Towards a Theory for Understanding the Open Source Software Phenomenon

By Kasper Edwards*

Departmnent of Manufacturing Engineering and Production, Technical University of Denmark
Building 303 east, room 150, 2800 Lyngby, Denmark. (email: ke@ipl.dtu.dk)

ABSTRACT

This paper presents the main theoretical building blocks for the understanding of Open Source Software (OSS) development. Two different bodies of theory are presented, 1) Economic theory of goods where it is demonstrated that OSS is a pure public good. 2) Theory of epistemic communities, which are coupled with situated learning and legitimate peripheral participation to allow learning processes as a central part of understanding OSS development.

Five mechanisms of open source software are presented, 1) Barriers to entry in OSS development, 2) Why OSS reach a state of maturity unlike its commercial counterparts, 3) Reversed peer-review as a part of the development process, 4) Why the maintainer is a central figure in development and 5) How it is possible to collaborate without collaborating in OSS projects and still have a synergetic effect.

Kasper Edwards is a Ph.d. student at Departmnent of Manufacturing Engineering and Production at the Technical University of Denmark. The title of his project is "Technological Innovation in Software Industry" the project is focusing on the Open Source Software development process. Kasper Edwards has a background as a M.Sc. (Eng) and has worked for several years as an independent computer consultant. He is an experienced computer user and administrator of both Linux and the different flavours of Windows (9x/NT/2K). As a hobby he maintains a PC gaming network in a youth club powered by Linux servers. As part of both project and hobby Kasper Edwards is engaged in the SSLUG (Skåne Sjælland Linux User Group, a 6000+ member user group), and participates in mostly technical but also political discussions.

---

* I would like to thank associate professor Jørgen Lindgaard Pedersen for helpful comments and suggestions on earlier drafts on this paper and Mairanne Edwards for through language corrections. I assume full responsibility for any remaining vulnerabilities.

Table of contents

1 Introduction

Open source software is not a new phenomenon. It might even be an old phenomenon that may be traced back to the early days of computing and programming. Knowing very little of this period that starts in the 1940ies, we skip to the early 1970ies where AT&T developed the Unix operating system. AT&A were not allowed to profit from this new and revolutionising operating system. Due to accusations of abuse of their telecommunications monopoly AT&T were only allowed to make money from telecommunications. They were, however allowed, to research other areas not related to telecommunications. In 1978 AT&A were divested into smaller companies – the Baby Bells, and restrictions on the business were revoked. In the period from 1970 to 1978 AT&T gave away the source code for the growing Unix operating system for free, and Unix lived and evolved in academic communities.

The Unix operating system was living in academic communities most often as a research object. After AT&T were divested, Unix were no longer free and as a reaction Free BSD (a Unix variant) were created at the Berkeley University. Also the Free Software Foundation were formed. (For a description of the beginning of Unix see Salus 1994 and Edwards 2000a). In the 1990ies the Linux operating system were developed and lived a quiet life until approximately 1998, where all of a sudden open source software were reaching headlines. This was mainly due to Netscape loosing the browser war to Microsoft and making a last attempt to continue as open source software.

The awareness of a free, open source alternative operating system has sprawned many new projects to create software for all sorts of purposes. From the quiet backwaters of academia, open source software is now competing head to head with the world's largest software vendors. Competing companies are nothing new to the world but competing philosophies, - this is very different. Commercial software vendors sell their software for money, where as open source software is given away for free.

The continued development of open source software is interesting, and it is important to understand the mechanisms, which lie behind open source software development. For the understanding of open source software development, this paper proposes two distinct different bodies of theory and a number of mechanisms about how open source software development works.

The first theory is classical economic theory of pure public goods but interestingly it is pure public goods are privately produced. It is believed that the theory of privately produced, pure public goods are the only part of main stream economics that may explain open source software development. The treatment of open source software and public goods in this paper makes an analysis, and concludes that open source software is a privately produced, pure

public good, and ends with a brief outline of different contributions to the debate that might explain the underlying rationale.

The second theory treats open source software development projects as epistemic communities, and couples this with situated learning. Epistemic communities came about in a long search for theory that could explain the special group phenomenon, which seems to exist in open source software development. This theory will provide a perspective very different from the economical, and will shed light on the dynamics of the development process. The epistemic communities approach it intended to be linked to economic understanding by treating epistemic communities and situated learning as mechanisms which drastically reduce the cost of co-ordination. Much like the invisible hand of Adam Smith which in essence reduce co-ordination costs by collapsing all information about a product into one unite – its price.

The mechanisms presented are unconnected descriptions of parts of the open source software development process. These are inspired by writing by other scholars, and are a critique to their perception of open source software.

The theory and mechanisms presented in this paper is sometimes illustrated with interviews with Danish open source software developers. These interviews were conducted in August 2000. These interviews were personal and explorative, seven interviews have been conducted each lasting between two and four hours. The interviews were taped for later transcription.

1.1 Introduction to Open Sources Software

Open Source Software (OSS) is computer software, which comes with a license that is very different from the licenses that are used in commercial software like Microsoft's Office suite or the Windows™ operating system. Commercial companies have typically relied on very strict licence schemes, which allowed the user a minimum of rights, as the license does not transfer ownership but only the right to use the software. Usually no warranty for the functioning or potential damage caused by the software were provided. Users were not allowed to copy or modify the software in any way, in other words: The software came as is; Any problems arising from use of the software were for the user to take care of. Even if a user were able to fix some of the problems in his new software, the user was not allowed to do so and the source code to the software would not be provided. The source code for a piece of software is the human-readable instructions that make up the program. When the source code is compiled, a special program (a compiler) translates the human-readable source code into a machine-readable code, which a computer can understand.

Crucial to elaborating on or modifying a program is access to the source code. As noted the license to open source software is very different from that of commercial software. The name open source software hints one of the special properties of the license: The source code for the software is freely available – open source. In this paper the words 'software' and 'program' is used in a similar way. The usual understanding of a program is that it is contained in one file, whereas software is a more generic term that covers programs and/or applications, which run on computers as well as the many programs or sub-programs that constitutes, for instance, an office application. Software usually consists of several or many programs performing different functions for the software. This is done to make the software easier to maintain and design.

Apart from access to the source code, open source software licenses have other properties which are very different from that of the usual commercial software license. An open source software license[1] grants the user:

- The right to access and review the source code
- Access to the source code may not cost more than shipping and handling[2]
- The right to modify the source code
- The right to use the source code or parts of in his or her own product
- The right to freely distribute the software or source code

To understand the implications of these five bullets, a few further comments seem in place. The five bullets provide the ability to obtain a piece of software at little or no cost and to use it. Also the five bullets provide anyone with the option to turn a piece of software into a product and distribute the derived software. It is noteworthy that it is allowed to sell the software at any price one may choose, as long as the source code is available at a price no more that shipping and handling. Needless to say, Open Source Software come cheap at little or no cost.

However, open source software are covered by copyright, and most if not all programs contain a reference to a license and a copyright notice from the copyright holder. Given the rights that open source software license provide the user, the copyright means that the author retains the right to the program. Other people are not allowed to make changes to a program

[1] At least six different types of licenses exist, where the specifics wary, mentioned here are key common features. For a description of license types see Perens, Bruce, 1999.

[2] It is assumed that source code in most cases can be accessed at a website.

and distribute the program under the same name as the original author. Full credit must also be given to the source, which is the original author.

Intuitively it seems that this type of license will be of little use if any use at all, assuming the developer wishes to make a profit from his creation. There is an almost complete transfer of rights from the author to the user, and the ability to appropriate benefits from ones effort is limited. Assuming that firms and people wish to profit from their creations, the incentive for a software producer, be it a firm or a private person, to apply a open source software license seems non-existent. Given the rights which an open source software license grant the user, it is not likely that a developer be it a firm or person will be able to profit from selling the software. If a copy should be sold, the buyer is allowed to distribute it at his own pleasure, and why shouldn't he cover his expenses by selling 10 copies at a tenth of the price – after all he's allowed to do so. It is doubtful that any company will be able to pay salaries and rent by using such a licensing scheme.

However, the picture just described is far from the truth although the incentives cannot be understood under the normal economic assumptions open that source software is being developed by both firms and private individuals. In some areas like webservers open source software products is claming in excess of 60% of the market[3]. The Linux operating system is another prominent example of open source software. Linux is fast becoming the dominating Unix operating system with growing backup from commercial vendors. IBM is backing their Linux serves with 24 hour 7days support and other vendors are following this example.

In stark contrast to the problems of understanding the incentive to develop open source software, adoption or use can easily be understood within the framework of standard economics. The incentive to use a product is extremely high, if the product has the required quality, and if the price is low. In the case of the Apache webserver the price is negligible - it can be downloaded at no charge from many websites[4], and it is included in most Linux distributions[5]. The volume of software which is developed and offered to anyone, who wishes to use it, is overwhelming. A quick look at SourceForge[6] shows that they host 14,198 projects, and currently have 108,060 registered users. A project is a commonly used reference

---

[3] Netcraft Web Server Survey, accessed January 4th 2001.

[4] Apache may be downloaded from http://www.apache.org

[5] A Linux distribution is a complete Linux operating system with many applications usually distributed on a CD-ROM. Many companies market Linux distributions under a bran

d name scheme, examples are RedHat Inc., SuSE Gmbh., Mandrake Inc. and many several others.

[6] SourceForge, accessed January 18th 2001.

to a particular development effort i.e. the development of a certain piece of software. Other websites like SourceForge exist, Freshmeat and the Linux portal Linux Online[7] are examples of web sites, which host many projects. The projects range from applications to projects want to use Linux on other hardware than originally intended (called a port). All these projects and many more apply open source software license, and could thus be used and further developed whatever the particular need might be.

In general the organisation of open source software development is structured as many different projects some of which are dependant on each other. Some projects are overlapping in their goal and others try to archive the same goal. Open source software development is dependent on the Internet to provide infrastructure for communications and transport of all things that may be digitised. Websites relating to the different projects provide means of communication from the developers. Newsgroups and email provide two way communications between developers and users. Real time communications are provided by chat-forums. Organisationally most open source development projects have a central person or group of people called maintainers. People who wish to contribute to a program or the software may then communicate their comments or their changes to the maintainer. Often this communication is done using one or more dedicated newsgroups/mailinglists, where discussions are open to all interested parties. Contributions are sent to the newsgroup for discussion and possible acceptance by the maintainer. If accepted, the maintainer includes the contribution[8] in the software, and makes the new version available to the public.

Often the maintainer is the person who has written the first version of the software in the project. The maintainer can to a certain degree control the development of the program, but contributors are at any time free to use the source code in a new project of their own and create a competing project. However, competing projects can not use the name from the original project, and may thus have problems creating a user base. However, the maintainer has an obvious interest in attracting as many co-developers as possible, since this will speed up development of the program. (For a discussion of the relationship between maintainers and co-developers see Edwards 2000b).

[7] Linux.org, projects section accessed January 18th 2001.

[8] It is assumed that contributions are source code to be included in the software.

2 What to Understand/Research?

The introduction intuitively raises many questions: How can this be? Why do people contribute? Why do people give away the fruit of their efforts for free? How did it come this far? All these questions and many others become even more accentuated, when one realises that open source software products are important players in certain segments of the market for software.

