CVS properly and accidentally erases lots of elements. It happened for an Italian translator, who inadvertently erased three complete programs from the CVS repository. The programs were later restored, although it required some effort, and it serves to illustrate that CVS access should not be granted to anyone.

Maintainers of a program or an element do not stay maintainers of the same area for a lifetime. They are perfectly free to do so, but from time to time maintainers loose interest in a program and find new and more challenging programs. It often happens that a developer creates a project, and suddenly 10 other developers are contributing code to the project, because they use it. When the developer no longer needs it, he might start on another program, and just maintain the program he started. Then he might say:

> "Hmmm, Jody, I'm just sitting and fixing this and you simply write too much code. Don't you think you should take over?" [Christiansen 2000]

In this way the maintainer duty of the different elements passes from one person to another. Still, each package contains a maintainer file stating who is the maintainer, and who has been working on the package, thus making it easy to identify the maintainer.

The actual development in the GNOME project uses mailing lists, but more importantly IRC (Internet Relay Chat, see Picture 6). IRC is a real-time chat system, where participants can write messages to each other and actually discuss things in real time. There is common GNOME developers' channel, where people meet, and this channel serves as a general forum developers. It should be noted that the channel is not open to the general public, only GNOME developers are allowed.



Picture 6: The X-Chat program in action talking in the #gnome channel. The right pane shows people present on the channel and the banners show active channels.

Using IRC has the advantage of being real time, and thus developers can discuss problems…

> "Yesterday I discovered something in the latest CVS version of Gnumeric. When I inserted something it would erase the frame around the object. It was just a single line that had to be fixed then it worked again. So I just wrote it on IRC, it's much easier to explain than using email. People can ask 'I don't understand'. In email you just press erase – there are 300 mails to be read anyway" [Christiansen 2000]

Some of the maintainers of the different GNOME elements are usually to be found in their respective forum, for instance, translators and people writing documentation can be found in the #docs channel. Anyone with questions about translation can proceed to the #docs channel, and be almost sure to reach some of the persons working on the subject and get an instant answer. Chat programs like the one shown in Picture 6 contain a paging function to catch people's attention. If two or more developers have created a private forum to discuss matters relevant only to them, other developers can page the developer they need to reach. Although each participant in an IRC discussion has to write complete sentences and transmit them, communication via IRC is efficient. Abbreviations, acronyms, and sign language are quickly developed and used as seen on Picture 6, where sign combinations like :-) are used to illustrate a smily face.

The social element of IRC should not be neglected, and KC underlines that it is indeed nice to small talk with the friends you have made while working on GNOME. There is a lot of small talk about vacations and day-to-day issues. KC notes that seemingly

a lot of time is wasted chatting and discussing irrelevant issues for the programming problem at hand. Yet, it does serve to keep developers interested, as it is nice to be part of a community. Discussing things thoroughly before implementing them seems to make it easier to create the first version, and the first version is usually better than just a rough private implementation. Some of the initial design problems will be caught, and this shows in the results.

9.5.7 Poul-Henning Kamp

Poul-Henning Kamp (PHK) is a FreeBSD core team member, who has been working for many years on the FreeBSD operating system. FreeBSD is a Unix like operating system descending from the original BSD Unix, which is recognised as being very reliable, for instance, the internet site Yahoo runs on a farm of FreeBSD servers. PHK also served as release engineer for FreeBSD version 2.0. PHK runs a private consulting company focused on software development, and he has worked in various positions as a software developer. Furthermore, he is working on a book in collaboration with two other writers on the subject of network timeservers, and claims to have the most precise timeserver in the world with a drift within the 10-nanosecond range. The interview, like the interview with Kenneth Christiansen, was indeed very elaborate and covered many areas of open source software and FreeBSD development. This section will focus on PHK's work as a FreeBSD core team member developing code and integrating code from other developers.

FreeBSD is an operating system that resembles Linux in many respects and runs most of the same software as well. FreeBSD and Linux live together peacefully but have historically appealed to different groups. Whereas Linux got hold of university students and younger people in general, the FreeBSD camp is populated by an older generation:

> "I believe if we take a look at the two developer groups there is an average age difference of about 5-10 years. I believe it makes a difference. I don't have the youthful energy to go out and demand my $50 back from Microsoft[48]" [Kamp 2000]

The two camps occasionally look to each other for ideas on how to solve a particular problem. Linus Torvalds has mentioned that he of course takes a look at how FreeBSD has implemented things.

FreeBSD is organised around a core team that decides what goes into FreeBSD, and what does not. The core team is responsible for the development of FreeBSD, and the team also has the last word when choosing the direction of development. Members of the core team are not members by birthright but chosen by the other members.

FreeBSD uses CVS to manage their source code, and thus everybody has complete access to the latest version of the source code. Privileges to commit changes to CVS, on the other hand, are not granted to anyone and it is the job of the core team to decide and approve who is allowed to commit. Currently about 250 people around the world have CVS commit rights. Anyone can suggest that a person should have CVS commit rights:

> "If it's a name you have seen a couple of times before or you might even have received patches from the guy or you know him personally. Well, um, yes" [Kamp 2000]

[48] Some years ago a Windows refund day was announced and a large group of Linux users drove to Microsoft and demanded to get their money back – which they did.

Others may insist on filing one bug report after another, all documented, and with patches. When this becomes a burden for the core team, the only resolution is to grant the person commit rights to CVS and make sure that he cleans up any code he may have messed. In other occasions, although rare, people come forward and ask the core team for commit rights. Still others have to be persuaded to accept the commit rights, and when people have accepted, they must pass a small ritual. They have to write their name in the right place in the manual and write a small note about themselves. Often a mentor is appointed to look after the newcomers, just to hold their hand and show them how the project works. Usually the newcomer gets accustomed to the ways of the project and is on their own in no time.

Even though many people have commit rights, it is not allowed to just commit changes, everyone must obey the 72-hour review rule. If a developer has made changes, the changes must be announced and put up for review within at least 72 hours. 72 hours was chosen, as three days pass, and this allows people to come home from weekend and review the changes. If no major objections have been made within the 72 hours, the developer can commit the changes

The FreeBSD project uses the same means of communication, as do most other projects like mailing lists and web pages. But recently chat has become accepted, and the FreeBSD developers have created their own non-public chat forum. The forums are logged, and the developers can at any time review the logs. The advantage of chat is that it goes beyond email and is real time. Email has the nasty tendency of making people disagree and too often angry at each other, due to lack of detail in expression:

> "If you compare with a conversation, you completely lack the ability to emphasize things. There is no modulation at all. The word stands as the word stands, you can add smiley's but what does it mean? We are back to letters and never has anyone been as insulted face-to-face as they have been by a letter" [Kamp 2000]

Chat has the advantage that misunderstandings can be cleared instantly, and insults are avoided. There are times where both emails and chat are insufficient, and the only way out is to grab the telephone and discuss things.

As part of his consulting work, PHK was involved in modifying FreeBSD to suit the needs of a small American Internet Service Provider (ISP). The small ISP was interested in launching a new product for individual hosting, and the idea was to allow customers to have full control of a computer and let them have their own IP-address. This would allow customers to run web or mail servers from the computer. The computer, however, was not to be a real computer but a virtual machine. From the customers point of view this would appear to be a complete individual computer.

The small ISP was already running FreeBSD on their production machines, but it would require modifications to the FreeBSD kernel to implement the wanted feature. It would require a feature called "jail", where each virtual machine runs in a jail and is unaware of any other virtual machines running on the same computer. Technically the jail required changes to all places where the kernel checks what user is requesting what from the operating system – rather security critical.

PHK agreed to implement the feature, and in the course of doing so he discovered that the way FreeBSD checked for the superuser was poorly implemented. Consequently implementing the feature also resulted in a major cleanup:

> "It takes a bit of ice in the stomach to do it because the 362 places that had to be changed was all over the kernel devices, memory management, and file system" [Kamp 2000]

The feature was implemented, and the small ISP was able to offer virtual hosting to their customers. The changes went into CVS and became a part of FreeBSD.

PHK, like other open source software developers, is interested in software licensing but takes a preference towards the BSD license. The BDS license, unlike the GPL license, allows the licensee to modify the code, publish it, and it does not require the changes to be public. And here lies the core of PHS's argument in favour of the BSD license. The BSD allows firms to use the code and mix it with their own proprietary code without having to fear that competitors can obtain the source code and learn secrets.

A company called Juniper Networks manufactures routers using high-level processors that are among the best in the world:

> "One of the founders of the firm needed something to run in the high-level processor. If he ran Linux on the processor, then all his proprietary code has to be GNU (GPL license). If he chooses to run FreeBSD on the processor, he can keep all his proprietary drivers and oddities to him self. We have got a lot back from him, we got serious network fixes because it is easier for him if they are integrated in FreeBSD then he doesn't have to maintain them at home" [Kamp 2000]

It is more important that people use the code rather than contribute back. Recently PHK discovered that one of his password scramblers had found a way into some of Ciscos' routers. A friend of his at Cisco confirmed that the source code contained his name and copyright. The program was licensed under the beerware license, and the source code at Cisco contained a note: "If anybody meets this guy, buy him a beer". As PHK never made any money from open source software development, he felt that people might as well use his software. The BSD and Beerware license offer greater potential for adoption, than does the GPL license. Had PHK used the PGL license he would not have made more money than he has done from the BSD and Beerware license.

Experiences like Cisco's use of his code are very satisfying and serve as a pad on the back implying that this is good code.

IBM got entangled in some of PHK's software that was licensed under the beerware license. Unlike Cisco, IBM was in serious doubt of how to interpret the license; was is just one beer or one beer per copy of the program, or was it the customers of IBM who should to provide beer for PHK?. Consequently, the legal department of IBM contacted PHK and asked just how much beer they would have to provide in order to satisfy their license obligations. PHK answered:

> "I explained to them that since they had to consider it (the amount of beer) they obviously did not like the software enough, and therefore it did not matter. – They were very relieved" [Kamp 2000]

9.6 Summary

This chapter has presented the history of Unix and Linux and tried to describe open source software from a users perspective. Open source software development was exemplified with Linux and FreeBSD and a number of quantitative surveys were referred.

Seven mini cases based on interviews with open source software developers were presented. The mini cases focused on particular development stories in order to shed light on how agents acted in an open source software development projects.

The following chapter (Chapter 10) uses the mini cases for the purpose of testing the model software development and consumption.

Part III: Test and Discussion

The first part of the thesis was about the existing knowledge about open source software, developing research questions, and presenting the theoretical tools, which are to be used.

The second part of the thesis presented new knowledge. A model of open source software development was developed and coupled with three licenses. The coupling between the three licenses and the model resulted in an understanding of the involved agents behaviour. Part II ended with a presentation of the empirical evidence gathered during the project. From the raw interview transcripts found in Volume III, seven mini cases were deduced. The mini cases were individual development stories from the persons interviewed.

The third and last part of the thesis tests the model developed in part II and discusses the complete thesis. Chapter 10 tests the model developed by using it to analyse the development stories. The complete thesis is discussed in chapter 11 where all main chapters and results are discussed. Lastly a conclusion is offered in chapter 12.

10 Testing the Model

This chapter will test the model software development and consumption developed in chapter 7. The model software development and consumption will be used as an analytical tool, and the seven mini cases presented in the previous chapter (Chapter 9) will be analysed. In this respect, the question asked is: "Does the model provide insight into the behaviour of the seven interviewed developers?" The model is considered successful, if it is possible to use the model as an analytical tool and to understand the behaviour of the interviewed developers. The judgement of the model's fitness will be passed as part of the following chapter "Discussion" (Chapter 11). This chapter only focuses on the analysis.

The chapter will begin by discussing how to use the model as an analytical tool and then proceed to analyse the cases extracted from the seven interviews.

10.1 Using the Model as an Analytical Tool

The model focuses on two types of agents: The maintainer and user-developers. The maintainer is the person or firm, who develops and distributes a program. The user-developer is the person or firm, who uses the program, and depending on the license, makes modifications to the program. The user-developer can then decide whether to keep the modifications private, distribute the modifications as open source, or distribute the modifications as closed source.

When the model was developed, the theoretical argument played a central part in deducing the reasons for the maintainer and user-developer to make either of the possible choices in the model. The theoretical argument is still interesting, but only as far as it is an explicit part of the model. Theoretical references going beyond the model will not be used, as it is the model, which is to be tested at its own merits.

The model focuses on a development situation, where a number of agents interact with the intent to develop a particular program. Interaction is tied to a particular program, which the agents use and/or develop. The model analyses the behaviour of the maintainer and the user-developer, and therefore it is important to identify the maintainer and the user-developer.

The model is tied to the license and states that behaviour is a consequence of the specific license. For this reason the first step in the analysis is to identify the license of the program, around which the mini-case revolves.

The maintainer and the user-developer are identified in each of the cases by analysing, to whom the user-developer returns his modifications. A user-developer will return modifications to the perceived maintainer of the program. This makes it easier to identify several levels of maintainers and user-developers, and to establish how they change over time. For instance, a person may maintain a particular version of the program for which there is a separate user-base. The question arises: Is the person a maintainer or a user-developer? In this case the person is a maintainer, as he has users, who turn to him with problems and patches.

The situation becomes a bit more blurred, when the maintainer of the patch tries to get the patch included in the main program. Now who is who? The maintainer now

becomes a user-developer seeking to have his work included in the program maintained by the maintainer.

It is important to make the distinctions between the maintainer and user-developers clear and to explicitly state, what type of agent is related to a given situation. Behaviour changes, as the agent changes type. For instance, a maintainer is more concerned with the quality of modifications to include in his program, while a user-developer is more interested in just having his desired feature included as soon as possible.

The procedure of using the model as an analytical tool will be to briefly mention the development case. This serves to refresh the memory and to identify the program developed or used in the development story. Identifying the program is important, as it also identifies the license used, and thus the model that will be used in the analysis. The next step is to analyse the agent's type, and this serves as the basis for understanding the agent's behaviour. In the event of the agent changing type, if the program changes maintainer or is included in a larger program, this must be identified and the impact on behaviour explained.

10.2 Henrik Størner

Henrik Størner (HS) was involved in two development stories: 1) The Linux Kernel and 2) Windows95 Long File Names.

10.2.1 The Linux Kernel

HS implemented a new feature, which allowed only the selected parts of the kernel to be compiled thus reducing compile time significantly. The code, which HS modified to do so were pre-existing and licensed under the GPL. Thus the modified and derived also had to be GPL. This places the analysis within the model of behaviour relating to the GPL license.

The scripts were pre-existing, and the infrastructure of the kernel was pre-existing as well. HS was implementing a new feature on top of existing work and sought to have the changes included in the program. This categorises HS a user-developer, who chooses to have modifications to the program distributed by a maintainer. The maintainer was Linus Torvalds, as he was the person to comment and include the modifications in the following release of the scripts.

As a user-developer, HS was using the program distributed by the maintainer (Linus Torvalds). User-developers have two possible choices: 1) Just use the program or 2) Make modifications. HS had been a user for a long time but was beginning to be annoyed with the compile time of a complete kernel. This prompted HS to make modifications to the program.

