UNDERSTANDING OPEN SOURCE COMMUNITIES

An organizational perspective

Ruben van Wendel de Joode

UNDERSTANDING OPEN SOURCE COMMUNITIES

An organizational perspective

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op maandag 26 september 2005 om 13:00 uur

door Ruben VAN WENDEL DE JOODE
doctorandus in de bedrijfseconomie

geboren te Winterswijk.

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. mr. J.A. de Bruijn

ISBN: 9056381385

Printing and binding: Febodruk BV, Enschede

PREFACE AND ACKNOWLEDGEMENTS

In 2001 my daily supervisor showed me an article about a special kind of Internet community. The name of the community was Linux. According to the article, a group of volunteers met on the Internet and held discussions online to create highly complex software. To my mind that was strange; why would people collaborate voluntarily? What is in it for them? After reading more articles, having numerous discussions with colleagues and monitoring some of the Internet communication between the volunteers, it struck me that a more fascinating aspect of Linux is the way the group is organized. How is it possible that volunteers from all over the world, who might never see each other in real life and who have different backgrounds and interests, are able to create complex software? This question remained in my mind throughout this research.

It was also in my mind when I did my first interview. My first interviewee seemed the stereotypical Linux developer: long gray hair, a long beard and highly intelligent. Somewhat hesitantly, I explained to him (most open source developers are indeed men) that I was interested in the organization of communities like Linux. He believed this to be a simple question. It would hardly need detailed research. He said, "Linux is anarchy! Everybody does what he feels like doing." Obviously, this answer raised even more questions: How can such behavior result in software? Who makes the decisions? How are conflicts resolved? Again, he explained, the answers were rather simple: Linus Torvalds is the project leader and he usually makes the right decisions. Perhaps without knowing it, my first respondent had introduced me to the two dominant scientific explanations as to how a community like Linux is organized. At the very least he convinced me that Linux and other 'open source communities', which is the more general term for Linux-like communities, would be both an interesting as well as a difficult object of analysis.

My first interviewee explained to me one other thing about open source as well. He said I had to understand that people in open source communities like to share their knowledge, ideas and even their inventions. This, according to him, is one of the basic principles in the communities. And indeed, research shows that people who participate in open source communities frequently do so because they enjoy sharing their knowledge and they hope to learn from the knowledge they receive from others. Together participants are able to achieve great things: they develop software programs that are surprisingly reliable and used by many individuals, corporations and governmental organizations worldwide.

One of the most important lessons I learned from talking to the many open source developers and enthusiasts is that you cannot write a complex software program without the help of others. These other people should not be confined to software programmers with similar skills and interests. On the contrary you need people who have different software development skills *and* even people who lack such skills altogether. Each performs a part in the quest to together improve the quality of the software.

To share information, ideas and knowledge is not only important in the creation of software. It is at least as important to openly share ideas and to use the ideas of others when writing a dissertation. In the process of writing this dissertation I have had the help of many people who had very different ideas about open source communities and about my work. In hindsight I can safely say that all of them had an important role in the completion of this work. Their information, comments, suggestions and ideas deserve a special word of thanks.

In the course of the research I spoke with many people who in some way were actively involved in open source communities. I thank the people who spent time talking with me, provided me a look inside the communities and showed me why open source communities are truly fascinating and fun. Special thanks go to Andries Brouwer, Jo Lahaye, Dirk-Willem van Gulik, Ray Dassen, Guido van Rossum and John 'Maddog' Hall.

Open source communities have attracted the attention of academic researchers. Some of them had valuable input in the making of this dissertation. Here thanks are due to Gaby Rasters, Bart Knubben, Biella Coleman, Karim Lakhani, Rob Peters, Yuwei Lin and Shay David. I especially thank Sebastian Spaeth and Walt Scacchi for their comments on a previous version of the concluding chapter.

Many people played a part in transferring my findings into an actual dissertation. Here I thank my colleagues from the Department of Policy, Organization & Management. They created an inspiring and fun environment to work in. Martijn Kuit, thanks for advising me to do PhD research. Thank you Ellen for being a fun roommate. A special word of thanks also goes to fellow PhD student Mark for listening when I flooded him yet again with ideas and frustrations about the dissertation. And of course thanks for the many lively discussions about our favorite soccer club! Michel and Hans, I thank you for your patience in reading my texts, providing me with useful comments and suggestions, and most importantly for letting me make all the usual mistakes. Thanks to Charlotte Hess, Erik-Hans Klijn and Victor Bekkers who helped me with ideas and suggestions to improve my work. I thank my peers at the TPM faculty; Alexander, Heleen, Leon, Linda, Maura, Mirjam and Sonja for their many comments about previous versions of this work. Coaches Cees van Beers, Pieter Bots and Pieter Vermaas, thank you for your comments and good ideas. The people from the Betade group thank you for the lively discussions on a wide variety of topics. Thank you Tineke Egyedi for being such a pleasure to write together with. Michelle Luijben, thank you for your work editing previous versions of the manuscript.

Finally, I am much indebted to my friends and family. Some deserve a special word of thanks. Thanks Alexander for assisting me in the formal ceremony and Rian for always listening to my tedious stories about the research, especially in the first years of my doctoral study. I thank Jeroen for always widening my horizon with new ideas and management concepts. Special thanks also go to Sander for doing such great work during and after the interviews we conducted in the United States. I thank both of my parents. You are the best teachers I have had. Thank you for helping me to understand what the important things in life are and what it means to have a *home*. Yeun Mee, thank you for believing in me and trusting me. You give me great confidence. Without you life would be much less exciting and fun!

TABLE OF CONTENTS

CHAPTER ONE

INTRODUCTION

Open source communities are groups of sometimes hundreds if not thousands of individuals with different interests, backgrounds and motives. Many of these people are volunteers who are not paid to participate in the communities. Furthermore, many never get to see each other in real life. They meet virtually, on the Internet. Yet they are able to collectively develop software that is highly complex, that has proven to work and that is viewed as a viable alternative. The question that initially triggered this research is 'how?'

Open source communities in a nutshell

Openness of the source code

An increasing number of software programs are being developed in open source communities. The software created in these communities is known under a wide variety of names, of which 'open source software' and 'free software' are probably the most common.[1] The communities consist of hobbyists, programmers employed by companies, students, freelancers and computer-illiterates, all of whom contribute various amounts of their time and labor to the software's development and maintenance. In the communities, the source code of software is not treated as a secret. Instead, programmers agree that both the software and the corresponding source code should be open, visible, downloadable and modifiable for anyone who is interested (Von Krogh & Von Hippel 2003). The software and the corresponding source code are thus said to be in the *commons* (Benkler 2002a, Bollier 2001b, Boyle 2003, Bruns 2000, McGowan 2001).

To explain the idea of the commons, Bollier (2002) draws a parallel between open source communities and the rise of community gardens in New York City. He describes how "a group of self-styled green guerillas" (p. 16) started to plant flowers and trees on sites that the market had abandoned because they were presumably worthless. As a result of these efforts, 800 community gardens sprung up in various places in New York City. They became a central part of the lives of the people who lived near these formerly abandoned sites. The neighborhood residents themselves would and still do maintain these gardens. They are volunteers who receive no money for their maintenance work and they do "not treat the sites as interchangeable units of land" (p. 17). Rather, they treat the gardens and their efforts in maintaining the gardens as a commons, as something used, maintained and shared by a community of volunteers.

This is also what occurs in open source communities. Individuals share their work; their inventions and contributions, with others in the community.

A strange organization

The belief that the source code of software is something that should be open and publicly available differentiates open source communities from most companies in which software is developed (Bollier 2001a). Another difference is the organizational structure of open source communities. At a first glance the communities' organizational structure seems strange, since many of the 'usual' mechanisms that coordinate individual efforts appear to be absent. For instance, individuals who participate in the communities are not located at one physical location. Participants are spread across the globe and may not even know what their colleague developers look like, although they might frequently interact. This is because many never physically meet (e.g. Osterloh et al 2003a).

Open source communities are said to lack individuals or institutions in charge of a master plan. There is no one to decide in which direction a community as a whole should move. "Central control to make long-term 'strategic plans' and decisions is absent in the Linux community" (Van Wendel de Joode & Kemp 2002, p. 522). Similarly, Kuwabara (2000) writes, "The Linux project has neither top-down planning nor a central body vested with binding and enforcing authorities. Its power, the source of its bubbling creativity, is instead in the ceaseless interactivity among its developers."[2]

Open source communities lack labor contracts or more general contractual relationships that tell participants what to do and how to do it (Franck & Jungwirth 2003). Instead, the majority of open source programmers are volunteers (Hertel et al 2003). They decide what they want to work on and how they want to do it. "Work is not assigned; people undertake the work they choose to undertake" (Mockus et al 2002, p. 310).

Open source communities are frequently reported to lack clearly defined organizational boundaries (Fielding 1999, Raymond 1999b). There is no consensus about who or what is inside a community and what is outside. "Membership in the community is fluid; current members can leave the community and new members can also join at any time" (Sharma et al 2002, p. 10).

The presence of internal pressures

The absence of the abovementioned mechanisms triggers questions about how these communities are organized. These questions become all the more interesting when we realize that the organization of open source communities faces a number of pressures. First, pressure is caused by the fact that the source code of open source software is open and freely available to anyone who wants to download and install it. This would suggest an absence of incentive to invest time and effort in the development of open source software, since anyone can enjoy the benefits of the good for free. Everyone is tempted to free ride (Olson 1965). Open source communities thus face a potential scenario in which many users free ride, which jeopardizes the continuity of software development and improvement within the communities (e.g. Becking et al 2005, Fitzgerald & Kenny 2004).

Second, pressures are caused by the threat of incompatibility. Any programmer in the open source community can adapt and improve the source code of the software to meet their own specific needs and wishes. Combined with the fact that open source communities have no clearly defined boundaries, which enables anyone to enter a community and write source code, there is a great deal of opportunity to diverge. Theoretically, the opportunity to create divergent lines of the software could result in fragmentation and give rise to problems of incompatibility (Axelrod & Cohen 1999, Egyedi & Van Wendel de Joode 2004).

Third, pressure emanates from the numerous conflicts that confront open source communities (Elliot & Scacchi 2002, McCormick 2003, Van Wendel de Joode 2004b). As mentioned earlier, the communities consist of a wide diversity of participants; people with different backgrounds, nationalities and interests. The diversity among participants is one of the reasons why conflicts emerge. The presence of conflicts constitutes an internal pressure, which could result in inertia, fragmentation and worse (Jehn 1995, Jehn & Mannix 2001) and hence threaten the continued development and maintenance of the software (see for instance Mannix et al 2002).

The presence of external pressures

Next to internal pressures, open source communities face external pressures. These arise because the communities are intertwined with the software market. This means they are confronted with software companies that want to earn a profit. Furthermore, they face patent and copyright regimes. The goal of intellectual property rights (IPR), like copyright and patents, is to give inventors ownership rights on their technological advances (Nordhaus 1969). IPR gives inventors the right to exclude competitors from duplicating, reverse engineering and applying any innovation that is so protected (Arrow 1962). IPR also serves another goal, namely, to ensure the disclosure of knowledge to the public domain. It is a common notion that "knowledge generates further knowledge" (Benkler 2002b, Harison 2002). IPR is meant to make innovations known and thus stimulate the transfer of knowledge between companies and individuals and encourage further advances.

Open source communities do not rely on copyright or patents to protect software and source code. Yet the communities do deal with companies that use IPR. This has consequences for the open source communities. For example, a study showed that "283 software patents not yet reviewed by the courts could potentially be used to support claims of infringement against Linux."[3] The fact that companies own patents on pieces of the software could stifle the ability of open source communities to continue to develop and maintain software (Benkler 2002a, Boyle 2003, Kahin 2002, Vemuri & Bertone 2004).

Open source communities also face other external pressures. What, for instance, are the effects of companies' growing interest in getting involved in open source software? How do volunteers interact with and protect themselves from companies with commercial interests (Van Wendel de Joode 2004c, Van Wendel de Joode et al 2003)?

Open source software is successful

Based on the above – the source code is open and available and the organization of the communities faces both internal and external pressures – one would seem safe to assume that software developed in open source communities is unlikely to be successful. Empirical data, however, suggest differently.[4] Data show that some open source programs are quite dominant in today's software markets – though this is not the same as to claim that software developed in open source communities is necessarily better than proprietarily developed software. The quality of proprietary software and open source software differs for each individual software program, and the relationship between the quality of the software and the way in which it was developed is as yet little understood. The only claim made here is that in certain segments of the market open source software has gained a sizable share and thus has apparently reached a satisfactory level of quality, which in itself can be considered a surprise.

Essentially, this section introduces open source software as more than a hobby of a large number of volunteers worldwide. Instead, it has become an important organizational form for creating and maintaining software in today's software market. The value and apparent intelligence[5] of this organizational form are demonstrated by the success of a number of open source programs.

Some open source programs dominate their segments of the software market. The program Sendmail is a good example. Sendmail, an open source product, currently provides the standard method of routing e-mails on the Internet. In 1998, Sendmail was estimated to handle 80 percent of all e-mail traffic (Lerner & Tirole 2002b). Another example is Apache. The first layout of Apache was developed in 1994. It was soon picked up and currently the Apache HTTP Web server hosts 67 percent of all active websites on the Internet, much more than the 25 percent share held by Microsoft.[6] In the Netherlands the content management system MMBase is a good example. This open source software has been adopted by an increasing number of, primarily public, organizations, like schools,[7] municipalities[8] and public broadcasting networks.[9]

Increasing numbers of companies too are deciding to support their mission-critical business processes with open source software. One example is the New York Stock Exchange.[10] Another is *Amazon,* which is said to owe a large part of its success to open source software, since the Amazon.com website is built on it.[11] Or consider IBM. This company has invested billions of dollars in the development and marketing of open source software.[12] Furthermore, much of its hardware now automatically has open source software installed on it.

Worldwide many governments and municipalities have adopted open source software and recommend and sometimes even mandate the use of open source. In the United States, the City of Newport decided to radically shift its information technology policy. Its current intention is to use open source software.[13] In May 2003 the city council of Munich decided to switch 14,000 desktops away from Microsoft to open source software. Currently, the city is in the midst of the transition (Laan 2003).[14] In Brazil two small government agencies recently made the shift away from proprietary software to open source alternatives. Furthermore, IBM recently signed a letter of intent to boost the adoption of other open source programs.[15] In the Netherlands, open source has gained a prominent place on the political agenda since the publication of the report *Software Open U!* (2001). Written by two Dutch members of parliament the report advocates the use of open standards and recommends the adoption of open source software. The Dutch government has since acted on the report's conclusions, erecting the Open Standards and Open Source Software (OSOSS) program. OSOSS "encourages the use of open standards and informs about open source software."[16] Its website lists reference projects in local and national governments in which open source software has been adopted. Included in this list are, among others, open source initiatives in the Dutch municipalities of Groningen and The Hague.[17]

Certain open source programs are comparable in quality to proprietarily developed software as measured by the number of defects per 1,000 lines of source code. An example of a qualitatively comparable product is Apache, which is "found on par with commercial equivalents."[18] Research shows that Apache has 31 software defects in 58,944 lines of source code. This is equivalent to a defect density of 0.53 per 1,000 lines of source code. This is similar to the defect density of comparable proprietarily developed software programs, which have an average defect density of 0.51 per 1,000 lines of source code.[19] Another, even better example of quality in open source software is the operating system Linux. Research by The Reasoning consulting group compared six operating systems on their implementation of a key-networking component and concluded that the Linux kernel performed better than the five proprietarily developed operating systems. The study found that the open source operating system had "8 defects in 81,852 lines of Linux kernel source code."[20] This results in a defect density of 0.098 per 1,000 lines of source code.

Research question

The fact that open source software has received so much attention, is being adopted by so many businesses and governments, and is of comparable quality to proprietarily developed software, raises a variety of questions. How are volunteers possibly able to create such qualitatively good products? How do volunteers protect the software against companies that would like to use intellectual property rights to appropriate it? How are the contributions of such a large number of volunteers integrated? How are decisions made about which contributions to include and exclude? What about quality control of the individual contributions and the overall product? How do the volunteers cope with the wide variety of interests and goals that are likely to exist among them?

This research aims to provide answers to the above questions. All of the issues raised, however, can be boiled down into a single problem question, which is at the heart of this research:

How are open source communities organized and how do they sustain themselves?

To organize in the research question refers to the fact that the motivation of individuals in open source communities is not enough. As a later section of this introduction argues, most of the current research focuses on the question of why individuals are motivated to participate in the communities. Yet it must be made clear that to organize means more than just to getting individuals motivated; it also means 'to coordinate,' to arrange and order individuals' efforts. *To sustain* relates to the fact that the existence of open source communities is far from self-evident. Much research on open source communities appears to take their current and future presence as a given, as something that needs no further inquiry. While such research may focus on many relevant questions, it neglects the question of how individuals safeguard their community over longer periods of time.[21] How do the communities protect their continuity in light of internal and external pressures? This research aims to fill both gaps in current knowledge on open source communities.

Research on open source communities: the state of the art

An increasing amount of research is performed on open source communities. The state of the art is highly diverse and adopts a multitude of perspectives. This section maps a number of important strands of recent literature. They are: a) the motivation of individuals to contribute their time and effort, b) the use of metaphors to explain the organization of open source communities, and c) the role of collective mechanisms in open source communities.

The motivation of individuals to contribute their time and labor

One question that has been the focal point of much research on open source communities is what motivates programmers and users to participate in the development and maintenance of open source software (e.g. Hars & Ou 2002, Hertel et al 2003, Lakhani & Von Hippel 2003)? Recent research in this area is quantitative in nature and uses surveys to inquire into individuals' motivation to participate in open source communities. Two examples of such research are Hertel et al. (2003) and a project funded by the European Commission called Free/Libre and Open Source Software (FLOSS).[22] Hertel et al. (2003) focused specifically on the Linux kernel community. In a survey they explored the motives of 141 contributors. Perhaps their most important conclusion was that the participants' engagement is determined in particular by their identification as a Linux developer, by their desire to improve their own software and their tolerance to investing time (Hertel et al 2003, p. 1159). The FLOSS survey targeted both individual developers and companies and public institutions. A total of 2,784 developers completed the online survey. The study shows that most of the respondents are men and relatively young. They further had strong backgrounds in information and communication technologies and enjoyed relatively high levels of education (2002).

The first writings on open source communities suggest that volunteers participate for altruistic reasons (Wichmann 2002). Volunteers were said to want to help others in the group, to contribute to the group effort and to promote the community (Butler et al forthcoming). However, the focus of research on individual motivation has now shifted and is currently much more centered on an explanation based on rational and individual profit-seeking actors.