Research in Open Source Software is a growing field of interest, but a common understanding of the phenomenon there is yet to established. Papers treating open source software development often use non-conventional economic[9] theory or theories from psychology[10] and anthropology[11] also the economics of career concern[12] have been applied to explain why open source software are developed and how.

This paper understands open source software as a construct which is developed and used by both private individuals and professional companies. This paper has chosen economic rationality as the foundation for understanding OSS development. The author being an engineer M. Sc. (Eng.) by training, finds it easy to understand why a few people could be engaged in the development of a computer operating system for the sake of technical interest and leisure. But the sheer scale of what might be called the open source movement makes it impossible to apply 'engineering for fun' as a basic rationale for explaining and understanding Open Source Software. The perspective of this thesis is one of economics and technology, and as such explanations will be found in economic theory coupled with an understanding of technology.

The meta-question which guides this paper and which later will guide the final thesis could be stated as: "What is the nature of open source software development?". This is a way of expressing that there must be some logic to open source software development. This logic should make it possible to understand who, why, how, and what are the economic decisions and in which quantities open source software are developed. In essence these are basic questions[13] that economics try to answer.

---

[9] Raymond, Eric S. 1999.

[10] The Linux Study

[11] Raymond, Eric S. 1999.

[12] Learner, Josh and Tirole, Jean, 2000.

[13] Stiglitz, J.E. ,1996, p.11.

In most areas of society, where goods have been produced and sold, economic rationality has provided plausible explanation to central questions of price, allocation and quantity. Based on two principles: 1) Voluntary transaction and 2) Mutual beneficial exchange; the invisible hand of Adam Smith has made sure that goods were produced in sufficient quantity, and allocated to those wanting (and willing to pay). For privately produced goods the exchange involve on one hand a transfer of ownership, and on the other hand a transfer of some kind of value perceived to be equal or more than that of the good being sold.

Analysing the open source community it is evident that goods are produced and transactions are conducted on a voluntary basis. Also one must assume that the exchange of goods is mutually beneficial given the fact that goods are offered and consumed. In the open source software community goods are offered, but not on a basis of direct exchange between parties. Usually the goods are offered at a web site, and the users are free to download the goods and use them. When this happens a transfer of rights from the producer to the consumer happens but no payment for the good are required. The consumer incur a cost from using the internet and an opportunity cost from time spend, but the producer receives no profit for the goods that he delivers. It must be noted that a transfer of ownership does not take place. Software it sold as a set of rights, which the producer grants the user.

At this point the basic economic understanding fails - rights are transferred, and it should be assumed that a payment equal to the marginal benefit of the goods received by the consumer should be transferred to the producer. Understanding open source software as a private, good where ownership or rights are transferred in return for some kind of payment does apparently not produce any insight into open source software development. Without being able to assign any direct cost or price to open source software it becomes impossible to analyse open source software using regular economics of supply and demand. If one considers drawing a demand/supply curve, it might be possible to calculate the cost of developing open source software as an opportunity cost, but the price per unit becomes a problem, since the open source software license allows for infinite reproduction thus lowering the cost to zero. Given these problems, it is clear that other types of theory must be applied to understand open source software development, and luckily the realm of economics has much else to offer.

A note on words, when discussing software there is little point in using the term producing, which implies a process where the same software it constructed over and over again. Contrary to production of material goods, productions in the sense of creating multiple copies of a piece of software is not difficult neither expensive. Software development may be

compared to prototyping in a manufacturing context with the slight difference that the final and approved prototype may be copied at a microscopic fraction of the development cost, and the copies are identical to the original. However, using development in an economic context will create ample room for misinterpretation and wrong associations to the economic literature. Therefore the term production will be used in the economic context when describing what actually is software development. Also the word user in a software context is close to a consumer in economics, both use a product or program. When applying economic understanding the word consumer will be applied in order to use a wording consistent with economic literature.

2.1 Conceptual Framework

Having established that supply, demand and private goods and perfect competitive market economics is an insufficient set of tools to understand open source software development, the search goes on for another theoretical framework. The economic concept of goods has a lot more to offer than just private goods and perfect markets. As previously showed private goods allowed for an unsuccessful, tentative analysis of open source software, and provided little or no insight into open source software development. Moving away from private goods opens the door to a vast body of literature that analyses goods with other properties then that of private goods.

Understanding open source software as a certain type of good will allow open source software to be treated within the realm of economics, which, as noted, has a lot to offer in terms of understanding allocation, production and consumption of goods. This will also move focus away from individual producers of open source software and provide a macro type of perspective. This is relevant, because understanding will be derived from properties of the good, that is, open source software in itself. It will then be possible to understand, how these properties affect the users and producers, which interact with the good. Also it is far more interesting to shed light on some of the general mechanisms that must be at work in open source software development. It is easy to get carried away by the idiosyncrasies of various high profile open software developers and their sometimes large egos.

However, the macro approach will provide little understanding of the group dynamics, which must exist in open source software development. To that end a micro perspective must be applied which is done by switching focus to the context in which the open source software is used and developed. Inspired by interviews with open source software developers and the authors participation in the open source software community, it is assumed that this context is a group of people rather than an individual. It is natural in open source software development

for people contribute or collaborate due to the simple fact that the source code is available and therefore it is possible to participate in the development. Also, open source software is free of charge and the licenses are explicit that the software comes 'as is', implying that no one should expect others to add features, which they themselves feel are very important. It seems that these groups of open source software developers posses special properties, where an explicit interest in using and producing knowledge precede other aspects of interaction in the group or community. Groups with these and other properties, which will be explained in a latter chapter, have already been examined in the literature and are known as epistemic communities[14].

2.2 The Units of Analysis

As with other sciences the result of ones study is dependent on what one chooses to analyse. The macro perspective which uses economic theory and the micro perspective based on epistemic communities provides two levels of analysis with different units of analysis.

The unit of analysis when applying the macro perspective is the 'good'. The good is open source software which users and producers consume and produce. The good have, as mentioned, some special properties which stems from the good itself as a technology and a thing which can be used and produced in certain ways. A later analysis will uncover these properties. The key element in this analysis is the understanding of open source software as a good in an economic sense.

The micro perspective removes any relation to economic incentives and the unit of analysis becomes the group dynamics, and how these dynamics influence the individual developer. In essence the unit of analysis is the epistemic community which surrounds a particular piece of open source software and its development.

[14] Buchart, H. and J. Marx, 1979, p. 107

3 Theory

Two very different bodies of theory are to provide understanding of the observed phenomenon. The theories are corresponding to two different levels of analysis, which in turn correspond to two different units of analysis. When analysing open source software as a good, the unit of analysis is the software itself, and this makes it possible to understand how people interact with the good, and how properties of the good affect interaction. When people interact with a good, they are either users or producers. The other level of analysis is that of the community of practitioners which collaborate in a project to create a specific piece of software. It is the hope that applying theory of epistemic communities to open source software projects will yield an understanding of how and why people collaborate on software projects and give away the results for free.

The first part of this chapter presents economic theory of goods, and a classification of properties relating to a good is offered. The logic of the first part of the chapter is that goods have different properties, and these properties affect how the good is consumed and provided. Therefore the first part of the chapter resembles a deductive process, deducing the expected behaviour related to consumption and provision when goods pose certain properties.

The second part of the chapter presents the theory of epistemic communities. Epistemic communities are a way to understand, how experts collaborate to achieve a common goal for no other reason than achieving that particular goal. Thus epistemic communities capture an important property of open source software projects: No or at best little hope to profit economically from ones effort. Of course, other criteria has to be met before people entering into an open source software project could be assumed to be part of an epistemic community.

3.1 General Theory of Goods

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 are consumed, produced and in which quantities. The market is the place, where goods are exchanged and is considered a very efficient way to produce and allocate goods. However, the marketplace 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[15]. If this is not possible, then the market, as a mechanism that

[15] Kaul et. al. in Kaul, Inge et. al. (eds.), 2000, p. 3.

determines the price of goods, and thus what to be bought and sold and in which quantities, seize to be an efficient allocater. In this situation other mechanisms begin to influence how goods are allocated and produced.

Goods are described by two properties: 1) Rivalry in consumption and 2) Excludability[16].

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 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 can not 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 nonrivalry in consumption. Nonrivalry exists, when the good can be consumed by one individual without detracting, in the slightest, from the consumption opportunities still available[17]. Looking at flowers is an example of a good that has the property of nonrivalry in consumption. People looking at the same flower and thereby consuming the good, do not diminish the value of the good, and thus other people may also consume the good.

2) Excludability

Excludability refers to the ability to exclude people from consuming a particular good, as with rivalry in consumption opposite extremes exist. 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 the 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 are non-excludable. The radio station transmits radiosignals, 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, and the cost would exceed the potential profit, and exclusion would be done at prohibitive cost. The scale of exclusion from excludability to non-excludability is closely associated with the cost of

[16] Ibid.

[17] Crones, Richard & Sandler, Todd, 1996, p. 8, 145.

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 assumed that this reserve is large, very large it is not economically feasible to erect a high fence around the reserve, so that people would have to 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.

3.1.1 Private Goods

Starting in the top left corner of the matrix goods of this type is rivalrous 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 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 rely on laws of private property, i.e. society by law recognises that people has ownership of their creations. This ownership could in turn be transferred at a certain price – often determined by the market.

3.1.2 Commons

Commons are characterised by nonexcludability 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 arise from the fact that commons are all around us like the air we breath or the ozone layer. Excluding people from consuming these goods would be impossible or at best extremely expensive, and thus not feasible.

3.1.3 Club Goods

The club good is characterised by excludability and nonrivalrous 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 persons consumption of cable TV does not affect other people's ability to consume the same good, and neither does one persons use prohibit others. However, excludability is easy, and the producer can charge a fee for consuming the good.

3.1.4 Pure Public Goods

Last is the pure public good which is nonrivalrous in consumption and 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. The good is the protection from the suns 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 also one persons consumption does not reduce the value of other people consuming the good.

3.1.5 Mixed Types of Goods

Goods should be understood as a continuum from the private good to the pure public good. In between lie a range of goods that has degrees of nonrivalrous in consumption and degrees of non-excludability. Any good that has properties of nonrivalrous in consumption and/or non-excludability is to some degree a public good.

3.1.6 Quantity and the different types of goods

The four types of goods exist only, when certain assumptions are met. A public freeway can 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, but the public is then limited to owners of cars. 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 raises 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-rivalrous. 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 will exhibit changing properties and thus become a different type of good when a capacity limit is reached.

3.1.7 Provision of Goods

Private goods can, as mentioned in section 3.1.1, be owned and this ownership can be transferred. When consuming or producing the good, all the costs are incurred by the individual, and the cost is equal to the cost incurred by society; 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 nonrivalrous in consumption, and this has an impact on the producers of the good. Because producers of club-goods don't incur a cost when producing an extra unit of the good, there are huge benefits to be gained from this type of good. With unit cost pr. consumer are falling with each additional consumer falling, club goods are the source of natural monopolies.

Goods where exclusion is not possible pose other problems to the allocation of the good. Obviously, if a good is free and people can not be excluded from consuming the good, the good will be in high demand and in danger of depletion. Public goods like fish in the sea have within the past 20 years come under heavy pressure from overfishing. This has resulted in steps being taken by the political establishment to prevent depletion of this important natural resource. This 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. This is of course 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. Externalities arise when a person or a company performs an action, but does not incur the true cost.