User-developers, who choose to make modifications, have a further choice of keeping the modifications private or to distribute them. If distributed, the source code for the modifications must be publicly available.

The modifications made by HS reduced the compile time significantly, as unwanted parts of the kernel were not compiled. If HS had kept the modifications private, HS would not have sustained the advantage from his modifications. HS would have had to make his modifications to all new kernels before saving time during compilation.

The original modifications took about a weekend to complete, and undoubtedly future modifications would take less time to complete. However, the modification involved

modifications to the make files[49] controlling the compilation process. Kernel developers frequently modify the make files when new features are added to the kernel. HS would then have to edit the new and modified make files, each time a new kernel was distributed. This process is likely to consume more than the half hour of compile time, which motivated the development effort in the first place.

The opportunity cost of HS' development effort must be considered to be very high, since the work was done in HS' spare time, and he already had a fulltime job. HS could have chosen to perform actual paid work in the weekends but has chosen to spend the weekends as leisure time.

The time frame, which HS had available for leisure i.e. toying with Linux, testing new kernels, was small. Thus waiting time represented a large opportunity, which was cost equal to the fraction of available time consumed by unnecessary compiling.

Being able to avoid spending time with unnecessary compiling maximises the time available for leisure and provides an incentive to make the modifications. The modifications took a considerable amount of time – in effect all the available (and valuable) leisure time of one weekend. The only way the time investment would become feasible was, if the modifications made by HS were included in the kernel distributed by the maintainer of the kernel (Linus Torvalds). The cost of personally maintaining the modifications would easily surpass the waiting time of compiling the kernel.

The make files used by everyone to compile the Linux kernel represented a simple use-product, as most users would consume the same use-product. Unlike large complex programs, the make files were not manipulated by users, but usually only by developers adding elements to the Linux kernel. Still the use-product desired by HS was not readily contained, and HS decided to invest time in developing the desired functionality.

10.2.2 Windows95 Long Filenames

In this development story HS engaged in a development effort in order to make the Linux kernel support the new file system for Windows95 allowing Linux to make use of the long file names.

The feature of accessing a Windows95 file system already existed as a patch but was not in the official kernel, which required that HS had to patch his kernels before compiling them. It also required HS to fetch the patch from a slow FTP server somewhere in Berkeley. Both were time consuming tasks that took time from the interesting aspects of testing new kernels.

Given the existence of a patch and the fact that HS based his modifications on this patch, makes HS a user-developer. The person who was maintaining the patch, Gordon Chaffee, is the maintainer, with whom HS interacted. The patch was licensed under the GPL, and thus behaviour of the agents must be understood within the GPL model.

In order to avoid wasting time when patching the kernel and downloading the patch, it had to integrated into the mainline kernel. HS was well aware of this and also of the work required for doing so. HS (the user-developer) contacted the maintainer and proposed that the patch was rewritten to fit into the mainline kernel. However the maintainer was not interested in doing the work, and thus HS had to do the work, if he should avoid constantly

[49] Make file(s) is one or more files containing instructions for the compiler about how a program should be compiled.

patching the kernel. Although the maintainer was not interested in doing the work, the maintainer was interested in discussing the modifications made by HS and making sure that the core logic was sane.

The involvement of the maintainer in the development helped reduce development time, as HS could focus on merely restructuring the code, while the maintainer made sure that the core functionality was working.

When HS and the maintainer had finished cleaning up the code, the new patch was sent to Linus Torvalds for integration in the kernel. HS was responsible for doing so, and by trying to integrate the patch in a larger project it would appear that Linus Torvalds became the new maintainer. This, however, is not the case. The former maintainer (Gordon Chaffee) and HS were still the persons with intricate knowledge of the patch and would be considered maintainers, i.e. problems found in the code would be directed to them.

The advantage to both the maintainer, Gordon Chaffee, and the user-developer, HS, was the lowered cost of using the patch. HS did not have to spend time obtaining the patch and patching the kernel, and Gordon Chaffee no longer had to actively maintain the patch. Kernel developers, whose projects touched the code, now became responsible for any changes they made, and for the way in which this affected the functioning the of long filename driver. Anyone disrupting the functioning of a driver or other features would be responsible for fixing the error themselves.

The further advantage of the integration, which in particular applies to Gordon Chaffee, is the improvement achieved by restructuring the driver to rely on existing infrastructure of the kernel. When the infrastructure is updated and improved, the long filename driver automatically benefits.

It would have been possible for HS and Gordon Chaffee to keep the changes private, but the whole point of the rewrite was to avoid the extra work when testing a new kernel. Both HS and Gordon Chaffee are individuals with no incentives to commercialise from the driver. The driver was already licensed under the GPL, and this would have made it impossible for them to distribute the driver without making the changes public.

Both developers, HS and Gordon Chaffee, obtained an increase in human capital for doing the work, and also their names in the credits file. Whether the increase in human capital has actually paid off, is unknown. It is, however, certain that the long filename driver has been associated with HS and Gordon Chaffee. And although the changes were done a long time ago, the kernel still carries reference to their work:

```
"VFAT extensions by Gordon Chaffee, merged with msdos fs by
Henrik Storner" – Source code for Linux version 2.4.19
```

10.3 Keld Jørn Simonsen

Keld Jørn Simonsen (KJS) was active in the Danish localisation of Linux and has translated for a variety of programs and distributors. The case focuses on KJS involvement in translation, and although it is not software, the development process and organisation appear to be similar to software development. For this reason the model for software development is also used to analyse KJS involvement in translation and in localising Linux.

The basic work performed by translators is translating the files containing words and phrases used in the programs. A translator downloads a file containing a list of words and returns the list when the translation is completed. Translators do to some extent coordinate

their effort to avoid double work and keep translations consistent within and between programs. Consistency in terms and phrases is most important, and the aim is to have the same wording and use of phrases across distributions.

The files that KJS received are under the GPL license placing KJS in the GPL model, where KJS can either keep the modifications private or distribute them publicly. KJS returns translated files to the maintainers of the program, and thus KJS becomes a user-developer. KJS is involved in several translation efforts, and a single maintainer cannot be identified. Instead there are several maintainers each of which maintain a program that are is in the process of being translated.

The interaction between the maintainer and user-developers is different from software development, as the maintainer does not have the capability to serve as a quality reviewer. Thus the maintainer can disclaim responsibility for the translation and include it with minimal effort.

The maintainer is interested in having the widest possible potential user base, as this provides the maintainer with the highest degree of exposure. Whether the maintainer is a firm (Linux distributor) or an individual a large, potential user base provides an incentive for including the translation in the program or distribution. This is also the incentive for the maintainer to provide the necessary infrastructure in the program to allow multiple languages to be used. It requires an investment on behalf of the maintainer to make his program able to handle multiple languages. A program with no support for additional languages can use simple print statements to pass messages to the user. Multiple language support requires the program to be designed using the Gettext package. Gettext is an open source system that allows a program to use message catalogs for storing messages.

Thus we can easily understand the incentive for the maintainer to provide infrastructure and include translations in the program, and although the maintainer does not directly enhance his personal use-product, it does allow the program greater exposure to users. Some of these users might indeed find the program sufficiently interesting to offer patches that enhance the maintainers' use-product.

User-developers, i.e. the translators, do not - on the other hand - have the same incentives as does the maintainer. The user-developer (KJS) was performing a time consuming task of translating programs or distributions. KJS was performing the translation in his spare time, and he mentioned that some of the translations had taken three to four days to conduct. For his work KJS received a packaged Linux distribution from the distributor worth approximately 70$. The cost of his time, assuming he was working 8 hours a day for three days, would be $70/(3*8)=2,9$ $/hour roughly half the Danish minimum wage. However, if we include the Danish tax of 50%, the value of the $70 is doubled if it was to be bought of a normal salary. While the tax perspective adds to th incentive the pay for performing translation is only roughly equal to the Danish minimum wage and must be considerable lower than JKS' opportunity cost. KJS has been working as an independent consultant and his opportunity must be at least 5 times the minimal wages. From a monetary perspective there is little incentive to become a translator, which indicates that a monetary incentive is not enough to explain the involvement of user-developers.

KJS could decide to use his translations privately and just enhance his private use-product. Given the pace that new distributions were released or programs updated, it would require a large proportion of KJS' time to obtain the desired use-product. This gives the user-developer an incentive to return translations to the maintainer.

User-developers are organised in groups discussing their work and making sure that translations are consistent across programs and distributions. KJS referred to regular meetings where translators would meet face-to-face. In the model the user-developers were discussing their modifications with other agents, who modified as well. The model also contains a link to the maintainer, but the maintainer is not part of the discussion.

Being part of a group that shares an interest can in itself be very motivating, and in light of the model some of the explanation of the efforts of user-developers should be found here. Reputation effects are also present, as the translators become recognised by their peers and users of the translated programs. The reputation effects can be viewed as user-developers gaining human capital from their translation work. The translations are proof of their ability to translate from one language into their native language. This skill could easily be demonstrated to a possible employer by showing the working programs with Danish translation.

However, translation skills are less precise than software development skills, as the translation files do not carry direct reference to the actual person translating a particular set of lines. Therefore, capitalizing from a skill demonstrated in translating might be harder then for software developers. The effects are nonetheless present.

Lastly personal interest and a desire to obtain a particular use-product also serve as an incentive. KJS has a deep personal interest in localisation and has worked on the subject on several levels.

10.4 Jens Axboe

Jens Axboe (JA) worked as a fulltime open source software developer employed by the SuSE Linux distributor. JA's development stories were exemplified by two projects: 1) CD-ROM maintainer and 2) PacketCD.

10.4.1 CD-ROM Maintainer

The function of the CD-ROM maintainer is to continually maintain the CD-ROM layer[50] in the Linux kernel. New drives come out all the time, and occasionally new CD-ROM formats are introduced, and it is the job of the CD-ROM maintainer to ensure that they are supported.

The CD-ROM layer is part of the kernel and licensed under the GPL license thus placing the behaviour of JA within the GLP model. JA took over an existing body of code and had no influence on the original choice of license.

Historically, JA got involved in the CD-ROM layer by sending fixes and updates to the previous maintainer. Thus JA started as a user-developer developing the CD-ROM layer and returning the changes to the maintainer. The original maintainer did not have the time to maintain the CD-ROM layer, and JA took over and became the official maintainer. The official maintainer is the person, who receives patches, updates, and bug reports for the part of the kernel he maintains.

As CD-ROM maintainer JA becomes a maintainer, who interacts with a number of user-developers. The CD-ROM layer is part of the Linux kernel, but the modular design of the Linux kernel makes it a self-contained par of the kernel, and the CD-ROM layer can be

[50] The CD-ROM layer is a broad reference to kernel functionality and drivers required to use CD_ROMs under Linux.

modified separate from the kernel, as long as the interface between kernel and CD-ROM layer is not changed.

When JA has made his changes to the CD-ROM layer, they must be included in the official Linux kernels. In this situation JA changes from being a maintainer to being a user-developer agent. It should be noted that it is not just Linus Torvalds, who maintains an official kernel, as Linus Torvalds is currently only maintaining the 2.5.x development series of the kernel. Marcelo Tosatti is the official maintainer of the 2.4.x series kernels, and still other persons maintain the 2.0.x and 2.2.x series.

The model distinguishes between firms and private developers, and JA has the potential to change role. The Linux distributor SuSE employs JA, but his position at SuSE allows, within limits, JA to choose what to work on. On occasions JA has to perform software development work in the direct interest of SuSE. For instance, when a new version of SuSE Linux is about to be released, JA must help making the distribution ready, and it often entails creation of a custom kernel containing the features, which the business strategy of SuSE dictates. SuSE does not dictate the work of JA in his capacity as a CD-ROM maintainer, although the work is performed in time paid for by SuSE. JA is supposed to work full time on the Linux kernel development, which includes being CD-ROM maintainer. Thus JA should be analysed as an individual, despite the fact that JA is an employee with a firm. When accepting the offer to become maintainer of the CD-ROM layer, JA becomes locked up in a behaviour, where he must develop the code and distribute the modifications. It is in the nature of the maintainer to continue making modifications and distribute new versions. The code was under the GPL license and modifications to the code are considered derived work, which must also be licensed under the GPL.

There are high signalling effects associated with maintaining something as generic as the CD-ROM layer, which is used by practically every Linux user. This provides a clear incentive to fulfil the role as maintainer, although the maintainer must expect to be buried in bug reports, if he makes an error. If the maintainer is able to solve the bug reports, the signal effects will be even higher, as the users, who are directly affected, will be personally grateful to the maintainer.

JA has an incentive to accept modifications from user-developers, as this may potentially lighten his workload. As maintainer JA must, however, review the modifications, before they are sent to the kernel maintainers for inclusion. If the modifications submitted by user-developers are of high quality, accept of modifications will be a net benefit to JA.

10.4.2 Packet CD

Packet CD was a kernel patch, which JA wrote to support packet writing to CD-ROMs. The patch was written from scratch, giving JA the option to choose the licence. JA chose the GPL license, and this section will analyse the incentives behind his choice.

Packet CD was conducted as a personal initiative by JA and was not initiated by SuSE. This makes JA a private developer, who acts as an maintainer. Although Packet CD is a patch for the kernel, it is not a derived work, and as a maintainer JA has the choice of keeping the patch private or distributing it.

If JA had chosen to distribute the program under an MS EULA style license, JA might have been able to earn a profit from the sale. However, Packet CD is a source code patch for the kernel, and it cannot be integrated in a GPL program without being GPL. JA might have developed packet CD as a binary driver and released this under a MS EULA

license. However, such a license would not have been possible, as JA's employer SuSE would have objected to JA developing his personal software business while being employed by SuSE.

This leaves the maintainer with a choice between keeping the patch private or distributing the patch under one of the open source software licenses. Keeping the patch private does not provide the maintainer with any benefit, as he will not receive any feedback, and no modifications or signalling effects can be enjoyed.

The advantage of choosing one of the two open source software licenses would be that JA derives some signalling effects. But JA would also receive possible contributions from user-developers using the patch, and this would improve the use-product on the patch at little cost to JA. The functionality provided by the patch is desired by many user-developers and did not exist, before JA developed the patch. This provides an incentive for JA to distribute the patch under an open source software license.

The GPL license has the advantage over the BSD license in the sense that the maintainer (JA) can expect a higher degree of reciprocity. User-developers are more likely to return modifications to the maintainer under the GPL license, and this would be the incentive for JA to distribute the patch under the GPL license.

10.5 Peter Makholm

Peter Makholm (PM) is a Debian maintainer of the game Slash'Em, which is released under the GPL license thus placing the analysis in the GPL model. PM is a private developer maintaining Slash'Em and other Debian packages in his spare time. A Debian maintainer packages a program (Slash'Em) into a Debian package that is easily installable and upgradeable for Debian users.

The users of the Debian Slash'Em package consider PM the maintainer of the package and will complain to PM, if the package in some way does not function. This makes PM a maintainer in relation to the users of the Debian package.