According to Weber (2004), empirical evidence suggests that altruism cannot be the primary motive explaining why individuals participate in open source communities. Weber refers to the credits list attached to open source products, which does not fit in with an explanation based on altruism (for a discussion on the role of the credit's file see also Raymond 2000). Indeed, empirical data supports the claim that many programmers participate to receive personal benefits. Some of the benefits identified are: a *reputation* (e.g. Dalle & Jullien 2003, Lakhani & Von Hippel 2003, O'Mahony 2003, Sharma et al 2002), *learning* and improving programming skills (e.g. Hertel et al 2003, Von Hippel & Von Krogh 2003, Lakhani & Wolf 2003), meeting a *personal need* with a software program that has a certain functionality (e.g. Edwards 2001, Hars & Ou 2002) and having *fun* (Lakhani & Von Hippel 2003, Torvalds & Diamond 2001).

Explaining the organization of open source communities; the use of metaphors

Many of the earlier writings on open source communities set out to provide a general understanding of the communities. These writings frequently adopted metaphors to help identify the characteristics of open source communities and to convey how they differ from other software development groups. There are many such metaphors. Examples are: open source communities as guerilla networks (Raymond 1999a), as religious movements and even as sects.[23] Some metaphors have found their way into more scientific writings about open source communities. One of the best examples is by Eric Raymond, who wrote the frequently-cited book *The Cathedral and the Bazaar* (1999b). In the book, the difference between software development in open source communities and in proprietary environments[24] is compared to the difference between processes in bazaars and cathedrals. The bazaar metaphor is addressed in more detail below. Other metaphors are the open source community as a 'gift economy,' as a 'scientific community' and as a 'self-organizing system.' These too are introduced below.

Open source communities as great babbling bazaars. Raymond (1999b) argued that open source communities consist of developers who have different agendas and approaches. Furthermore, centralized coordination is said to be absent. Individuals in the communities are claimed to behave like people in a bazaar. This is contrasted with the cathedral-like software development methods that are, according to Raymond, employed in software companies. In the latter, coordination is top-down, while in the former coordination arises spontaneously (Raymond 1999b). Individuals in the community behave as in a free market or ecology, where through implicit and informal codes of conduct coordination is achieved and the software created (Raymond 1999b). The analogy of open source communities as bazaars has been adopted and extended by a multitude of researchers (for instance Demil & Lecocq 2003, González-Barahona & Robles 2003, Kuwabara 2000).

Next to the proponents of the bazaar metaphor there are also researchers who contest the metaphor. They argue that the analogy is too simplistic and ignores many of the complexities in the communities (Bezroukov 1999, Iannacci 2002, Zeitlyn 2003).

Open source communities as gift economies. Exchanges between programmers in open source are not based on money. Instead, many researchers argue that they are based on the principle of gift giving (e.g. Bergquist & Ljungberg 2001, Ljungberg 2000, Markus et al 2000, Raymond 2000, Zeitlyn 2003). The concept of the gift economy can be traced back to Mauss (1990) who described a wide range of communities in which gift giving laid the fundament of exchange. A gift economy relies on the principle of reciprocity and an implicit requirement to give (Mauss 1990). In these systems "a gift is not so much a physical resource as a social and moral system by which sharing, collaboration, loyalty and trust are cultivated" (Bollier 2001b, p. 11).

Indeed, there are some indications that the principle of gift giving is important in open source communities. "Open-source contributors have told us that they enjoy the sense of 'helping others out' and 'giving something back'" (Markus et al 2000, p. 15). A respondent interviewed for this research argued, "It is nonsense to believe that in open source you do not receive anything. If you do what you are good at, others will do the same. I receive a lot from others, which I could not have done myself. In the gift economy everybody is better off."[25] As such, participants in the communities are said to create and sustain dynamic relationships with one another based on the exchange of gifts (Zeitlyn 2003).

Other researchers contest the explanation of open source communities as a gift economy (e.g. Iannacci 2003). One of the underlying assumptions of the gift economy is abundance. "The society of open-source hackers is in fact a gift culture. Within it, there is no serious shortage of the 'survival necessities' – disk space, network bandwidth, computing power" (Raymond 2000). Much critique of this metaphor relates to this idea of abundance, which is argued to be incorrect (e.g. Wayner 2000). For instance, "Raymond's statement that the larger the number of open-source projects, the smaller the programmer pool does imply an overall problem of resource scarcity in terms of coding talent that contradicts his overall assumption of resource abundance" (Iannacci 2003, p. 4).

Open source communities as scientific communities. A third explanation of the organization of open source communities is based on a comparison with *scientific communities*. Bezroukov was one of the first authors to compare open source with science. His 1999 paper contests the analogy of open source communities as great babbling bazaars of activity for a number of reasons. Among these are that the bazaar analogy suggests (i) the communities lack rules and norms and (ii) they consist of ideal cooperative people. Instead, Bezroukov argues that open source communities do have some informal rules and norms, which are quite similar to the rules and norms found in science. Furthermore, this metaphor is said to explain the nature of the fights and arguments between participants in the communities.

Other researchers have also adopted the analogy with science, yet the metaphor appears to be less popular than some of the others. Its proponents point out that both communities tend to believe that no property rights on 'inventions' should be granted, and that inventions should instead be freely shared. As in science, the person who first publishes an invention is said to reap the benefits, in the form of credit (Dalle & Jullien 2003). The motives of people to participate in open source communities are also said to be similar to those in scientific communities (Bezroukov 1999, Bonaccorsi & Rossi 2003a). Among them are building a reputation among peers and learning and developing skills (Bonaccorsi & Rossi 2003a, Bonaccorsi & Rossi 2003c).

Open source communities as self-organizing systems. Many researchers argue that open source communities are self-organizing systems (Axelrod & Cohen 1999, Bekkers 2000, Kuwabara 2000, Madey et al 2002). Garzarelli (2003), for instance, writes, "The open source philosophy assures a 'self-correcting spontaneous' organization of work" (p. 4). The claim here is not that hierarchy is completely absent from open source communities, but that no specific entity or person created the organization of open source communities. The organizational structure is the result of and has emerged from the interactions between participants.

The term 'self-organization' has its origin in biology and is especially relevant in the analysis of so-called 'social insects.' Basically, these are insects that live in groups. Examples of social insects are ants, termites and bees. One of the observations of these social insects is described as follows:

> Without centralized control, workers are able to work together and collectively tackle tasks far beyond the abilities of any one of the individual. The resulting patterns produced by a colony are not explicitly coded at the individual level, but rather they emerge from myriads of simple nonlinear interactions between individuals or between individuals and their environment (Theraulaz et al 2003, p. 1265).

This quote contains two important notions, namely the absence of centralized control and the fact that the collective patterns emerge from local interactions, without being coded into them (e.g. Andrew 1989, Bergmann Lichtenstein 2000, Bonabeau et al 1999, Resnick 1994, Simon 1996). The collective patterns in self-organizing systems are not purposefully engineered. Instead, a self-organizing system can be understood only by understanding the actions of 'lower-level agents' in the system and the rules that drive these actions. Therefore, to understand how a self-organizing system is created and functions, the argument goes, we need to understand how the actions of the individual agents aggregate to behavior on a system level.

Resnick (1994) simulates the emergence of phenomena like traffic jams, flocks of birds and slime-mold clusters. He showed that these phenomena result from interactions on a lower level and not from central control (Resnick 1994, p. 141). One thing that he describes as having surprised him the most is that while demonstrating his computer programs, observers would always try and explain the system through some form of central control. It is the idea that "a pattern can exist only if someone (or something) creates and orchestrates the pattern" (Resnick 1994, p. 4). He argues differently and demonstrates that certain systems cannot be understood with a notion of central control. Rather, they must be comprehended through the interactions between lower level agents.

Thus, to argue that open source communities resemble self-organizing systems would imply that open source communities are neither purposefully engineered, nor can they be understood in terms of centralized control. To claim that open source communities are self-organizing means that open source communities are organized through the actions and interactions of the individual participants. Some efforts have been undertaken to model processes in open source communities based on the ideas and concepts of self-organization. For instance, in a working paper Dalle and David (2004) use an agent-based experiment to understand how development efforts are allocated in open source projects.

There has been quite a lot of critique on the description of open source communities as self-organizing systems. One point of critique is that the communities are more hierarchically organized than the metaphor suggests. "[T]he community is far more hierarchically organized for the actual development of software code than suggested by the metaphor of a population of interacting agents" (Kogut & Metiu 2001, p. 260). Furthermore, the critique argues that self-organization fails to explain "*how* those local interactions add up to 'global' order" (Weber, 2004, p. 132).

The role of collective mechanisms in open source communities

A growing body of literature moves away from general metaphors and focuses on specific parts of the organization of open source communities. It does not aim to understand and characterize all the aspects. Topics addressed and researched are, for instance, the role of users in processes of innovation in the communities (Von Hippel 2001, Shah 2003), the role of modularity in both the community structure and the structure of the software (Narduzzo & Rossi 2003) and how conflicts arise and are resolved (Elliot & Scacchi 2002).

These strands of research have contributed a more detailed understanding of open source communities and resulted in a great number of mechanisms that are said to promote coordination. These mechanisms are not claimed to be unique to open source communities; many of them might also be found in other methods of software development.

This section discusses some of the most frequently cited mechanisms. Its goal is not to be exhaustive, but rather to give an idea of the variety of mechanisms that have been identified.

Automated mailing lists. One of the basic tools to support coordination in open source communities is mailing lists. They are used to communicate all sorts of information to the participants in the communities (Bauer & Pizka 2003, Edwards 2001, Kogut & Metiu 2001). Most of the mailing lists are automated, which means that tasks like joining, handing out passwords and making changes to mail addresses are performed automatically.

Software modularity. Many researchers call attention to the fact that open source software is modular, which basically means that big and complex software programs are divided into smaller parts. These smaller parts are relatively easy to understand and ensure that programmers remain relatively less dependent on each other (Benkler 2002a, Bonaccorsi & Rossi 2003c, Garzarelli 2003, Kogut & Metiu 2001, Langlois 2002, Lerner & Tirole 2002b, p. 28, McKelvey 2001b, Moon & Sproull 2000, Narduzzo & Rossi 2003, Tuomi 2001).

Open source licenses. Many communities have licensed their software with a specific type of license. Examples of open source licenses are the General Public License (GPL) and the Berkeley Software Distribution (BSD) license. These licenses allow others to adopt and make amendments to the software and are said to be necessary to ensure collaboration in the communities (Boyle 2003, Dalle & Jullien 2003, Lerner & Tirole 2002b, McGowan 2001, O'Mahony 2003)

Project leaders. Many communities have one or more clearly identifiable project leader (Bonaccorsi & Rossi 2003c, Dafermos 2001, Fielding 1999, Hann et al 2002, Lerner & Tirole 2002b, Markus et al 2000, McCormick 2003, Moon & Sproull 2000). Generally, leadership is 'given' to the person who makes the first lines of source code publicly available. The actual activities performed by project leaders are different for each community. For instance, the leader might maintain the version that is commonly accepted as the official version of the software, as well as promoting the software and informing organizations considering adopting the software about how to deal with the larger community of developers. Project leaders thus perform an important role in coordinating the efforts of the participants (Egyedi & Van Wendel de Joode 2004). Nonetheless, there is no universally accepted image among researchers of the exact role that project leaders play and their degree of influence on the individual contributors. Von Hippel and Von Krogh (2003), for instance, argue that project leaders are different from most managers in 'traditional' companies, because they cannot mandate and enforce (p. 218). They argue that further research on the role of project leaders is highly relevant, and propose that project leaders should perhaps be compared to a coach in a sports team.

Concurrent Versions System. Many researchers have identified the relative importance of the concurrent versions system (CVS) in achieving coordination in open source communities (Bauer & Pizka 2003, German 2002, Hemetsberger & Reinhardt 2004, Von Krogh et al 2003a, Scacchi 2004, Shaikh & Cornford 2003). The CVS is a system that supports the development activities of individuals and allows multiple developers to simultaneously improve a certain piece of source code. Adopting a CVS thus reduces the need for coordination among the participants in a community.

State-of-the-art research leaves our research question unanswered

The first category of explanation, that is, individual motivation, is highly relevant and has been the focal point of many publications on open source. Indeed the question of motivation is one that needs to be answered first in order to understand the organization of open source communities. However, an organization is more than a collection of motivated individuals. Research focusing on motivation leaves many questions unanswered about the organization of the communities. For instance, how are activities divided among the individuals that participate in a community? Based on what criteria do individuals decide to participate in one community rather than another? What are the criteria for selecting a certain piece of source code and integrating it into the existing code base? Who makes the decision? These and other questions need to be answered to understand the organization of open source communities; yet they cannot be answered with a focus limited to motivation.

The second and third category of explanation focus more on the actual organization of open source communities. Yet they too are unable to provide an answer the research question. The second category consists of research that primarily draws parallels with existing and better understood communities and organizations. It uses metaphors, explaining the communities as babbling bazaars of activity, as scientific communities, as self-organizing systems and as gift economies. The use of metaphors enables the identification of mechanisms and processes that have previously gone unnoticed or which were believed to be unimportant. This is a key means to characterize the type of organization of a community and to understand how certain elements and mechanisms in the communities are related to one another (e.g. Morgan 1986, especially pp. 12-14 and 339-344). However, the dilemma of the metaphors described in a previous section is that they do not move beyond the level of analogy.[26] They are based on presumptions; on mechanisms that are assumed to be present, but which by and large are not tested and confirmed by empirical facts. In short, they leave many questions unanswered. For instance, what does it mean that the organization of open source communities can be compared to scientific communities? How does this explain the rise of open source communities? How are decisions made? How are conflicts resolved?

The third category groups a great number of research efforts which identify mechanisms that are argued to contribute to coordination and collaboration in the communities. This line of research is highly valuable and provides crucial insights into the communities and the way they are organized. However, in reading this literature it becomes clear that many gaps remain. For instance, to have an understanding of an organization implies more than a discussion of one or more mechanisms. This too leaves us with a number of questions: What is the organizational model that underlies these mechanisms? What is the interrelationship between the mechanisms? What goal does a mechanism serve? Is a particular mechanism crucial in achieving coordination or collaboration? Are four or five mechanisms sufficient to explain the organization of a community and to ensure its sustainability? To answer these questions, some framework of organizations is needed. Such a framework would impart a better understanding of the role, function and potentially the importance of a particular mechanism. It would also provide an opportunity to better understand the interrelationships between multitudes of mechanisms.

The framework: reconciling differences in state-of-the-art research

Self-organized anarchies versus institutionalized communities: two extremes?

As the previous sections showed, one group of researchers aims to explain the organization of open source communities through the adoption of metaphors. These metaphors portray and characterize the communities as highly decentralized organizations. The communities are said to be 'bazaars' that lack centralized control or 'economies' in which individuals voluntarily exchange gifts. Or interacting individuals who 'self-organize,' which means that their interactions result in complex patterns of behavior on the collective level. In the explanations there is no room for institutions. Collective mechanisms to coordinate the efforts of individual participants are, by and large, believed to be absent.

Many of these metaphors are being challenged both implicitly and explicitly by a growing number of researchers who have identified a broad spectrum of collective mechanisms. These mechanisms are said to contribute to coordination and they suggest a certain level of institutionalization in the communities.[27] Much research, for instance, points to the presence of an institutional licensing scheme, which ensures that the source code of open source software remains open and available (Dalle & Jullien 2003, Franck & Jungwirth 2003). O'Mahony (2003) identifies formal institutions, like formally erected foundations, that protect the future of open source software. Finally, some communities have institutionalized their leadership structure. The community of participants might hold yearly elections to choose a new project leader (O'Mahony & Ferraro 2004) who directs processes of software development and maintenance.

The two explanations appear to be opposites. One group of researchers characterizes open source communities as self-organizing, with order said to spontaneously emerge. Others, however, identify a great number of mechanisms that institutionalize processes on a collective level. They argue that coordination in the communities does not emerge from the interactions between participants, but is institutionalized – that is, they are at least partly the outcomes of institutionalized collective arrangements.

Reconciling competing explanations: community-managed common pool resources

This research adopts a framework that incorporates the key ideas of both explanations. The framework comes from research on community-managed common pool resources. Elinor Ostrom (specifically 1990, 1999) and other researchers, although some implicitly, demonstrate how communities of individuals are able to self-organize. The explanation of how they are able to self-organize and ensure sustainability is sought in institutions. Institutions influence individuals' behavior in such a way as to enable them to organize. Thus, in research on community-managed common pool resources, self-organization and institutions are not treated as mutually exclusive, but rather as complementary.

Chapter two provides a detailed explanation of research on community-managed common pool resources. The chapter introduces eight design principles[28] which are said to explain how and why certain communities succeed in organizing themselves, whereas others do not.

For now, it is important to note that this research adopts these same design principles to create a framework that can reconcile the two strands of state-of-the-art research on open source communities. The design principles refer to institutions that need to be present to ensure that communities can organize and sustain themselves (Anderies et al 2003, Ostrom 1993).

Other reasons to adopt lessons from research on community-managed common pool resources

There are other reasons to adopt the lessons from community-managed common pool resources. Most important, it does not presume self-organization to be present, unlike the literature that starts with the assumption of self-organization. Rather, the framework identifies the collective functions that are needed for the communities to organize and sustain themselves – and leaves open the possibility that these functions are performed through self-organizing individuals.

The second reason is that it combines rigorous empirical analysis with concepts from research on self-organizing systems. State-of-the-art research on self-organization in social systems and organizational theory are both said to lack such a framework (Edmonds 2002, De Jong & Van der Voort 2004, Laland & Brown 2002). Research on community-managed common pool resources combines extensive empirical research on social systems with concepts adopted from self-organization. The eight design principles create a coherent set of characteristics that are claimed to be needed to understand how self-organizing communities are organized and how they sustain themselves.

The third reason is that the source code of open source software shares features with other, more traditional, common pool resources. Common pool resources face the threat of appropriation. Appropriation refers to a process in which individuals claim ownership of parts of the resource. Too much appropriation threatens the future of the resource. Open source software faces a similar threat. The invocation of intellectual property rights would allow others to appropriate open source software. This external pressure was described in a previous section of this chapter and is acknowledged and written about in a large body of literature on open source software (e.g. Boyle 2003). The presence of this threat has led researchers to claim that it is fruitful to adopt lessons from research on community-governed common pool resources as a framework to describe and understand open source communities (e.g. O'Mahony 2003).[29] Adopting these lessons helps us to identify the processes and structures in open source communities that protect the source code from the threat of appropriation. The next chapter provides a more elaborate discussion about the similarities and differences between the source code of open source software and common pool resources.