Pollution is the classic example, where a company pollutes a river, thereby saving the cost of cleaning the waste. In turn the pollution reduces the value of the river e.g. by killing a percentage of the wildlife, thereby reducing the 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. The company may also take action to educate their low skilled female workers with the purpose of increasing their performance. However, education of low skilled female workers have a positive effect on child survival. The company does not directly enjoy a benefit from reduced child mortality, and thus there is a positive externality from their action enjoyed by society and the women in question. What is labelled as a positive or a negative externality is dependent on perspective.

Pure public goods are as noted nonrivalrous in consumption and non-excludable, which makes them susceptible to overuse and supply problems. If it doesn't cost anything and people can't 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[18] "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". If a public good appears to be produced in sufficient quantity, we have a phenomenon that should be studied in details.

3.2 Open source software as a good

We now know that different types of goods exist, and that properties of the different types of goods have an impact on their provision and consumption. It is then important to determine what type of good open source software is. When the properties of open source software are known, the type of good can be determined and further analysis can be made. This section will begin by analysing software as a mere technical thing with no property rights attached.

Open source software is software like all other software: A sequence of instructions to be interpreted by a computer, and perform actions accordingly. The software in it self is stored on a computer or some sort of digital medium, which a computer is able to read. Being a digital manifestation software may be copied without loss of quality, and this applies to all things digital. Of course the software may be printed as either source code or binary code for at person to read. Whichever, the software then is external to the computer and can not be fed

[18] Stiglitz, J. E. in in Kaul, Inge et. al. (eds.), 2000, p. 311.

directly to the computer without transcription from paper to the computer. When on paper software has an analogue representation, which does not translate into a digital form without some human intervention.

Software exists in all shapes and sizes and is made to solve different tasks. No piece of software exists that covers all areas of use, and still more software is being developed due to computers in almost every job function. Most software solves a specific purpose. Like word processing software solves the problem of writing text and creating a layout for the text, email clients are software for sending and receiving email, and the list goes on. Software helps people perform different tasks some of which can only be performed by use of a computer with the proper software. Other tasks are made easier and faster resulting in a productivity gain. Given the increasing number of computers in society and their increasing application in solving daily tasks and performing work related tasks, it is no wonder that software is perceived as good which is often both attractive and necessary.

It is assumed that when a consumer receives a piece of software, the consumer is allowed to use the software as much as he pleases. This assumption is made to counter emerging 'pay per use' licensing schemes where software is a service, and the consumer only pays when he actually uses the software. Pay per use software has been on the 'wish list' of software companies for a long time as a partial solution to counter a growing problem of software piracy. Online services that use a 'pay per use' system have yet to experience a breakthrough in the market. Limited bandwidth internet connections and poor quality internet connections have been a problem for these types of services. Online services must be very robust and reliable in order to persuade the average computer user to use these services. Many users experience occasional, if not frequent problems with internet connections, and the thought of not being able to work due to some third party fault keep these services from widespread adoption. There shall be little doubt that once network connections and services become mature, these services will become common in the market. Open source software does not exist in this form, and software with this special property/licence is beyond the scope of this paper and will not be discussed further.

3.2.1 Software as a Good

When removing all property rights attached to software, it is a good with interesting properties. Software is a good that may be copied without loss of quality, and therefore the n'th generation copies are identical to the original. To determine the properties of software without any property rights, we must examine the way software are used by the computer. When software is executed on a computer, the computer reads the files which make up the software from some sort of storage, either a disk - or perhaps a runic stone. Regardless of

storage type the information in the files are *read* from storage and subsequently executed by the computer. Before 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 remain 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. Theses files are not considered part of the software but results from using the software.

Because software is not changed, software is available for subsequent use by the same or other persons. The economic property that software here exhibits is nonrivalry in consumption. The software good can be consumed by one individual without detracting, in the slightest, from the consumption opportunities still available to others. There are special types of software, like a malicious computer virus, that will change its behaviour upon execution by altering its files. This type of software will not be cowered in this paper, mainly because they are developed for solely destructive purposes and not adopted on purpose.

The other interesting property, excludability, is a much more complex matter. Whereas nonrivalry in consumption is mainly a property that stems from technical properties of how computers read and execute software, excludability touches behavioural maters as well. The question is: Can the owner/provider/possessor of some software exclude others from using the software, and at what cost. If it is easy to exclude others from enjoying the benefits of the software good, then excludability does exists.

We know that software is stored on some sort of medium be it a CD-ROM, harddrive or other computer readable medium. In the infancy of computers software were stored on rolls of paper tape with punched holes to be read by a tape reading device. Access to software will then require access to the medium in order to make a copy or to run the software directly from the medium. The keyword here is access, access to the medium on which the software is stored. If the owner/provider/possessor of the software is the only person to possess a copy then this person can easily control access to the medium for instance by locking the CD-ROM disk or harddrive in a safe. In this perspective, where access is controllable software is excludable.

Another situation arise, when the software is distributed from the owner/provider/possessor. In the above example a single person, perhaps the creator, had possession of the only existing copy of the software in which case he was able to exclude others from access to the software and from the possibility of enjoying the good. The creator is likely to have spent a substantial amount of time and money when developing his software. To earn rent from his creation he decides to sell his software to anyone interested. He will charge the sum of 10 credits for the software. The first customer buys the software and

enjoys the good of the software. The first customer may enjoy the good of the software over and over again, as the software is not consumed upon use. We know from the discussion of rivalry in consumption that software exhibits nonrivalry in consumption, and therefore software may be copied without detracting from the value of the good.

The first customer has five friends with the same need for software. Not being restricted by intellectual property rights the first customer decides to cover his costs and sells to each of his friends a copy of the software for the sum of 2 credits. Now, the first customer has covered his costs and done his friends a favour by saving them 8 credits each. The five friends 5 friends each to whom they sell the software for 0,4 credits. In this case software will spread like wildfire with price being reduced to nothing. Five people distributing the software to five friends each and repeating the process five times will result in 3125 copies being distributed at prices going down to 0,0032 credits – based on the assumption that each seller just covers his costs.

This illustrates an interesting property of software, which is actually a combination of technical properties and human behaviour. The important technical property is that software exhibits nonrivalry in consumption, that software may be copied, and that the copy and the original are identical. This also illustrates the first copy problem[19]: The first copy is expensive, because the cost is equal to the cost of development. Additional copies on the other hand cost very little to produce.

No matter the cost nor the incentive to distribute software to cover ones costs, the ability to redistribute software relies on the fact that buyers may potentially be excluded from enjoying the good. We can then conclude that technically software exhibits excludability, as it is possible to exclude others from enjoying the good.

3.2.2 Property rights

There are, however, certain rights which a creator of software may claim, which change the properties of software. Society has agreed that intellectual property rights should be granted to anyone who produces intellectual work. The intellectual property rights are descendants from property rights, where people can own land and enjoy protection of their land from the authorities. According to the Bern Convention anyone who creates a piece of art should enjoy legal protection of his creation. The Bern Convention was by the 1st January 1994 ratified by 105 nations[20] and provides this legal protection of creations, that does not

[19] Shapiro, Carl & Varian, Hal R., 1999, p. 21.

[20] Koktvedgaard, Mogens, 1994, p. 38.

require the creation to be registered anywhere. If a person wishes to enjoy protection from the Universal Copyright Convention it is required that the international copyright mark ©, name of the copyright holder and year of publication is stated in the publiction. When enjoying copyright protection one is not allowed to make copies of others creations without permission. Copyright is a central right, since anyone who makes a program or just a line of code automatically[21] receives copyright to his creation, in fact even my private shopping list is covered by copyright.

Copyright and other types of legal protection are in themselves pure public goods, which society provides. The basic copyright protection grants the creator protection from other people copying his work, i.e. people are not allowed to make copies without permission from the copyright holder. By declaring the terms of use (the license) of the software, the author (the copyright holder) grants some rights to the users of the software (the licensees).

When software is sold as a product or provided for free as open source software, there is a transfer of rights from the copyright holder to the licensee. The license defines the terms of use for a particular piece of software, and states what the licensee may and may not do with software at hand. Commercial software companies are usually very restrictive when defining the license terms only allowing the users (customer) the right to use the software and do not permit any copying, distribution or reverse engineering. Needless to say, this is done to ensure profits from sale of software and to protect their intellectual property rights.

As such the license is central when understanding properties of software and which type of good a particular type of software is. By defining a license where the licensee is not allowed to redistribute the software, the copyright holder retains the right to distribute the software and makes it illegal to distribute the software. Licenses restricting the licensee to just use the software and not permitting distribution or copying in any form, transform the software into a good, which exhibits excludability. If the license further permits only one user per license, the good will also be rivalrous in consumption. Most commercial software companies enforce tight restrictions on redistribution, copying and the number of copies in use at the same time. With these properties commercial software is private goods. It is of course a condition that people obey the law. The author is well aware that software is pirated and distributed illegally, which however does not affect the theoretical consequences of licensing influencing properties of software.

[21] According to the Bern Convention the acquisition of legal protection of a creation is instant.

3.2.3 Open source software

In sharp contrast to the licenses of commercial companies are the open source licenses. There are many different types of license referred to as open source software. In this paper open source software are defined as software who's license comply with the Open Source Definition[22] (OSD). In OSI's own words:

> "Open Source Initiative (OSI) is a non-profit corporation dedicated to managing and promoting the Open Source Definition for the good of the specifically through the OSI Certified Open Source Software certification mark and program." [23]

OSI does not define or promote a specific license but rather specify requirements which a license must meet in order to conform to the Open Source Definition.

As noted in the introduction open source software has a license which grants the user far more privileges than does commercial software. We may call it closed source software since commercial companies never allow public access to the source code for their software. The licensee has the right to freely distribute the software and the right to modify the software. With open source software it is required that the source code is distributed, and thereby it is possible to modify the program. Open source software permits the creation of derived works, i.e. a person may take some open source software, modify it and redistribute the new software. It is required that the source code for the software is publicly accessible. Open source software place no restrictions on the number of people using the same copy of the software. The copyright holder may require that derived works carry a different name. Even if the software is redistributed under a different name, the original author retains the rights to the lines of code in the software, which the original author wrote.

This makes open source nonrivalrous in consumption since there are no restrictions on the number of users allowed to use open source software at the same time. An open source software license permits and even encourages distribution of open source, software and it is explicit that no group or person may be excluded from use. This makes open source non-exclusive in consumption. It can then be concluded that open source software is a pure public good.

3.2.4 Summing up open source software as a good

Analysis of the type of good, which software is has demonstrated three things:

---

[22] The Open Source Definition

[23] Ibid.

- Software without any property rights attached is nonrivalrous in consumption and exhibits excludability, because assess may be denied, and thus it is a club good.
- Intellectual property rights makes it possible to transform software into a private good by prohibiting distribution, copying and more than one user per license.
- Open source software uses a license which encourages free distribution and prohibits exclusion of groups of people thus making open source software a pure public good.

Further it can be concluded that the license applied to software is instrumental when understanding what type of good the software is. The license defines the limits for proper use, and defines when the line to unlawful use is crossed. Often, however, the line is blurred and must be determined in a court of law.