However, when PM packages the Slash'Em program into a Debian package and observes bugs in the program, PM must interact with the maintainers of Slash'Em, thus making PM a user-developer. When PM acts as a maintainer in relation to the Debian users, PM is redistributing the program. The redistribution may contain Debian specific modifications and also general modifications to the program, and as the original program is released under the GPL license, it is not possible for PM to redistribute the program under a different license, and any modifications made must be publicly available.

PM is tied in his behaviour by the maintainer role, which dictates that he must release a new package, whenever a new version of the program has been released. If the maintainer does not package new versions of the program, users will wonder if the program has been orphaned, and that might result in other agents beginning to distribute the package. This only leaves a simple choice for PM, who can either continue or stop being a maintainer for the Slash'Em package.

Although PM does not derive a monetary benefit from maintaining the package, there are incentives for continuing to do so. PM derives a signal effect as a maintainer, signalling the ability to make Debian packages, and he also receives acknowledgement from users of the packages, who are appreciative of the work performed by PM.

Debian is a Linux distribution without any commercial backing, and all work and packaging of programs are done by volunteers. Providing something back to the

community, which has provided the very distribution which PM uses daily is intrinsically rewarding. It signals that PM is not a mere user but a maintainer who take part in providing packages for the common Debian project. The fact that Debian is non-profit and maintainers are volunteers increases the potential signalling effect.

There is also a personal benefit from turning a desired program into a Debian package. As a Debian user, one prefers to have all programs installed using the Debian package manager, and this makes day-to-day maintenance very easy to perform. Should the package not be available, PM has a choice of creating the package or installing the program from source. When creating a package PM is adding a personal use-product to the Slash'Em program while incurring an opportunity cost from the time spent during the operating. The personal benefit is a clean system, where all desired programs are installed using the Debian package manager. When subsequent versions of the program are released, there is also a cost associated with creating the Debian package. This cost is significantly lower assuming that the program has not changed in any way that could affect the Debian specific modifications. Having completed the package, the additional cost of making the package available to other Debian users is negligent.

The incentives for providing the Slash'Em program as a Debian package can be summarised as a combination of personal benefit, signalling incentives, and being part of the community of Debian package maintainers.

When PM interacts with the maintainers of the Slash'Em program, PM assumes the role of a user-developer. The program was released under the GPL, and any user-developer is obliged to make public any modifications distributed. However, as PM acts as maintainer of the Debian package, PM has a high incentive to return modifications e.g. bug-fixes to the maintainer. The fewer problems contained in the Slash'Em package, the less fuss PM is likely to receive from users of the package. As a user of the program PM also has an incentive to enhance his personal use-product and remove some of the errors contained in his personal use-product.

As a user-developer, PM could have made modifications to the program and kept them private, however, being a maintainer and personal user it is a highly unlikely choice. Should PM decide not to return modifications to the project and distribute is own enhanced version of the Slash'Em game, he would be allowed to do so by the license. There would be higher signalling incentives by providing a more stable version of Slash'Em for the Debian community. However, it would be a continued effort to maintain an improved version of the game for the Debian community. There would be a significant amount of work associated with it, as new features and improvements from the original program would have to be merged in the Debian specific version. Maintenance is only reduced, if bug-fixes are returned to the maintainers of the program.

10.6 Claus Sørensen

Claus Sørensen (CS) is engaged in open source software book writing. The books are released under an open source software license similar to the GPL, requiring that copies and derived works be released under the same license. CS participated in the book writing effort by acting as a sort of quality reviewer fixing grammatical as well as and orthographical errors.

Although the book license is not identical to the GPL license, the above mentioned similarity places the book and agents, who manipulate the book, within the framework of

the GPL license. In the extracted case, CS is modifying the content of a pre-existing work, and this makes CS a user-developer.

As a user-developer CS has the choice to use (read) or modify the book. And if CS chooses the latter, there is a further choice of keeping the modifications private or distributing them.

The book contains several different use-products from which different readers will consume their preferred use-products. One reader might just read the book and never look back, while other readers will use the book as a reference. Some readers are more sensitive to grammatical errors than others, and this is also part of the use-product of some readers.

The use-product that CS is demanding is one, where orthographical and grammatical errors are few and far between. As the use-product in the book is not the one desired by CS, he must choose to either make the required modifications or not.

Reading the book with the obvious errors will make the value of the reading experience low and distract from the knowledge contained in the book, and thus there is an incentive to make modifications to the book. Making modifications also has the added bonus of focusing attention to the contents of the book, thus enhancing the learning experience.

The purpose of reading the book was to get a brief update on security issues, before visiting a paying customer. Making modifications to the book enhances the personal use-product of the book to CS plus there is an increased learning experience from working intensively with the text.

CS might choose to keep his modifications private and have a personally enhanced version of the book. This would be particularly valuable, if CS was to read the book over and over again. It is not likely that the book will be read cover-to-cover more than once, and if it is used again, it will be as a reference.

Returning modifications to the book project provides value to CS in two ways: 1) Making modifications and being credited for doing so has signalling effects, and CS will later be able to use the work on the book as a reference for customers. 2) Contributing to the book is also a valued activity appreciated by the community, and CS will derive some signalling effects in respect to the open source software community.

In other words, if the book is so annoying that one cannot help correcting the errors, there is an additional learning bonus as well as positive signalling effects to be gained.

10.7 Kenneth Rhode Christiansen

Kenneth Rhode Christiansen (KC) is a GNOME developer, and he is contributing to the development of several GNOME programs. All GNOME programs are published under the GPL license, placing KC in the GPL model.

KC has worked on several projects within GNOME and has returned many modifications to various maintainers, which makes KC a user-developer. All of the work done by KC is conducted in his spare time, and this places KC in the category of individuals.

As a user-developer KC has the choice of just using the program or making modifications to the program. KC explicitly expressed his dismay with the all too obvious lack of coherence, and he felt compelled to try to change it. KC's discontent was such a motivating factor, that it would have been impossible for KC to continue to just privately

use the programs. KC needed to get involved and influence the development of the desktop, which he felt was most promising.

Thus KC chose to modify the program, and this left KC with a choice of either keep the modifications private or return them to the maintainer. The maintainer has not been further identified, as KC has interacted with several maintainers within the framework of the overall GNOME project.

Developing modifications to the program or programs requires an investment in understanding the code, so that bugs can be identified and modifications made. The cost of doing so is equal to the opportunity cost suffered by KC. KC is a student and is doing the work in his spare time, and occasionally he spends time that should have been devoted to studying. The cost can then be divided into a monetary opportunity cost suffered from KC, who could have earned money from a job in his spare time, and a cost of not focusing on the task at hand [Lerner & Tirole 2002]. The latter being a delayed cost and not felt immediately. The costs might appear as lower grades or longer period of study.

KC is a computer science student, and this fact will offset the cost of not focusing on his studies, as a lot of real world programming competence will be gained by contributing to the GNOME project. The learning effects associated with open source software development are accentuated by KC being a computer science student. The increase in human capital is accelerated in time, as KC is both enhancing his skills in regard to his studies and his skills from contributing to GNOME. The two learning experiences are, however, not synchronised, and the full effect from theoretical learning combined with practical programming driven by personal motivation cannot be achieved.

KC desired a use-product not contained in GNOME, when he began using it, and given the source code KC was able to help implementing the desired use-product. The desired use-product was not just a feature but also numerous small fixes and improvements. It is interesting that KC actually embarked on the project, as it seems obvious from the interview that KC knew that a lot of work was needed, before the desired use product could be obtained.

KC had a choice to either keep the modifications private or return them to the project. When KC began contributing to GNOME, the development pace was only picking up, and as time passed, more and more developers joined the effort. This gave a tremendous development pace, where GNOME evolved from day to day, and programs matured. Keeping the modifications private would have resulted in KC having to re-implement his own modifications on top of the constant stream of new versions coming out of the GNOME project.

By returning the modifications to the project, KC became part of the group of agents who modify GNOME. The persons, who are part of this group, obviously share some of the ideas of KC, and this is motivating in itself. The GNOME project made extensive use of IRC, which is an interactive communications medium. KC underlined the importance of meeting and talking to friends when joining the GNOME.

Being part of this group also has a positive effect on development as KS and other developers could discuss modifications before and during implementation. Design errors could then be caught in due time, before KC or other developers wasted time implementing a modification, which would turn out to be wrong. Important changes and minor but annoying bugs could be communicated through IRC directly to the maintainer of the particular code. This reduced development costs and increased the personal joy of being part of the project.

10.8 Poul-Henning Kamp

Poul-Henning Kamp (PHK) is a FreeBSD core team member, who among many other things has modified the FreeBSD kernel to suit the needs of a small American Internet service provider (ISP).

The work for the small American ISP required PHK to implement the "jail" feature directly in the kernel of FreeBSD. FreeBSD is licensed under the BSD license placing this development story in the realm of the BSD model. The small ISP commissioned the jail feature, and PHK was hired to do the job, which again turns the agent into a firm. The code that was manipulated was pre-existing thus turning PHK into a user-developer.

The modification was ordered, and the small ISP needed the feature to support a new business opportunity. PHK and the small ISP entered the modification stage, in which the BSD license provided an additional choice, namely the choice of distributing the changes as closed source software. The small ISP was within its rights to make modifications to FreeBSD and sell the modified version as open or closed source software.

The choice for an ISP to choose a BSD licensed operating system in the first place is obvious, and predicted by the model. The BSD license allows the ISP to keep private any modifications enhancing competitiveness. It is very plausible that this was part of the consideration when choosing to base their infrastructure and products on FreeBSD.

Keeping modifications private was an obvious choice for the small ISP and PHK. The modifications to FreeBSD were made to accommodate a specific business need for the small ISP, which would gain a competitive advantage. The model predicts that the small ISP should have made the modifications and have kept these private. This leads to the assumption that the small ISP would not be competitively threatened by returning the modifications to the FreeBSD project, or that the cost of maintaining the modifications would offset the advantage. Another answer would be that the model is an insufficient analytical tool for this case.

To clarify this, a follow-up email was sent to PHK, revealing that the small ISP was given a choice of paying the "full service price" or the lower "FreeBSD related price" [Kamp 2002]. The small ISP gained a cost advantage in two ways by choosing to return the modifications to FreeBSD: 1) An immediate cost from PHK making the modifications to the lower "FreeBSD related price" and 2) A long-term cost saving from not having to re-implement their desired use-product when new versions of FreeBSD was released.

In the case of Juniper Networks, which manufactures routers using FreeBSD, the founder had a choice between using Linux licensed under the GPL and FreeBSD licensed under the BSD license. Juniper Networks chose FreeBSD, as this allowed the company to use and modify the source code without having to publish it, if the software was distributed. This is particularly important for a firm selling hardware with software either installed or imbedded. Each time Juniper Networks sells a router; it is accompanied with software, which legally constitutes a distribution. Under the GPL license distribution of a binary program requires the source code to be publicly available.

The situation of a firm selling the original program with modifications is very different from a firm using the program as a production platform. The GPL license is appealing to the firm using the program as a production platform, as this ensures a higher degree of reciprocity from other users. The BSD license is, on the other hand, the most obvious choice for firms selling hardware including the modified software. The firm is then able to keep the production secrets to itself without being forced to make its modifications public.

While the BSD license results in a lower degree of reciprocity, it is certain that modification not competitively sensitive will be returned to the project. This lowers the cost of maintenance for the firm and perhaps even spurs further improvements in the area.

10.9 Summary

This chapter has used the model developed in chapter 7 to analyse the seven interviewed agents. The analysis has served to test the usability of the model, and the conclusion is that the model can indeed be used as a tool for understanding the choices made by agents, who are involved in software development and use.

11 Discussion

It is the purpose of this chapter to discuss the thesis, which is divided into: 1) The main elements, 2) Discussing Research Questions and Results, 3) What should, in hindsight, have been different, and 4) Contribution to Open Source Research. The purpose of the discussion is to act as the devil's advocate and discuss the thesis as a whole.

11.1 Discussing Main Elements

The main elements of a thesis are background literature, methodology, theory, and empirical evidence. This section will discuss each of the four themes and highlight their strengths and weaknesses.

11.1.1 Literature

Every research begins by researching existing literature and forming a basis for understanding the phenomenon and the problems of the particular field of research. When this project began in November 1998, open source software was not only a new phenomenon but also a new term invented in the spring of 1998. Open source software, it turned out, was an old phenomenon in a new disguise.

The open source phenomenon were characterised by many firms and individuals volunteering their time and effort to create software and provide this for free. Researching this phenomenon revealed two documented examples of a similar phenomenon. The first example was development and use of computer programs at MIT [Levy 1984] in the 50ies. The second example was development of Unix in the early 70ies.

These documented examples illustrate how source code is shared and developed as a collaborative effort. However, the two examples appear to differ greatly from the observed open source phenomenon. The early code sharing at MIT was done predominantly by students, who mostly belonged to a club for technically interested students. Another difference was the availability of computers in the 50ies, or rather the lack of available computers. This was in itself a powerful incentive for toying with the computers, and as very limited software was available, developing code was one of the only things that the computers could be used for. The development of Unix was also different from the observed phenomenon. Unix was developed by AT&T and spread to researchers, who contributed to the system. Computers were scarce, and only researchers and selected students had access.

These examples serve to prove that code sharing and common development had existed earlier, but the conditions had changed radically. It was not possible to use the early literature as a base for this project.

Fortunately, a few papers had actually been produced analysing open source software as a separate phenomenon. The period from 1998 to 2001 saw a steady production of papers on open source software. The papers focused on various aspects of the open source software phenomenon; some of them focused on the economic aspect hereof, and they are the ones included in the literature review. Their theoretical as well as their empirical understanding has inspired and has been used.

The search for literature has been a continuing effort, but papers written later than 2001 have not been incorporated in the thesis. A number papers has been produced since then, and some could even have been used in this thesis, however, the development in literature has been followed.

When comparing the selection of papers used in the literature review with papers available today, only a few papers are obvious candidates for inclusion. Two recent papers focus on the very licensing issues, which are treated in this thesis: 1) "A Framework for Understanding GPL copylefting vs. non copylefting licenses" by Aigrain [2002] and 2) "The Scope of Open Source Licensing" by Lerner & Tirole [2002]. However, time did not permit inclusion of these.

The literature section in this thesis is large, taking up little over 10% of the volume, and is justifiable given the rather pristine state of open source software research at the beginning of the project. The literature review serves as a baseline from which this project has evolved.

It is believed that the selection of papers presented in the literature review provided a sound foundation for the later research in this thesis. No paper has been considered of such importance that they should have been included in, or could have changed the initial understanding of open source software significantly. No paper has been observed to contradict the findings of this thesis. This is partially because the findings in this paper are new, and open source software as a field of research is just beginning to explore the importance of licenses.

11.1.2 Methodology

The project began with the assumption that the hypothetical deductive methodology was the correct approach to answering the meta-question of why open source software is being developed. However, as the project progressed, it became increasingly clear that the understanding of open source software was far from sufficient to warrant this approach. The hypothetical deductive approach has implications for the process of solving the meta-question. It is a process, in which a sound understanding of the phenomenon is used as a basis for developing a theoretical model that is tested. This process proved to be completely unsuccessful and frustrating, because the basic understanding of the phenomenon was far too incomplete.