The fourth reason for adopting lessons from community-managed common pool resources is that the latest research on open source communities has identified mechanisms that contribute to coordination and collaboration. However, a coherent framework that combines these findings into one organizational model of open source communities is currently missing. The eight design principles introduced in chapter two do provide such a framework. The framework constitutes a means to position the mechanisms relative to one another and to explain their role in the community.

Relevance of this research

This research has social relevance and is relevant for policymakers for the reasons presented below. The reasons for the scientific relevance of this work are also presented and discussed.

Policy relevance

There are a number of reasons why this research is highly relevant for practitioners and policymakers. First, more and more companies and governments are switching or planning to switch to open source. Also, in many countries policymakers are actively promoting the adoption of open source software. Many reasons are used to justify this choice, for instance, it is claimed that: the costs of open source software are low, open source software stimulates the local ICT market and open source provides more freedom to users.[30] However, a generally accepted understanding of what open source communities are, how the software is developed in the communities and whether the communities provide a sustainable way to produce software is by and large lacking. Furthermore, it is unclear what the implications are for organizations adopting open source software. How should they interact with the developers in a community? How can they influence the direction of software development in the communities? What open source software packages should they select and which packages should they ignore? To answer such questions, further research on the organization of open source communities is needed.

Second, as explained in a previous part of this chapter, patents and copyright threaten the future of open source communities. Many governments are considering extending patent laws and the length of copyright. Yet at the same time research suggests that software developed in open source communities is innovative, gives rise to level playing fields and stimulates local economies (e.g. Dalle & Jullien 2003, Von Hippel 2001, Markus et al 2000, Vendrik & Van Tilburg 2002). Given these positive effects, the presence and impacts of patents and copyright on open source communities need further investigation.

Third, a number of captains of industry in the software market are currently changing to a so-called 'inner source approach.'[31] The idea is to modify the internal structure and process of software development to an approach based on open source communities. However, a good understanding of what open source communities actually are and how they are organized is largely absent. The inner source approach is based on many assumptions about the communities, but the question is whether these assumptions are correct. Therefore, research on open source is needed.

Scientific relevance

There are also a number of reasons why this research has scientific relevance. First, open source communities consist of professionals who are part of organizations that are not engineered and that seem to be self-organizing, evolutionary. The scientific question is how are these professionals organized and what can we learn from this? A better understanding of open source communities might result in a new organizational model and the identification of new organizational concepts to manage and organize professionals.

Second, a great number of assumptions in the literature do not seem to hold for open source communities. For instance, a large body of literature argues the need for patents to stimulate and ensure innovation in the software market (e.g. Cowan & Harison 2001). However, open source communities create innovation without reliance on patents. This results in a number of scientifically relevant questions: What triggers innovation in open source communities? Does innovation in the communities depend on chance? Or are there fundamental characteristics that explain why software development in the communities is innovative? Could these characteristics be applied to the entire software market, which would mean that patents are unnecessary in the software industry? A thorough understanding of open source communities is one of the most important requirements for answering these questions.

Third, globally, many researchers are trying to understand and explain how professionals in open source communities organize their activities. Yet as of yet no satisfying organizational model has been put forth that can explain the organization of these communities. The fact that open source communities are hardly understood and an organizational model is lacking makes research on open source communities relevant.

Structure of the book

Table 1.1 depicts the structure of this book. The second chapter presents the theoretical framework, introducing research on common pool resources in some detail. This chapter answers questions like 'What are common pool resources?' and 'How do they differ from other types of goods?' What are the problems facing common pool resources? What are the solutions to overcome these problems? How are communities of utility-maximizing individuals able to overcome the problems facing common pool resources? A substantial part of this chapter will be used to introduce and explain eight design principles. Basically, the design principles are believed to explain why certain communities are able to organize themselves and achieve continuity and why others are not. Each of the design principles addresses a separate aspect or issue of the organization of 'successful' communities.

Chapter three describes the research methodology. This chapter describes how the empirical data for this research was gathered and analyzed. More importantly, it presents the strategy for answering the research question. Chapters two and three will perhaps be less interesting for those readers who are mainly interested in open source communities. These readers may want to fast forward to chapter four.

Chapters four through ten focus on empirical observations concerning open source communities. These chapters are structured according to the design principles. Each chapter discusses a design principle. A single question underlies each chapter: 'Can we identify mechanisms or institutional arrangements that address or solve the functions described in the design principles?' To answer this question, each chapter investigates and discusses the existing structures and mechanisms in open source communities, and tries to tease out their role in the communities' organization.

The final chapter is the conclusion. This chapter integrates the findings of the separate chapters into one organizational model of open source communities. This model sheds a light on how the functions in each of the design principles are addressed and thus answers the research question.

Table 1.1 – Outline

Chapter 1		Introduction
Chapter 2		Theoretical framework
Chapter 3		Methodology
Chapter 4 - 10		The design principles
	Ch. 4	Boundaries
	Ch. 5	Provision and appropriation
	Ch. 6	Conflicts
	Ch. 7	Collective choice
	Ch. 8	Monitoring and sanctioning
	Ch. 9	Nested enterprise
	Ch. 10	External recognition
Chapter 11		Conclusion

Notes on chapter one

[1] In the remainder of this research the term 'open source' will be used most. This goes against the direct wishes of the person who first coined the term 'free software' and who is considered to be one of the most important leaders in the communities. He, Richard Stallman, asked us in a personal interview to please be sure to "always mention free software *and* open source. That gives us equal attention, so we can't complain." The continuous use of both terms, however, would detract from the readability of this text. Therefore, a choice had to be made. 'Open source' is used here because it seems to be the more widely used term. The controversy surrounding both terms is worth noting: 'open source' is said to refer to the more commercial term and 'free software' to the ideology (see, for instance, Van Wendel de Joode R, De Bruijn JA, Van Eeten MJG. 2003. *Protecting the Virtual Commons; Self-organizing open source communities and innovative intellectual property regimes*. The Hague: T.M.C. Asser Press.) The choice for 'open source' should not be understood as a position taken by the author in this controversy.

[2] Linux is one of the best-known examples of open source software.

[3] Quoted from a position paper by Open Source Risk Management (OSRM). The paper is available on the Internet: http://www.osriskmanagement.com/linuxpatentpaper.pdf (August 2004).

[4] The surprising fact that communities are able to organize themselves effectively has also been addressed in other types of communities. Examples include file sharing communities on the Internet (Cunningham BM, Alexander PJ, Adilov N. 2004. Peer-to-peer file sharing communities. *Information Economics and Policy* 16: 197-213) and innovative consumer goods communities (Shah S. 2003. *Community-Based Innovation & Product Development: Findings From Open Source Software and Consumer Sporting Goods*. dissertation thesis. Massachusetts Institute of Technology, Cambridge, MA).

[5] The term 'intelligence' "refers to the ability to achieve outcomes that fulfills desires as much as possible." (March, JG 1999. *The pursuit of organizational intelligence*. Oxford: Blackwell Publishing, p. 1)

[6] The figure is from the website of a company called Netcraft. "Netcraft offers a range of services in the areas of World Wide Web Publishing, Internet Security, and Contract Systems & Network Management" (cited from their website: http://www.netcraft.com/info.html, September 2002).

[7] The Trimbos Institute has implemented MMBase as a content management system (http://www.mmbase.org/index.jsp?page=21341&portal=202&o=0&newsnr=20122, March 2004).
[8] Two municipalities in the Netherlands, namely Leeuwarden en Amsterdam, have now adopted MMBase as their content management system (CMS) (http://www.webwereld.nl/nieuws/16704.phtml, March 2004), but more municipalities will follow, as MMBase is the preferred CMS in a combined initiative of municipalities in the Netherlands to get a better presence on the Internet (http://www.egem.nl/index.php?query=mmbase&amount=0&blogid=1, March 2004).
[9] The Dutch public broadcaster VPRO developed the first version of MMBase. Other public broadcasters that use it include the EO and the NOS.
[10] From an article on the Internet: http://www.it-director.com/article.php?articleid=2125 (November 2003).
[11] From an article on the Internet: http://www.oreillynet.com/pub/a/oreilly/ask_tim/2004/amazon_0204.html (March 2004).
[12] From an article on the Internet: http://news.com.com/2100-1001-275388.html?legacy=cnet (November 2003).
[13] From an interview with the IT director of the City of Newport. The interview is available on the Internet: http://www.linuxdevcenter.com/pub/a/linux/2004/01/15/andy_stein_interview.html (March 2004).
[14] There are also many references to this project in articles on the Internet, for instance: http://www.wired.com/news/infostructure/0,1377,62236,00.html (March 2004).
[15] From an interview with the director of Brazil's National Information Technology Institute available on the Internet: http://www.wired.com/news/infostructure/0,1377,61257,00.html (March 2004).
[16] Cited from their website: http://www.ososs.nl/index.jsp?alias=english (March 2004).
[17] From the website: http://www.ososs.nl/index.jsp?page=198 (March 2004).
[18] Cited from an article on the Internet: http://www.infoworld.com/article/03/07/01/HNreasoning_1.html (March 2004). The choice for the word commercial is somewhat unfortunate, as open source software can also be commercial. Use of the word proprietary would be more appropriate. Proprietary highlights the fact that the owner develops software with exclusive rights and considers it private property, which is the most fundamental difference between open source and other types of software development.
[19] From the website: http://www.infoworld.com/article/03/07/01/HNreasoning_1.html (March 2004).
[20] Cited from the Internet: http://www.reasoning.com/news/pr/02_11_03.html (July 8, 2003).
[21] One exception is the article: O'Mahony SC. 2003. Guarding the Commons: How Community Managed Software Projects Protect Their Work. *Research Policy* 32: 1179-98 She explicitly addresses the question of how open source communities protect themselves against external pressures. She then focused on the boundaries that were constructed in open source communities. The next chapter will argue that boundaries are just one part of the answer to the research question.
[22] For a more detailed description of this project and for the final report see the website: http://www.infonomics.nl/FLOSS/index.htm (November 2004).
[23] There are many articles in which references can be found to open source communities as some kind of religion or communistic sect. For example: http://www.eweek.com/article2/0,1759,1600832,00.asp (February, 2005). Or consider the title of an e-book, which was published in 2000 and is called: "The New Religion: Linux and Open Source."
[24] The open source development model is usually contrasted to the proprietary software development model. In the latter the source code is treated like a blueprint that should be kept secret, see for instance: Edwards K. 2001. *Epistemic communities, situated learning and open source software development.* Presented at 'Epistemic Cultures and the Practice of Interdisciplinarity' Workshop at NTNU, Trondheim
[25] Translated from Dutch.
[26] The claim in this paragraph is not that every explanation based on metaphors ignores empirical facts and remains on the level of analogies. The claim is that the four metaphors, as introduced in this chapter, do not move beyond this level. They largely ignore empirical facts and complexities. Their ability to explain the processes and structures in open source communities is therefore limited.
[27] In a personal correspondence with a fellow PhD researcher of open source communities.
[28] The term 'design principle' does not refer to the term 'design' as it is typically used in engineering. In other words, they cannot be used to actually build and construct a community. The following chapter will describe in quite some detail what the author, who initially coined the term, means with the term.
[29] This was also the conclusion of a one-day workshop at the EASST conference in Paris, August 27, 2004, at which many participants agreed that the rhetoric in both settings was similar and that analysis along these lines is useful.

[30] These claims are heard at many conferences and workshops about open source software. They are also found on numerous websites that argue in favor of open source software.
[31] The term 'inner source' is currently in use at companies like HP and Philips.

CHAPTER TWO

THEORETICAL FRAMEWORK

The unit of analysis in this research is the open source communitiy. Chapter one introduced the research question: 'How are open source communities organized and how do they sustain themselves?' The framework that is used to answer this question is based on research on community-managed common pool resources. Before introducing this particular body of research, this chapter first introduces common pool resources and explains what they are. It argues that this particular type of resource faces two challenges: (i) a collective action problem and (ii) a 'tragedy of the commons' problem. Much, primarily empirical, research on common pool resources indicates and argues that communities of rationally acting individuals can overcome the two conceptual challenges that are inherent to common pool resources. Extensive fieldwork on and conceptual models of communities confirms the fact that individuals are able to organize themselves and create a sustainable institution in which common pool resources are managed. The claim is that the lessons from this research are valuable and that they provide a promising start to analyze the organization of open source communities.

There are a number of reasons why lessons from community-managed common pool resources are adopted to create the framework that underlies this research. One of these reasons stands out, namely, research on community-managed common pool resources reconciles the two strands in current state-of-the-art research on open source communities. The previous chapter argued that researchers either describe open source communities as self-organizing with coordination achieved through the spontaneous and self-correcting actions of agents, or as entities with institutional mechanisms for achieving coordination. Research on community-managed common pool resources demonstrates that both explanations can co-exist and provides a framework to do justice to both strands in recent research on open source communities.

On the nature of common pool resources

In economic theory a 'common pool resource' or 'common good' is a distinct type of good. To understand the characteristics of common pool resources, they can be offset against three other types of goods (e.g. Levacic 1991). The four types of goods are typically characterized along two dimensions. The first is that of *excludability*; whether or not people can be excluded from consuming the good. The second is that of *consumption*, which can be either joint or subtractive. *Joint* means "one person's consumption of the good does not reduce the amount available to another" (Kollock 1999, p. 223). *Subtractive* means that one person's consumption of the good does reduce the amount available to others (Levacic 1991, Ostrom 1990, Ostrom 1999, Ostrom 2000, Thomson & Schoonmaker Freudenberger 1997).

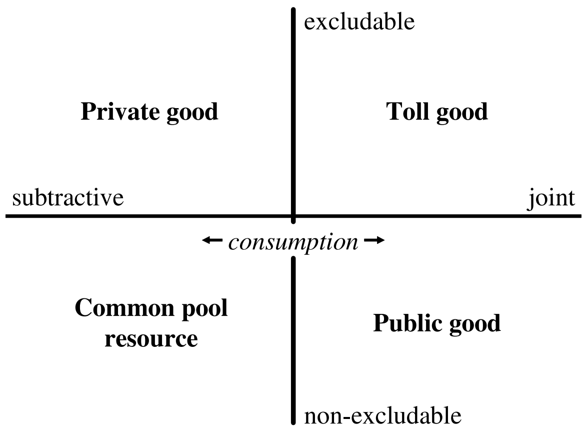

Figure 2.1 - Four types of goods

Placing these two dimensions on two axes creates the matrix presented in figure 2.1. As the figure shows, a *private good* is a good from which people can be excluded and where consumption is subtractive. Such goods include most commercial items bought in stores, for example, bottles of mineral water, stereos, televisions and clothing. From a *toll good* people can also be excluded, but consumption is joint. Examples of toll goods are a zoo or amusement park. A *public good* is a good from which people cannot be excluded and where consumption is joint, like streetlights or the safety of a nation. The final type of good is the *common pool resource*. From a common pool resource people cannot be excluded and consumption is subtractive.

According to Hess and Ostrom (2001), the characteristics of consumption and excludability are inherent to a resource and the distinction between the four types of goods can be interpreted in only one correct way. At the same time, they acknowledge the fact that people frequently use the term 'common pool resource' to identify goods which in reality are not.[1] They argue that a common pool resource is frequently confused with a management regime, namely the common property regime. The term 'common property resource' (Dietz et al 2002, Hess & Ostrom 2001, Ostrom 2003) is used to illustrate their point.

The common property regime

A common property regime, or 'commons', is a property regime in which multiple owners have unrestricted access to a given resource and, maybe more importantly, no owner has the legal right to exclude another owner from accessing and consuming the resource (Heller 1998, pp. 623/624). This is different from a regime of *open access* (Bromley 1992, Bruns 2000, Dietz et al 2002, Runge 1992, Steins et al 2000). In an open access regime no property rights have been recognized, which in Latin is called *res nullius* (Bromley 1992). In such a regime no one can formulate rules to limit entry and use of the resource (Dietz et al 2002).

Instead, in a common property regime multiple owners exist and they can decide to regulate entry and use of the good. In this regime a group of people can govern the good as collective entity (Oksanen 1998). The individuals are thus "no longer entirely free to decide for themselves how to make use of the commons, as in a private-property arrangement, but participate in a process of collective choice that sets limits on individual use" (Oakerson 1992, p. 47). Thus, a common property regime consists of rules that limit access or use of the good, whereas an open access regime per definition has no such rules.[2]

Collective action, the tragedy of the commons and the usual solutions

The distinction in types of goods is important for at least one reason, namely, because one of these categories faces particular challenges. These challenges have been identified in economic literature and originate from the fact that common pool resources face externalities, which occur "when the actions of one economic agent affect the welfare of others in a way that is not reflected by market prices" (Selz 1999, p. 22). The two challenges are known as the 'collective action problem' and the 'tragedy of the commons.'

The collective action problem

Common pool resources share with public goods the fact that they are non-excludable. Therefore, people who haven't paid for the good can still consume it. This characteristic gives rise to what is known as the *collective action problem* (Olson 1965). The central idea here is that people behave rationally, meaning they will try to maximize their own individual utility. People who can have a good for free will not pay for it, according to this axiom, nor will they contribute to its development and maintenance. Why should they? They are much better off consuming the good without paying. This is not to say that people do not feel a need for the product. However, even if they did feel a need it is considered rational for them not to participate in the good's development and maintenance (Olson 1965).

The problem of a public good is that every individual will very likely face the same trade-off, which means that for each individual it is rational not to participate in the development and maintenance of the good. This could mean that the good is not created, even though collectively people would benefit from producing the good.

The tragedy of the commons

A common pool resource differs from a public good in that its consumption is subtractive. This characteristic creates the second problem facing common pool resources, which is usually referred to as the 'tragedy of the commons' (Hardin 1968). In a common pool resource every individual has an incentive to consume as much as possible. Yet consumption limits the amount of the resource available to others due to its subtractive nature. Many natural resources, like fishing grounds, have a natural rate of regeneration. In time the resource will be depleted if the rate of consumption exceeds that regeneration rate. According to Hardin (1968), it is rational for every individual to consume additional units of the resource. This will almost inevitably result in the depletion; that is a 'tragedy,' of the common pool resource.

Solution 1: privatization and the market

The discourse on common pool resources has long been dominated by the idea that there are only two models to solve the problems facing this resource category. One is coordination through privatization and the market and the other is coordination by hierarchy (Ostrom 1990, Ostrom et al 1999, Rose 2002).

Coordination via the *market* is achieved through the price mechanism. Though the price mechanism leads actors to pursue their self-interest, this pursuit yields coordination among the actors as an (un)intended side effect. This process is better known as the 'invisible hand' of the market (e.g. Witt 1997). The invisible hand, however, is present only in so-called 'perfect markets.' In the absence of such perfection, markets fail to coordinate individuals' efforts. This is usually referred to as market failure, which could be the result of an externality. The non-excludability of common pool resources creates such an externality, meaning that prices will not reflect the social costs of, say, overfishing (Selz 1999).