3.2.5 What to Expect from Treating Open Source Software as a Good

In the past sections we have seen that open source software may be understood as a pure public good. It is then clear that any theoretical understanding derived from the realms of economics should be built on the economics of public goods. What is particularly interesting is the fact that open source software is privately produced. The theory should address privately produced public goods, which is interesting but certainly not without difficulties. As Stiglitz pointed out, privately produced public goods will be undersupplied[24]. It is not possible at this point to answer whether open source software as a good is underssupplied or not. It is the hope that analysing open source software development as a pure public good will provide some insights as to what are the mechanisms of open source software development.

Another approach to understanding provision of open source software has been proposed by Learner and Tirole[25]. As mentioned earlier Learner and Tirole try to understand the incentives in open source software development as the economics of qualifying for a job. Developers in a open source software project contribute code to acquire skilles, get a reputation and write a CV. The logic is that the CV is a direct qualification for a later job. Further participation does not require a formal education and may be considered an actual qualifying education in the software sector. The clever thing in their approach is their focus on the individual incentive, which means that they do not have to address the issue of privately produced public goods. It is, however, not certain that the approach offered by

[24] Stiglitz, J. E. in in Kaul, Inge et. al. (eds.), 2000, p. 311.

[25] Learner and Tirole, 2000

Learner and Tirole provide an exhaustive answer to why open source software is provided and continues to grow. This paper is not the place for such a discussion and, the issue will be dealt with in the final thesis or a separate paper.

3.3 Epistemic Communities

In the following sections the theory that are to be used in the micro perspective of the analysis is presented. As mentioned, the micro perspective has its focus on the level of the open source software project itself. The aim of the micro perspective is to understand the motivational factors of open source software development and also how innovation is conducted. It is the impression that some of the factors wich motivate developers of open source software should be found in the communities where, development takes place. As such light must be shed on both the group and its dynamics and the single participator of the group who acts as both a person and as part of the group. There shall be little doubt that a group and a single person behave differently when facing the same situation. The single individual is prone to be intimidated when facing a larger opponent, while the individual when in a group will feel a sense of security even when facing a likewise larger opponent. It is not the purpose of this paper to elaborate on the theory of groups versus individuals, but rather to state that it is recognised that there are differences in the way a group and a single person behave under the same circumstances.

In search for literature that could ameliorate the apparent lack of understanding at the micro level, the epistemic communities' approach came up. The decision to use the epistemic communities' approach was triggered by some similarities in the open source movement and the communities, which the epistemic communities literature tried to analyse. People collaborating on open source software projects (developers) are organised in a network where nobody has any formal power to influence others. The developers are equal – so to say- they do however share an interest in developing open source software and sharing this software with others. Developers are not tied together in any relationship which includes monetary compensation, and developers are free to enter and leave projects as they see fit. These characteristics are very similar to those found in communities analysed by Holzner & Marx[26] and Haas[27]. The settings in which the theory is applied differ, but important similarities like the just mentioned exist. The theory of epistemic communities that Holzner & Marx and Haas present differ to some extent, perhaps due to the different communities they analyse.

[26] Holzner and Maax, 1979.

[27] Haas, P.M. 1989.

This has implications for the epistemic communities' approach and the theory which must be 'adjusted'.

Going through the literature concerning epistemic communities, epistemic cultures[28] was examined as well. The epistemic cultures' approach had little to offer in terms of understanding open source software development. Methodologically epistemic cultures employed a comparative analysis to show the differences between two epistemic cultures. Karin Knorr-Citina and her team analysed the culture at CERN and compared those findings with a similar study of bio-engineers. Of particular interest was an analysis of how the two cultures viewed the object they studied. The CERN people used a particle accelerator as their primary tool and through their work developed an almost personal relationship with the accelerator. However interesting, understanding open source software development as a particular cultural phenomenon is not the ambition of this thesis. The methodology used by Karin Knorr-Citina and her team were observations and interviews, and the quality of their research can in part be attributed to the huge amount of time spent in the field studying the manifestations of culture. Thus, the epistemic cultures' approach was abandoned.

The epistemic communities' approach on the other hand seems to offer an analytical approach to understand how communities interested in producing and using knowledge come about, and how individuals enter into the community. Two contributions are foundation for the theory of epistemic communities, these two come from Holzner & Marx and Haas. The theory presented in these two contributions are, as mentioned, not identical, and an understanding unique to this thesis will have to be developed.

The following description of epistemic communities is based on Holzner and Marx's[29] approach using epistemic communities to describe the American medical profession. Also used in the following is a later publication by Peter M. Haas[30] who offers a different take on epistemic communities in which he relates international policy coordination to an act, where epistemic communities can play a large part. Haas in his paper recognises the work of Holzner and Marx and elaborates on some of their findings, while distancing his approach in other respects.

---

[28] Knorr-Cetina, Karin, 1999.

[29] Holzner, B. and J. Marx, 1979.

[30] Haas, P. M., 1989.

3.3.1 Defining Epistemic Communities

Holzner and Marx[31] seem to be the first to introduce the term epistemic communities and to use the term to distinguish this type of work community from others. Epistemic communities put a high priority to epistemic criteria which are criteria related to production and use of knowledge. In epistemic communities epistemic criteria have primacy over other interests and orientations, as such these communities will place the interest in using and producing knowledge higher than in social relations within the community. In the epistemic community social relations depend on the relations formed around the interest in generating and producing knowledge. This means that friendships and other relations are initiated as a consequence of knowledge being communicated between participants in the community. One person may need knowledge to complete at particular task and turn to other participants for that knowledge. Being part of the same epistemic community participants know more about the other persons' strengths and weaknesses in different areas related to the knowledge activity in the community than they know about their personal traits.

Epistemic communities may be organised as both formal organisations and informal groups, where the latter seem to be the most common. The reason for using the term epistemic communities is to emphasise that these communities differ from other types of communities. As Holzner and Marx note:

> "The key element is the primacy of epistemic criteria for activities that involve the creation of new reality constructs or their application to situations of practice."[32]

An example of epistemic communities is scientific communities where, epistemic criteria have a dominant position. Epistemic criteria such as formulating theory, which is intrinsically logical and empirical testing of theory through reproducible experiments have been institutionalised within the scientific community.

The term epistemic communities thus designate those knowledge-oriented communities in which cultural standards and social arrangements interpenetrates around a primary commitment to epistemic criteria in knowledge production and application[33].

Put this way, the before mentioned scientific community is not the only type of community that can be defined as an epistemic community. Any community that centres around creation of knowledge or uses some special kind of knowledge may be said to be an

---

[31] Holzner, B. and J. Marx, 1979, p. 107.

[32] Ibid.

[33] Ibid.

epistemic community. Most interestingly is that these communities be they users of a special body of knowledge, producers or both, tend toward the structure of an epistemic community. Even the most esoteric knowledge traditions such as serious aura readers and healers might be categorised as epistemic communities in the same way officially recognised knowledge disciplines like material science may be.

Haas is more formalistic is his definition of epistemic communities and defines them as:

> "… a network of professionals with recognised experience and competence in a particular domain and competence in a particular domain and an authoritative claim to policy relevant knowledge with in that domain or issue-area."[34]

This definition clearly shows that the intentions of epistemic communities are to explain the policy impact of epistemic communities on policy formulation. The definition is explicit in stating that epistemic communities consist of professionals and people with relevant knowledge and competencies. However, in a footnote[35] it is noted that epistemic communities need not be made up of natural scientists and professionals thus broadening the definition to include all types of groups which possess some body of knowledge. Haas mentions Holzner and Marx as proponents of a definition of epistemic communities in which the members of the community are bound together by a shared faith in the scientific method as a way of generating truth. Based upon readings of Holzner and Marx, there seems to be a point missing in their definition where the community is bound together by a primary commitment to production and application of knowledge[36]. However, the analysis performed by Holzner and Marx is conducted on the American medical profession, which is an epistemic community based on the tradition of natural science. According to Haas definition people are bonded by particular knowledge and truths and their verity and application, a kind of a thought collective.

In the definition of Hass an epistemic community may consist of participants from various disciplines with various previous experience, and they have: 1) A shared set of normative and principled beliefs, providing a value based rationale for the social action of community members; 2) Shared causal beliefs, which are derived from their analysis of practices leading or contributing to a central set of problems in their domain and serving as

---

[34] Haas, P. M., 1992, p. 3.

[35] Ibid.

[36] Holzner and Marx 1979, p. 108.

the basis for elucidating the multiple linkages between possible policy actions and desired outcomes; 3) Shared notions of validity – that is, intersubjective, internally defined criteria for weighing and validating knowledge in the domain of their expertise; and 4) A common policy enterprise – that is, a set of common practices associated with a set of problems to which their competence is directed, presumably out of the conviction that human welfare will be enhanced as a consequence[37].

What is apparent from the two definitions is that they are somewhat specific and has been adopted to their specific area of analysis. In this thesis the definition of epistemic communities will have to be adjusted to the situation of open source software programming. Doing this introduces some methodological issues, which need to be discussed. When defining epistemic communities in open source software development a filter is made, which will be used for categorising the observations in a later thesis. Making a wrong definition could mean that the right groups or epistemic communities will be excluded in the analysis. However, there is a tacit understanding of which groups should be included in the definition. These groups consist of people developing open source software, and ideally the various open source software projects should fall within the definition of an epistemic community. It is expected that these groups differ in several respects such as their area of interest in programming. Some groups may think that memory management is the only thing that matters, and others may feel that file systems are very special, and still others have different takes on what is interesting. The groups are also expected to differ in the way they communicate and use and produced knowledge. Still it is believed that there are common traits which will make it possible to make and use a common definition of the epistemic communities in open source software development.

On one side – the theoretical side – a definition is made which is intended to match the other side – the empirical side. These two sides should make a perfect match and demonstrate a correlation between theory and the real world. It is, however, very possible that these two sides do not make a perfect match or match at all. Does this mean that the theoretical work is a failure? No, on the contrary! In this situation the theoretical definition will serve as a filter, not with the purpose of seeing the matches but to identify what the differences are. In turn this will require the theoretical definition to be adjusted, if not completely rewritten, and the theory related to that very definition will have to be adjusted as well. A wrong definition means that a thorough discussion of the findings and differences will have to be made. Predictions made from a theory relying on the wrong definition will be shattered, since the

[37] Hass, P. M. 1989, p. 3.

logic on which the predictions where made no longer hold true. Depending on the severity of the difference between the theoretical definition and the empirical, evidence the theoretical predictions can in best case be adjusted, and in a worst case they will have to be rejected.

The purpose of a theoretical definition of the epistemic communities of open source software development is to make clear what traits are expected to be found when analysing open source software projects. This is also important in the light of the methodology used in the thesis, which is hypothetical deductive. As such the definition serves as a first step in the deduction of what type of behaviour one should expect from these epistemic communities. The theory of epistemic communities couples with a definition of epistemic communities, and some situational parameters – not yet defined – should make it possible to deduce a hypothesis about the nature of epistemic communities in open source software development. There shall be little doubt that using the hypothetical deductive methodology in an area of research as new as this, is a dangerous path to follow. It does, however, make it possible to state some hypothesis based on theory that may be either confirmed or falsified. Whatever the case a thorough discussion of conclusions and interpretations has to be conducted. Especially the point where empirical evidence is interpreted or translated into the theoretical framework is sensitive.