As a consequence, the hypothetical deductive approach was abandoned in favour of an explorative approach. This approach favoured gaining a hands-on empirical understanding of the actual phenomenon: Using open source software and understanding how it was developed. This is not to say that using theory was abandoned, quite the contrary. The explorative approach has been instrumental in formulating the model of software development and consumption, which makes extensive use of theory.

A questionnaire was developed (see section 3.5.1), the purpose of which was to explore and uncover a broad area of topics ranging from motivation to how actual development was conducted. A number of open source software developers and community members were approached for an interview, and seven individuals responded positively. Open source software was installed, and use and problems were solved through news groups and mailing lists.

Changing to an explorative approach using interviews proved successful and provided a sound understanding of the phenomenon, which became the driving force of the

study. Using and gaining experience from open source software was also an eye-opening experience, as it provided an understanding of open source software.

The disadvantage of the explorative was the rather unfocused questionnaire that tried to cover as much as possible in each interview. The literature available at the time did not permit a more focused approach, and the theoretical understanding was nonexistent due to the failed hypothetical deductive approach. In this respect the explorative approach was the best choice at the time.

The model developed in this the thesis was tested using interviews that were conducted, before a theoretical understanding was established. Naturally this gives rise to concern regarding the integrity of the test and its conclusions. A valid concern is whether the model has been created to fit the behaviour reflected in the interviews. While the model was developed with the interviews in mind, the model has not been developed to match the interviews. The interviews documented, how the respondents acted, and tried to uncover the incentives, albeit not very successfully due to the unfocused explorative approach.

The model, on the other hand, deduces behaviour of agents based on incentives, theory, and mechanisms. Thus integrity of analysis is upheld by using the model to interpret behaviour of agents and uncover the underlying incentives.

The described development stories beginning in section 9.5 can be seen as a possible source integrity pollution. The development stories were extracted from the transcribed interviews, (see Volume III containing the raw interview transcripts) and the actual process of extraction is questionable. The method was to analyse the interviews by marking sections, where the respondent discussed a particular development story. For instance, Henrik Størner discussed his work on the Windows file system driver, and all references to this were marked. Subsequently, the marked paragraphs were written as a separate development story. The process of marking corrections is considered to be unproblematic, while the transcription into a separate development story is dangerous. The model of open source production and consumption was finished at this point, and it is not entirely impossible that unconscious analysis has influenced on the way in which the development stories were written.

This could only have been avoided, if a second round of interviews had been conducted asking focused questions relating to the model and it would also have had reintroduced the hypothetical deductive methodology with its emphasis on testing validity of theory and model. Time constraints did not permit such an endeavour.

11.1.3 Theory

Three theoretical elements were selected as tools for understanding why open source software is being developed. The elements were: 1) Theory of economic goods, 2) Economic theory of externalities, and 3) Theory of competing technology and lock-in.

The theory of economic goods was chosen in the belief that open source software analysed as an economic good would go a long way in explaining the phenomenon. The analysis of open source software as a simple economic good in section 6.3 concluded that the theory of economic goods was unable to account for the reason, why open source software is being developed. However, the understanding of open source software as an economic must not be under estimated as the analysis of the properties of consumption has been instrumental for the subsequent model development. The analysis also established that the type of license was able to determine the type of good. A more through analysis of open source software as an economic good combined with independent theoretical

development might have pushed results further. However, the static understanding of the good with no feedback mechanisms was the major limiting factor of the theory. It was also this limitation, which inspired the later development of the model of software production and consumption.

Following the interviews, it became clear that the economic mechanism was one, which did not require direct transactions between agents. Economic externalities fit this definition, as the externalities are benefits or costs imposed on one agent by another agent without any consent. The theory, however, needed to be used in a framework to be useful, as, per definition, it includes all actions performed by agents affecting other non-consenting agents.

This framework was established by developing a model of how open source software is being developed and consumed. Theory of externalities is well suited as a tool to understand agents' choices in the model. The theory of externalities is actually quite a large topic and a separate area of research. This thesis only relies on very simple elements of the theory in pursuit of understanding what the externalities are, and how they affect agents' behaviour.

This was done intentionally because of the novel nature of the researched phenomenon. Like the hypothetical deductive approach, use of complex theory requires a deep understanding of the studied phenomenon. If a deep understanding is not present, it becomes impossible to use the theory, which then introduces uncertainty instead of clarity. Such a deep understanding was obviously not present in more than the first half of the project. Only after analysing the interviews in the light of the model and thus using simple theory of externality, a good understanding of the phenomenon has emerged. Thus, only at the end of project, a more advanced theoretical approach could be used.

The third theoretical element was theory on competing technology and lock-in. Software often represents compatibility regimes, where an application only functions properly in conjunction with another specific component. For instance, Microsoft Word is only able to use and save all features using the .doc file format, which is proprietary. As Microsoft Word is the most used word processor and consequently people need to be able to process .doc documents.

This lock-in effect is powerful and an important determinant of both firms and individuals choices. The theory of competing technologies was included as an element to help understand incentives for maintainers, as firms, developing and releasing the first version of a program. The theory provides incentives for the reason why firms would release the first version of a program under an open source license.

The use of the theory is at a conceptual level, and the effects in numbers are never approximated. Although this is a weakness, it has never been the intention to calculate benefits from choices in the model. From the outset, the intention has been to provide a conceptual explanation as to why open source software is being developed, and the model developed in this thesis does just that. Using the concepts from the theory of externalities and the theory of competing technologies, the model of software development is created. The model is capable of deducing a logical explanation of why open source software is being developed.

The questions then arise: Does the theory capture all relevant elements, and could the same model and conclusions have been reached with a different theory? The model appears internally logical in the sense that agents behave according to the theories, and this brings our understanding of why open source software is being developed one step further.

The model could not have been developed using other theoretical perspectives, and in this respect the model is sensitive to the economical behaviour assumed in the used theories.

The model does not capture other motivating factors like interpersonal and psychological factors. Motivation and incentives derived from being part of a group, a larger social movement, or political incentives are not accounted for in the model. These interesting aspects are one level lower, than the ones examined and explained by the model. There is evidence of master-apprenticeships in open source software projects, which might be a motivational factor. The motivation and learning taking place in master-apprenticeships could be examined in the situated learning perspective [Lave & Wenger 1991]. The recent interest in open source software from the political establishment is not accounted for in the model. A perspective of epistemic communities [Edwards 2001] could play a part in examining, how open source software interest groups are formed alongside political interests.

11.1.4 Empirical Evidence

The empirical sources used are interviews, websites, and open source mailing lists, and only the interviews have been systematically documented. The websites have served to provide general overview of open source software, while the open source mailing lists have provided insight into the working of open source software development.

Seven interviews with open source software developers were conducted and documented. The interviews were broad, in-depth, and they all lasted several hours. The strength of the interviews is their scope, and that they cover a huge area of topics. As mentioned in the methodology section, this approach has both advantages and disadvantages. The obvious advantage is the broad range of topics covered, which is desirable considering the explorative perspective. The disadvantage is the lack of depth in the interviews making them more difficult to analyse using the model. The interviews were not designed to support the model, as the model was not yet conceived at the time of interview. The number of interviews is also fairly low, and more interviews would have been desirable, had the model been conceived earlier.

The quality of the interviews can only be judged by whether they have provided useful insight and have brought the project forward, and they have! The interviews created the foundation for the later model by providing through insight as to how open source software is developed.

The question is of course whether more empirical evidence should have been collected, and whether this would have changed the model or conclusion.

Since the interviews were conducted, a few studies of open source software have been conducted by Lakhani et al. 2002[51] and Ghosh [2002], both of which are quantitative. A qualitative study has also been conducted [Lakhani and Von Hippel 2000], and nothing in these studies indicates that the interviews presented in this thesis are not generally representative. Consequently, it is believed that further explorative interviews would not have changed the perception of open source software.

Specific and precise qualitative studies reflecting the understanding presented in the model might have contributed further to the model. The study could have been performed as a questionnaire, but the preferred approach is qualitative interviews. Questionnaires

[51] FLOSS survey

have the advantage of being easy to distribute in numbers, but one can only expect to get answers to the questions asked. This is the main disadvantage, especially when testing a new model that appears logically sound but is not empirically proven. To this end, the qualitative interviews hold an advantage, as the researcher will be able to capture issues outside the scope of the model, and besides, qualitative interviews provide much more input to the process of developing and refining a model. However, qualitative interviews are very time consuming, and this fact prohibited a second round of interviews.

11.2 Discussing Research Questions and Results

This thesis set out to provide an answer to the fairly ambiguous meta-question: "Why is open source software being developed?" The question was motivated by the observation that vast amounts of open source software were being developed, which had production level quality and could be obtained for free. Software development is a time consuming activity, and it seemed strange that many talented developers would use their time making software for free. The meta-question was further divided into four research questions:

```
1) How is open source software being used, exchanged, and
marketed.


2) What characterises the open source software development
process?


3) How do properties of software (license - and technical
properties) affect open source software as an economic good?
And how does the type of good affect behaviour?


4) What are the externalities of open source software?
What is the source of these externalities?
And how do these externalities affect behaviour?
```

11.2.1 Research Question 1

"How is open source software being used, exchanged, and marketed." This question has been answered indirectly in many different chapters. The use of open source software was among other things illustrated by the Apache web server, which is dominating the market for web severs. It was also illustrated how open source software could be used as a regular desktop system.

Open source software is being used more and more, but it is mostly in the server rooms that open source software is replacing proprietary software. On the desktop, open source software is gaining momentum, but as the review showed, novice users must be willing to learn and experiment in order to get their system working. But there are signs that Linux and open source software are ready to be used on the desktop. Recently Xandros[52] released a Linux distribution, based on Debian and aimed at the desktop, and reviewers claimed that it just worked, even with Microsoft Office2000 [Gasperson 2002]. Success stories have surfaced in the press about firms turning to Linux for their day-to-day computing needs. For instance, the Yorkshire Police has just begun migrating applications to Linux and expect to save £1 million a year using Linux on their 3500 desktops

[52] Xandros is a company marketing a Linux distribution named Xandros Desktop, see [Website] www.xandros.com

[Williams 2002]. So, there are clear signs that Linux and open source software are heading for the desktop. The major obstacle so far has been the lack of user friendliness and incompatible applications and file formats. The usability issues are being resolved but incompatibility with the Windows platform remains a problem. Microsoft Office is the default file format when exchanging documents. Two things have helped: 1) Improved import filters and 2) The ability to run Microsoft Office on Linux. Improved import filters reduce dependency on Microsoft products provided there is a high quality word processor for Linux The release of OpenOffice and increasing maturity of other office applications has made professional quality office applications available for Linux. The ability to use native Word documents in these programs is greatly reducing the cost of switching to Linux from Windows. Microsoft is still in control of the file formats and can change the format as they please, and this is the real competitive advantage of Microsoft.

Open source software is primarily being exchanged using the Internet, and this has been the assumption when developing the model of software production and consumption.

Open source software is distributed and marketed both as distributions like a Red Hat distribution containing countless open source programs, and as free download from the Internet. Some distributions can only be bought as boxed sets, while others allow the latest product to be downloaded for free from the Internet. The GNOME and KDE desktops are mostly bundled with distributions but are also available as a download from the Internet. To the knowledge of this author, there is no open source software that is not available as a download from the Internet. In the early days of open source software, when it was called free software, and the Internet did not exist, it was distributed on tapes. The Internet made this method of distribution obsolete and replaced it first with CD-ROMs and then with downloads from the Internet. Large compilations of open source software are still available on CD-ROM and, depending on the available Internet connection, this can often be a good alternative to downloading huge amounts of software. Still, the Internet makes the development cycle of open source software much faster, as changes can be propagated to all interested parties at once.

While not the main focus of this thesis it is interesting to ponder the question of the viability and strategy of Linux distributors. It appears that Linux distributors are employing a strategy which builds on the principles of lock-in and installed base. By providing free downloads of a distribution a Linux distributor is creating an installed base, where the lock-in is tied to ease of updates and packaged programs. Even though RedHat and SuSE are using the same package format (RPM) they are to some extent incompatible. This has the effect of tying users to a particular distributor. This does not per se generate additional revenue as some sort of excludability must be facilitated in order to offer a product worth paying for. Both SuSE and Red Hat charge a subscription for automatic updates of their distributions. Naturally users are offer a free 6 month subscription when buying a distribution, by the end of which the users will realise the time saving and convenience of signing up for a new period.

As most of the updated packages are open source it is not possible to exclude access for those users who whish to do it for theme selves. To counter this, Linux distributors such as Red Hat has introduced forced expiration of their distributions. This means that a given version of the distribution will no longer be maintained by Red Hat after a particular date. Naturally firms are not interested in forced upgrades and are offered to buy enterprise versions which have a significant longer life.

Based in the above observations it can be concluded that the strategy of the Linux distributors appears to be one of gaining installed base, and then sell service and upgrade subscriptions.

11.2.2 Research Question 2

"What characterises the open source software development process?" This has been indirectly answered in the model of software production and consumption. The model assumes a particular development cycle and organisation, where a single point is responsible for releasing new versions of the program in question. The specifics of open source software development such as editor, integrated development environment, and versioning system have been deemed outside the scope of this thesis.

The interviews, however, touch on the subject of how open source software developers collaborated. The development cycle used in the model is easily identifiable in the interviews, the tools may vary, but the basic principle and seven distinct steps in open source software development can be identified:

1. A maintainer releases a program and source code
2. A User-developer downloads software and source code
3. The User-developer identifies problems or needed features
4. The User-developer modifies the original program
5. The User-developer returns modifications to the maintaner
6. The modifications are discussed with other agents who develop
7. The maintainer reviews the modifications and perhaps includes changes in the program
1. The maintainer releases program and source code

Open source software development can be characterised as a distributed development model, where anyone interested can participate in development. Development is usually organised around a maintainer, who is responsible for releasing new versions of the project software. The maintainer has the final say as to what is and becomes part of the project software. Some open source software projects like FreeBSD are organised around a core team distributing the role of the maintainer to a group of people. The fact the FreeBSD allows many people to upload to the CVS repository does not invalidate the seven-step development process, as modifications must survive 72 hour without major critique before being committed. This corresponds to the steps, where a user-developer returns modifications to the project, modifications are discussed, and modifications are included in the program. The difference being that responsibility has been delegated to minimize core team workload.

11.2.3 Research Question 3

"How do properties of open source software (license - and technical properties) affect open source software as an economic good? And how does the type of good affect behaviour?"

The first part of the question was answered in section 6.3 where it is shown that the type of economic good is determined by the license of the software. The MS EULA uses copyright to turn a program into a club good, and the GPL license turns a program into a public good. The analysis of open source software as a simple economic good illustrated that the theory of simple economic goods was inadequate for answering the meta-question.