One solution to overcome this problem is the allocation of *property rights*. Property rights are used to divide a resource into smaller, tradable units and to assign private ownership of each individual unit (Levacic 1991, Ostrom 1990). Dividing the resource into smaller units effectively removes the presence of the externality associated with common pool resources, because users can now be excluded from individual units of the resource. The units can also be traded on a market as if they were private goods.

Assigning property rights also has disadvantages, however. One is the problem that certain common pool resources cannot be logically divided into units (Levacic 1991, Ostrom 1990). How can a fishing grounds or air be divided? Another problem with property rights is the creation of inequality, which could result from separating the resource into units. What if, for instance, one area of the resource is more profitable, for example, if one unit of land has substantially more rainfall than other areas? Who is to receive what unit of the resource (Ostrom 1990)?

Essentially, the price mechanism coupled with private property rights stimulates the rise of coordination. It is one way to solve the collective action problem and prevent a tragedy of the commons. This model, however, comes with a set of its own predicaments.

Solution 2: a central authority

Another model for stimulating coordination and overcoming the problems of a common pool resource is through some form of central control (Hardin 1968, Levacic 1991, Ostrom 1990). The appointment of a central authority is frequently considered to be the exact opposite of assigning individual property rights and letting the market do the work. Markets are unstructured, decentralized and operated by an invisible hand. Central authorities, on the other hand, are considered to act in a structured and controlled manner. In a way, central authorities replace the invisible hand of the market (Powell 1990). A central authority is appointed to control the creation, use and maintenance of a resource. Through the use of incentives and punishments, central authorities are said to be able to control individuals' behavior so as to prevent a tragedy of the commons and to solve the collective action problem.

Like the market, however, the solution of a central authority has a number of shortcomings. For example, hierarchies can have rather harmful effects in situations where the resources and activities in them are highly complex and dynamic. Under these conditions a central authority is hardly able to oversee the entire problem facing the resource. It is therefore likely that the central authority will influence the system negatively. Indeed, many examples are reported in the literature where the interference of a central authority has stimulated destructive behavior. In these situations central authorities did not resolve the problems, but rather did just the opposite (Ostrom 1990, Scott 1999). Another problem of a hierarchy is the presumption that punishments and incentives lead people to exhibit the desired behavior. Much literature, however, points out that this need not be the case (e.g. Frey 1997, Ostrom 1990). Finally, central authorities often have difficulty taking into account and dealing with the strategic behavior of actors in the system (Kuit 2002, Ten Heuvelhof et al 2003).

A third model: self-organized and self-governed communities

The rhetoric on common pool resources has long been dominated by the claim that the state and the market are the only possible avenues to overcome the collective action problem and to prevent a tragedy of the commons. However, as research on common pool resources progressed, a growing number of examples provided empirical proof that in certain situations a depletion of a resource was prevented – apparently without any market or state that had achieved this. Rather, research shows that other, informal types of institutions succeeded in preventing resource depletion. These institutions are communities of local people who proved able to organize themselves to collectively solve the problems facing common pool resources (Poteete & Ostrom 2004).

Elinor Ostrom was among the first researchers to describe these community-managed common pool resources. She wrote her findings, which are based on a great number of case studies, in a book called *Governing the Commons: The Evolution of Institutions for Collective Action* (1990). In the book she claims that the state and market are not the only ways to overcome the problems facing common pool resources. Instead, she argues in many situations local people are much better able to overcome the problems. She shows that people are able to self-organize and create a community in which they collectively ensure the continuity of the common pool resource.

It is important to note that Ostrom's findings do not depart from the basic principles of economic thought. The individuals in communities described in Ostrom's book are not altruistic. Instead, they want to maximize their utility and act in their own self-interest (see Ostrom 1990, pp. 33-40). Yet these individuals themselves create communities in which they trade some of their short-term gains to create and sustain a community structure in which the common pool resource is governed. To Ostrom, the reason lies in the institution of the community. It is the design of the community that creates a set of incentives and disincentives that influences the behavior of the agents in such a way that they collectively – and without state intervention or a market – are able to assure the continuity of the common pool resource.

Institutions in research on community-managed common pool resources

The concept of 'institution' is widely used, but has different meanings in different disciplines. Indeed, Ostrom (1990) gives a central role to the presence of institutions and their ability to influence the actions of individuals. She writes, "Communities of individuals have relied on institutions resembling neither the state nor the market to govern some resource systems with reasonable degrees of success over long periods of time" (p. 1). To understand what Ostrom and other researchers on community-managed common pool resources mean by institutions, this section describes and defines the term.

A large body of literature addresses the notion of institutions (Goodin 1996, Hendriks 1999, Nelson & Sampat 2001). According to Nelson and Sampat (2001), the one thing that unites researchers using the term 'institutions' is their focus on human interaction (p. 38). In their article, they identify three ways in which the term is used.

> Some use the term to refer to standardized behavior patterns per se. Others use the term to refer to factors and forces that constrain or support these patterns of customary behavior, like norms and belief systems, or the rules of the game, or governing structures… Others tend to define institutions in terms of a broader social and cultural context within which particular rules and organizational forms take shape (2001, p. 38).

Ostrom uses the term 'institution' to refer to factors and forces that constrain or support the behavior of individuals and that give rise to regularities in patterns of human behavior (Crawford & Ostrom 1995). Hendriks (1999) writes:

> Ostrom does not focus on actual behaviour – as do those investigating administrative conventions – but on the 'working rules' which condition that behaviour. In her work there is only an indirect link between actions and institutions. Institutions, for her, are prescriptions and rules that signal the boundary conditions for behaviour, not the behaviour itself (p. 72).

Thus, in her work, institutions are sets of working rules that influence the actions of individuals in an action arena (e.g. Ostrom 1990, Ostrom et al 1994).

The literature commonly distinguishes two types of institutions. One type is informal. Informal institutions are, for instance, norms and behavioral codes of conduct (De Jong 1999, North 1991). Informal institutions typically emerge; they are said to be organic (Hodgson 2003). Institutions can also be formal. As the name implies, these consist of formal rules that "may be imposed and enforced by direct coercion and political or organizational authority" (March & Olson 1989, p. 22). Examples of formal institutions are procedures, arrangements, contracts and laws.

Elements of any institution are 'institutional arrangements' (e.g. Tang 1991), which are also referred to as 'rules of play' or 'rules of the game.' Ostrom writes, "'Institutions' can be defined as the sets of working rules" (1990, p. 51). According to Klijn (1996), these rules regulate interactions between individuals and are by definition not logically connected to an individual actor. Rules exist because individuals know of their existence and because in their actions they are influenced by them. However, people have the option of not following them, they could opt to ignore them (see also Ostrom et al 1994). In this research the term 'rule' will be used differently. They are considered to be behavioral rules, i.e. the rules that underlie the actual behavior of individuals. The differences between both types of rules are explained in more detail the next section.

Self-organizing communities and the role of rules

Ostrom focuses on one particular type of institution that is able to sustain a common pool resource, which is a self-organizing community. She uses this term to highlight the fact that neither states nor markets created these institutions. Instead, individuals crafted the institutional rules themselves. They created and adopted sets of rules to influence their action arena. In *Governing the Commons* she argues that this type of institution, in which individuals create their own rules and voluntarily subject themselves to them, has by and large been ignored by researchers. What is missing is "a theoretical explanation – based on human choice – for self-organized and self-governed enterprises" (1990, p. 25). However, in her book Ostrom remains somewhat vague as to what exactly self-organization and self-governance mean. If we were to extract a definition of self-organization it would be that it is a form of "collective action whereby a group of principals can organize themselves voluntarily to retain the residuals of their efforts" (Ostrom 1990, p. 25).

In a more recent article she pays more attention to 'self-organized groups.' She claims that such groups are forms of and should be viewed as *complex adaptive systems*. She writes, "I draw on recent research by Holland (1995) and colleagues at the Santa Fe Institute to discuss the attributes and mechanisms of a different form of general organization – a complex adaptive system – that is not the result of central direction" (Ostrom 1999, p. 497). This statement introduces another term for self-organizing systems, namely 'complex adaptive systems.' They are defined in Holland (1995) as "systems composed of interacting agents described in terms of rules" (p. 10) and they "exhibit coherence under change…and they do so without central direction" (pp. 38, 39).[3]

Like Ostrom, Holland refers to rules. Yet each points to a different type of rule. Ostrom refers to working rules, to rules that influence the action arena of individuals (Ostrom & Ostrom 2004). "Rules define the actions that individuals may, must, or must not take" (Costanza et al 2001, p. 17). She writes, "One should not talk about a 'rule' unless most people whose strategies are affected by it know of its existence and expect others to monitor behavior and sanction nonconformance" (1990, p. 51). In her definition of rules she leaves room for both informal and formal rules that influence interactions between individuals.

This definition of a rule is different from the way rules are defined in most research on complex adaptive systems or self-organizing systems. Generally, in the literature on self-organizing systems rules also form part of the explanation of how individuals interact. However, these rules are not collectively agreed. Individuals, or generally, actors or agents,[4] are unaffected by rules. Instead, the rules are connected to an agent. Holland writes, "It is useful to think of an agent's behavior as determined by a collection of rules" (1995, pp. 7). The basic premise in complex adaptive systems is that agents are not bound or restricted by collectively agreed rules; instead they are autonomous and have their own agenda (Bonabeau et al 1999). An individual or agent in such a system acts based on its "internal rules and the state of its local environment" (Kelly 1994, p. 22). Rules in complex adaptive systems are approximations or simplifications, and they are believed to underlie the actual behavior of individuals (e.g. Waldrop 1992). To highlight the fact that these rules describe actual behavior, this research uses the term 'behavioral rules,' borrowed from De Jong (1999). This does not mean that all agents in a complex adaptive system behave similarly in similar situations. On the contrary, they can and will behave differently (Resnick 1994).

The different ways in which Ostrom and other researchers on self-organizing systems perceive rules draws attention to a difference in the way both analyze the question of how coordination is achieved in these systems. Ostrom analyzes coordination that is achieved by individuals who define and create rules which are collectively understood and which influence the actions of individuals. These rules "change the structure of incentives in situations" (Ostrom 1986, p. 6). They affect the action arena of individuals who are then likely to display a certain type of behavior. Collectively this can fuel coordination and prevent a tragedy of the commons. The 'rules of the game' are collective arrangements. In contrast, the rules in other works on complex adaptive systems are not agreed upon. They are not rules of the game, but rather rules that underlie the actual behavior of individuals. They are behavioral rules.

Perhaps this difference in the use of the term 'rules' can best be understood by the object that is analyzed in both strands of research. Ostrom and other researchers on community-managed common pool resources do *not* focus on actual behavior. Thus, the rules do not describe actual behavior. Returning to Holland's definition, research on self-organizing systems aims to understand individual behavior. In this line of research rules are adopted to understand and model this behavior.

Despite the differences in the way in which coordination in self-organizing systems is believed to be achieved, there is one important similarity, which is the attention that both strands of research give to collectives that are able to achieve coordination in the absence of a supervisor. Individuals are believed to be able to achieve collective action and to coordinate their activities in the absence of "an external ordering influence" (Bonabeau et al 1999, p. 9).

Explaining how communities sustain common pool resources

Empirical research on community-managed common pool resources has resulted in the identification of a set of characteristics shared among the successful communities. 'Successful' refers to the fact that these communities were able to sustain a productive resource over a long period of time (adopted from Ostrom 1990). Before introducing the characteristics, an example of a self-organizing community is introduced.

Example of a self-organizing community

Ostrom (1990) details how farmers in the regions of Murcia and Orihuela, both situated near Alicante in Spain, have been able to share among themselves the water available to irrigate their lands. Farmers in both regions depend primarily on rainwater and water from the Segura River to survive. These farmers have somehow been able to divide the water among themselves, even though the quantity of rainfall has always been limited and has greatly fluctuated through the years. The fact that the farmers depend on water for their survival and that the water is scarce, explains why conflict and tension about the way the water is divided "has always been just beneath the surface of everyday life" (Ostrom 1990, p. 69). Still, they have been able to create institutions that have survived for more than 1,000 years and which specify and regulate how much water each farmer is allowed to use.

The irrigation systems are a typical example of a common pool resource. First, it is difficult if not impossible to exclude farmers from using water from the irrigation system. The farms are located near the river or a smaller canal, meaning that farmers can appropriate water from the system if they wanted to. Second, consumption of water from the irrigation system is subtractive; a farmer cannot use water that has already been appropriated to and used by someone else. The question therefore is how the farmers have collectively been able to prevent the collective action problem and a depletion of the irrigation water.

In Murcia and Orihuela, the right to use water is tied to ownership of land. In both cities land ownership is highly dispersed, having been assigned long ago and remaining stable for many centuries. The farmers are grouped in communities. In Murcia 30 communities of farmers exist. Orihuela is smaller and counts 10 such communities. The question is, 'How do the farmers, that is, the owners of the land, in each community divide the water between them?' In other words, who is entitled to what share of the resource and when?

The basic principle to divide the water from the irrigation system is to assign each farmer a fixed time slot to withdraw water. The advantage of this system is that farmers know exactly when and for how long they can obtain water. They can thus exactly plan their activities and will probably be quite efficient with the water they are allowed to draw from the common system. The disadvantage of this system is that it is unclear how much water is available to each individual farmer. Furthermore, the system is quite inflexible, "particularly as farms are bought and sold, divided or combined" (Ostrom 1990, p. 76). In a severe drought there is a chance that the procedure will no longer work, simply because there is too little water to go around. In such a situation, officials in each community post a new schedule that indicates which crops are given precedence and the rotation schedule is adjusted accordingly.

To ensure that each farmer upholds and acts according to the rotation system, the communities have appointed guards. These are usually farmers from the community who are employed and nominated by their fellow farmers. In addition to upholding the rotation system, they also help other farmers in distributing and collecting water from the system.

The communities are connected through so-called *huerta*-wide organizations. *Huerta* is the name given to "well-demarked irrigation areas surrounding or near towns" (Ostrom 1990, p. 71). In Murcia the *huerta*-wide organization meets annually to elect the members of an executive commission and approve the annual budget and taxes. In Orihuela the organization meets once every three years "to elect a water magistrate, …lieutenant, and a solicitor" (Ostrom 1990, p. 77).

Finally, both communities have created central water courts, where disputes among farmers from one community or between farmers from different communities are settled. Interestingly, each city has organized its courts in a radically different way. In Murcia an assembly was created consisting of representatives from each of the 30 communities. It settles disputes by majority vote. The mayor of Murcia votes only in cases of a tie. In Orihuela disputes are brought before a single judge, who imposes sentences or tries to bring the involved farmers to an acceptable agreement.

Eight design principles to explain sustainability in the communities

The challenge is to understand why the communities in Murcia and Orihuela are able to manage their common pool resources while other communities are less successful. Having analyzed and compared a number of successful and unsuccessful communities, Ostrom (1990) concludes that the explanation lies in eight so-called 'design principles' (see also Hess & Ostrom 2001, Kollock 1996, Ostrom 2000, Sekher 2001). Each of the eight design principles is considered to be "an essential element or condition that helps to account for the success of these institutions in sustaining the [common pool resources] and gaining the compliance of generation after generation of appropriators to the rules in use" (Ostrom 1990, p. 90). The fact that the design principles are essential conditions means that the principles are believed to be present in all communities in which a common pool resource is managed and in which a tragedy of the commons is prevented. The absence of one of the principles could explain why a tragedy of the commons occurs. Table 2.1 lists the eight principles.

Table 2.1 - Design principles of long-enduring communities[5]

1.	Clearly defined boundaries
2.	Congruence between appropriation and provision rules and local conditions
3.	Collective choice arrangements
4.	Monitoring
5.	Graduated sanctions
6.	Conflict resolution mechanisms
7.	Minimal recognition of rights to organize
8.	Nested enterprise

The eight design principles are characteristics common among the communities that are able to successfully sustain a common pool resource over a longer period of time. These characteristics relate to the institutions of these communities. They are generalized descriptions of the rules that have been crafted and adopted by individuals in the communities. Ostrom (1993) writes, "These eight design principles are quite general. The specific rules that suppliers and users of long-enduring irrigation systems have crafted to meet these principles vary substantially in their particulars" (p. 1910). Thus, the design principles describe the goals or the functions that the institutional arrangements – that is, the working rules – are intended to meet or achieve.

Ostrom is cautious about the principles and claims they are merely a first attempt at building a coherent set of design principles (1990, p. 90). Yet the principles have been adopted and lay the foundation of the work of a large number of researchers (e.g. Buck 1998, Davies 2001, Muchapondwa 2002, Sarker & Itoh 2001, Sekher 2001, Tucker 1998), indicating their continued value and relevance to understand how communities are organized. This conclusion is further strengthened by the observation that Ostrom has continued to use and explain these same eight design principles (Anderies et al 2003, Ostrom 1993, Ostrom 2000).

Introduction to the eight design principles

The first design principle is *clearly defined boundaries*. With clearly defined boundaries the members of the community can be sure that no 'outsider' will reap the benefits of their efforts (Agrawal 2002, Holman 2000, Ostrom 1992, Ostrom 1999, Steins et al 2000). Boundaries thus reduce people's possibility to consume without adding to the resource, an act referred to as 'free riding.' Having boundaries should increase people's motivation to collaborate. Boundaries are also important because they stimulate group members to interact more frequently (Kollock & Smith 1996), which should stimulate coordination.

The second design principle is *congruence between appropriation and provision rules and local conditions*. The community members should define (i) appropriation rules that regulate the number of units of the resource that every insider is allowed to consume, and (ii) provision rules, which regulate what every insider should contribute to the development and maintenance of the resource (Ghate 2002, Ostrom 1999, Ostrom et al 1994, Tang 1992). These rules should be in congruence with the nature of the good and local conditions. Generally, the more complex the community and good, the more difficult it will be to achieve this congruence.

The presence of *collective choice arrangements* is the third design principle. This addresses the need for mechanisms that enable decision making in which every community member is involved and to which they should adhere (Ghate 2002, Ostrom 1990, Sproule-Jones 1998). One thing that involves every group member is the rules that govern community members' behavior. Generally group members are the ones who best understand the problems facing the community and the resource. Therefore, they should be able to adjust the rules. This will increase the chance of a better fit between the rules and local conditions.

The fourth and fifth principles are *monitoring* and *graduated sanctioning of rule violators*.[6] Monitoring and sanctioning is needed to ensure compliance with the operational rules of the community (Agrawal 2002, Buck 1998, Hess & Ostrom 2001, Tang 1992). It is important that group members perform this monitoring themselves and that sanctions remain low for first-time violators. By keeping the first sanctions low, group members acknowledge the fact that in certain situations infraction might be acceptable or even necessary.