Before a definition is made, it is necessary to outline the premises for the definition. By now it should be clear to the reader that participants in open source software development projects are thought to be part of an epistemic community. What is less clear is the exact definition of epistemic communities. This definition must be able to capture various groups of people, and each group should be understood on its own premises, i.e. as an epistemic community. The definition should be the common denominator of the different groups. At this point it is possible to list some of the features that seem to separate the open source software developing groups from other groups:

1. Their activities are directed towards using and developing open source software.
2. The resulting software is available to anybody, at no cost higher than shipping and handling.
3. Anyone is allowed to contribute to the project.
4. Anybody is allowed to use and modify the resulting software.
5. There is no immediate monetary gain from joining a open source software project.
6. There are no formal structures that allow a developer to direct the efforts of others.

The first bullet is to make sure that the activities of the group are directed at developing open source software. The second bullet is a consequence of the software being open source, the definition require that the software has to be available for anybody. Given that these

projects are open source and the source code is freely available, anyone may contribute to a project and is also allowed to do so. Whether the contribution is included in the project is another matter, and depends on the rules for contribution to the particular project. In a project like the Linux kernel there is a single maintainer (Linus Torvalds, the originator), who single-handedly decides what goes into the kernel project and what not. Also a consequence of being open source is bullet number 4. Bullet number 5 is different, because this states that projects are not commercial entities where wages are paid when joining the project or contributing to the code. In effect this bullet makes sure that people do not join a project to gain a quick profit. People may later, or as a consequence of their effort, get a paid job, but that is a secondary benefit from participating in open source software development. The 6th and last bullet states that projects are conducted on a voluntary basis, and that even though an informal hierarchy may exist, there are no formal structures to reward or punish developers for contributing or not contributing to the project. Developers may get criticised for their actions, but it is not possible to fire a developer or deduct the mistakes from their wages – there usually are no wages.

In open source software development epistemic communities are communities of people engaged in developing open source software. The epistemic element is embedded in the activity of developing software. Development of software as well as other development activities require knowledge and understanding of the problem at hand in order to develop a solution. As the word develop suggests, this activity often incorporates use of new knowledge, or that knowledge has to be generated to solve the problem. In this part of the definition there is a clear relation to the definition of Holzner & Marx, emphasising the elements of usage and production of knowledge.

Holzner and Marx are not specific when defining the organisational setting or the characteristics of the persons in the community. Haas on the other hand tries to be more precise in his definition, and states that epistemic communities are " …a network of professionals with recognised experience….". It seems that the epistemic community of open source software development is indeed a network in the sense that the developers are interconnected and thereby form a network. The definition of a network is not made explicit by Hass, and generally the term network is rather ambiguous. However, the term network seem to capture the image of collaborating developers using the internet as a medium to facilitate their distributed development efforts. The internet facilitates most kinds of communication and also provides transport for the software being developed. Even though Haas states that the network consists of professionals, he, as mentioned, downplays this very element and writes that it need not be professionals, but anyone with a strong claim to a body

of knowledge. A network is perhaps the best description of the organisational structure of open source software development. A network conveys the idea of dynamic elements and the network being in a constant flux of change.

Haas' description of the four elements seems to be appropriate to include in the definition of the open source software epistemic community. Open source software developers within an epistemic community share the same normative and principled beliefs, i.e. they value the same elements for instance the strong belief that software patents are not good, and that one should react to this. They share the same causal beliefs, whereas Haas ascribes this to a shared understanding of the causal relation between policy actions and descried outcome, this is not the understanding in this paper. In this setting the shared causal understanding relates to the developers' understanding of programming within their domain of expertise. There are a few buts in the shared causal understanding, since it is required that developers perceive the problems at hand in the same way. This is not always the case, since elements such as experience and education will affect ones causal understanding. However, the essential element in epistemic communities are use and production of knowledge. The exchange of ideas and knowledge will ensure that the epistemic community over time will converge in the way they perceive problems. The actual technical solution to a problem is not likely to converge, as there are many way to solve the same problem and a solution will be highly personal. However, programming style are likely to converge to increase readability of the code in general.

As Haas also points out, there is a shared notion of validity as to what is a good and what is a bad solution to a problem.

The last criteria is the common policy enterprise which, of course, is very specific to epistemic communities, which seek to influence policy. Open source software epistemic communities do not seek a common policy enterprise, they do, however, seek to solve a programming problem common to the participants of the community. Therefore this point will be interpreted, as there is a shared notion of what is a valid solution to a programming problem. It is not given that all members of the community will be able to make what is recognised as a good solution. The members are, however, able to recognise a good solution to a problem, when they see one. The notion of validity will change over time, as the epistemic community grows and accumulates knowledge, and then gained insight will influence on what is regarded as good or bad solutions to a problem.

3.3.2 Summing up the Definition of Epistemic Communities

Open source software epistemic communities are defined as communities of people engaged in open source software developing activities. People regard epistemic criteria such as using their knowledge and expanding the boundaries of their knowledge higher than other criteria when engaged in the community. It is not a necessity to produce actual source code to be part of the epistemic community, some members function as catalysts to others when discussing programming problems.

The actual trigger of when a person enters and becomes part of an epistemic community is not exact. People may enter the community by beginning to receive the discussions and programming results of the community. In this phase the person begins a process of socialisation into the community that may result in the person is becoming part of the epistemic community. Being part of an epistemic community requires two things: 1) The person begins to take an active role in the community, in reality this means that the person begins to make contributions to the community in the form of either source code or participating in discussions in the epistemic community, 2) The person must feel a sense of belonging or even identification towards the epistemic community, and this happens when there is a correlation between the persons' interests and the purpose of the community. This is also a consequence of a situated learning process in the community, which is a consequence of the exchange of knowledge and the software development process. It is argued that learning and identity is closely related[38], and that one can not take place without the other. Following this logic, active participation in the development of software in the community is in effect a process of socialising, and learning from the epistemic community and using this knowledge will result in a feeling of identification with the epistemic community.

The organisational setting of an open source software epistemic community is a loosely coupled network, people are free to join and leave the epistemic community as they please.

The participants of an open source software epistemic community share the same normative and principled beliefs in relation to the epistemic community. This must be understood in a narrow sense, where the participants share normative and principled beliefs directed towards the efforts in the epistemic community. Whereas Haas seem to let a causal logic run from the normative and principled beliefs to the epistemic community, this paper claims that the causality runs the opposite way from the epistemic community to the normative and principled beliefs. It should be remembered that open source software

[38] Lave, J. & Wenger, E., 1991.

epistemic communities direct their efforts towards solving a software problem. When engaging in an epistemic community the person wishes to solve a programming problem, which means that person shares the same normative and principled belief that this is a problem worth solving. This (in Haas' words) is in effect the value based rationale for the actions of the community members. The participants share the same causal beliefs, in respect to the programming problem they wish to solve. Lastly, the participants share notions of what is a valid solution to a programming problem.

3.3.3 What to Expect from Epistemic Communities

Epistemic communities will provide an understanding of open source software development from the perspective of an epistemic community defined by the open source software development project. The epistemic communities' approach can identify sources of motivation, which stem from being part of a productive and knowledge generating group. The theory also provides criteria for identifying epistemic communities and describes the mechanisms which work in these communities. The criteria and the theory should be used as part of a hypothetical deductive methodology to test whether predictions based upon the theory can be confirmed empirically.

3.4 Legitimate Peripheral Participation

In the effort to find theory that could be used to understand the micro perspective various theories about learning, social learning, knowledge generation and knowledge communities have been surveyed. Leave and Wenger[39] with their interesting theory about situated learning and legitimate peripheral practice present what seems like a valid theory for explaining of the micro perspective. However, Leave and Wenger do not try to explain the motivation for entering into a learning situation, neither do they have any intention to do so. Motivation is assumed to be present, and the theory is used to analyse different learning situations and describe their differences. Examples used by Leave and Wenger include butcher apprentices in a supermarket, Xerox copy repairmen, Mayan midwives, and tailor apprentices in Bangladesh. The main conclusion that can be drawn from Leave and Wenger's book is that learning is highly situational. Learning takes place in a situation, where a learner or practitioner is engaged in the solution of a problem related to a thing, for instance repairing a Xerox machine. The point is that although knowledge can be thought of as a textbook, a real situation will encompass many subtle parameters, which are difficult if not impossible to convey using a text book. Further, in a learning situation where the apprentice

[39] Ibid.

is guided by a teacher, it must be legitimate for the apprentice to participate. The legitimate aspect comes into play in the day to day relationship between apprentice and master, where the apprentice must feel that his questions are welcome. The apprentice must be allowed to participate at his own skill level and to ask silly questions. In short the apprentice must be allowed a legitimate peripheral participation.

The focus on learning and communication is not part of the epistemic communities theory. Epistemic communities are defined as groups of people acting professionally and sharing the same normative beliefs. In essence it is assumed that the members of the community perceive problems the same way, and when processing problems reach the same conclusions. News groups, mailing lists, and other open source software development forums are clear examples of the fact that people perceive problems very differently and propose different solutions to the same problem. Thus, already before using the epistemic communities' approach, it is clear that there are some weaknesses with its narrow focussing on small communities.

Situated learning and legitimate peripheral participation is the remedy to this problem. By coupling situated learning, legitimate peripheral participation and epistemic communities it is possible to understand the development process, which is actually a learning process for the involved parties. In this perspective the involved parties are the epistemic community – the people who participate in the open source software development project. The learning process in open source software development takes place in two spheres 1) The private and 2) The collective sphere.

1) The Private Sphere

This is the development situation in the home of the individual developer. In this situation the developer tests and explores different solutions to problems which he wants to solve. The development is facilitated by a trial and error process in which the developer tests different solutions on his computer. Perhaps trial and error give the wrong impression, it is rather a recursive process where different principles are implemented in order to truly understand the problem and make the 'right' solution.

2) The Collective Sphere

This is where the developer interacts with the epistemic community using mailing lists, chat, news groups, and what not. Emails are sent with questions and answers. Many ideas are discussed in the collective sphere.

The two spheres are by no means separate and exist side by side. Often developers have a keen eye on what is happening on different mailing lists and in chat rooms when they are working. Developers Jens Axboe[40] and Kenneth Rohde Christensen[41] explained how they always, like many of their developer friends, had their email client and chat client running when working. The two developers noted that the community feeling were quite strong when logging on to a chat forum and they were greeted by friends.

Situated learning and legitimate peripheral participation should shed light on these two spheres. Beside the two spheres learning takes place on different levels, and is reflected by the different capabilities that developers possess. A newcomer to a project will often need a substantial amount of guidance to get started. Here the more experienced developers must allow the newcomer to participate legitimately and respect questions which have been asked a thousand times before[42]. Here knowledge flows from the master to the apprentice. Another situation is where developers discuss pros and cons of different implementations or ideas for implementations. Here no ready answer exists, and communication is part of the actual development process. The latter situation is a learning situation, but seems more like the understanding of development in epistemic communities.

3.4.1 Summing up Legitimate Peripheral Participation

It has been shown that epistemic communities might be lacking understanding of the learning process, which is an integral part of open source software development – or just development. To remedy this problem legitimate peripheral participation and situated learning have been introduced as a theory that should be integrated in the epistemic communities' approach.