The second part of the question focused on the effects on behaviour. A maintainer, as either a firm or individual, would choose a program licensed under the GPL or BSD, as this is freely available. The theory of simple economic goods does not allow any feedback between the maintainer and the user-developers, and thus the choices between comparable programs are reduced to the cost of the program.

For firms acting as maintainers, the preferred license choice is the MS EULA type of license, as this turns the program into a club good that can be sold for profit.

Individuals, acting as maintainers, have no incentive to distribute their program under either of the licenses, as individuals per definition are not interested in profiting form their work. Here the theory of simple economic good shows its deficiencies, as it can only account for incentives derived directly from the transaction. In this perspective the transaction ends, when the maintainer has distributed the program to a user-developer. Any motivation from having users or from expecting some degree of reciprocity from users cannot be explained by this theory.

For firms the theory is adequate when explaining why firms release software under the MS EULA license. However, it does not explain why firms engage in open source software development, which we know they do.

11.2.4 Research Question 4

"What are the externalities of open source software? What is the source of these externalities? And how do these externalities affect behaviour?" This research question reflects the presumption that the basic economic concept behind the model software development and consumption is externalities. The three questions are answered by developing the model, and they conclude that there are massive externalities in open source software development. Externalities arise from one agent's modification to a program, which affects the value of the program for many other agents. One agent fixing a problem in a program results in many agents enjoying the benefits hereof. Externalities of open source software are tied to the non-rival nature of open source software in terms of licensing and digital nature. Externalities would be offset, if there were substantial costs associated with distribution, or if rivalry in consumption existed.

It is the external effects of one agent's actions, which provide an incentive for the next user-developer to participate as a user or developer. If one agent adds value to a program, this value is available to the next agent wanting a similar use-product. The added value is another way of saying that the available number of use-products is increased. User-developers will only use a program, if the desired use-product is contained in the program. The available number of use-products thus serves as a barrier to entry for users. The number of available use-products also serves as a barrier to making modifications to a program. The more use-products contained in a program match the desired use-product of a user-developer the lower the barrier to begin modifying a program.

Open source software is a broad term encompassing a variety of licenses, and the model only covers two open source licenses, the GPL and the BSD. Externalities of software are defined by the license, and the model illustrates how the MS EULA offered

externalities as knowledge of how to solve usage problems with particular use-products. But the MS EULA does not offer any externalities associated with making modifications to the source code and modifying the number of use-products in a program. The GPL and BSD license on the other hand provide fertile ground for externalities, as the licenses allow user-developers to make modifications.

The two open source licenses provide different expectations as to the amount of modifications returned to the maintainer. The GPL license requires modifications, which are distributed, to be made publicly available, whereas the BSD makes no such requirement. User-developers making modifications to a program can expect a higher degree of modifications to be returned to a program licensed under the GPL than under the BSD license. Expectations form a self-reinforcing loop, where expectations of little or no modifications made by other user-developers raise barriers to entry.

The model developed does not explicitly discuss if the type of program influence the behaviour of agents. It seems appropriate to assume that different types of programs induce changes in the incentive structure of agents. For instances we may categorise one type of programs infrastructure programs. Infrastructure software needed by many people, which is not source of a competitive advantage, will lead user-developers to expect both individuals and firms to return modifications to the maintainer. High expectations can lead to both strategic waiting and lowered barriers to entry. Strategic waiting is countered by the effect of individual use-products, where an agent waiting cannot be sure that other agents have the exactly same problem. Agents who enter into strategic waiting cannot predict when - or even if - their particular use-product will be developed. A recent discussion on the Linux kernel mailing list underlines this:

> "I've been getting more and more people talking to me looking to pay people to fix small Linux bugs but having problems finding smaller companies. Obviously wanting to send $1000 to have someone fix a driver simply doesn't work when you talk to big companies.
>
> One thing the FSF do which is rather sensible is keep a list in the packages of people who you can pay to fix stuff in them. I asked on Linux-kernel and got a small initial set of company responses. hopefully more will appear once its merged." [Cox 2002]

The above post and the discussion that followed illustrated that indeed user-developers are willing to pay for obtaining a particular use-product. For firms needing a particular functionality, strategic waiting is not an option, and paying to have their personal use-product fixed is a sensible choice.

11.2.4.1 The Meta question

Having answered the research questions, the next question to answer is, whether the Meta question: "Why is open source software being developed?" has been answered. The Meta question should be seen as more than the sum of the four research questions and relying on the four research questions. The four research questions have been introduced in order to define a sort of research path towards understanding why open source software is being developed. The most obvious answer to the meta-question is that, well, it lies in the model of software development and consumption.

Open source software development must be viewed as a process. The process view indicates the way open source software has developed, and is not reversible and might not

even be reproducible. The dynamics of the model is a result of user-developers making modifications to a program and returning the modifications to the maintainer. All these little modifications provide a constant stream of improvements and added use-products. But the dynamics of a single program is far from enough to provide an incentive for a user-developer to participate. It is the dynamics of the system, the open source software system as a whole that is important.

As Kenneth Christiansen explained about the development of GNOME, it was a constant process of building on top of existing components, and enhancing them when needed. If the underlying components did not support the functionality of new programs, the developers had to make changes to the underlying components. This is evident in open source software development in general. The Linux kernel is dependent on C compilers, and currently the GNU C compiler is the preferred choice, in turn other programs rely on functionality supplied by Linux. The programs written for either KDE or GNOME require the presence of KDE or GNOME respectively to function. Some programs may function in either environment, but the special features supported will only function in its native environment. It is the dynamics of many agents each providing a little, and some a lot. Linux would never have reached the desktop without a credible window managers like KDE or GNOME. It is the growing system of open source software that is attractive and drives adoption from both users and developers.

It is far from certain that open source software would have evolved into a mature range of products, had Linux been released 10 years later. The dot.com bubble played a major part in providing momentum for open source software. Today open source software has reached a level, where some areas are economically viable. The continued existence of Linux distributors and the increasing use of open source software is a testament to this. Still the vast majority of open source software developers are unpaid, and this is not going to change any time soon. It is inherent in open source software that many contributions will come from individuals, simply because these agents are making modifications to a program in order to obtain a particular use-product not supplied by others. Such narrow market segments will never be attractive for companies.

11.3 What Should, in Hindsight, have been Different

This project took just about four years to complete, and much has been learned in the process. A similar project conducted today would naturally be done in a different way. In the following, a similar project is outlined but designed with the state of the art of open source software research in mind.

This project was in the beginning outlined as a project that would use the hypothetical deductive methodology to test a theoretical understanding of a specific phenomenon. However, when the project began, there was very little literature or knowledge on the subject, which is a prerequisite for using the methodology. Much time was used to just understand the concept of open source software, before actual research could be undertaken.

A project today would not have to waste time just understating the concept of open source software. There is now a growing body of literature on the subject, and a new project would quickly be able to navigate the different theoretical positions.

During this project, open source software has moved from being mostly developed by individuals to being developed by both individuals and firms. I predict, the public sector to become the next major player within a three-year timeframe. There are many signs of

the public sector being interested in adopting open source software, the main argument currently being significant cost savings.

A project today would take the interest from individuals, firms, and public sector for granted. The public sector is not a specific player in the model presented in this thesis, but could easily fit the description of a firm with special incentives. Those incentives might be to use the native population to develop open source software and thereby increase national competence.

The empirical approach to this project has been inadequate because of the initial project design being hypothetical deductive and the general empirical understanding not sufficient for such an approach. Should the project be redone, the research design would be explorative and much more focused. It is evident that the questionnaires cover a broad range of topics, while not focusing enough on the topics relevant to test the model. This project has focused on a selection of individual open source software developers, some maintainers and some user-developers. The individuals were chosen randomly, not because they were either maintainers or user-developers. A project today would change the analytic focus to cover the project rather than the individual agents, as this is the focus of the model. A selection of, say, 10 case-projects would have to be selected and followed over a period of time. The number is rather high, because it must be expected that some projects will die, and that others will see much varying activity. The Ph.D. project would then track the changes made by each agent to the programs. A specific questionnaire would be used to examine the reason for agents making specific modifications, and they would have to be simple and possible to fill out within a time frame of 5-10 minutes in order to get a high response rate. Some respondents should be selected for in-depth interviews to uncover other aspects than the ones covered in the questionnaires.

Preferably the open source software projects would give their prior consent, before beginning the data-collecting phase. The questionnaire could then be built into the software projects revision control system and be a natural part of participating in the project. This would further require the questions to be very specific and easy to answer, as user-developers would have to answer the questions every time a modification was returned to the maintainer.

11.4 Contribution to Open Source Research

The main contribution of this thesis is the model of software development and consumption. It is a model, which is able to provide an explanation as to why open source software is being developed. The model is interesting, as it explains why firms and individuals develop and contribute to open source software. The model also illustrates how the license of a program changes behaviour of agents, and this can be used ex ante to determine the optimal license of a program.

The model uses signalling effects from Lerner & Tirole [2001] as part of the incentive mechanism, which is coupled with opportunity cost. Central to the model is the two terms: Use-product [Bessen 2001] and user-developer [Johnson 2001]. These two terms has been coupled and is the basic terminology used in the model.

The model also represents a subtle contribution in terms of the organisation of open source software development. The model very explicitly separates the role of the maintainer and the user-developer. Lerner & Tirole [2001] uses a distinction between a project leader and programmers, but the role of users is neglected. The purpose of

separating the two roles is that the two roles have distinctly different incentives, which must be accounted for.

On a personal note, it is hard to see a model, which took years to develop be disseminated in no time. This is the problem of ideas: It takes time and effort to develop new ideas and once stated they appear simple and easily understandable.

The next contribution is empirical, which is a description of seven person's involvement in open source software development and their reasons for doing so. The raw interview, although Danish, make some interesting reading regarding the open source software community. Most research into open source software development has been focused on quantitative studies, which are needed to provide a larger picture of how, where, and why people contribute to open source software development.

I also feel that the analysis of development cases of the seven interviewed persons represents an individual contribution. The contribution is both an analysis of the incentives and choices, and a test of the model as an analytical tool.

11.5 Are the Results Idiosyncratic or General?

The model of software development and consumption has been tested on seven mini cases. In each of the mini cases the model offered insight as to the incentives of the agents involved. However, as noted the mini cases has been extracted from seven interviews with Danish developers. Naturally suspicion arise whether is has special importance that the developers interviewed are Danish. Are Danish developers different from other developers around the world?

Thankfully this appears not to be the case! The developers interviewed are often collaborating with developers from around the world and in these projects Danish developers represent but a fraction of the total number of developers. Jens Axboe is one of few kernel developers and certainly the only Danish developer maintaining the CDROM layer. Peter Makholm is the only Danish Slash'Em developer.

It appears that these developers, when developing open source software, become a part of the project and develop according to the values of the project rather than being a special Danish characteristic.

Had the interviewed developers been working on the same project it would clearly have raised valid concerns regarding the general application of the model – but this is not the case.

While not exactly documenting I have also followed both the Linux kernel, Amanda and ZINF projects. A tentative analysis of these projects appears to confirm the general application of the model.

The desire for a particular use-product can clearly be identified in al of the three projects. In particular Amanda and ZINF it is obvious that new users who are capable of modifying the program are very active in the first period following adopting the program. This motivate existing developers to comment on the proposed modifications are they are interested in not having to erect mistakes made by the new developers.

For the Amanda project it is obvious that development is driven by a community of professional network administrators. This community is interested in developing a backup solution, which solves the need of their organisation. This means that some modifications are inspired by the need of their users. For instance, one Amanda developer has developed a web interface allowing users to retrieve from backup at the click of a mouse.

In the Linux kernel things are more complicated as agents often represent commercial interests, which are not always explicit and thus incentives may be difficult to directly observe.

The Cost of Being Too Late is also apparent in ZINF when development is picking up pace. Often developers are asking if their modification has been integrated and are asking how to make a diff against the latest version in order to make sure that the modification does not have to be redone.

In conclusion it appears that the results and the model indeed may have general application. The model is arguably not idiosyncratic but on the other hand it has not been established that the model has general application – the truth may lie somewhere in-between.

11.6 Summary

This chapter has discussed the main elements of the thesis. The chapter began by discussing the main elements consisting of literature, methodology, theory, and empirical evidence. Next the research questions were discussed one by one. The third section discussed the lessons of such a large project and what should be done different if such a project should be undertaken one again.

The contributions to open source software research made by this thesis were highlighted and contrasted with some of the literature used as a basis for the thesis.

Lastly, it was discussed whether the results of the thesis were purely idiosyncratic or generally applicable. The discussion found that no evidence was in favour of the results being idiosyncratic. It was argued that the model and some of the mechanisms of the model was easily used to explain the behaviour of agents in other open source software projects. For this reason the results were concluded to have a more general nature.

12 Conclusion

This thesis set out to answer the meta-question: "Why is open source software being developed?" An increasing number of references to free and open source software in the press motivated the question. Netscape had lost the browser war to Microsoft's Internet Explorer and had announced the release of their source code. Releasing the source code was an unprecedented move in the software industry, which naturally caught the attention of the press.

Looking closer at open source software, a puzzling behaviour was observed: Individuals and firms produced (and still do) high quality software and made it available to anybody interested. The software developed was made available as source code, making it even more puzzling as the source code can be compared to the secret Coca Cola recipe. Often software firms had given some of their software away for free to stimulate adoption. The source code, however, was never released and was guarded as the most important trade secret. In open source software this picture was turned upside down, as users were allowed not only to have the source code but also to make modifications and create derived works. The software was made available free of charge as a download from the Internet.

The size of the phenomenon and the composition of agents were important factors in stating the meta-question. Open source software was not just developed by a gang of eager hobbyists doing a little thing in their garage. A "we are doing it for fun" explanation would suffice for a little program developed by individuals in their leisure time, but open source software was developed by both firms and individuals. The quality of open source software was impressively high, and some open source software claimed market dominance.

To an engineer with a vested interest in economics this behaviour seemed strange to say the least: Firms and individuals were investing time and money in software and providing it for free. Incentives for this behaviour were nowhere in sight, and it was decided that this thesis should dare to attempt an explanation.

This last chapter is divided into three parts: 1) Overview, 2) Results, and 3) Perspective. The overview provides an overview of the chapters and their contents. Results are presented in the following section and are related to the meta-question and the research questions. Lastly, a perspective is provided, which discusses the perspectives of this thesis, perspectives of open source software, and perspectives of further research into open source software.

12.1 Overview

Chapter 1 gives an introduction to open source software development and provides two examples of very successful open source software projects. The introduction outlines the structure of the thesis and discusses use of terms like free and open source software.

A survey of the literature was conducted in chapter 2, which revealed some contributions to understanding open source software development. However, the answers provided by the literature fell somewhat short of answering the meta-question of why open source software is being developed. It is evident from the literature review that open source software development is in its early stages. The amount of papers available at the beginning of this project was very low, approximately three or four. As the project

progressed, research in open source software has increased, and papers are now produced frequently.

The advent of the opensource.mit.edu research community website has made access to working papers and published papers easy, and the site serves as a focal point for researchers. Also noteworthy is the monthly journal FirstMonday that has spearheaded the first publications about open source software and continues to do so.