The presence of *conflict resolution mechanisms* is the sixth principle. Conflicts will always occur. Certain rules, for instance, will eventually be debated and such a debate should be possible. Without conflict resolution mechanisms debates and conflicts can easily frustrate development and maintenance of the resource and might cause a depletion of the resource and the end of the community (Buck 1998, Smith 1999).

The seventh principle is that *external authorities do not challenge the rules of the community*. To ensure that the rules are effective, the principles must be embedded in a larger legal context. Community members are less likely to obey the rules of the community if they know that external government authorities do not acknowledge and enforce them (Tang 1992).

The eighth principle is *multiple layers of nested enterprises*. This principle addresses the need of the community to organize differently in different situations. It is especially important when the common pool resource is large and complex, in which case various rules and structures may be necessary (Ostrom 1990).

The eight design principles applied to Murcia and Orihuela

Most of the design principles can be found in the example of Murcia and Orihuela. Consider the first design principle, the presence of clearly defined boundaries. In both cities boundaries were drawn based on land ownership. Only farmers who own land are allowed to use water from the irrigation system. Appropriation rules are also present: each farmer is assigned a fixed time slot to withdraw water from the system. However, in a severe drought the system is revised and a new schedule created.

The example also provides a number of hints that collective choice is present. First, there is the presence and appointment of officials. Collective choice must be present to determine how the officials are appointed and what their responsibilities are. Second, the election process for a water magistrate, a lieutenant and a solicitor suggest the presence of collective choice. There must be rules that govern these processes. Third, the nomination procedure for the guards is a collective choice arrangement.

In Murcia and Orihuela guards are appointed to perform monitoring and sanctioning activities. The guards are said to uphold the rotation system. They do so by monitoring and thereby will probably report any infringement of a rule. The example lacks a description of the type of sanctions used to sanction an infraction and whether they are gradual.

Both cities also have a conflict resolution mechanism. In each a central water court was erected where conflicting parties are brought together. The conflict is resolved through a voting procedure, which is different in the two cities.

Ostrom's (1990) description of the irrigation systems in Murcia and Orihuela does not include information about external authorities and their attitude towards the rules and procedures. It is therefore unclear whether external recognition is present.

Finally, the last principle is the presence of multiple layers of nested enterprise. In both cities the communities are nested in multiple layers. First, each community is nested in the city. Furthermore, each community is organized and participates in *huerta*-wide organizations. In the cities and the *huertas* a number of additional organizational processes take place.

Developments in research on common pool resources

Many researchers showed interest in Ostrom's findings and soon a new school of thought started to form. A large part of this school consists of people who either work at or are associated with the Workshop in Political Theory and Policy Analysis at Indiana University (Ostrom 1999). This group soon began to add empirical evidence to the analysis of common pool resources and some, though certainly not all of the researchers, adopted the eight design principles as defined by Ostrom.

Most of the research on common pool resources remains close to the early initiatives, meaning case study research. Studies focus on relatively small communities, which are primarily locally organized, and on mainly natural resources (Ostrom 1990, Ostrom 1998, Ostrom 1999). Among the sectors frequently analyzed are agriculture, fisheries, forestry, grazing, land tenure and use, water resources and irrigation and wildlife.[7] Together, this research has contributed much data to the body of knowledge on common pool resources (Ostrom 1999).

In some of the research the focus is on a single design principle. Research of this type provides empirical evidence of the relevance of the design principles as well as insight into how they are implemented in different situations and under different conditions. An example of such a study is Gefu and Kolawole (2002) *Conflict in Common Property Resource Use: Experiences from an Irrigation Project*. This publication describes the rise of conflicts between the different stakeholders in an irrigation project in Nigeria and analyzes the various modes of conflict resolution (Gefu & Kolawole 2002). Another example that focuses on conflicts and the way in which they are resolved is a study by Ba (2000), which provides a case study of the Delta area and the associated dry areas in Mali.

There is also a large body of research on common pool resources without a direct link to Ostrom's eight design principles. Little of this research uses a specific framework to analyze the management of common pool resources; rather, it analyzes and studies different and innovative questions (e.g. Berge 2003, Berkes 2003, Carlsson 2003). There is also research which adopts another framework, namely the Institutional Analysis and Development (IAD) framework.[8] Many studies have adopted this framework to structure their analysis of institutions and to understand how an institution can affect the action situations facing individual actors (Ostrom 1999).

Yet there is one common denominator in these case studies, namely that they – implicitly or explicitly – question the ability of states and markets to prevent a tragedy of the commons. They, moreover, provide detailed information on how locally devised, self-organized and self-governed institutions are able to solve the collective action problem and prevent a tragedy of the commons.

Today research on common pool resources is no longer exclusively focused on relatively small-sized communities. Instead the research has diversified and gone in a number of directions, most of which can be grouped in one of three categories:[9] the global commons, theory building and the virtual commons.

The global commons

Next to research on relatively small common pool resources, a growing number of studies focuses on what is called the 'global commons.' As the name implies, this line of research looks at natural resources that are not restricted by national boundaries but are global and which would seem to require management and protection on a global level. One book that centers on this type of common pool resource is *The Global Commons: An Introduction* written by Susan Buck in 1998. Among other frameworks she adopts five of Ostrom's eight design principles to present and analyze the institutional frameworks of four global commons. The four commons are the Antarctica, the oceans, the atmosphere and outer space, and telecommunications. In her book Buck primarily focuses on formal institutions that transcend national boundaries. These include international treaties, United Nations conferences and other international bodies.

One difference that sets this type of research apart from more traditional research on smaller-scale common pool resources is its focus on formal and global institutions instead of local communities (see also Farrell & Morgan 2000, Sinha 2000). Yet it nonetheless contributes to the body of knowledge on common pool resources. Buck (1998), for instance, shows that Ostrom's design principles can explain why certain institutions on a global level sustain a resource, whereas others do not. Farrell and Morgan (2000) highlight the importance of local, bottom-up strategies to manage common pool resources, even in a global setting. Like Ostrom (1990) they plead against the idea that more formal top-down control provides the only logical solution to assure the continuity of common pool resources. This attention to the interaction between a global level of a decision making and a local, decentralized level of rule conformance is at the heart of most research on global common pool resources (e.g. Berkes 2000, Karlsson 2000, Mudiwa 2002).

Theory building

A growing body of research aims to develop, build and improve theory on common pool resources. One observation concerning research on the management of common pool resources is that it is dominated by individual case studies, which have a high level of variety and do not share a clear theoretical basis. There is, however, also research that uses the findings from the case studies and has as its goal to improve the theoretical underpinnings of research on the management of common pool resources. Examples of such studies are Agrawal (2002), Oakerson (1992), Rose (2002), Falk et al. (2002) and Ostrom et al. (1994). These authors' efforts to bring maturity to research on common pool resources is evidenced in four ways (these four ways and their explanation are adapted from Stern et al 2002).

First, there are many initiatives to develop standard typologies related to a wide variety of aspects of common pool resource management. These typologies "allow researchers to focus attention on a tractable number of variables and then to state and systematically examine research hypotheses about them" (Stern et al 2002, p. 446). Second, there is a growing focus on contingent research hypotheses. The example given in Stein et al. (2002) addresses research that aims to understand how the various forms of variety impact the ability of communities and agencies to manage a common pool resource. Third, the research increasingly aims at identifying and understanding causal relationships between variables. Fourth, there are many attempts to integrate the results from the wide variety of research being done. Stein et al. (2002) argue that single case studies are not appropriate vehicles to achieve this integration. A better method to achieve such integration is, for instance, controlled experimental research (e.g. Walker et al 2000).

The virtual commons

Finally, a growing body of research analyzes the so-called *virtual commons* or *information commons*. This line of research adopts concepts and the associated research approach from traditional common pool resources to analyze goods, infrastructures and services in a digital, networked environment. Most of this research focuses on specific types of virtual commons, namely, those that seem to be most similar to more traditional common pool resources.

One of the best examples of such research is by Hess and Ostrom (2001). They extensively discuss the applicability of the thoughts about and principles of common pool resources to different types of information. They argue that much research on the virtual commons adopts the name 'commons' without actually referring to a common pool resource. They use the term to refer to public goods, as the consumption of the good is not subtractive but joint. There are many examples of such research (Berkes 2000, Bollier 2001a, Bollier 2001b, Kollock & Smith 1996). However, next to this research is also research that does actually study the Internet as a common pool resource. Such research either focuses on (i) the technological infrastructure or (ii) social networks (Hess & Ostrom 2001, p. 64).

An example of the first type of research is that by Noonan (1998). He argues that the adoption of thought from the study of common pool resources shifts the focus to those parts of the Internet that are prone to problems of congestion. "The metaphor of 'information superhighway' is particularly apt here, because both resources present commons prone to 'traffic jams.' Too many users can overload different links in the network chain, reducing the value of other transmissions congested at that point" (Noonan 1998, p. 189). Along a similar line of reasoning, Bernbom (2000) adopts the analogy and argues that especially the physical network infrastructure should be viewed and analyzed as a common pool resource that is highly vulnerable to congestion.

Research that considers social networks on the Internet as a common pool resource is scarcer. One example is by Kollock (1996) and Kollock and Smith (1999), who argue that the eight design principles are a promising starting point for analyzing and constructing virtual communities. One could also argue that O'Mahony (2003) is an example of the second category of research. She argues that source code in open source communities shares characteristics of common pool resources and therefore the communities need to defend themselves against external pressures.

Discussion: the applicability of research on common pool resources

A number of questions can be raised about the adoption of a research framework based on the lessons of community-managed common pool resources for analyzing the organization of open source communities. Two of these questions are particularly relevant here.

Is source code a common pool resource?

A first and important question is whether the resource in open source communities is a common pool resource. Software and the corresponding source code are commonly considered to be public goods (Kogut & Metiu 2001, Kollock 1999, Kuwabara 2000, Lakhani & Von Hippel 2003, Ljungberg 2000, Markus et al 2000). The consumption of source code is *joint*, because one's use of source code does not affect the amount available to others. In other words, the source code of software can be copied indefinitely (Shapiro & Varian 1998). *Exclusion* of users is difficult. Every copy on a compact disc, website or peer-to-peer network can be multiplied without loss of quality. The fact that source code is a public good would suggest that lessons from research on common pool resources are inappropriate.

However, as briefly discussed in the previous chapter, there is at least one reason why open source software is a common pool resource, or at least, why it "shares some features with nonrenewable resources or common pool resource problems" (O'Mahony 2003, p. 1181). The presence of intellectual property rights allows organizations and individuals to appropriate software from the public domain and treat it as private property (Boyle 2003, Hunter 2003). This creates a threat to the future availability of open source software, as the source code is then no longer free to be shared among the participants in the communities.[10] To protect the future availability of source code, the communities require "more protections than those offered by the public domain. To remain open and publicly available, it must be protected from proprietary appropriation… Use of it will not diminish in the present, but the future stream of benefits is at risk" (O'Mahony 2003, p. 1182).

Thus, the presence of intellectual property rights enables the appropriation of source code and introduces a form of consumption of the software that is subtractive. Open source communities must create ways to protect themselves against the appropriation of source code and hence to ensure the continuity of open source communities (also Van Wendel de Joode 2004a, Van Wendel de Joode 2004c).

Are open source communities like communities in which traditional common pool resources are managed?

A second question stems from the differences between the communities. Open source communities are relatively large, connecting up to thousands of people. The members of an open source community are geographically dispersed and they are relatively independent of each other and of the resource. Most research on community-managed common pool resources focuses on much smaller communities that are geographically concentrated, with community members depending on the resource for their survival (Hess & Ostrom 2001). These differences give rise to the question of whether these types of communities can be compared.

One argument to claim that such a comparison is useful is provided by Kollock (1996). He writes that online communities face challenges of social interaction and it is all but understood how communities should solve these challenges. He propagates the design principles by Ostrom as shedding light on how communities might be able to solve such social challenges. In a sense, this would imply that the design principles are relevant to open source communities, even though open source communities differ from the communities described in community-managed common pool resources. The differences do raise a number of methodological issues, which will be discussed in the next chapter.

Notes on chapter two

[1] Page 33 provides some of the arguments that explain why open source software should be analyzed as a common pool resource.
[2] According to Runge, the difference between open access and a common property regime is not as visible as sometimes suggested by theory. “Often, what appears to the outside observer to be open access may involve tacit cooperation by individual users according to a complex set of rules specifying rights of joint use. This is common property.” (Runge CF. 1992. Common Property and Collective Action in Economic Development. In *Making the Commons Work; Theory, Practice, and Policy*, ed. DW Bromley, pp. 17-41. San Francisco: ICS Press).
[3] The selection of these two quotes is based on Ostrom who uses the same quotes to define complex adaptive systems in Ostrom E. 1999. Coping With Tragedies of the Commons. *Annual Review Political Science* 2: 493-535
[4] The term ‘actor’ or ‘agent’ refers to more than just individuals. Actors and agents refer to “a unit engaged in some kind of undefined action. It may be an individual, a department, an organization, a tier of government or even a nation state”; page 84 in De Jong M. 1999. *Institutional Transplantation: How to adopt good transport infrastructure decision-making ideas from other countries?* Delft: Eburon
[5] Adapted from Ostrom (1990).
[6] Ostrom (1990) also discusses principle 4 and 5 together.
[7] From the website: http://dlc.dlib.indiana.edu/view/subjects/ (June 2004).
[8] The IAD framework consists of the following rule sets: 1) boundary rules, 2) position rules, 3) authority rules, 4) scope rules, 5) aggregation rules, 6) information rules and 7) payoff rules (Ostrom E. 1999. Coping With Tragedies of the Commons. *Annual Review Political Science* 2: 493-535, Ostrom E, Gardner R, Walker J. 1994. *Rules, Games, and Common-Pool Resources*. Ann Arbor: The University of Michigan Press). The difference between the IAD framework and the eight design principles is that the IAD framework

provides a method to analyze all sorts of institutions. In a sense it is a generic and descriptive analytical framework. In contrast, the eight design principles are prescriptive and much more focused on self-organizing communities in which common pool resources are managed. For these two reasons the eight design principles are adopted in this research and not the IAD framework. Next to differences there are also many similarities. For instance, the first cluster of rules of the framework is “boundary rules affect the characteristics of the participants.” (Ostrom E. 1999. Coping With Tragedies of the Commons. *Annual Review Political Science* 2: 493-535) The first design principle states that communities must have clearly defined boundaries. It is obvious that both refer to the same issue.
[9] These four categories have been reformulated and regrouped based on the information from two sources, namely the collection of articles and research papers available on the digital library on common pool resources (http://dlc.dlib.indiana.edu/, June 2004) and the book *The Drama of the Commons* (Ostrom E, Dietz T, Dolšak N, Stern PC, Stonich S, Weber EU, eds. 2002. *The Drama of the Commons*. Washington: National Academy Press). Obviously, these four categories are artificially chosen and there will be many examples of papers that fit within more than one of the four categories; there will also be papers that fit in none of the categories. The categorization does create a framework to position some of the major trends in research on common pool resources.
[10] This point also came up in many of the interviews that were held during this research.

CHAPTER THREE

METHODOLOGY

There is an apparent lack of understanding of the organization of open source communities. The goal of this research is to synthesize a new organizational model from key elements in the current literature, as well as new empirical data. The model is intended to serve as input to discussion and reflection and as a guide for further research on the subject. To create such a model, this research first aims to provide an understanding of the processes and structures in the communities. In other words, it is explorative and adopts qualitative methods. There are a number of reasons why this approach is taken.

First, the first chapter of this book already argued that the current state of the art on open source communities is unable to answer the question of how the communities are organized and how they are able to sustain themselves. One reason was that open source communities are still new and therefore relatively unexplored. Another reason was that previous research has primarily resulted in a number of unrelated and sometimes mutually exclusive answers. Thus, a commonly accepted understanding of the structure of and processes in open source communities is lacking. Because of this lack, sense-making and interpretation become an important part of any research on open source. These two activities belong to the domain of qualitative research (Denzin & Lincoln 1994) and they make quantitative research less appropriate.

Second, it is not clear what exactly an open source community is and what it is not. The first chapter argued that open source communities appear to contradict many 'traditional' organizational structures and processes. According to Aldrich (1999), the basic characteristic of any organization is the presence of boundaries. However, the introduction of this book already argued that open source communities generally lack contracts that bind the individuals to a collective. The fluidity of the boundaries makes open source communities hard to define and to grasp, which is another reason why a qualitative research method is appropriate.

The next pages present and describe the research strategy, the methods of data collection, the way in which the data was analyzed and the structure of presentation of the observations and conclusions.

Research strategy: the design principles as a conceptual answer

Basically, the answer to the research question is believed to lie in the design principles, which should yield a conceptual answer to the research question. According to Ostrom (1990), the design principles form a coherent set of essential elements or conditions that explain[1] why certain self-organizing communities are able to coordinate individual efforts and sustain a resource, generation after generation. The principles are transferred to open source communities to understand how the communities organize and sustain themselves. They are used as guidance; as a way to decide where to look for mechanisms to understand how the communities are organized. The principles provide a heuristic to give meaning to processes, structures and mechanisms. The design principles act as propositions, which means that they are used as a way to decide where to look for an answer to the research question (Yin 1989).

The design principles in a slightly revised form

Table 3.1 – The framework

1.	Clearly defined boundaries
2.	Congruent appropriation and provision rules
3.	Access to conflict resolution mechanisms
4.	Access to collective choice mechanisms
5.	Presence of monitoring and graduated sanctioning
6.	Multiple layers of nested enterprise
7.	External recognition

The theoretical framework adopted here is based on the lessons from research on community-managed common pool resources. Specifically, the eight design principles are adopted, as first identified by Ostrom (1990). Table 3.1 lists the design principles once again. The design principles in this table are a somewhat simplified version of Ostrom's original principles. Furthermore, some of the design principles have been rearranged. There are a number of reasons for doing so. First and most visible, the design principles *monitoring* and *graduated sanctioning* are combined into one design principle, as in Ostrom (1990). The reason for combing them is that both principles are highly related and they address the same issue, namely, ensuring that the rules as defined by the community are adhered to by the individual members of the community. Second, the ordering of the design principles has been somewhat changed. The principle *conflict resolution mechanism* is placed before the principle *collective choice*; and the principle *external recognition* is moved to the back to become the last design principle. The reason for altering the ordering is a practical one. The ordering of this book is based on the design principles and the reordering improves the readability and understandability of the individual chapters.

The research question and sub-questions

The main question in this research is:

How are open source communities organized and how do they sustain themselves?

The research question can be divided into seven sub-questions:

1. How is the principle *clearly defined boundaries* addressed in open source communities?
2. How is the principles *congruent appropriation and provision rules* addressed in open source communities?
3. How is the principle *access to conflict resolution mechanisms* addressed in open source communities?