[40] Jens Axboe is CD-ROM maintainer and employed by SuSE, Interview conducted the 11th of September 2000.

[41] Christiansen, Kenneth Rohde is a GNOME developer and language translator. Interview conducted the 21st of September 2000.

[42] The authors personal experience from participating in the game server and general technical Linux newsgroups.

4 Observed mechanisms

The two main bodies of theory have been presented, and we now change perspective and address some empirical observations which have yet to find a theoretical foundation. The observations in this sections are explained by use of heuristics, which are a plausible explanation to the observed phenomenon. These observed mechanisms are interesting due to the novelty of the open source software phenomenon and hopefully, though not in this paper, the observed mechanisms will be explained theoretically. However, part of studying a new empirical field is to describe observed phenomena and try to explain them.

This chapter presents a discussion of barriers to enter into open source software development. Next it is observed that open sources software unlike commercial software tend to reach a state of maturity. When this state is reached little or no further development is conducted on the software. In the next section, the peer review model which have been proposed some of the explanation of the success of open source software becomes heavily criticised. Next, an explanation as to why the maintainer is the central figure is provided. Next it is observed that open source software development allows people to collaborate without collaborating. And lastly a description of the different actors in open source software is proposed.

4.1 Barriers to entry

Barriers to entry describes the barriers to enter into open source software development either by starting a new project or by participating in an existing project. Two different types of barriers depend on the situation, starting a new project with no pre-existing code or elaborating on an existing project. These two types of barriers will be called 1) Upstart barriers and 2) Barriers to enter an existing project.

1) Upstart Barriers

Upstart barriers are barriers which must be overcome in order to start a new open source software project. It is important to stress that this barrier is strictly related to starting a new project. The barrier arises from the cost or time which one has to spend to start a new project. The time is not consumed by the organisational problems of setting up a new project but is rather related to the high intellectual costs associated with development of the first draft of the software. The first version is breaking new ground, and thus time has to be invested in designing structure and a proper foundation for the new program. The new software concept has to be analysed and broken down into functional entities which are suitable for programming. Later, when the structure of the program has begun to emerge, the program must be implemented which in it self is time consuming. Often the first version of a

program is discarded entirely, because the initial assumptions of how the program would function turned out to be wrong, and starting all over would be easier and faster than repairing an implementation which turned out to be broken[43]. Interviews with developers Poul Henning Kamp[44], Jens Axboe[45] and Kenneth Rohde Christensen[46] have confirmed that a proportionally large part of the intellectual effort is consumed in what may be called a design phase. As Poul Henning Kamp described, the process of development was one where the first period of time was spend trying to understand the problem, searching for previous examples of the problem and its solution. When a rough idea had emerged, it was quickly implemented as a draft to test its functionality (Implementation is usually made in the C programming language). What followed was a trial and error process, where different implementations were tested. Very often the implementation of the structure turned out to be insufficient. Poul Henning Kamp explained:

> "First, I build the foundation and then I build the house on the foundation.. sometimes it just turns out that the house was not right on the foundation….In these cases I treat it as a learning experience and start over again – now I know what went wrong and where to put things"[47]

In a distributed environment as the open source software community with no central allocation of resources, it is not possible to know whether another person is working on a similar problem. It is also difficult to know and find out who is interested and thus with whom one might discuss ones thoughts. When the preferred communications medium is a written medium like email and chat, where design of sketches and "back-of-the-envelop" drawings are not possible, it is a problem to enter into an illustrative and creative dialog. A design phase produces many fastly drawn sketches and rough ideas expressed in a illustrative language consuming many words and phrases to express what is not yet known. Verbal communication is much better suited for this type of interaction than is email or other types of written communication. More precisely, written communication may be used, but it is not efficient and may easily impede the creative process.

Given the size of the OSS community, there is a possibility that others might be working on a new project based on roughly the same idea. This is a further deterrent for

---

[43] Developer term for poorly functioning software.

[44] Kamp, Poul Henning, is a FreeBSD core team member, interview conducted 27th of September 2000.

[45] Axboe, Jens, interview conducted the 11th of September 2000.

[46] Christiansen, Kenneth Rohde, interview conducted the 21st of September 2000.

[47] Kamp,Poul Henning, interview conducted 27th of September 2000.

starting a project: Why waste the effort, if the software needed will be available in the future, and all one has to do is wait!

The possibility of later expansion of the program is dependent on a good initial structure, which makes later extensions and modifications easier. Maintainers of the GNOME[48] open source project have written an introduction to potential co-developers, who wish to contribute. An excerpt illustrates the point:

> "Cleanliness. Clean code is easy to read; this lets people read it with minimum effort so that they can understand it easily.
>
> Consistency. Consistent code makes it easy for people to understand how a program works; when reading consistent code, one subconsciously forms a number of assumptions and expectations about how the code works, so it is easier and safer to make modifications to it.
>
> Extensibility. General-purpose code is easier to reuse and modify than very specific code with lots of hardcoded assumptions. When someone wants to add a new feature to a program, it will obviously be easier to do so if the code was designed to be extensible from the beginning. Code that was not written this way may lead people into having to implement ugly hacks to add features.
>
> Correctness. Finally, code that is designed to be correct lets people spend less time worrying about bugs and more time enhancing the features of a program. Users also appreciate correct code, since nobody likes software that crashes."[49]

When a first draft is created and the program or software - depending on size and structure - is available to the public, the barriers to enter that particular project have been lowered substantially. At this point the structure of the software is laid down, and a working example may be tested and, if need be, extended. If the original developer follows the guidelines for good coding style, it is easier for others to understand the code and contribute to the development.

2) Barriers to Join an Existing Project

---

[48] GNOME is the GNU Network Object Model Environment. The GNOME project intends to build a complete, easy-to-use desktop environment for the user, and a powerful application framework for the software developer. See http://www.gnome.org

[49] Quintero, Federico M. et.al, 1999, accessed 26 April 2000.

With the advent of working code[50] the barriers change and become barriers to join an existing project, in short: Barriers to join. Barriers to join means that there is working code and an active project. In this situation the program will begin to evolve, as we are assume that people find the basic idea interesting and wish to contribute to the project. The maintainer may alone continue the development effort, but needless to say the effort of one person will produce less then that of many.

Having working code, crude it may be, provides insight of how it might be possible to solve the programming problem at hand. This significantly reduces the frustration of moving into uncharted programming territory, and also reduces the design phase overhead, which would otherwise have been needed. This lowers the barriers to join an existing project.

Projects are not the solution to all problems of open source software development, and the barriers to join do not continue to lower, as the project grows and co-developers contribute. The program will begin to grow, and with this growth it will be more and more difficult for newcomers and even co-developers, who have been in the project, to retain an overview of the project. When this happens the barriers to entry raise again, as the overhead required to understand the code becomes larger. One must also take into account that there is a relation between the need for the required feature and the effort that must be made to be able to contribute. If a person needs a feature and is willing to invest, say, 20 hours in the project and he must spend 19 hours just to make sense of a large fur ball of badly structured code, it is needless to say that the remaining 1 hour will probably not solve his probem. Interviews with both Poul Henning Kamp[51], Jens Axboe and Kenneth Rode Kristensen revealed that reading other peoples' code in order to understand a program, before the actual contribution can be made, was little fun but an investment that has to be made. The time needed to understand the code was highly dependent on the particular code and previous experience with similar structured code.

To keep the barriers to join from raising, the maintainer or co-developers will have to split the program into several smaller pieces that can be managed more easily. Splitting the code into smaller pieces is called modularization, as the code is split up in modules. There is more to modularization than just splitting up and organising. Modularization is about defining interfaces between the different parts of the code and organising the software into well thought-out functional entities. An interface defines the type of data and/or arguments passed into a module and returned to the calling program. Making interfaces and modules

[50] Hacker term describing a draft of some software that demonstrates the intended functionality.

[51] Kamp,Poul Henning, interview conducted 27th of September 2000..

facilitate further development and the making of new modules. Creators of new modules can read the interface specifications and learn about the data and arguments which his new module can expect to receive.

With modularization the barriers to entry are lowered again, and this will encourage potential co-developers to contribute. The step from the little program to a piece of software with several subprograms is substantial, and will often require some effort from the maintainer. As development continues barriers will be raising and lowering, depending on how and when new modules are created. To keep the barriers to entry low the maintainer has to keep the size of the modules manageable. It is all about reducing the area in which a potential co-developer will have to be proficient. The better the interfaces and smaller the modules, the less there is to understand, and subsequently it is more likely that potential co-developers become actual developers.

Like newcomers to a project have problems with understanding the code and the functioning of the program, the maintainer has a limited capability to understand contributions. Depending on the amount of time which a maintainer has committed to a project, the maintainer should focus on the size of contributions which he is willing to accept. Following the logic of keeping the investment in understanding a program low, the contributions to a project should be small and incremental in order for the maintainer and other co-develpers to quickly understand the changes. Readers of Kernel Traffic[52] will remember numerous posts by Linus Torvalds urging people to keep the patches small, incremental and well explained, or they would be rejected. This will also make it easy for co-developers to stay up to date with changes made to the software.

4.2 Open Source Software Becomes Mature

An interesting aspect of open source software development is the observation that open source software tends to reach a state of maturity. Maturity can be observed as a decline in development activity and a subsequent longer period of time between new releases. The declining development activity is also evident in the amount of traffic on the mailing lists related to the project. Most open source software projects rely on mailing lists as central means of communication between people who are interested in the project, including both users and developers. Often different mailing lists exist to accommodate different needs. The decline in development activity sometimes indicates that the software is not maintained for long periods of time. However, changes in the environment in which the software functions

[52] Kernel Traffic is a digest version of the development of the Linux Kernel. URL: http://kt.zork.net

may require users of the software to add features or make the software compliant with new standards.