The meta-question is approached in chapter 3, and the reviewed literature is used to conduct a preliminary analysis and tentative answer to the meta-question. The meta-question is then approached under the assumption that the type of agent and properties of software determine behaviour of agents. Agents are divided into firms and individuals, and properties of software consist of technical properties and license properties.

The meta-question is then divided into four research questions based on an assumption of the theoretical tools and methodological approach:

1) How is open source software being used, exchanged, and marketed?
2) What characterizes the open source software development process?
3) How do properties of software (license - and technical properties) affect open source software as an economic good? And how does the type of good affect behaviour?
4) What are the externalities of open source software? What is the source of these externalities? And how do these externalities affect behaviour?

Research questions one and two were used to focus on the empirical exploration and gain an understanding of open source software development. The third question focuses on the license issues of open source software and tries to examine open source software as an economic good, the characteristics of which are determined by the license. The fourth question reflects an initial belief in the economics of externalities as being an important perspective for understanding open source software.

It was the intention to use a hypothetical deductive methodology to test theoretical explanations of the meta-question and research questions. However, the amount of literature and likewise the low level of general knowledge did not warrant a hypothetical deductive approach. The hypothetical deductive approach assumes a detailed understanding of the empirical field and a firm theoretical foundation from which to deduce hypotheses. Consequently, the project switched to an explorative approach, although the hope of using the hypothetical deductive methodology was never completely forgotten.

This resulted in search of a model – a theory of understanding why open source software is developed, based on which hypothesis could be deduced. It was proposed in chapter 3 that the behaviour of agents participating in open source software was a consequence of properties of open source software and type of agent. Properties of open source software are comprised by technical properties of software and properties of the particular license. Agents were divided into firms and individuals, based on the assumption that the two types of agents will have significantly different incentives. Firms must make a profit to pay salaries and can only afford to participate in open source software development, if there is a payoff. Individuals do not have to profit or make a living from developing open source software, it might even be an intrinsically rewarding leisure activity performed in their spare time. In short, firms are profit maximising, and individuals are utility maximising.

Economic theory was selected as the tool for understanding behaviour of agents as a function of the license and the type of agent. Three different theoretical approaches were selected and presented in chapter 4. The theoretical approaches were 1) Economic goods, Externalities, and 3) Increasing returns to adoption.

In order to understand open source software in particular and software in general, the theory of economic goods was used to understand a simple situation of production and consumption of a program good, and how the license influenced the type of good. The theory of economic goods was presented in section 4.1.

Open source software is having access to the source code for a program and making changes to the program. Individuals and firms are making modifications to a program developed by other agents, without compensating them for their initial work. This simple observation motivated the choice of the second theoretical approach, which is theory of economic externalities. Externalities are effects outside the price mechanism, i.e. if a farmer kills the annoying fox, the other farmers do not compensate him. This theory has been instrumental when analysing and developing a model of the behaviour of agents as a function of license and agent type. The theory of externalities were presented in section 4.2.

Lastly, it was evident from just observing the software market that software exhibits special behaviour in adoption processes. Compatibility and installed base are powerful factors influencing agents' choices between competing technologies. Increasing returns to adoption is present in the model and influences incentives for developing open source software. The theory of increasing returns to adoption was presented in section 4.3

A plethora of different software licenses exists, and it was argued that the license is a key element in defining behaviour of agents. It is believed that software licenses represent a set of rules defining how agents may interact and use the program. While it is neither possible nor feasible to analyse the behaviour of agents in the light of all possible licenses, a selection of eleven licenses were analysed in chapter 5. The licenses were then compared schematically, revealing their differences and similarities. Chapter 5 also provided background for understanding intellectual property rights and licensing in general.

Agents' behaviour from a perspective of simple economic goods was analysed in chapter 6, which began by analysing the properties of software (i.e. technical and license properties) and the two types of agents. This resulted in selecting three licenses based on the analysis of eleven licenses in chapter 5. The licenses chosen were the MS EULA representing a proprietary license, the GPL license representing the strictest open source license, and the BSD license representing a liberal open source license.

Being familiar with licenses, technical properties, and an understanding of the two types of agents, the analysis proceeded to software as a simple economic good. The analysis revealed that software as an economic good changes type depending on the license. This was the first indication that licenses do indeed have a profound impact on behaviour. The analysis, however, also highlighted the weaknesses of the theoretical framework of economic goods.

Chapter 7 developed the model of software development and consumption. The model was developed using an approach where the simple relationship between maintainer and user described in chapter 6 was gradually extended with feedback loops. The model of software development and consumption is developed by analysing incentives for each type of agent in light of the license of the particular software and the theory of externalities. The

resulting model of open source software development was able to explain the incentives for developing and using open source software and its dynamics.

Chapter 8 elevates the analysis to a higher level by considering the selection process taking place before agents enter the model of software development and consumption. The chapter analyse a situation where agents desire a particular functionality and enter the market in search for such a program. Agents can choose between programs licensed under the MS EULA and the GPL. For the sake of simplicity the BSD license is not included as it offers no other options than do the GPL license.

The explorative nature of the study has lead to a large collection of empirical evidence presented in chapter 9. The chapter begins by outlining the history of open source software, which provides a perspective on the old age of the open source software phenomenon. Many are unfamiliar with the details of open source software, and the empirical chapter remedies this by describing open source software using Linux as an example. Lastly, the empirical chapter presents open source development stories from seven interviewed developers. The developers were interviewed using the interview guide presented in section 3.5.1. The questionnaire was deliberately not focused, as it was conducted in the explorative phase of the project. In this respect it is well suited to test the model, as the questionnaire was not developed with the model in mind. However, there is also a major drawback to using the questionnaires, as they are not focused on the model, and thus details of the model will only be observed accidentally.

The open source development stories are analysed in chapter 10. The analysis reviews each of the development stories using the model software development and consumption as an analytical tool. The analysis uses terminology of the maintainer and user-developer from the model and matches them to the agents in the development stories. Each agent's incentives are analysed depending on the license of the program developed. The analysis reveals the model of software production and consumption to provide a rational explanation for the agent's behaviour.

Chapter 11 contains a discussion of the thesis in general. The chapter begins by identifying results and contributions made in this thesis. The results are then discussed in respect to their validity, and the discussion turns to examine how this thesis would have been different, had the knowledge of today been present when the project started. Lastly perspectives are discussed.

12.2 Results

The result of this thesis is the answer to the meta-question: Why is open source software being developed? Which is comprised of the four research questions. The answer to the meta-question is more than the sum of the four research questions, and therefore they will be presented as well. The four research questions are therefore answered first.

12.2.1 Research Question 1

"How is open source software being used, exchanged, and marketed?"

The question was posed to focus on the empirical study, prepare for the theoretical work and subsequent development of the model. The research question was purposely explorative in its nature at the expense of precision and clarity. Use refers to how firms and individuals are using open source software. Exchange refers to how firms and individuals exchange open source software, i.e. how is it distributed between agents. Market refers

directly to the commercial side of software distribution and exchange, i.e. how and under what conditions is open source software marketed.

12.2.1.1 How is Open Source Software Being Used?

Use of open source software is subject to constant flux, and its use has changed significantly during the project. When the project began in November 1998, open source software was renowned for its high quality in server services. Open source software had for a long time been accepted as capable of delivering best of breed solutions in areas like web serving, file and print services, DNS service, email, routing, and others.

During the project interest in open source software has exploded, and one of the fist signs was IBM's announcement of their willingness to spend one billion $ on Linux. Many other large corporations have since expressed significant interest in Linux.

At the beginning of the project, Linux and open source software were hard to use on the desktop, and very few people considered open source software as an alternative to Microsoft windows on the desktop. Many applications existed, but they were crude, and installation and configuration were painful.

Today Linux and open source software is increasingly being considered as replacements for Windows desktops. Linux and open source software has in four years reached a level of usability, where it is possible to install and use the software even for a novice user. The review of three major Linux distributions revealed that a complete Linux installation could be performed including a plethora of applications, and a working system could be obtained within an hours work.

The commercial Linux distributors like Red Hat, Mandrake, SuSE, which are reviewed, and many others are trying to offer cost effective solutions to fit the needs of the average desktop user. So far Linux has yet to overturn the undisputed dominance of Windows on the desktop. There are many reasons for the lack of acceptance on the desktop. One is, that Linux does not enjoy the same application support, as does Windows. Users are not used to Linux, and significant switching costs are associated with replacing a Windows desktop by Linux. Currently Linux distributions has reached a level, where it is very easy to install Linux and get a basic system running with lots of open source applications. However, customizing the system to fit personal preferences is still more difficult and requires far more detailed knowledge than does Windows.

12.2.1.2 How is Open Source Software Being Exchanged?

The Internet is instrumental in exchanging open source software. Open source software is exchanged using the Internet, where it is accessible to anyone interested; either the projects are hosted at one of the available open source software hosting sites, or the project is hosted at a private/corporate/institutional website.

Hosting websites such as SourceForge offers source code control, tools for bug tracking, communications services such as mailing lists, and other services. Hosting websites offer a complete system and interface for developing, distributing, and exchange open source software. Users can use the websites to download the latest versions, upload modifications, or communicate with other users and developers of a program.

If developers choose not to use the hosting websites, a project can easily be represented on the Internet by making the software available on a website.

Open source software is also distributed using other media such CD-ROMs, and it is possible to buy a compilation of software for a reasonable price. This will often be a preferred choice for agents with limited access to the Internet. Open source software is also distributed via magazines containing a CD-ROM with open source software. The Free Software Foundation is one example of an organisation that prepares special compilations of software at a price.

12.2.1.3 How is Open Source Software being Marketed?

Marketing of open source software is in itself an interesting topic. Being open source, the software can be packaged and distributed by anybody, who wishes to do so. However, the focus here is the commercial Linux distributors, who package and configure a complete Linux operating system including the best open source applications available. Linux distributions are marketed and sold as normal closed source software in shrink-wrapped boxes. Distributors add value to the distributions by making the software easily installable and by packaging the installed programs using one of the available package formats (RPM or deb). Firms can then offer easy upgrade of the software and relieve users of the problems of downloading and compiling from source.

The review of three Linux distributions also revealed that while Mandrake and Red Hat offered their distributions as a free download, SuSE did not. The two former distributions could be downloaded and burned on to a CD for subsequent installation and use. The only thing users did not get by not buying the shrink-wrapped version was the printed manual, which is often handy as a desktop reference.

The review also discovered that although the three distributions used the same package format, some degree of incompatibility existed. This is apparently the distributor's way of tying users to a particular distribution: Maintain program compatibility while subtly differentiating the package system. Users will prefer to use packages designed for their own distribution, which might create an increasing returns regime with a dominating distributor.

The strategy of the Linux distributors appears to be one of gaining installed base, and then sell service and upgrade subscriptions.

12.2.2 Research Question 2

"What characterises the open source software development process?"

While focusing on the process and not on the incentives for agents' choices in the process, the open source software development process can be characterised as a distributed development model, as developers on a single project are geographically dispersed.

Open source software development is organised with one or more persons taking responsibility for each project. Usually the one person responsible is referred to as a maintainer, and a group is referred to as a core team. The maintainer and the core team serve the same function of releasing new versions of the project software, deciding what functionality is part of the next version, and acting as quality review service.

The maintainer of a program is a single person, who may individually decide what to accept from other developers. The maintainer is in a position to accept or reject patches from other developers. By being the central point for new releases of a program, the

maintainer has complete control of the program development. However, it should be noted that the maintainer has little power to direct the effort of other developers.

Core teams have an organisational form, where a group of people work together to perform the duties of the maintainer. The interview with Poul-Henning Kamp revealed the workload to review patches was one reason for using a core team. A core team is also a democratic approach to managing an open source software project, which may reduce personal tension and change moods of the maintainer.

Large projects based in a single maintainer like the Linux kernel does not place the complete patch review workload on Linus Torvalds. Subsystems of the kernel have individual maintainers, and changes to a particular subsystem must be approved by the relevant maintainer before being sent to the Linus Tovalds. The Linux kernel is distributed with a file containing a list of the maintainers of each subsystem.

Open source software projects rely on Internet based forms of communication and most projects must have a website to accomplish general communication to potential users. If a project is not visible or available on the Internet, no users will ever learn the existence of the project.

Most communication between developers rely on mailing lists, however, the interview with Kenneth Christiansen revealed that real-time chat was fast becoming widely used. Chat has the advantage of being real time, and a sudden problem can be cleared up instantly – provided the persons in question are on-line.

Poul-Henning Kamp mentioned that the FreeBSD developers sometime resorted to using plain old telephones to clear up problems. Telephones were not used much, but when things were complicated, telephones were the only usable media. No other place (literature, websites, or interviews) has mentioned telephones as a means of communication. Poul-Henning Kamp mentioned that this was probably due to the close interpersonal contact being much more intimidating than just sending an email or chatting to someone already online.

The specific elements of open source software development can be characterised as a recursive process repeating the following seven steps. The maintainer is the agent, who releases the first version of the program, and the user-developer is a potential user and developer (in the illustrated seven step process, the user-developer chooses to use, modify, and return modifications to the maintainer):

1. A Maintainer releases a program and source code
2. A User-developer downloads software and source code
3. The User-developer identifies problems or needed features
4. The User-developer modifies the original program
5. The User-developer returns modifications to the maintaner
6. The modifications are discussed with other agents who develop
7. The maintainer reviews the modifications and perhaps includes changes in the program
1. The maintainer releases program and source code

And so the process repeats itself.

12.2.3 Research Question 3

"How do properties of software (license - and technical properties) affect open source software as an economic good? And how does the type of good affect behaviour?"

This question was answered in section 6.3 and it focused on answering the meta-question specific to the theory of economic goods.

12.2.3.1 How do properties of software (license - and technical properties) affect open source software as an economic good?

The technical properties of software refer to properties of software when examining a program on a computer connected to the Internet. Software can be copied in unlimited numbers without loss of quality, and when connected to the Internet it can be mass distributed at insignificant cost. Software with no restrictions imposed by copyright and licenses are non-rival in consumption, and availability on the Internet is non-excludable.

Technical properties of software are, however, not the only factor affecting rivalry in consumption and excludability. Copyright grants the producer of a program exclusive right to distribute and copy a program. The copyright holder can through a license transfer any amount of right, he may choose, to third party. Copyright ensures the copyright holder a strong set of rights, which, upon creation, makes software good a club good. This was also the intention of copyright law, to ensure exclusive rights to those producing intellectual property. Intellectual property had to be protected, as the cost of production is high, and the cost of copying and reproduction is disproportionably low.

The analysis concluded that the copyright holder is able to control the type of good depending on the license of the software. A program licensed under the MS EULA is turned into club good, as the license only allows the licensee to use the program. Open source software licenses like the GPL and the BSD license turn a program into a pure public good. Open source software licenses permit anyone to copy, modify, and distribute the program, as they please, and consequently the good constituted by the program is non-excludable.

Open source software turns the principle of copyright upside down and uses copyright to enforce a strong set of rights to users of the software.