4. How is the principle *access to collective choice mechanisms* addressed in open source communities?
5. How is the principle *presence of monitoring and graduated sanctioning* addressed in open source communities?
6. How is the principle *multiple layers of nested enterprise* addressed in open source communities?
7. How is the principle *external recognition* addressed in open source communities?

The first chapter explained that state-of-the-art research on open source communities is divided on the question how the communities are organized. One line of research focuses on a collective level and they argue that institutions can explain how the communities are organized. Another line of research focuses on the actions of individual developers; they seek an answer on the level of the individual. The aim of this research is to reconcile the two lines of research and the analysis will therefore include both levels.

A selection of communities

There are many open source communities and all appear to be different. For instance, some communities are said to have just one project leader, while others have a board or management committee. Yet despite these differences "they all follow some basic rules that allow them to interconnect" (Costigan 1999, p. xviii). The challenge in this research is to identify and understand the basic principles of the organization of open source communities, at the same time realizing that there is no such thing as *the* open source community. To identify and understand these basic principles this research focuses on a selection of communities. The selected communities are relatively large, involving many contributors ranging from hobbyists to multinationals. Their software is widely adopted and used, consisting of hundreds of thousand or even millions of lines of code. Both the communities and software receive coverage in a wide variety of media. Finally, the communities are relatively old compared to other communities that one might find on a website like SourceForge, because they have all existed for at least seven years and one for more than twelve years. The age and the size of the communities implies that they have gone through various stages of change and growth, "part of the evident success of (say) an operating system like GNU/Linux must be related to the way this software adopts to a changing technical reality community, as well as with increasing contributors to the project" (Bauer & Pizka 2003, p. 171). In short: these communities are the most critical cases in light of the research question. They consist of many participants, have many sub-projects, the software consists of many lines of source code and interdependencies in the source code are complex.

The most important reason for analyzing relatively large communities is that for larger communities the chance is higher that the design principles identified in Ostrom (1990) are present. The design principles are said to explain why communities are able to *sustainably* manage a common pool resource. Adopting the design principles to understand the organization of open source communities thus means that the communities should also to some degree meet this criterion. To Agrawal (2002), sustainability of community-governed common pool resources is a measure of the "durability of institutions that frame the governance of common-pool resources" (p. 44). Furthermore, sustainability is said to be related to the resource as well as the institution that manages the resource (Dietz et al 2002, Sekher 2001). Although a definition based on durability is still relatively vague and open for different interpretations, it does indicate that sustainability is related to time and can be at least partly based on the age of the community and the resource (Costanza et al 2001). In community-managed common pool resources, sustainable communities are typically those that exist for more than 100 years. However, this time frame is hardly applicable to open source communities. The environment of software development is "characterized by high levels of uncertainty, ambiguity and ignorance" (Dalcher 2003, p. 423). Software development projects have to make do with "knowledge that is elusive, tacit, incomplete, ephemeral and ambiguous" (Dalcher 2003, pp. 422, 423). Due to complexity and high dynamics, lifecycles in software development are extremely short. A period of five years[2] might be considered short in terms of traditional common pool resources, but it is long in terms of software development and maintenance. A community that exists for more than five years can by these standards be considered old and perhaps even sustainable. This is especially true when we contrast this time frame with the many communities that are active for just a limited number of months.[3]

Furthermore, analysis of the bigger communities is likely to yield more interesting observations. Compared to smaller communities, the bigger communities face many more challenges in collaboration and coordination. The large size of these communities implies that members are more geographically spread, and thus have different backgrounds and goals as to what the software is supposed to do. This is likely to result in a greater number of conflicts, which must somehow be dealt with. Most of the larger communities also create and maintain software that is generally larger in lines of source code and more popular in commercial uptake. This again poses serious strains on the organization of the communities. The software is more complex and thus development and maintenance are more challenging. Commercial entities have different requirements for the software than some other users, and they will somehow want to include these in the next release. To summarize, for larger communities organizing and sustaining themselves is believed to be a bigger challenge and thus more fascinating to analyze and try to understand.

Short introduction of the selected communities

The observations and findings in this research focus on the analysis of a selection of communities and foundations. They are the Apache community, the Linux kernel community, the Debian community, the Python community and the PostgreSQL community. Next to these communities a number of interviews were performed with developers and users (i) from smaller communities and (ii) from other large communities. These communities are not described here, as in general only one interview was performed. The five communities described below are ones that were analyzed over a longer period of time and using at least three data sources (see appendix B).

The *Apache* community is probably the most successful of the five projects in terms of adoption of the software.[4] The foundation of the Apache Web server was laid by the National Center for Supercomputing Activities (NCSA) in the United States. The NCSA server was initially adopted by a small group of software programmers. They used the software at their companies and were granted NCSA permission to improve it. A mailing list was created and a community of programmers began to emerge. Through the years the community attracted more and more programmers who were interested in the software and it started to adopt tools to better manage the individual contributions. In 1998 IBM decided to replace its existing systems with Apache. Later, many more companies followed in adopting the Apache software. The figures slightly differ depending on the source, but over the years Apache's market share appears to fluctuate around 65 percent. Furthermore, the software now includes a rapidly rising number of lines of code and the community is split into a great number of different projects.

The development of the *Linux kernel*[5] started in 1991 with an Internet message on a mailing list from its original creator Linus Torvalds inviting others to take a look at a small software program he wrote. In his message Torvalds provided the source code of the kernel and invited other people to suggest improvements. At first ten people downloaded the source code and five sent back bug fixes, code improvements and new features (Naughton 1999). Torvalds then took the time to review the responses and explain why he chose to ignore or incorporate a suggestion. Many people on the Internet were attracted to the software. As a result, a thousand people had downloaded the Linux kernel only a year after Torvolds' original message. By then, the kernel had become a functional operating system that counted 40,000 lines of code.[6] From then on, the development of Linux went amazingly fast. In less than ten years Linux became a reliable product and an alternative to Microsoft (Wayner 2000). Currently it provides the basis of a number of commercial software distributions and is adopted by large multinationals and improved by thousands of volunteers and paid programmers.[7]

Debian is claimed to be the only significant Linux distribution that is not commercial.[8] The start of Debian was marked in 1993 by Ian Murdoch, whose aim was to move to an open distribution of Linux. At first, a small group of volunteers collected a set of rather randomly selected packages. Parallel to this effort Murdoch spent much of his time and effort to create an environment in which it would be easy for volunteers to contribute. A community began to evolve which focused on a limited selection of items that were believed to form the core of the distribution. Furthermore, an organizational structure was created, of which a system of rotating leadership is perhaps the most visible attribute. Murdoch remained leader of the community until March 1996, when Bruce Perens took up the leadership role. Currently, as of 2004, Debian has its eighth project leader and has attracted thousands of developers. The Debian distribution now consists of thousands of packages.

Python is a programming language created by Guido van Rossum at the Dutch Center for Mathematics and Computer Science (CWI).[9] He created the first version of Python in response to the languages that were available back then. Posting the software on the Internet attracted the attention of a number of other programmers and together they improved the software. A community started to form, which Van Rossum named Python. In 1995 he moved to the United States where he continued to lead the development effort. Currently, the language is widely recognized and adopted by companies, schools and governments.

The fifth community is the *PostgreSQL* community. PostgreSQL is a database software program that started out as a prototype at a California-based research institute. The institute saw no real use in it and almost entirely abandoned the project. In 1995 a single worker maintained the software. That was when Bruce Momjian discovered the software and decided to participate in its development.[10] Like the other communities, PostgreSQL started to attract other programmers who contributed to its development. Currently, PostgreSQL is one of the most popular database programs and involves hundreds of programmers and other contributors.

Methods of data collection

There are two reasons to adopt multiple strategies in parallel to collect data on open source communities. (i) Most of the interviewees are highly professional and have much tacit knowledge (Nonaka & Takeuchi 1995, Schön 1983). The interviews demonstrated that the interviewees lack a common understanding of how the communities are organized. Reliance on interviews alone could generate a false image of the communities. (ii) There is no common understanding among researchers of open source communities. See the first chapter for a more elaborate discussion of the differences in state-of-the-art research on open source communities.

The four sources of data used in this research are: interviews, secondary literature sources, direct observation and archival records. Most of the empirical data was gathered between October 2001 and September 2002. For each community a minimum of three sources of data was used. This means that the data from each of the communities meets the requirement of data triangulation (Janesick 1994, Stake 1994, Yin 1989).

Interviews

Sixty in-depth interviews were held. Many of the interviews were long; some of them took more than four hours. All except two interviews were face-to-face. In-depth interviews were used due to the lack of a shared understanding among the interviewees as to how open source communities are organized. Under these conditions the use of questionnaires would have been less applicable. In-depth interviews made it possible to move beyond the level of metaphors and abstract observations. Most interviews generated a high level of detailed information about the organization of open source communities.

Appendix A presents the list of interviewees. Most were programmers and contributors from at least one open source community; they constituted 37 respondents in total. Of them, four are or were project leaders in an open source community. Respondents who are not programmers had some other relationship with open source communities. The two largest groups are managers or employers in companies that had adopted open source software but did not contribute to open source communities, 14 in total, and people who write about open source communities, 7 in total. Of the latter group three interviewees were doing PhD research on open source communities and two were editors in chief of open source journals.

Most of the interviews were conducted in Dutch. Five interviews were done in German. About half of the respondents, 29 in total, were interviewed in the United States. Three of these interviews were in Dutch; the other interviews were in English. Throughout this book, quotes are used as part of the argumentation. The Dutch and German quotes have been translated into English.

The interviews were semi-structured with open-ended questions. This means they offer a mix of elements from unstructured and structured interviewing techniques. No questionnaires were used that had "preestablished questions with a limited set of response categories" (Fontana & Frey 1994, p. 363). Neither were the interviews unstructured, as the questions were not based on observations only (Fontana & Frey 1994). Instead, they were semi-structured, focusing on the functions as they are described in the eight design principles. For example, related to the design principle 'monitoring,' respondents were asked questions like 'What do you feel is counterproductive behavior and why?' Does this type of behavior occur regularly? How do you know this type of behavior occurs? Do you regularly check whether someone else acted this way? Based on the answers from previous interviews, as well as responses given during the actual interview, the background of the respondent and data collected from other sources, the respondents were asked more specific and detailed questions about certain issues (in light with the process described in Travers 2001).

All but three interviews were done in person. Three were done via e-mail. The personal interviews held in the Netherlands and Belgium were all taped and later transcribed. This was also done for one interview in Germany. The other interview in Germany and the interviews in the United States were concurrently written down by a research assistant. As soon as possible after the interviews were held, I read these transcripts and added my field notes to them.

In total, four of the interviews were so-called 'group interviews' (see Fontana & Frey 1994). These four were also semi-structured.

Secondary literature sources

Open source communities are heavily analyzed and researched objects of study. One of the primary sources of research papers, articles and books is the open source website hosted by the Massachusetts Institute of Technology (MIT).[11] The site was created at the end of 2001 and at the time of this writing lists the names of 444 people who are, for different reasons, interested in open source communities. Of these, 278 are connected to a university or other kind of research institute. The number of papers listed on the site is 147. Next to this compilation of papers, an increasing number of articles on open source are being published in journals, books and conference proceedings. Many of these provided inputs for this research.

Direct observation

Direct observations were performed in a variety of locations and settings. One important source of direct observation was at conferences, workshops and so-called 'user meetings.' Three large open source conferences were attended, namely the O'Reilly Open Source Convention in San Diego in July 2002, the Open Source and Free Software Developers (FOSDEM) in Brussels in February 2001 and the Holland Open Software Conference in Amsterdam in May 2005. Next to these three large conferences, many smaller seminars in the Netherlands were attended. One such meeting was a yearly members' meeting of the Dutch Society for Open Source (VOSN) held in Utrecht (the Netherlands) in 2001. At the conferences and seminars, I listened to presentations by a wide variety of people who were somehow connected to one or more open source communities. Furthermore, the interactions among the participants were observed. Field notes were drawn up during and after the meetings.

A number of interviews were held at the programmers' workplaces. In seven interviews the respondents demonstrated their actual activities, for instance, showing how they modified source code and returned the source code to the community. They also explained and demonstrated how they interact with others in open source communities.

Open source programmers and users frequently organize what are called user meetings. These are small workshops or gatherings at which a wide variety of people share their experiences, ideas, the latest gossip and news. The author attended three such meetings, namely those organized by the Utrecht Linux User Group, the Polder Linux User Group (PLUG) and the Greater New Hampshire User Group.[12] At these meetings field notes were made.

Finally, the author visited the physical location of the Free Software Foundation (FSF). As explained later in this book, the FSF performs an important role in open source communities. The visit to the FSF gave an impression of the status and setting of the foundation and its role vis-à-vis open source communities.

Archival records

Many activities in open source communities take place on the Internet. These activities are usually archived. For instance, both the actual uploading of a new piece of open source software and the e-mails on mailing lists are archived. These archives were extensively analyzed for this research. In a sense, there is no clear distinction between analysis of archival records and direct observation. This is because though the Internet enables actual, real-time observation of e-mail correspondence between programmers, much of the analysis took place in retrospect. E-mail correspondence is therefore categorized under this heading.

The most important information source on the Internet were the resources directly linked to a particular community. Especially the weekly summary of the major happenings on the Linux kernel mailing list were studied extensively in the period between November 2001 and June 2002. A log was created of relevant observations.

Data analysis

The primary goal of data analysis in this research is to retrieve the substantive meaning in the data[13] and create a link between the data and the theoretical framework. This was done in several steps: (i) making sense of the empirical data, (ii) relating empirical concepts to the theoretical concepts and (iii) checking the findings.

Making sense of empirical data

Data analysis was started in parallel with the data collection process. A large part of this process was to reduce and make sense of the empirical data. Another substantial part was the creation of a living document in which the most recurring data from the data collection phase was grouped. This grouping resulted in the identification of a great number of concepts.[14] These concepts were rather loosely defined and remained close to the terms used by the respondents (see for instance Spencer et al 2003). An example of such a concept is *elegance*. This concept came up in a number of interviews, in discussions on mailing lists and in the secondary literature. The challenge became to understand what the concept meant for the organization of open source communities. Does it have a role? And if so, what? In the subsequent interviews the concept of elegance was then brought up and the interviewees were asked what elegance meant to them and what they perceived as the role of elegance. As such, a better image of elegance and its role in the organization emerged. A similar process was used to better understand other concepts like modularity, project leadership, voting systems and open source licenses.

Relating empirical concepts to the theoretical concepts

At the time the research proposal was finished (September 2001) the seven tentative design principles had been defined in correspondence with Ostrom's eight design principles. Seven separate documents, one for each design principle, were created to collect the findings and relate them to the principles.

To structure the findings, the design principles were first defined and operationalized. For example, for the design principle *monitoring and graduated sanctioning* a theoretical investigation was conducted to analyze what monitoring and sanctioning are, how they are defined in the literature and why they are needed. This led to a definition of the terms, which made it possible to identify which of the empirical data were related to the design principle monitoring and sanctioning and which not.

The second step was to look for institutions in open source communities that might fulfill the function of a design principle. An example is the presence of *project leaders*. In some communities the role of project leaders is highly institutionalized. For instance, in the Debian community yearly elections are held to appoint the new Debian project leader.

The third step, if an institution was found to be absent, was to look for alternative ways in which the function as addressed in the design principle might be fulfilled. In some instances, data and concepts could be easily connected to just one of the design principles. In other cases, the process proved much less straightforward, because (i) some concepts seemed to have a role in more than one principle and (ii) a number of issues were difficult to relate to a design principle. One example is the Debian community's so-called *orphanage*. As two interviewees explained, the orphanage is a virtual place where package maintainers can drop their package if, for whatever reason, they are no longer interested in maintaining it. Others in the Debian community can now take on the role of package maintainer if they so choose. The question was then, 'What role does the orphanage play in the Debian organization?' Analyzing the use of the orphanage and the transcripts in combination with the design principles resulted in the conclusion that the orphanage enables and facilitates the provision aspect of Debian.

The fourth step, if no alternative to the institution could be found, was to understand whether and why the absence of a function would be a problem.

A check of the findings

There were a number of checks to verify the findings and conclusions of this research. First, the knowledge and information in this research were accumulated through a process of feeding them back into the data collection. This is especially true for the findings generated from the interviews. The findings from each interview were immediately written down and then used in the subsequent interview. Respondents were asked their opinion about previous findings. They were asked to explain if and why they agreed with a finding from a previous interview. Thus, interviews were used to check the information and findings generated from previously held interviews, resulting in an accumulation of knowledge.

Second, most of the information and findings gathered through one data source were cross-checked with other sources. Observations from interviews, for instance, were cross-checked with data from mailing lists, secondary sources of data, etc.

Third, the overall conclusions of this research were presented and checked in two different academic settings. The first was the semi-annual conference on common pool resources in Mexico in August 2004. The second was an international scientific workshop on open source software and open source communities in Paris, also in August 2004. Both occasions resulted in feedback, which was incorporated in a new version of the conclusions. Parallel to this trajectory the main findings and conclusions were sent to scientists who are experts in a variety of disciplines and have an interest in open source communities. They too provided comments on the findings of the individual chapters and the overall conclusions. Seven researchers provided comments and suggestions for improvements.

Method of presentation: structure along the design principles

The structure of this book is based on the design principles that are central to this research. There are a number of reasons for choosing this presentation. First, this structure allows a focus on each of the principles. The principles are believed to be the factors that shed light on the organization of open source communities. A focus on each individual community would quickly zoom into the specifics of that community and lose sight of the underlying principles. Second, the field research showed that while there are many small differences between the communities, in general the communities are highly comparable. Because they share many characteristics, a structure that focuses on the communities individually would produce a much less interesting account. Third, much of the secondary literature is clustered around themes and topics, more so than around individual cases. A thematic structure along the seven design principles thus provides a more logical connection with existing literature.[15]

Structure of the individual chapters

Each chapter starts with a theoretical investigation of the design principle under scrutiny. The goal is to get a feeling for the design principle and the functions it addresses. These investigations are primarily based on research on common pool resources and descriptions of the design principle. Some principles, however, are grounded in other theories or are also addressed in other disciplines. For this reason, some chapters supplement the description of the design principle with ideas and knowledge from other disciplines.

The chapters demonstrate how the design principles address a certain organizational problem, for instance, conflict resolution. In this case, the design principle conveys the fact that the presence of conflicts can constitute a threat to continued coordination among community members. To understand how the design principle manifests in open source communities, the first step is to establish whether and how conflicts are present in open source communities and how they manifest. The largest part of each chapter is devoted to an analysis of the way in which the design principle, and its inherent aspects, are implemented and/or addressed in open source communities. This entails a procedure of giving meaning to the mechanisms and processes that were extracted in the data analysis. Each chapter ends with a conclusion.