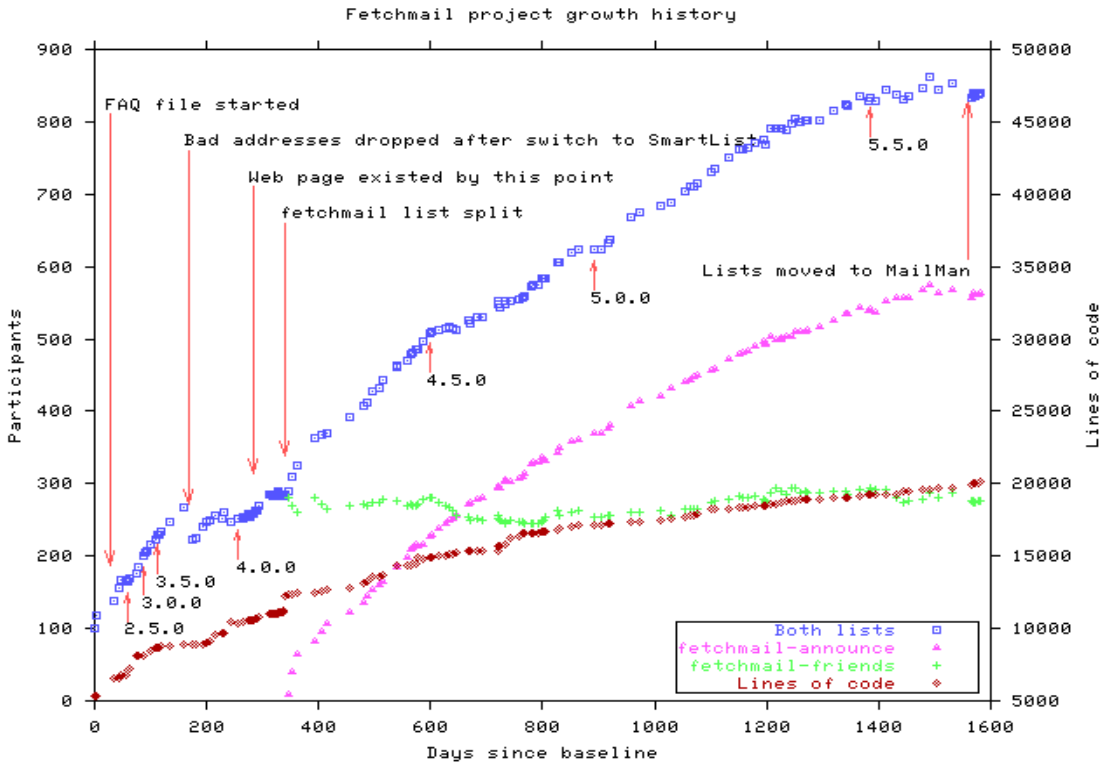

Figure 1: Trends in the Fetchmail projects' growth history.
[Source: http://www.tuxedo.org/~esr/fetchmail/history.html]

These observations have been documented in the Fetchmail[53] project, where data for the project activity have been published[55]. Assuming that lines of code is a rough equivalent to the amount of features in a program, it is revealing that the lines of code (brown marks in Figure 1) reach a stable level, and that the activity of the Fetchmail mailing lists also reaches a stable number. The green line in Figure 1 shows 'Fetchmail friends'. This list was the basis of the development effort and is an indication of interest in development. Both traffic on the 'Fetchmail friends' list and the lines of code show a clear indication of the project no longer growing and gaining features. There is still some activity indicating that bugs are being actively hunted and fixed. An enquiry to Eric S. Raymond about the growth trends in the projects produced the following answer:

[53] The Fetchmail development have been documented in: Raymond, Eric S. 1999.

[55] "Trends in the Fetchmail project's growth", accessed: March 05 2001.

> "I think the trend line for the "friends" list reflects contributionlevels pretty accurately. It's been basically flat since fetchmail came out of beta " [56]

When Fetchmail came out of beta[57], the program was considered complete in many respects, of course bugs needed to be fixed, but don't they always? It would have been possible for people to request additional features or to contribute new features themselves. The fact that the contribution level has been basically flat indicates very little need for new features or that it was not worth the effort to add new features.

A key difference between commercial software and open source software is that commercial software is revenue generating. Commercial software companies rely on the sale of their software as a source of income, and are thus dependant on a continuing revenue stream. Commercial software companies rely on intellectual property rights and the legal system to prevent illegal distribution. Modifications and extensions to their software are hindered by not allowing people access to the source code of their various products, and they are explicitly stated in the software license.

Software unlike other goods does not get worn, and does not have to be replaced within a certain time. The context in which the software is used may change and may subsequently require that corresponding changes are made to the software, or new hardware may force changes to be made. But otherwise software does not get worn out.

Historically wear and tear has provided a continuing revenue stream to companies. Software companies do not have this type of income, and must find other ways to keep the revenue stream. A company may choose between two strategies: 1) Further development of existing products, and 2) Creation and development of new products.

1) Further Development

Already having a product also means having a user base and potential customers who are likely to be interested in improved versions of their product. By following this strategy a company will have to offer improvements of the software to their customers – like new features. These features often have to visible, like for instance changes in the user interface. Microsoft's Office line of products are a perfect example of this, with each passing version of Office, many new features have been added to persuade customers to upgrade. Upgrading may be made at a reduced cost compared to buying the full new version, however, this strategy ensures Microsoft a substantial continuing revenue stream from upgrades alone.

---

[56] Raymond, Eric S., Email received the 15th May 2000.

[57] A beta version is a development/test version of the software. Coming out of beta means that the software is considered fit for production use and no obvious errors are present.

2) Creation and Development of New Products

Creating new products is a very uncertain course of action. A new product has to find its user base and prove itself in competition with other programs. This makes the option of labelling a product as mature and discontinue active development very unlikely. In commercial companies this would send a signal of a product ready for retirement thus lowering the products' value in terms of future profits. A company would have to find a replacement product that could "take over" the revenue stream from the mature product altogether, a very risky path. In effect, companies following a maturity strategy would be actively shortening the life cycle of their products, and this is not a logic strategy for companies expected to be profit maximising.

Open source software on the other hand uses a licensing scheme, which allows anybody to distribute copies at their own leisure. The profit prospects from selling software, which people are allowed to freely distribute are small, if not none existing. Making money from open source relies on added value to the software. Value is added by supplying easy access to updates, by packaging and bundling software, and by making it easily accessible to the customers. However, there is little economic perspective in producing open source software and selling the software in itself. Knowledge, however, and services related to knowing is a way of generating profits from open source software development. Those who have created the software have a deep understanding of the software and can easily act as consultants for other people or companies requiring services related to use of the software.

Not relying on software as a product that has to generate a continuing revenue stream leaves little incentive for the developers to continue development of the software and add features, when the wanted features are already present.

4.3 The Theory of Reversed Peer-Review

It has often been hailed that open source software development resembles production of knowledge in scientific communities[58]. The central argument is that peer review is a central mechanism in both the open source and scientific community. The scientific peer review happens when a scientist publishes a paper or presents a paper in some kind of forum, usually a conference or a workgroup. The paper is presented to the scientist's peer, who in turn comment on the paper. The result is a better paper, things which are not clear, or weak arguments are exposed, and the scientist may on the basis of these comments improve his paper. In essence a scientist receives helpful insight and comments regarding his product.

[58] Raymond, Eric S, 1999.

Raymond[59] attributes the high quality of open source software to peer review. The idea is that when a program and its source code is publicly available, many people will look at the source code and offer comments and criticism.

A closer look at the actual process and motivation does, however, tell a different story of something which might be called reversed peer review. As noted, the maintainer is the person who maintains a program or a piece of software depending on size, here referred to as a program. The maintainer is responsible for releasing new versions of the program, and as such all changes (referred to as a patch) to the program must go through the maintainer to be merged into the program for subsequent release. Basically there are two types of patches i.e. bugfixes and changes in features. A bugfix is a patch intended to fix certain problems in the program without removing or introducing new features. A patch with new features introduces new capabilities to the program, a graphics program, for example, may need new features in order to be able to read files from a competing graphics program. This would be a typical patch introducing new features.

Returning to the reversed peer review, the maintainer receives the patches from people who have used the program and felt the need to fix a bug, which was a problem to them or to introduce a new feature, which they needed. The maintainer feels responsible for the program and will not include anything in 'his' program, and thus the maintainer will closely review the patches received before including them in the program. A maintainer may well choose to reject a patch which does not meet the maintainer's expectations regarding quality or the direction, which the development may take when including new features. Often, when a maintainer rejects a patch, he will return the patch along with a description of why he chose to reject the patch. The maintainer is interested in making 'his' program grow and is thus interested in attracting co-developers, who will help him maintain and further develop the program. This is also the reason for the maintainer to spend time explaining co-developers why a patch is rejected. It is in the maintainer's interest to make the potential co-developer improve his patch and re-send it to the maintainer.

In this description the direction of the peer review is the opposite of peer review in the scientific community, the author reviews the changes, which his peer suggest to his paper. Therefor it may be said that open source software development relies on reversed peer review as a mechanism for quality assurance.

[59] Ibid.

There is, however, some truth to the observations of Raymond 1999, as it is true that users will submit bug reports and in that sense review the first draft of the program. Users who take interest and responsibility in the program and become co-developers may send patches containing bugfixes to the maintainer. This process may at a first glance appear to be peer review, but it is not. Discussing how bugs are discovered with CD-ROM maintainer Jens Axboe it is appears clearly bugs are discovered not because people like[60] to read the source code for a program, before they use it, and nor do they, but because the bug causes the program to malfunction. When a program malfunctions, and one has the means to fix it, most people cannot resist the urge to fix the problem and get on with their work. This process is not at all like peer review, self defence seems like a more appropriate term.

4.4 Why the Maintainer is a Central Figure

When analysing open source software development is apparent that all projects are organised around a central figure – the maintainer. Given the fact that every person could create his own project and is free to elaborate on an existing project or even create a new project based on another project it is interesting that projects do get formed and attract co-developers. As mentioned in the introduction an open source software license allows anybody to freely distribute the software and modify and elaborate on the software in anyway he feels. Still open source software is organised around a maintainer, so why is the maintainer the central figure in open source software development?

It has been proposed[61] that the maintainer of an open source software project is a very central figure, who resembles a leader in many ways. As a leader the maintainers' tasks are to: Provide a vision, Divide the project into smaller and well-defined tasks (modules), Attract programmers, and Keep the project together. For these reasons the maintainer is very central, since most decisions would need involvement from the leader. This proposition has been criticised[62], and an alternate understanding of why the maintainer is a central figure is proposed.

Three reasons are offered all of which seem to work together (at the same time) in a project. The three reasons are not judged by their relative importance.

---

[60] Of course some people like to do this, and some companies require that their software is scanned for malicious code before put into production.

[61] Lerner and Tirole , 2000, p. 21.

[62] Edwards, Kasper 2000b.

4.4.1 The Legal Perspective

The person, who writes the first lines of code to a program or a piece of software, is called the originator. The originator writes the first lines of code, and gives his infant program a name. The originator automatically receives copyright to his creation – the lines of code, and the name he has chosen. Although the originator by applying the GPL license surrenders some of his rights to the licensees of his program, he still retains some central rights. The originator and the maintainer may be one and the same person but need not to be. If the originator loose interest in the project, he may pass on the duty of being maintainer to someone which is eager to further develop the project. In this paper the maintainer and originator is defined as one and the same person, and the situation of transferring maintainership is not discussed.

The maintainer (being the originator) has the legal right to the name of the software and with this right he may choose what should be called by that name. As a maintainer he receives all contributions from co-developers, who have elaborated on the software. The maintainer decides which contributions should be included in the software. The power to decide what goes into the software is essential, because the features and bug fixes which are contributed, will be associated with the name, of the software. It is like a brand name where customers associate many properties like quality and ease of use with the brand name, and this gives the product an advantage over non-brand name products.

Should someone decide to start their own project based on the code from a known project, they will have a hard time getting a user base. Most users are familiar with the existing software and know its strengths and weaknesses. Not being able to gather a user base also influences on the development of the software in general, since many co-developers are found in the user base. Therefor the smaller user base will be an impediment to the new software due to the unknown name.

4.4.2 Co-ordination Service

From an economic point of view the maintainer provides a co-ordinating service for the participators of the project. The maintainer is the person, which merges patches from all co-developers, and often the maintainer is a actively engaged in the development. This gives the maintainer an intricate knowledge of what parts of the code are being actively developed. If someone decides to go ahead and develop some feature, he should check with the maintainer to see if someone else is working in that area. If this is the case the maintainer will ask the two co-developers to co-ordinate their efforts and if possible co-operate on the task at hand. The good maintainer will ensure that development of the software is a smooth, ongoing project and ease up the tensions between disagreeing co-developers.