12.2.3.2 How Does the Type of Good Affect Behaviour?

The analysis assumed a simple static situation with two types of agents agents: Maintainer and user-developer. The maintainer developed a program and distributed it to user-developers, who consumed the program.

Whether being a firm or an individual, the user-developer has an incentive to use open source software, as it allows the user-developer to use, modify, copy, and distribute the program freely. Thus, given a choice of comparable programs licensed as either open source or a MS EULA style license, user-developer will choose open source software, regardless of the license being BSD or GPL.

The maintainer is the developer of the program, and there is a cost associated with developing a program. For firms, the cost in terms of salaries is equal to the development cost. For individuals, the cost of development is equal to the opportunity cost associated with time spent on developing the program.

Firms acting as a maintainer have no incentive to distribute a program under one of the open source software licenses, as the MS EULA style license is the only license turning

the software into a club good. As a club good, a firm may charge a price for use of the program and thereby cover their costs and make a profit.

Individuals acting as a maintainer have no incentive to even distribute a program. If individuals are interested in profiting from their program, they transform into a firm and distribute their software under a MS EULA style license. Individuals do not derive any benefit from distributing their program under an open source license, they rather incur an additional opportunity cost from time spent on distributing the program.

12.2.4 Research Question 4

"What are the externalities of open source software? What is the source of these externalities? And how do the externalities affect behaviour?"

12.2.4.1 What are the externalities of open source software?

A program in itself does not posses any externalities, as externalities per definition arise from actions performed by agents. When examining open source software as a continuing process of development and releases of new versions of programs, externalities become evident.

Each and every time a firm or individual makes a modification to a program and the modification is included in the next releases of the program, the value of the program has increased. Firms and individuals are merely enhancing their use-products i.e. adding a needed feature of fixing a bug. However, by doing so firms and individuals confer an appreciable benefit to users of the next version of the program. This creates a positive externality, which other users can benefit from. However, the act of creating a modification does not create externalities, it is the distribution and inclusion of a modification in the next version of a program.

Users of the next version of the program might enjoy the benefit of the modification made by the developer depending on the overlap of use-products. The more the user's use-product is identical to that of the developer, the more he will enjoy benefit from the modifications, and the larger the externality.

The maintainer will, on some occasions, reject modifications from user-developers, and in that case a modification is not included in the next release. Depending on the rejected agent's response, possible externalities are either lost or diminished. If the agent decides that the modification is not worth making publicly available, possible externalities are lost. If the agent decides to maintain a separate set of modifications, publicly potential externalities exist, but agents will incur a search cost from obtaining the modifications as well as a cost from implementation.

Negative externalities arise, when a program is modified in a way, which changes its expected behaviour or removes features. Users, who rely on this feature, will find their use-product not functioning when upgrading to the latest version. Users must then decide to either continue using the previous version or incur a cost from adapting to the new version.

12.2.4.2 What is the Source of these Externalities?

The source of externalities in open source software development is the modifications added by firms and/or individuals, which are made publicly available. The size of

externalities depends on whether the modifications are included in the next version of a program or maintained in parallel.

However, the externalities can only exist because of the specific properties of software. Technical properties of software make software possible to copy without loss of quality. Consequently, technical properties ensures that modifications can be distributed at non-prohibitive cost. Had there been significant costs involved in obtaining or distributing open source software, the dominating properties would have been a club good – excludability due to cost of distribution, but non-rivalry in consumption. Externalities would still be present but could only be obtained at a cost. Distribution costs would lower the number of agents participating in the development cycle at lower development pace.

License properties are just as important, since they determine the type of good. A MS EULA style license creates a club good, for which the externalities are constituted by the tips and solutions to usage problems.

Open source software licenses create public goods, where externalities are tips and solutions to usage problems and modifications to programs. It is the modifications that provide significant benefits to other agents. Open source software licenses allow these externalities to exist, while the MS EULA style licenses ensure that modifications can only be created by the copyright holder i.e. Microsoft.

The GPL license is a noteworthy source of externalities by requiring all modifications, which are distributed, to be made publicly available.

The BSD license does not make this requirement and leave the individual persons or firms to decide, whether a modification should be returned or made publicly available.

With both of the open source software licenses it is possible to make modifications and keep them private. But, by not returning modifications to the project, a developer risks not being able to enjoy the benefit of externalities from other agents' modifications. The externalities are offset by incompatibility between the privately modified version and the official version, and the agent will incur a cost from re-implementing his desired use-product in the latest official version of the program.

12.2.4.3 How do these Externalities Affect Behaviour?

Externalities affect behaviour by making available to agents the benefits, which the agents in turn can modify to obtain a desired use-product. User-developers have an incentive to use open source software, because it is a public good and thus the benefits of a program can be obtained without incurring a license cost.

Externalities provide a benefit to users, the size of which is dependent on the agents desired use-product. When a maintainer releases a program, a large externality is available to user-developers. If the complete use-product is contained in a program, the user-developer has an incentive to use the program. If a large part of, but not the complete use-product of the user-developer is contained in the program, the user-developer will have an incentive to obtain the complete use-product through modification. The user-developer will modify the program, if the modification represents a least-cost alternative to obtaining the desired use-product. Obtaining the desired use-product also represents an investment, which the user-developer naturally wishes to cover by using the desired use-product.

The user-developer may choose to keep his modifications private, in which case the modifications do not generate any externalities, we shall call this user-developer UD1. However, UD1 is not the only user-developer, who is considering modifications, and other

user-developers (UD2, UD3,…., UDn) whose preferred use-product is not completely contained in the program, are facing the same considerations. The threshold for user-developers to begin making modifications will vary according to individual cost and value of the desired use-product.

If just one of the other user-developer distributes other modifications desired by UD1, such as bug-fixes, and the modifications are included in the following version of the program, UD1 will face a choice. UD1 must decide to use 1) the new version, 2) his old version, or 3) re-implement his modifications in the new version. Either choice will result in UD1 incurring a cost from 1) not being able to use his modifications, 2) not being able to enjoy the benefits of the new version, or 3) incurring the cost of re-implementing his desired use-product.

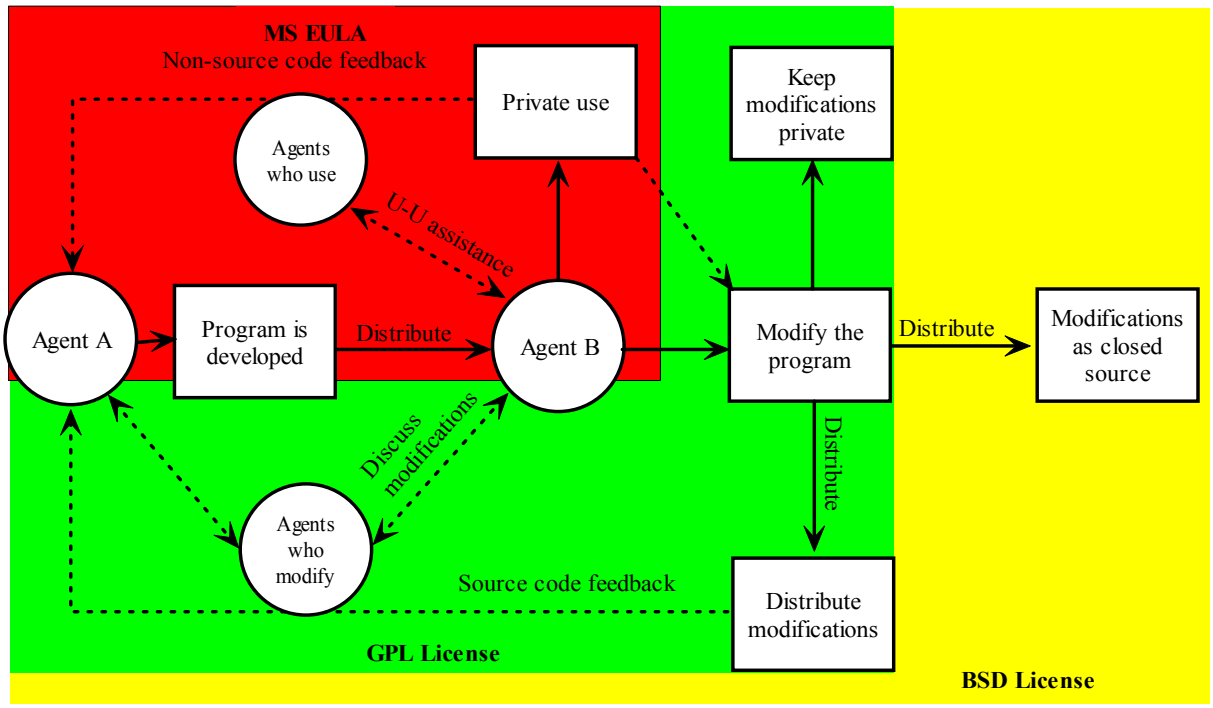
Thus, by not retuning modification to the maintainer, a user-developer risks incurring a cost from externalities generated by other user-developers. If the user-developer had returned his modifications to the maintainer, and if they had been distributed in the next version, other user-developers would have made modifications, which were compatible with the ones of UD1. UD1 could then enjoy benefits from externalities generated by other user-developers and his personal modifications. Externalities benefit other user-developers, and in the same vein they punish those, who keep modifications private.

Agents may also engage in strategic waiting for a desired use-product. In open source software development, strategic waiting may result in never obtaining the desired use-product. The nature of open source software development is for agents to implement their desired use-product. However, one agent's use-product is not per se identical to other agents' preferred use-product. For this reason, agents waiting have no guarantee that they will ever obtain their desired use-product, unless they implements the use-product themselves.

12.2.5 The Meta-question

"Why is open source software being developed?" This is the meta-question, which this thesis set out to answer, and now the time has come to present an answer.

The model (below) developed in chapter 7 forms the basis for answering the question. The model considers two types of agents, who develops and uses open source software. The model describes the behaviour of the maintainer and the user-developer(s) from an economic point of view and illustrates choices available to both the maintainer and user-developers given three licenses (MS EULA, GPL, and BSD). In the following the MS EULA is disregarded, since the question only relates to open source software. The details of the following argumentation can be found in chapter 7.



The question of why open source software is being developed can be answered by examining the behaviour of the maintainer and user-developer. The situations for the maintainer and user-developers are very different, because the maintainer is the first distributor of a program The maintainer is the person, who has copyright, and who is able to choose how to license the program.

As a firm, the maintainer will choose to license a program under the GPL license, if the program is to enter a market with a dominating program. The maintainer must hope that user-developers will adopt the program because of the advantages that the GPL license offers over a proprietary program. Releasing under the GPL allows a firm to distribute a program with fewer features while still attracting users. Thus a firm can minimize development costs when distributing a program under the GPL license. The program must, however, provide basic functionality and contain sufficiently desired use-products to warrant adoption. If no use-products or parts hereof are contained, the firm cannot expect user-developers to adopt the program.

As an individual, the maintainer, has an incentive to develop a little program under the GPL license, as this provide the best chance for adoption and subsequent modifications returned from user-developers. Modifications returned from user-developers are added features or bug-fixes to the program and valuable to the maintainer, as they improve his program and possibly improve his use-product.

The user-developer, as both a firm and an individual, has a cost saving incentive to use open source software. The licensing cost of open source software is zero and given comparable programs, open source software would be chosen in favour of proprietary software with licensing cost.

The user-developer, as both a firm and individual, also has the possibility of making modifications to the program. The user-developer will do so, if the cost of obtaining the

desired use-product by making modifications is lower than alterative choices. The program must also represent a significant value compared to the cost of modifying the program. If only a partial use-product is contained, the cost of making modifications becomes large, and the user-developer will consider developing a new program, in which case he becomes a maintainer.

User-developers' behaviour differs in their choice of whether to keep modifications private or distribute. User-developer are assumed to return modifications to the maintainer for inclusion if they distribute modifications. A firm will only distribute modifications, if they are not a source of competitive advantage. If the modifications are improvements to non-competitive infrastructure elements, they will be returned. Individuals will always distribute modifications due to maintenance costs.

Open source software is continuously being improved, and therefore the maintainer releases new versions of his program. One well-known example is the Linux kernel, where Linus Torvalds sometimes released new versions of Linux two times a day. If the user-developer had made modifications to a program and the modifications was not part of the next version of the program. The user-developer would then have to re-implement the modifications, if he were to use the latest version of the program. Provided the part of the program containing the user-developers' modifications were not changed re-implementing the modifications would be trivial but nonetheless extra work. Should the program have changed significantly it would be time consuming and perhaps even impossible to re-implement the desired modifications.

This is the pressure on the user-developers to return modifications to the maintainer. When modifications are returned and incorporated into the next version of a program, the maintainer and other user-developers will not unaware make changes, which ruin the modifications just included. The User-developer can then use the new versions, and may rest assured that his modifications are functioning. This is the mechanism, which forces development to continue. Every release of a new version ignites a new round of use and development to take place, and if just one user-developer or the maintainer makes a modification, the maintainer can release a new version and ignite the process once again.

In a sense open source software is being developed, because agents continue to make modifications. It is not possible to coordinate agents' actions nor is it possible for agents to predict when a new agent makes a modification and returns this to the maintainer. By waiting and not returning modifications to the maintainer, the user-developer risk a new version is released and thereby the cost of re-implementing the modification. If there is the slightest chance of agents making modifications the user-developer will rush

A program developed by the maintainer is distributed, and a user-developer obtains the program and discovers it to be interesting and containing almost his complete use-product. The user-developer chooses to make the required modifications to obtain his complete use-product. By returning the modifications to the maintainer, a new version of the program is released containing more use-products than the previous version. The new version will lower barriers to make modifications to the program for other user-developers. Thus this is a continuing process, where agents make small modifications to the program to satisfy personal needs. This generates externalities affecting other agents, who use the program, and receive a program with added use-products (more features or bug-fixes).

Because this is a continuing process, agents will expect the program to evolve, but, however, expectations are also tied to the license. The GPL requires modifications distributed to be made publicly available, while the BSD license allows user-developers to

release the modifications as closed source software. For this reason agents will have higher expectations to the continuing improvement of a program released under the GPL license.

12.3 Perspectives

Open source software is in its second youth and is showing potential to radically change our perception of software. Individuals and firms are used to think of software as packaged products with a clearly defined set of capabilities. Closed source software producers carefully select and price capabilities and create products matching customer needs and maximising profits. The software producers control, to a large extent, how software is to be used, and they completely control development of their own products. New features (many of which are not needed) and upgrades drive adoption of new versions of the software, and is the primary revenue generator. Often users are completely happy with the features present in the current version of a program, but as file and data formats are proprietary, they may be forced to upgrade to stay compatible.

Open source software changes this completely. Users have the power to modify the source code and implement any desired use-products. Open source also means open file formats, which are extremely important for ensuring interoperability between programs and avoiding lock-in to a single vendor's software portfolio. Not all firms and individuals posses the competence or have the time to make modifications to their open source software. In this case, firms, specializing in the particular program or the open source developer, who wrote the original code, can be hired to make the desired modifications.