Notes on chapter three

[1] Ostrom does not explicitly use the term 'to explain' in her definition of design principles. However, she does use the design principles *to explain* why certain communities have failed – or are prone to failure in the future – to sustain the organization and/or the resource in chapter five, Analyzing Institutional Failures and Fragilities, in Ostrom E. 1990. *Governing the Commons; The Evolution of Institutions for Collective Action.* Cambridge: Cambridge University Press.

[2] The period of five years is based on an interview with Lawrence Lessig who explained that five years is long in a market that is dynamic and in which product lifecycles are extremely short. The period of five years is, however, arbitrary and does not mean that communities that exist less than five years are automatically unsustainable. The number is used solely as a means to highlight that (i) software development is characterized by lifecycles that are much shorter than those in traditional common pool resources and (ii) open source communities that exist for, say, less than a year have not 'proven' themselves sustainable.

[3] In many open source projects listed on the SourceForge website (http://sourceforge.net/ accessed March, 2004) developers were active for a very limited period of time.

[4] This account of how the Apache community started is primarily based on a personal interview with one of the members of the Board of Directors of the Apache Software Foundation. He was involved in the community almost from its beginning.

[5] A well-written kernel can be compared with a fine hotel in which the guests are the software, the physical hotel is the hardware and the kernel is a "combination of the mail room, boiler room, kitchen, and laundry room for a computer… The guests check in, they're given a room, and then they can order whatever they need from room service and a smoothly oiled concierge staff. Is this new job going to take an extra megabyte of disk space? No problem, sir. Right away, sir. We'll be right up with it. Ideally, the software won't even know that other software is running in a separate room" (Wayner P. 2000. *FREE FOR ALL: How Linux and the Free Software Movement Undercut the High-Tech Titans.* New York: HarperBusiness).

[6] Josh McHugh (1998) *Linux: The Making of a Hack*, a Forbes article taken from the Internet: http://www.forbes.com/forbes/1998/0810/6203094s1.html (July 2001).

[7] See the introduction of this book for a more elaborate discussion of the market impact of Linux.

[8] See http://www.debian.org/doc/manuals/project-history/ch-intro.en.html (September 2004).

[9] This account is based on a personal interview with the creator of Python, Guido van Rossum.

[10] This account is primarily based on a personal interview with Bruce Momjian, who is one of the founders of the community and is still on the steering committee.

[11] See http://opensource.mit.edu/ (March 2004).

[12] Although the names might suggest differently, the two Linux user groups do not limit themselves to discussions about the Linux kernel.

[13] According to Spencer et al. (2003) the primary focus of analysis in public policy generally is to capture and interpret substantive meaning in the data. (Spencer L, Ritchie J, O'Connor W. 2003. Analysis: Practices, Principles and Processes. In *Qualitative research practice: A guide for social science students and researchers*, ed. J Ritchie, J Lewis, pp. 199-218. Londen: Sage Publications).

[14] These concepts are similar to what is known as *sensitizing concepts* (Blumer H. 1954. What is wrong with social theory. *American Sociological Review* 19: 3-10), which "give a general reference to empirical instances, later developing into more analytical, definitive concepts", (Spencer L, Ritchie J, O'Connor W. 2003. Analysis: Practices, Principles and Processes. In *Qualitative research practice: A guide for social science students and researchers*, ed. J Ritchie, J Lewis, pp. 199-218. Londen: Sage Publications).

[15] A similar structure can be found in Kaufman H. 1981. *The administrative behavior of federal bureau chiefs.* Washington: The Brookings Institution.

CHAPTER FOUR

BOUNDARIES

This chapter addresses the first design principle, *clearly defined boundaries*. The first section discusses the importance of boundaries in common pool resources. The second section explains why boundaries are needed to protect the communities and their software against a number of threats.

The sections that follow identify and discuss the boundaries present in open source communities. This discussion is oriented toward a number of empirical observations. The first observation is the presence of a trade-off, namely that of attraction versus protection. Participants want (i) to attract other participants and companies to work and improve their source code and (ii) to protect the source code from individuals and companies who want to appropriate it from the communities.

The second observation is that open source communities lack boundaries to limit their size. They have some minimal entry barriers, like a sufficient level of knowledge, but these hardly constitute the type of boundary suggested in the literature on common pool resources. Nor do such boundaries resemble those found in most companies. Commonly used boundaries like contracts or other such formal agreements are generally absent.

The boundaries created and adopted in open source communities center on the resource itself. Three types of boundaries are presented and discussed. The first is the open source license, of which an enormous variety have been created in the communities. To demonstrate the variety, several of the licenses are discussed. A more elaborate overview of the licenses is presented in appendix C. What is striking is that the licenses are not used as a static mechanism; they are not treated as finished products. Instead, they are adapted and updated to better meet changing situations and new challenges, and to modify the trade-offs made between attraction and protection of the source code. The second type of boundary is the presence of community websites to establish and enforce the boundaries of the source code and to educate participants about the implications of the boundaries. The third type of boundary is the 'open source' foundation. Many communities have erected foundations to protect the software and contributors from legal threats and to represent the communities externally.

The first design principle: clearly defined boundaries

Boundaries are considered to be a defining characteristic of organizations and other types of collectives (Aldrich 1999, p. 3). For example, "A minimal defining characteristic of a formal organization is the distinction between members and non-members, with an organization existing to the extent that some persons are admitted, while others are excluded, thus allowing an observer to draw a boundary around the organization" (Aldrich & Herker 1977).

A couple of elements in this definition warrant further investigation. The first is that an observer is said to be able to draw the boundary. In other words, the boundary does not necessarily have to be defined by the people in the organization. Understanding who is included and who is excluded allows the observer to judge the size of the organization and to determine its boundaries. Another element of the definition is the difference between members and non-members. Aldrich and Herker (1977) argue that this distinction is enabled by the fact that members are people who have been admitted to the collective, whereas non-members have not.

In his book *Organizations Evolving*, Aldrich argues that many businesses and volunteer organizations actively enforce the membership distinction and thus actively enforce their boundary rules. His examples include human resource departments and membership committees. Furthermore, he argues that members frequently have distinctive symbols of membership, like unique modes of dress or special vocabularies (Aldrich 1999, p. 3).

Boundaries in research on community-governed common pool resources

In her research, Ostrom (1990) found that in sustainable community governed common pool resources, communities defined and adopted at least one rule to determine whether someone should be admitted or excluded. She calls these rules 'boundary rules' and classifies these in three categories:

> Boundary rules can be broadly classified in three general groups defining how individuals gain authority to enter and appropriate resource units from a common-pool resource. The first type of boundary rule relates to an individual's citizenship, residency, or membership in a particular organization... A second broad group of rules relates to individuals' ascribed or acquired personal characteristics... A third group of boundary rules relates to the relationship of an individual with the resource itself. Using a particular technology or acquiring appropriation rights through an auction or a lottery are examples of this type of rule (Ostrom 1999, p. 511).

Empirical research on community-managed common pool resources shows that most adopt a combination of these three types of rules.

The design principle *clearly defined boundaries* refers not only to organizational boundaries. It also refers to the boundary around the common pool resource (Agrawal 2002). Ostrom writes, "Individuals or households who have rights to withdraw resource units from the CPR [common pool resource] must be clearly defined, as must the boundaries of the CPR itself" (Ostrom 1990, p. 90). Examples of boundaries of a common pool resource are, for instance, a fence or a ditch that surrounds a piece of grazing land.

Reasons for collectives to have clearly defined and protected boundaries

Literature identifies a number of reasons to define and protect boundaries: (i) to reduce uncertainty, (ii) to maintain autonomy and (iii) to protect against appropriation. Boundaries reduce uncertainty. "This…is important in that members' actions are considered more predictable than the actions of non-members" (Russ et al 1998, p. 127). Boundaries also create identity and autonomy. This is what Oommen refers to as the "we and they" distinction

(Oommen 1994, p. 5). Those who have crossed a boundary become a member and belong to the 'we' group, everyone outside the boundary is referred to as 'they.'

The role of boundaries in protecting against appropriation is specific to communities that govern common pool resources. Ostrom writes:

> So long as the boundaries of the resource and/or the specification of the individuals who can use the resource remain uncertain, no one knows what is being managed or for whom. Without defining the boundaries of the CPR and closing it to 'outsiders', local appropriators face the risk that any benefits they produce by their efforts will be reaped by others who have not contributed to those efforts (Ostrom 1990, p. 91).

Ostrom in the quote refers to free riding, which is the situation in which people reap benefits for which others bear the costs (Olson 1965). Olson describes free riding as one of the main reasons why people do not invest their time and effort in the development of public and common goods, of which open source software is an example. Why would anyone be willing to spend time developing a good that others can have for free? According to Ostrom (1990), the chance that people will invest their time and effort in the development and maintenance of a common pool resource increases when the collective of individuals defines boundaries that determine who can claim the benefits of the collective effort.

Why open source communities need boundaries: appropriation

Usage of software in general and of open source software in particular does not affect the amount of source code available to others. To use software developed in a community implies making a copy of the source code and installing it on a local computer. This does not diminish the original stock of source code, and therefore using the resource itself does not diminish the amount available to others (Benkler 2002a, Von Hippel & Von Krogh 2003, Lakhani & Von Hippel 2003, Markus et al 2000, O'Mahony 2003).

Usage of open source code can be compared to people who swim in a bay or who walk in a forest. The bay or forest needs some protection from people swimming or taking a stroll, but this protection need not be as stringent as that described as required for common pool resources. In the latter, protection is needed to limit appropriation from the resource, as this impacts the amount available to others. Too high levels of appropriation compared to the rate of regeneration could lead to a tragedy of the commons; that is, resource depletion. Similarly, in open source communities little protection is needed to safeguard the source code from people who just want to download it, as this does not affect the amount available to others. However, boundaries are needed to protect the open source communities against individuals and companies who want to appropriate the software and thus diminish the stock of source code available (O'Mahony 2003).

Many respondents interviewed for this research identified and explained this threat of appropriation. One member of the Apache Software Foundation (ASF)[1] Board of Directors talked about such threats:

> We weren't protected against lawsuits. What would happen if for example, employees of a certain company are working on the code? The company can sue us because the employee who

> put the code in did so during working hours and therefore the code belongs to the company. The company could sue us for that. A company can also try to get a patch in on which they have patent. If the patch gets included in an official version of Apache, the company would be able to start demanding a license fee of X dollars for every user using the Apache software.

This respondent named two threats, namely, copyright and patents, both of which are intellectual property rights (IPRs). He argued that people who have – willingly or unwillingly – included patches of source code that are subject to IPR, so-called 'infected code,'[2] can effectively jeopardize the future stream of benefits from the stock of source code. People who use the software are then effectively infringing on a copyright or patent and can be sued. Another threat is that the IPR holder can appropriate the source code from the software. This is particularly troublesome when the source code is (i) a large part of the software measured in lines of code or (ii) an essential part of the software, that is, a part that can hardly be removed without affecting the functioning of the entire program.

Both copyrights and patents pose a threat to the availability of source code. However, the difficulties of copyright are relatively easy to overcome. "Copyright on code is not an obstacle: we can write our own because we are programmers. We can write around copyrighted software."[3] A problem does arise when the source code used by the communities is based on a patented idea. According to the president of the Free Software Foundation (FSF),[4] it is then very difficult to rewrite the source code in such a way that it no longer infringes the patent.[5]

An example: the case of the SCO Group

There is a good example that illustrates the impact and the threat of companies or individuals that claim to possess an IPR on open source code. A company called SCO Group, Inc., recently sued IBM and a number of Linux end users.[6] The grounds for the suit is SCO's contention that the Linux kernel includes source code that is the property of SCO. Therefore, SCO wants to appropriate those pieces from the Linux kernel.[7]

In describing the lawsuits, participants in open source communities use words and phrases like "defending our community against predation"[8] and "preventing the open source community from being destroyed by SCO's greed and desperation."[9] Perhaps, these phrases over-dramatize what is actually taking place. They do, however, highlight developers' belief that predation, greed and appropriation can lead to the destruction of a community and its software.

Part of the reason for the high impact of SCO's claim is that the pieces of source code on which SCO allegedly has property rights are said to be key for the functioning of the Linux kernel; it is not straightforward to replace this source code. If SCO wins the lawsuits, the ruling would constitute a threat to the continuity of the community. SCO could, for instance, demand its source code be removed or demand fees for every user who downloads the software. This includes the many voluntary participants in the communities, most of whom are unlikely to be prepared to pay a licensing fee. Therefore, it is understandable that the SCO lawsuit has resulted in upset and anxiety among individual developers and corporate users of the Linux kernel.[10]

Conclusion: boundaries are needed

To minimize the chance of appropriation, communities are expected to need boundaries that protect the source code and ensure the continuity of the software development process (see also O'Mahony 2003). These boundaries can take two forms: organizational boundaries or resource boundaries (e.g. the fence).

Few organizational boundaries to limit the size of communities

Previous research has identified and described a number of less formalized organizational boundaries in open source communities. These boundaries are said to create a notion of belonging and provide a way to decide who is inside and who is outside the community. For instance, Lin (2004) describes how the attachment of individuals to certain artifacts connects them to a particular community and gives them a feeling of belonging to that community. Examples are T-shirts and mascots, like the Linux penguin and the FreeBSD devil. Edwards (2001) also argues that open source communities have boundaries to decide who is an 'insider' and who is an 'outsider.' These boundaries are informal and consist of, for instance, (i) the level of knowledge that potential participants need to understand the software and to be able to contribute to the development effort and (ii) the mastery of shared norms of conduct, as individuals who do not behave according to the norms will be sanctioned; they are, for instance, ignored or flamed (see chapter eight for more about sanctioning in the communities). One could also argue that the website and the mailing lists of a community demarcate its boundaries. Consider for instance the website www.apache.org, which is the domain name for many of the projects in the Apache community. In a sense the website delineates the boundary between the Apache community and the rest of the world.

What these boundaries have in common is that they are not intended to actively exclude participants. Even if they were, there is no means to enforce the boundaries. In that sense they are different from the boundaries described in research on community-governed common pool resources. In communities managing common pool resources, organizational boundaries have been created with the goal to exclude. Open source communities generally lack such exclusion mechanisms: the culture is to attract as many people as possible to participate. For this reason the boundaries of open source communities are much less formal and are arguably highly permeable (see also Fielding 1999, Franck & Jungwirth 2003, Osterloh et al 2003a). Membership in open source communities is claimed to be "fluid; current members can leave the community and new members can also join at any time" (Sharma et al 2002, p. 10).

This fluidity in membership gives rise to a number of potential problems. One is that people have little time to get to know one another. One could argue that in order to successfully develop a complex product like open source software, developers need to know one another and become familiar with how others operate. "As time goes by you get to understand how someone thinks and how someone does the job."[11] Another problem is that the absence of boundaries restricting the size of the community might give rise to a temptation to free ride (e.g. Olson 1965, Ostrom 1990). We already saw that there is no real need to restrict people from downloading source code, as this does not affect the amount of source code available for others. Still, theoretically speaking, boundaries are needed to restrict the size of the community for development purposes. The reason is straightforward; when people know each other they trust each other and can make all sorts of rules. A rule would, for instance, be needed to regulate how and when people are supposed to invest time and effort in the maintenance of the source code. Also sanctioning to ensure compliance with a particular rule is difficult when people do not know one another. Thus, not knowing one another increases the chance that rules and preferred forms of conduct will remain undefined or that people will not behave in compliance with these rules. According to research on community-governed common pool resources, however, such rules are needed to ensure the continuous development and improvement of the source code.

The next chapters, especially chapter five, identify and discuss a great number of mechanisms present in open source communities which largely explain why boundaries are much less needed than theory would predict. Basically, the reason is that open source communities have a large number of devices that create slack and redundancy, and thus make the development process and the resource more resilient to 'unwanted' behaviors.[12] These mechanisms also reduce the need for collaboration and formal planning, which explains why developers do not always need to know one another personally.

A trade-off: attraction versus protection

Individuals face a trade-off while they are creating and adopting boundaries in open source communities. Appropriation is not the only threat to the future stream of benefits from open source software. Another threat is a lack of development and maintenance efforts. These efforts are needed to keep the software updated and thus attractive to potential users. Consider the following statement by a Linux developer: "That piece of source code was pretty difficult to maintain, because the kernel constantly changes. To maintain the interface between the driver and the kernel was not at all easy." As this comment illustrates, developers must constantly update the interfaces between their drivers and the kernel. If they do not do this the drivers become worthless; the interface has to be up to date if the driver is to function the way it is supposed to.

If a driver or other piece of software no longer functions then the project will *die*. "People lose interest in a project. If development really stops then the project dies."[13] Effectively, a project is dead when the source code is no longer useful and ceases to provide benefits to potential users. Individuals then stop downloading the software and no longer contribute to the software's development and maintenance. The source code is likely to remain available, but hardly anyone uses it anymore. "Obviously you can still download my patch. You just can't really use it for the kernel anymore."[14]

To prevent a lack of development and maintenance, communities must attract users and developers. Communities need to draw contributors who understand the source code and have the skills to maintain it when new hardware is released or when interfaces change and to make changes when users have other, mostly new, requirements. The maintainer of BlueFish software describes the problem of attracting new developers:

> An unpopular program has few developers and then you must either have a lot of time to develop everything yourself or you always have to copy other programs. You can probably not keep up. If you want your program to remain popular then it needs to be able to run on different platforms, etc. Only then will you continue to attract new developers. It is like a chain

reaction, popularity leads to more users and to more developers, more add-ons and thus more users.

Developers in communities thus face a trade-off: They must guard the source code against people and companies who want to appropriate it and at the same time create incentives for, sometimes the same, people and companies to invest in source code development and maintenance. "Open source communities face conflicts concerning how to institutionalize access to common property. Intense debates have raged about how to provide a legal basis for open source software which will preserve accessibility while maintaining incentives for continued development" (Bruns 2000, p. 3). The dilemma is that stricter boundaries will lead to fewer volunteers who contribute to the development and maintenance of open source code.

Rising transaction costs is one explanation of why higher or stricter boundaries results in less participation (Coase 1937). Individuals and companies have to make more effort to cross the boundary and thus to participate in the development and maintenance of the software. This increases the likelihood that the project will die out. But boundaries that provide little protection against appropriation might lead to depletion, as people and companies might appropriate source code from the commons and thus endanger the future flow of benefits from that common pool resource.