The power to co-ordinate rely primarily on the legal perspective, where the maintainer is the only person, who is allowed to distribute under that very name. Co-developers may choose to let the maintainer co-ordinate or to form their own project, and as described in the previous section there are often advantages connected to staying with the project.

4.4.3 The Practicalities' Perspective

Being a maintainer is also about the nitty-gritty of arranging the files in a logical structure, answering questions about the software, and helping users getting started. The maintainer should as well be actively engaged in keeping the project together and maintaining the projects' communications channels. Often this has to do with maintaining a web page for the project and keeping the web page up to date. Alongside the web page are the mailing lists and perhaps organising locations, from where the software may be downloaded.

Many of these tasks involve provision of an infrastructure for the project, this could easily be compared to regular housekeeping – not the main focus, but it has to be done! When performing these tasks the maintainer has to decide many details of the project, simply because the maintainer answers questions, and writes web pages and documentation. By doing the housekeeping the maintainer will be identified with the project and thus become an authority in the project.

4.5 Collaborating Without Collaborating

A special aspect of open source development is the possibility for people to collaborate on the same project, but striving to achive different goals, whereas the usual understanding of collaboration is one of people working together trying to achive the same goal. In a manufacturing company setting this translates into people doing their part of the tasks, which have to be done for a product to be manufactured. As explained in Adam Smith‘s example of pin producers there are tremendous gains to be achived from the division of labour. However, the pin example may not be a suitable analogy, since it demonstrated gains from organising the work process in a sequence of sub-processes, which were performed sequentially. Further, the process of developing software is very different from that of industrial production. Industrial production is characterised by repetition of the same tasks over and over again. Conversely, software development is, as the name suggests, a development activity, and rarely the exact same tasks are repeated. The same programming techniques may be used to solve different problems in different settings. Studies of programming have

revealed that only 1/6 of the programming effort is actual coding[63]. Much time is devoted to understanding the problem and researching possible solutions.

As explained in section 4.1 Barriers to entry, there are advantages to structuring software in modules, which are easily maintainable, and new modules may easily be added. When people are collaborating without collaborating they are in essence contributing to a project, which they view as a vehicle to host some particular feature. This may be viewed as scavenging or taking advantage of an existing project. But in fact it is quite the contrary, there is a synergy to be achieved.

The effort to implement a desired feature is much lower, if supporting infrastructure cat be found in a host program. The host program also benefits from contributions of new functionality. A new feature will extend the usability of the software (host + contributed feature) and make the software even more attractive to users. Some of the new users will take an interest in developing the software, and this will further improve the software. Contributors of modules (features) are likely to engage in development of the host program to ensure that their feature or module is performing well and receives priority, when development choices have to be made. It is in everybody's best interest to attract users and developers, who will help to identify problems and correct them. Modularization and/or well structured software is imperative when collaborating without collaborating. Developers must be sure that when they embark on a programming effort, the interfaces between host program and modules or between main- and sub-programs remain static for a period of time. In this period developers may work on their feature.

[63] Brooks, F., 1995 p. 20.

5 Methodology

In this paper the theoretical basis for the later thesis has been presented. The presented theory has yet to be related to the empirical reality, and this is where the methodological considerations come into play. Methodology provides the link between theory and empirical evidence by explaining how the two relate to each other. In this thesis it is believed that the hypothetical deductive method is well suited to test if the theory presented provides a valid explanation for empirical observations.

Having presented the theoretical basis the next step, when using the hypothetical deductive method, is to make empirical predictions based on the theory. When making theoretically founded predictions the presented theory is used in a strict logically manner. The aim is to make deductions based on theory, which states expectations to the empirical findings.

The procedure starts by defining the preconditions, which have to be true for the theory to be valid. The preconditions are related to the situations in which the theory is meant to apply. There is no point in using a theory outside its valid application area – no point in applying quantum physics to the movement of pool balls (Newtonian physics is better suited).

Next a number of situational parameters are identified. They are an interpretation or an initial understanding of the empirical field of study, which is to be examined later. The situational parameters are then passed to the theory as arguments to be processed. Using the logic and the explicit relations, which the theory states, the situational parameters are transformed to theoretically based predictions of what to expect, when conducting the empirical analysis.

The strengths of this method are the predictions, which may be falsified or confirmed in the empirical analysis.

The weakness of this model is that it is fairly strict in the sense that predictions not stated will not be found. In this situation either the theory or the situational parameters were wrong. This methodology is usually applied to areas of research, where the empirical field of study is far better known than is the case in this thesis. Knowing the empirical fields of study better, means that the predictions will be more precise, and thus it is possible to test the theory in greater details.

A crucial point in this methodology is the definition of the situational parameters. Should it be that these parameters do not at all match the empirical field of study, then is it is

very unlikely that the predictions based on these faulty parameters are entirely wrong. The task will then be to try to correct the predictions and, make the deductions to see if the theory has at least some validity, or if it should be rejected altogether.

However, this methodology does provide a chance to actually test the theory, instead of reverse engineering the empirical evidence.

6 Conclusion

This paper has introduced software in general and open source software in particular and has explained how the licenses of the software have a profound impact on the type of good, which the software is. Open source licenses require that anybody has access to the source code and is allowed to freely distribute the software. Software allows for unlimited reproduction at little or no cost. This makes open source software non-excludable and non-rival in consumption, which is the definition of a pure public good. It has also been described that open source software is made by private persons and/or private companies, and thus open source software is a privately produced, pure public good. This last statement counters the established beliefs that pure public goods are either under-supplied or subject to government intervention. This paper does not offer any solution to the problem, but points to economic theory of pure public goods as the right direction in which to proceed when trying to understand open source software development.

Leaving the economic frame of reference, epistemic communities are proposed as a theoretical perspective in which the group dynamics of open source software development should be understood. Open source software development projects are shown to have a close resemblance to epistemic communities. However, being a development activity, there is a high degree of learning in open source software projects, which is not easily explained with the epistemic communities' approach. To this end, situated learning and peripheral legitimate participation are coupled with epistemic communities in order to explain the learning processes. Following the presentation of the two main bodies of theory, focus is switched to characterise six observations or mechanisms, which are present in open source software development.

The barriers for individuals to enter into open source software development are explained as two different kinds: 1) Upstart barriers, which are high and characterised by a large initial design overhead. 2) Barriers to join an existing project, which vary depending on size and structure of the code base present in the project. Modularization is mentioned as one way of keeping individual parts of the software separate and to keep the part small and easy to maintain, and thus barriers low.

Open source software reaches a state of maturity unlike its commercial counterparts. There is little profit to be gained from open source software, and thus little incentive to continuously add new features to try to sell the software.

The quality of open source software has often been attributed to a scientific peer-review process, which is believed to be present. It is argued that this belief is wrong, and when there is a review process, it is a reversed peer-review.

In open source software development the maintainer is a central figure. This is explained by three observations: 1) The maintainer has the legal rights to the name of the project. 2) The maintainer is performing a co-ordination service for the project. 3) The maintainer performs the housekeeping of the project.

Open source software allows people to collaborate without collaborating. Some programmers may enter into a project with the narrow perspective of implementing a needed feature or function. Instead of starting a new project the programmer enters an existing project and uses this as a vehicle to implement his feature, and often there is a spin off to the 'vehicle' project, when the programmer is forced to make improvements to accommodate his own features.

7 References

Axboe, Jens, CD -ROM maintainer and employed by SuSE, Interview conducted the 11th of September 2000.

Buchart, H. and J. Marx, Eds. 1979, "Knowledge affiliation: the Knowledge system in society" Allyn and Bacon, Boston, MA.

Brooks, F. P., 1995 "The Mythical Man Month", Addison-Wesley.

Christiansen, Kenneth Rohde is a GNOME developer and language translator. Interview conducted the 21st of September 2000.

Crones, Richard & Sandler, Todd, 1996, "The Theory of Externalities, Public Goods and Club Goods", Cambridge University Press.

Edwards, Kasper 2000a working paper "The history of Unix development" URL: http://www.its.dtu.dk/ansat/ke/emp_work.pdf

Edwards, Kasper 2000b: When Beggars Become Choosers. *First Monday*, volume 5, number 10 (October 2000), URL: http://firstmonday.org/issues/issue5_10/edwards/index.html

Haas, P. M., 1989 "Do regimes matter? Epistemic communities in and the Mediterranean pollution control", International Organisation 43, 3.

Haas, P. M., 1992. "Introduction: Epistemic Communities and international policy coordination", International Organization 46, 1.

Holzner, B. and J. Marx, 1979, "Knowledge affiliation: the Knowledge system in society" Allyn and Bacon, Boston, MA

Kamp, Poul Henning, interview conducted 27th of September 2000.Kaul et. al. in Kaul, Inge et. al. (eds.), 2000 " Global Public Goods – international cooperation in the 21st century", Oxford University Press, New York.

Kernel Traffic. [web page] http://kt.zork.net weekly editions.

Knorr-Cetina, Karin, 1999 "Epistemic Cultures – How the sciences make knowledge" Harvard University Press

Koktvedgaard, Mogens, 1994, "Lærebog i immaterialret", Jurist og Økonomforbundets forlag.

Lave, J. & Wenger, E., 1991, "Sitauted Learning - Legitimate periphal participation", Cambridge Uni Press.

Learner, Josh and Tirole, Jean, March 2000, "The Simple Economics of Open Source", Working Paper 7600, National Bureau of Economic research, 1050 Massachusetts Avenue, Cambridge, MA 02138.

Linux.org [webpage] http://www.linux.org/projects/index.html [accessed January 18th 2001].

Netcraft Web Server Survey [Webpage] http://www.netcraft.net/survey/, [accessed January 4th 2001].

Perens, Bruce, 1999 "The Open Source Definition" in "Open Sources", eds. DiBona, Ockman and Stone; O'Reilly.

Quintero, Federico M. et.al, 1999 "GNOME Programming Guidelines". [Web page] http://developer.gnome.org/doc/guides/programming-guidelines/index.html#INTRO, [accessed 26 April 2000]

Raymond, Eric S. [Web page] "Trends in the Fetchmail project's growth" [Accessed: March 05 2001] URL: http://www.tuxedo.org/~esr/fetchmail/history.html

Raymond, Eric S. Email received the 15th May 2000.

Raymond, Eric S. 1999, "The Cathedral and Bazaar", in The Cathedral & the Bazaar, O'Reilly and Associates.

Raymond, Eric S. 1999, "Homesteading the Noosphere", in The Cathedral & the Bazaar, O'Reilly and Associates.

Salus, Peter 1994 "A Quarter Century of Unix", Addison-Wesley Publishing, Inc.

Shapiro, Carl & Varian, Hal R., 1999 "Information Rules – A Strategic Guide to the Network Economy", Harward Business School Press, Boston Massacusetts.

SourceForge, main page [webpage] http://www.sourceforge.net [accessed January 18th 2001].

Stiglitz, J. in in Kaul, Inge et. al. (eds.), 2000 " Global Public Goods – international cooperation in the 21st century", Oxford University Press, New York.

Stiglitz, J.E. ,1996, "Economics", Norton.

The Linux Study, [web page], http://www.psychologie.uni-kiel.de/linux-study/index.html

The Open Source Definition, The Open Source Initiative [web page] http://www.opensource.dk/docs/definition.html [accessed: 21. March 01]