On a larger scale this will redefine the software industry. Open source software will continue to improve and catch up with proprietary products, and give more and more users an incentive to save licensing costs. Firms providing software will not be able to make a living from licensing and will have to change their business model. Delivering required modifications to open source software will become a viable business.

The GPL license is instrumental in this perspective, as firms making modifications for other firms are unable to distribute the software without having to make the modifications publicly available. The BSD license on the other hand might be dangerous, as a firm, given a potent investment, will be able to hijack a project and deliver a closed source solution with enough added value to derail adoption of the open source version of the program.

The free software licenses, i.e. the GPL and the LGPL, are the only license types, which guarantee that a program once released will remain free, and that all modifications valuable enough to warrant their distribution will be made publicly available. The BSD license cannot guarantee that and has never implied so. The BSD license is good for business, as it can be turned into a proprietary product. The disadvantage of proprietary licenses and licenses like the BSD license is the high social cost of the software. Being proprietary, the software vendor controls the level of compatibility, and is naturally trying to achieve lock-in. The lock-in situation places the software vendor in a monopoly situation, where he may dictate price and obtain supernormal profits at the expense of society.

The research presented here is instrumental for firms or individuals, who are to choose a license for their software. Given the situation and the purpose for distributing the software, the optimal license can be chosen.

This research has not analysed the policy implications of the different licenses. However, as governments are spending significant amounts of money on software, it is

important that policy makers understand the implications for the economy. For too long policy makers have neglected to dictate the license for their custom-built software, but with this research the time has come to use licensing as an active element in policy.

We can make a tentative analysis of the policy implications for the three licenses. The MS EULA creates a private property regime, which is amplified far beyond proportion, as closed source software creates powerful increasing returns regimes. This creates effective monopolies, which no society should support.

The BSD license allows government developed software to become private property. The benefits are reduced research and development costs for firms or individuals, who obtain the software. However, as the license can be change to closed source, the BSD holds the same danger as the MS EULA, and perhaps even more. The BSD license is more dangerous, if the government software is widely adopted and large investments are made in development. A proprietary software vendor will in time change the compatibility regime and pursue a strategy to achieve lock-in and become dominating – in other words a monopolist. The other firms and individuals, who adopted the software, are now facing a huge sunk cost, as they cannot be part of the dominating compatibility regime. The lock-in lead-time for the BSD license is no longer leading to larger sunk costs when, and - not if! - it happens.

The GPL license is the only license guarding against lock-in, and is thus the only sane license choice for governments. No doubt, existing software firms will do everything in their power to fight such a decision. Software firms are well aware that the GPL license is the only license, which effectively prevent firms from becoming monopolists.

13 References

Arthur, W. Brian, 1989, "Competing technologies and lock-in by historical events", The Economic Journal, 99, March 1989, pp. 116-131

Aigrain, P., 2002," A framework for understanding the impact of GPL copylefting vs. non copylefting licenses", [website] http://opensource.mit.edu/papers/aigrain2.pdf

Axboe, J., 2000, interview conducted the 11th of September 2000.

Berne Convention, [website] http://www.wipo.int/clea/docs/en/wo/wo001en.htm

Bessen, J., 2001, "Open Source Software: Free Provision of a Complex Public Good", [Website] http://www.researchoninnovation.org/opensrc.pdf.

Bezurokov, N., 1999a "Open Source Software Development as a Special Type of Academic Research (Critique of Vulgar Raymondism), FirstMonday.

Bezurokov, N., 1999b "A Second Look at the Cathedral and the Bazaar", FirstMonday.

Brooks F. P., 1995, "The Mythical Man-Month - Essays On Software Engineering", 20th Anniversary Edition, Addison-Wesley Longman.

Brown, Z., 2001, "Kernel Traffic", #142, 19 Nov 2001 [website] http://kt.zork.net/kernel-traffic/kt20011119_142.html#5

BSA Press Release, 2002, 29. May [Website] BSA Press Room, accessed 31 May 2002, http://www.bsa.org/usa/press/newsreleases//2002-05-29.1116.phtml

Christiansen, K. R., 2000, interview conducted the 21st of September 2000.

Cornes, R. & Sandler, T., 1996, "The Theory of Externalities, Public Goods and Club Goods – second edition", Cambridge University Press.

Corbet, J., 2002, "Open source licensing helps racism?", Linux Weekly News, [Website] http://lwn.net/2002/0221/

Cox, A., 2002, "PATCH: Driver Maintainers", email to [Mailing list] linux-kernel@vger.kernel.org, Tue, 5 Nov 2002 18:22:31 +0000 (GMT)

CyberSource, 2001, "Linux vs. Windows – The Bottom Line", [website, accessed 23rd August 2002], http://www.cyber.com.au/cyber/about/linux_vs_windows_pricing_comparison.pdf

Edwards, K., 2000a "When Beggars Become Choosers", First Monday, volume 5, number 10 (October 2000), URL: http://firstmonday.org/issues/issue5_10/edwards/index.html

Edwards, K. 2000b, working paper, “The history of Unix development” URL: http://www.its.dtu.dk/ansat/ke/emp_work.pdf

Edwards, K. 2001, “Epistemic Communities, Situated Learning and Open Source Software Development”. The paper was initially prepared for the ‘Epistemic Cultures and the Practice of Interdisciplinarity’ Workshop at NTNU, Trondheim, June 11-12 2001.

Free Beer License, Version 1.0

Fielding, R. T., 1999, “Shared Leadership in the Apache Project”, Communications of the ACT, April 1999/ Vol. 42. No. 4.

Foster, E., 1997, “Best Technical Support Award” [Website] http://ww1.infoworld.com/cgi-bin/displayTC.pl?97poy.supp.htm [accessed 2. October 2002].

Gasperson, T., 2002, “Xandros Linux: "It just works," even with Windows stuff”, News Forge [Website] http://newsforge.com/newsforge/02/10/21/1749230.shtml?tid=23

Ghosh, R. A., 1998, “Cooking Pot Markets”, FirstMonday.

Ghosh, R. A. et al., 2002, “Free/Libre and Open Source Software: Survey and Study”, International Institute of Infonomics, University of Maastricht, The Netherlands and Berlecon Research GmbH, Berlin, Germany. [Website] http://www.infonomics.nl/FLOSS/report/

GNU General Public License, Version 2, June 1991 [Website] http://www.gnu.org/licenses/gpl.html

Halloween3, 1998, “Microsoft's Reaction to the "Halloween Memorandum" ” [Website] http://www.opensource.org/halloween/halloween3.html

Hammerly et al. ”Freeing the Source” in “Open Sources – Voices from the open source revolution”, DiBona et. at (eds.) 1999.

Henderson, V. & Garzik, J., 2002, “BitKeeper for Kernel Developers”, presented at Linux Symposium 2002, [Website] note that the file contain the complete proceedings. http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz

Harmon, A. & Markoff, J., 1998, “Internal Memo Shows Microsoft Executives' Concern Over Free Software”, New York Times on the Web, November 3, [Website] http://www.nytimes.com/library/tech/98/11/biztech/articles/03memo.html

Harris, S. E., 2001, “The Truth Behind the Great Server Heist”, [Website accessed 23rd August 2002], http://consultingtimes.com/Serverheist.html.

Honderich, T. (ed.) “The Oxford Companion to Philosophy”, 1995, Oxford University Press.

Jimmo.com, 2001, “The Great Linux-vs-NT Debate”, [website] http://www.linux-tutorial.info/Linux-NT_Debate/Cost_Comparison.html

Johnson, J. P., 2000, “Some Economics of Open Source Software”, [Website] http://www.idei.asso.fr/Commun/Conferences/Internet/Janvier2001/Papiers/Johnson.pdf

Johnson, J. P., 2001, “Economics of Open Source Software”, [Website] http://opensource.mit.edu/papers/johnsonopensource.pdf

Katz, M. L. & Shapiro, C., 1994, “Systems Competition and Network Effects”, The Journal og Economic perspectives, Vol. 8, Nr: 2

Kaul, I., Grunberg, I., Stern, M., 1999, ”Defining Global Public Goods” in, Kaul, I. et al. (Eds.), 1999, “Global Public Goods”, Oxford University Press, New York.

Kamp, P., 2000, interview conducted 27th of September.

Kamp, P., 2002, Email dated Wed, 23 Oct 2002 14:55:00

Kernel Traffic, 2001, URL: http://kt.zork.net/kernel-traffic/kt20010521_119.html#stats

Koch, Stefan & Geog Schnider, 2000, “Results from Software Engineering Research into Open Source Development Projects Using Public Data”,URL: http://exai3.wu-wien.ac.at/~koch/forschung/sw-eng/wp22.pdf

Koktvedgaard, M., 1996, “Lærebog i immaterialret”, Jurist og Økonomforbundets Forlag.

Kofler, M., 1997, “Linux – Installation, Configuration and Use”, Addison-Wesley, Harlow, England.

Krishnamurthy, S., 2002, “Cave or Community? An Empirical Examination of 100 Mature Open Source Projects”, FirstMonday issue 7 vol. 6. www.firstmonday.org

Lakhani et al., 2002, “ The Boston Consulting Group – Hacker Survey”, version 0.3, [Website] http://www.osdn.com/bcg/BCGHACKERSURVEY.pdf

Lakhani, K, and Von Hippel, E., 2000, “How Open Source Software Works: “Free” user-to-user assistance”, MIT Sloan School of Management Working Paper #4117

Lancashire, D., 2001, “Code, Culture and Cash: The Fading Altruism of Open Source Development”, FirstMonday isue6 vol. 12, www.firstmonday.org/issues/issue6_12/lancashire

Lave, J. & Wenger, E., 1991, "Situated Learning - Legitimate peripheral participation", Cambridge Uni Press.

Lerner, J. & Tirole, J.,2000, “The Simple Economics of Open Source”, Working Paper 7600, National Bureau of Economic research, 1050 Massachusetts Avenue, Cambridge, MA 02138

Lerner, J. & Tirole, J., 2002, “The Scope of Open Source Licensing”, [website] http://opensource.mit.edu/papers/lernertirole2.pdf

Levy, S., 1984, “Hackers – Heroes of the computer revolution”, Penguin Books.

Liebowitz, S. J. & Margolis S. E., 1994, “Network Externality: An Uncommon Tragedy”, The Jurnal of Economic perspectives, Vol 8, Nr. 2.

Linux Journal, 2002, “Linux Timeline”, Vol. 100, Publisher: Phil Huges.

Linux Weekly News, 2002, [Website] accessed 8th August, http://lwn.net

Makholm, P., 2002, Interview conducted on the 14th of September.

McKelvey, M, 2001, “Internet Entrepreneurship: Linux and the Dynamics of Open Source Software”, CRIC The University of Manchester & UMIST, CRIC Discussion Paper no 44.

Netcraft Survey , May 2002, [website] http://www.netcraft.com/survey/ accessed 28 May 2002

Open Source Definition, The, Version 1.9, [Website] http://www.opensource.org/docs/definition.php

OpenSource.Org, 1999, ”History of the Open Source effort”, http://www.opensource.dk/mirror/history.html.

Parens, B., 1999, “The Open Source Definition” in "Open Sources - Voices from the Open Source Revolution", Ed's Chris Dibona, Sam Ockman and Mark Stone, O'Reilly & Associates 1999.

Quarterman, J. S. & Wilhelm, S., 1993, “Unix, POSIX and open systems - The open Standards Puzzle”, Addison -Wesley Publishing Company, Inc.

Raymond, E. S., 1999a, “The Cathedral and Bazaar”, in The Cathedral & the Bazaar, O’Reilly and Associates.

Raymond, E. S., 1999b, “Homesteading the Noosphere”, in The Cathedral & the Bazaar, O’Reilly and Associates.

Ritchie, D. M., 1984, “The Evolution of the Unix Time-sharing system”, AT&T Bell Laboratories Technical Journal 63 no.6 part 2, October, pp. 1577-93

Salus, P. H., 1994, “A Quarter Century of UNIX”, Addison-Wesley Publishing, Inc.

Simonsen, K. J., 1999, Presentation at the Skåne Sjælland Linux User Group, 12/10-99, www.sslug.dk.

Stallman, R. M., 1999, “The GNU Operating System and the Free Software Movement” in "Open Sources - Voices from the Open Source Revolution", Ed's Chris Dibona, Sam Ockman and Mark Stone, O'Reilly & Associates 1999.

Stiglitz, J. E., 1996, “Economics”, Norton, W. W. & Company, Inc.

Stiglitz, J. E., 1999, ”Knowledge As a Global Public Good” in, Kaul, I. et al. (Eds.), 1999, “Global Public Goods”, Oxford University Press, New York.

Stiglits, J. E., 2000, “The Economics of the Public Sector”, W.W. & Norton & Company Inc..

Størner, H, 2000, Interview conducted 15th August.

Sørensen, C., 2000, Interview conducted 19th September.

The Apache Foundation, [website] accessed 23 August 2001 http://www.Apache.org/foundation/

The Open Source Definition, Version 1.9, [Website] http://mirror.opensource.dk/docs/definition.php

Tiemann, M. in Eds. DiBona et al., 1999 ” Open Sources – Vouces from the Open Source Revolution”, Oreily

Torvalds, L., 1991, 25th August, [News group posting] comp.os.minix. Can also be accessed at: Damgaard, Frank, 1999 “Linux History – 1” [website] http://www.sslug.dk/artikler/linux_history_1.html [accessed 16 May 2000].

Torvalds, L., 2000, ”Kernel Mailing List”, linux-kernel@vger.kernel.org”, Thu, 4 Jan 2001 16:01:22 -0800 (PST).

Torvalds, L., 2001, Kernel Traffic, URL: http://kt.zork.net/kernel-traffic/kt20010528_120.html#5

Torvalds and Diamond, 2001, “Bare for Sjov”, Forlaget Adlandia, Humlebæk, Danmark. Danish translation of “Just for fun – The Story of an Accidental Revolutionary”

Valloppillil, V., 1998, “The Halloween document” an internal Microsoft memorandum leaked to Eric S. Raymond who subsequently published an annotated version at http://www.opensource.org/halloween/halloween1.html.

Wheeler, D. A., July 9, 2001a “Why Open Source Software / Free Software (OSS/FS)? Look at the Numbers!” [website] accessed 21. August 2001, http://www.dwheeler.com/oss_fs_why.html

Wheeler, D. A., 2001b, “More Than a Gigabuck: Estimating GNU/Linux’s Size”, [website, accessed 27. June 2001], www.dweeler.com/sloc/redhat71-v1/redhat71sloc.html

Wheeler, D. A., 2002, “Why Open Source Software / Free Software (OSS/FS)? Look at the Numbers!”, [website, accessed 22nd April 2002], www.dweeler.com/oss_fs_why.htm

Williams, P., 2002, ” West Yorks Police pilots Linux desktops”, Vnunet, [Website] http://www.vnunet.com/News/1135986

Øst, A. and Edwards, K., 1996, “Standardisering i Økonomisk Perspektiv”, Institut for Teknologi og Samfund.