Open source licenses: enormous variety

One mechanism that has been created and adopted in open source communities to protect source code from appropriation is the open source or free software license. Such a license is "very dependent on copyright, but it turns it around, on its head. Instead of limiting use, it frees the source code."[15] Such licenses are the most important means to protect source code from appropriation and at the same time attract developers and companies to participate in development and maintenance. Yet, as this section demonstrates, a wide variety of licenses has been created, as many developers in the communities have their own outspoken ideas about how such licenses should deal with the dilemma of protection versus attraction.

Appendix C lists the licenses that, according to the website of the Open Source Initiative, comply with the open source definition. This definition sets out requirements to which an open source license must adhere. Licenses that do not comply with this definition are not considered to be open source and thus are not listed on the website. In September 2003 the site listed 48 unique licenses.

The next sections present and discuss five of the licenses. These are the General Public License, the Lesser General Public License, the Berkeley Software Distribution (BSD) license, the Apache Software License and the Mozilla Public License. The General Public License, the Lesser General Public License and the BSD license are described because they are the licenses most frequently used.[16] The Apache Software License though less used, is introduced because it is used by the Apache community, which is one of the communities receiving the most attention in this research. The fifth license is the Mozilla Public License. Bruce Perens, who is one of the most authoritative open source developers and evangelists, claims that many companies use the Mozilla license as a basis on which to create their own slightly revised, new open source license (Perens 1999). The Mozilla Public License thus serves as an example open source license for companies creating open source software.

The descriptions of the licenses include a discussion of the way in which each license achieves a balance between extraction and protection. The descriptions start with a short introduction of the origin of the license. Then the protection measures in each license are discussed.

The General Public License

The first version of the General Public License (GPL) was developed by Richard Stallman and released in 1989. Many projects use the second version, which was released in 1991. The GPL is quite large and contains a number of interesting clauses. Stallman himself describes the main philosophy of the license:

> What the GPL does is to prevent companies from blocking and enclosing the software. Do you know what enclosure means? Enclosure has its roots in history. Back in the 1600s the land was not owned by anyone, it was in the hands of the commoners. Then people came and put a fence around the land and effectively took the common land away from the commoners. This created a lot of hostility. The GPL prevents companies from taking the software away from the commons. It has become free software. Free software is the commons of the Internet. It is like a public park. Everyone can use it, but they cannot build a fence around it.[17]

To ensure that the software remains in the commons, the license grants every user a number of rights and restrictions. The GPL gives every user the right to use, copy, modify and redistribute software that is licensed under it. This is because developers can understand and change software only when they have access to its source code. The GPL therefore ensures access to the source code by prohibiting distribution of modified software unless the modified version is accompanied by the complete corresponding source code. In other words, the GPL forbids anyone to modify and distribute GPL-licensed software without providing access to the corresponding source code. Thus, the GPL prohibits people and companies from keeping modifications to distributed source code private.

The fact that the GPL forbids people from keeping modifications to distributed source code private does not mean that the GPL ensures that the source code is returned to the original author or community of developers (this claim is frequently made, see for instance Perens 1999). In fact, the GPL provides no such assurance. First, the source code may be kept private as long as the modified software is not distributed.[18] Second, the GPL does not require that, upon distribution of the modified software, the source code also be returned to the original author or community of authors. It merely states that the source code must be made available to the recipient of the modified software.

The GPL is subject to much critique, the main reason being what is popularly regarded as its *viral aspect*. If source code licensed with the GPL, irrelevant of the size of the code, is included in a program, then the GPL dictates that the entire program must also be licensed with the GPL. This condition, however, is not only the subject of critique, but is also surrounded by much confusion and legal uncertainty. One of the problems is to decide when the source code has actually become part of a program and when not. There is, for instance, no universally accepted definition of what a 'program' is. This leaves room for doubt and uncertainty as to what the GPL allows and what not.

Clause 7 of the GPL deserves special attention. It is the so-called *liberty or death clause.* According to Stallman, this basically states, "If you infringe the GPL once, then you are out. You've lost your distribution right. Usually after a warning the one who infringed will comply and we [the FSF] will give back the right to distribute."[19] This is one way to enforce the GPL against people who do not comply with the license.

The GPL aims to restrict companies and individuals from appropriating source code that is licensed under it (Stallman 2002). According to the editor in chief of the Linux journal, "GPL has a ruling that if anyone tries to collect a patent on a piece of source code in GPL'ed software, further distribution will not be allowed any longer." When other conditions on GPL code exist and when the user cannot satisfy both the additional conditions and the conditions as stated in the GPL then distribution of the source code or object code is no longer allowed. These 'other' conditions can originate from a court judgment, an alleged patent infringement or other reason.[20] There are then two ways in which the source code can again be distributed: (i) if the other conditions are no longer imposed on the user, because for example, the conditions have expired or (ii) if the source code on which the other conditions rest is entirely removed from the program.[21] The editor in chief of the Linux journal claims that this clause protects communities from individuals and companies wanting to enforce a patent on GPL-licensed software. "Thus the GPL does have a defensive measure against a patent attack, in other words, a firebreak against a patent-based attack."

This last statement, however, must be questioned. The patent holder can effectively cease distribution of the source code. In that situation, no individual or company can use or modify the source code any longer. Therefore development and maintenance of the source code would be halted, eventually leading to the death of the project. This is perhaps not the same as the actual appropriation of the software, but neither does it safeguard a community from the threat of patents.

The Lesser General Public License

The Lesser General Public License (LGPL) was also created by the Free Software Foundation and is almost identical to the GPL. Companies and individuals that distribute a modified version of an LGPL-licensed library must also provide access to the corresponding source code. The 'liberty or death' clause is included in the LGPL too. Yet there is one important difference, namely in the LGPL *the viral character is absent.*

As discussed earlier, when software licensed under the GPL is combined with other software to form a program then the entire new program is subject to the GPL. This viral character of the GPL is problematic for software libraries. Libraries, by definition, are used in combination with other programs. When a library and its related programs are installed and run on a computer they are considered to be one program. Thus, if the library were licensed under the GPL the entire program with all of its associated elements must be licensed under the GPL. Because the LGPL does not include the viral character, a program can link to a library licensed with the LGPL and still maintain its original license. The preamble of the

LGPL states, "We use this license for certain libraries in order to permit linking those libraries into non-free programs."[22]

The FSF claims that sometimes it makes more sense to use the LGPL. One such situation is when a library does the same job as many other libraries that are not open source.[23] If a library has to compete with other, proprietary, libraries (i.e. libraries that are not open source), the LGPL is the more logical choice. The reason is fairly straightforward; people and companies are generally not inclined to use software that forces them to re-license their own proprietary software and make it open source. This is especially true when their business model depends on sales of that software.

The Berkeley Software Distribution license

The Berkeley Software Distribution (BSD) license was developed by Berkeley University and used to license the BSD software. The license used to state that the university must be given "credit in the manual and in advertisements" (Wayner 2000, p. 92). This clause, however, was removed in 1999. Without the clause the license has become the equivalent of another popular open source license, namely, the MIT license or rather the 'X11' license.[24]

The basic principle underlying the BSD license is that anyone can use, modify and sell BSD-licensed software. The GPL requires everyone distributing modified versions of the software to accompany it with the appropriate source code. The BSD, however, has no such provision. The BSD is therefore said to be less restrictive (Lerner & Tirole 2002b, Wayner 2000). Much controversy surrounds the BSD license however. Many people simply feel that the license makes no sense. "Why do you want to say we know that people are taking our software and that they are contaminating it, but we are going to release it anyway?"[25] Companies and individuals can use, modify and sell software licensed under the BSD, and the license does not require them to share the source code of their modified versions. In other words, companies and individuals can keep software developed in open source communities private. This is probably why people in the communities make the following claim:

> [BSD licensed software is] prone to 'hijacking' by commercial software vendors: in other words, the commercial firm may add some proprietary code to the open source software and take the whole private. While the resulting software may (or may not) be superior, the firm disrupts the dynamics of the open source project by *de facto* privatizing it (Lerner & Tirole 2002a, p. 12).

Some respondents argue that the BSD is fairer than the GPL. They feel it should be possible for companies to invest in the development of open source software and make a profit from doing so.[26] Therefore, they should be able to distribute a proprietary version of the software. But what happens when the proprietary version becomes more popular than the open source version? In this scenario the open source version would attract fewer volunteers, which could result in the death of the project.

The Apache Software License[27]

The first version of the Apache Software License (ASL) was created in 1998. The reason for writing the license was IBM's involvement in the Apache project. The idea was to create a license that resembled the BSD license. This means that the ASL does not contain the viral aspect and that modified versions can be proprietarily distributed. The license differs from the BSD license as well. Most importantly, it explicitly stipulates that the names 'Apache' and 'Apache Software License' may not be used to "endorse and promote products derived from this software without prior written permission."[28] An ASF board member explained, "There could be someone out there who uses the Apache software under the name WinApache. Our license explicitly forbids this. You can use our software and make a commercial version of it, but the license states that you cannot use a name that has Apache in it."

The creators of the license thus wanted a new license that would give more protection to the names 'Apache' and 'Apache Software Foundation.' They felt the need for this extra protection even though the name Apache is trademarked. The creators were afraid that individuals and companies would free ride on the Apache reputation. The trademark alone was seen as insufficient to preserve their reputation and protect their 'brand name.'[29]

The Mozilla Public License

Many companies have created their own license to make software open source. Usually these licenses are similar to the widely used BSD license, the GPL, LGPL or ASL. In general, software licensed under corporate-developed licenses attracts fewer users and developers because the consequences of these licenses are unknown.[30] To many, especially individual users and developers, the aforementioned licenses, like the GPL and BSD license, are much safer to use than a license created by a company. The GPL, for instance, is used by a large number of communities and has been analyzed, discussed and accepted by a great number of lawyers, companies and highly respected developers. In contrast, most of the licenses created by companies are used in only a limited number of communities and thus are known and understood by a very limited number of people.

When Netscape decided to make Navigator open source, it created a license called the Netscape Public License (NPL). The NPL gave Netscape the right to re-license the modifications that were made by others. In other words, it reserved the privilege to take these modifications private (Perens 1999). Later Netscape created the *Mozilla Public License* (MPL), from which this provision was removed. The MPL is said to balance two extremes, namely, the unrestrictive BSD license and the restrictive GPL.[31] The MPL resembles the GPL in that it states that distribution of modified versions of MPL-covered software must be accompanied by the corresponding source code. Like the LGPL, however, the license is not viral. Code licensed under the MPL may be combined with proprietary code and the combination does not have to be licensed under the MPL. The MPL also differs from the LGPL in that it does not contain the 'liberty or death' clause.

Bruce Perens (1999) claims that many companies have adopted variations of the MPL for their own programs. The reason why the MPL is quite popular might be that the MPL, like the ASL, is *incompatible with the GPL*. Many open source licenses are incompatible with the GPL, which means that one cannot "combine a module which was released under that license with a

GPL-covered module to make one larger program."[32] Incompatibility also implies something else. It means that MPL-protected code can never include any code licensed with the GPL. This might be considered a disadvantage, but it can also be viewed as a clever defense strategy, as it protects the source code against the viral aspect of the GPL. When GPL code is included in MPL software, the code must first be removed before the MPL code can again be distributed.[33] This prevents MPL code from becoming GPL code.

The licenses compared

The licenses differ in many ways. The one thing each license has in common is that no one, except in certain situations the copyright holder, can restrict the use of the source code. This means that once the software is protected with one of the licenses and made available on the Internet, that particular version of the software remains available for others to use. In other words, the licenses share the goal of keeping open source software in the commons. Table 4.1 presents an overview of the licenses and their restrictions.

Table 4.1 – Five open source licenses and their restrictions

License	**Restrictions**
General Public License	- Distributed modifications to the source code may not be kept private - The entire program must be licensed under the GPL if GPL code is included - Distribution right is revoked upon license infringement - Distribution of source code is prohibited when conditions of the GPL conflict with other conditions
Lesser General Public License	- Distributed modifications to the source code may not be kept private - Distribution right is revoked upon infringement - Distribution of source code is prohibited when conditions of the LGPL conflict with other conditions
Berkeley Software Distribution	- In general, use of BSD-licensed code is not restricted
Apache Software License	- The name 'Apache' and 'Apache Software License' to endorse derived code may be used only with prior written permission
Mozilla Public License	- Distributed modifications to the source code may not be kept private

The license is likely to influence the number of individuals who join a community and contribute to the development and maintenance of open source code. For instance, the more obscure the license that protects the source code, the less people will tend to participate in the community – other things being equal. Table 4.2 lists several ways in which a license might limit the number of developers drawn to a project. An 'X' marks the limiting factors that correspond with that particular license.

Three categories are identified. The first is *a high number of restrictions*. The GPL and LGPL have quite a large list of restrictions. These restrictions erect a barrier to participation, as potential users and participants must invest time and effort to understand both licenses and to ensure that they act in accordance with the restrictions. The other three licenses are less restrictive. The BSD license, for instance, has very few restrictions. The original authors "designed the BSD license to be very liberal to please corporate donors [of Berkeley University]" (Wayner 2000, p. 50).

The second category is the *ability to keep the source code private* when software is distributed. Both the BSD license and the ASL allow anyone to adopt the software and make a proprietary version of it. This version can be an exact copy of the software, but might also be a modified version. The BSD license and the ASL do not require developers to make the source code available. This characteristic of both licenses is likely to attract more companies to work on the software. Members of the Apache community interviewed for this research said that a license that allows companies to keep the source code private makes more sense. They argue that a company should be allowed to make a profit from the software. On the other hand, this characteristic might scare off potential developers, as they might consider it unfair that their time and effort could be usurped by a company to make a profit.

Table 4.2 – Factors that could influence the number of contributors

License	**High number of restrictions**	**Ability to keep the source code private**	**Uncertainty about the license**
GPL	x		x
LGPL	x		x
BSD		x	
ASL		x	
MPL			x

The third factor is *uncertainty about the license*. The MPL, GPL and LGPL are said to be surrounded with more uncertainty than the other two licenses. The MPL is uncertain mainly because the license was created by a private company and is used in only a limited number of projects. Might the license contain clauses that favor Netscape? In any case, the license contains a great deal of legal language, causing many potential participants to wonder what exactly the clauses intend to accomplish.

The uncertainty associated with the GPL and LGPL is due to the ambiguity of some clauses and their uncertain legal status. What, for instance, do phrases like 'part of a program' and 'distribution' mean? What exactly is a distribution? Distribution within a company might not be considered an act of distributing. What about when software is distributed between independent strategic business units that are geographically separated? Does the term 'distribution' apply to a situation in which software is provided to a strategic partner?

Uncertainty regarding both licenses also originates from their legal status. The GPL and LGPL contain clauses that might not withstand the scrutiny of a judge. The vice-president of the FSF is confident that no company will risk the step of going to court: "No attorney would decide to go to court. Because the license is very strong they are very likely to lose." Given the large number of programs licensed with the GPL and LGPL, one might conclude that most individuals and companies have faith in the legal status of both. However, this degree of confidence in two licenses that have hardly been tested in court is surprising, to say the least. There are many reasons why the licenses might not pass the test of a lawsuit. Without going into too much detail, we can mention two reasons why the legal status of the licenses is doubtful.

The GPL and LGPL must be viewed as a *contract* between the author(s) of the software and the user. As a contract their legal status can be questioned in a number of ways. First is the question of whether adding the GPL or the LGPL at the bottom of the source code is enough to engage two parties in a contractual relationship. Second, what happens when someone other than the author distributes the software to a third party? Are third parties then bound by the contractual relationship between the author and the party from whom they received the source code? Third, the licenses do not state a specific term "for the rights it grants and limitations it imposes. Two circuits have held that a license that states no term is terminable according to applicable state law" (McGowan 2001, p. 298).

The GPL and LGPL are based on *copyright* and as such can be viewed as copyright notices. According to McGowan (2001) this results in at least one additional uncertainty, namely, do the requirements of the GPL and LGPL conflict with the notion of fair use?[34] "A standard fair use inquiry would ask in part how the copying would affect the market for the copied work. This question is designed to ascertain whether the copying would reduce the author's returns by a notable amount" (McGowan 2001, p. 288). The question of how the GPL and LGPL conditions would affect the market is difficult to answer. One could even doubt whether open source software development takes place in a market. Undoubtedly, it is a new and different means of software production. Hence, predicting the outcome of a court procedure is all but straightforward.

Licenses are dynamic boundaries

Given the diversity and sheer number of licenses it is no surprise that the licenses are dynamic. Another reason to conclude that the licenses are dynamic is the large number of licenses that have a version number higher than 1.0 (see also appendix C). Consider the GPL. "The GPL version 2.0 was developed in 1991. When writing a license, one can only see a limited number of years ahead. Currently the GPL 2.0 has trouble addressing certain issues."[35] There are a number of reasons why it is difficult to write a license that is able to properly protect the source code from appropriation and at the same time attract contributors. The reasons that were derived from empirical observations in this research are four: (i) the trade-off between attraction and protection, (ii) technological change, (iii) the tendency to constantly explore the limits of what is allowed by the licenses and (iv) competition.

The trade-off

A previous section of this chapter argued that participants in open source communities face a trade-off. On the one hand they want to attract as many new users and potential participants and contributors as possible. On the other hand they need to protect the source code against appropriation. Faced with this trade-off, they can make many different choices.

They may, for instance, decide to create a very protective license or a very open one. They may also write new and innovative clauses to prevent certain types of appropriation or to enforce a certain type of behavior among users. In other words, many alternatives are available to participants. The different choices explains the creation of the enormous number of only marginally different licenses. Changing preferences and situations are other reasons for the frequent updating and modification of existing licenses.

Technological change

Technological change is another factor driving the constant revision of certain licenses. The way in which software is used is constantly under revision. At the time licenses are written, authors cannot foresee all these different types of usages and therefore cannot anticipate all types of behavior. It is quite plausible that the authors will want to forbid some of these new types of behavior. To do so, they must rewrite and add text to the original license.

An example of such a technological innovation is the rise of Web applications running on servers. Users of a Web application do not make a local copy of a software program they are using. The application remains on the server of the website, meaning that the user of the website does not actually obtain a version of the software. Thus, the owners of the website are not distributing the software, which means that, according to the GPL version 2.0, the owners of the website need not reveal the source code of the Web application. However, the creators of the GPL would like to see that in such situations the source code is also made available.[36] To deal with the specifics of Web applications the GPL needs to be modified.

Stretching and exploring the interpretation of licenses

Essentially the above-described situation in which Web applications are exempted from the GPL would not be a problem, were it not that companies are constantly looking to maximize their profits without increasing their costs. To increase their profit they constantly explore the limits of what the licenses allow. One company, called Affero, became painfully aware of the breach in the GPL when it was used for Web applications.[37]

Affero had licensed the Web applications on its website under the GPL and included a button that enabled people to download the source code. Its reason for doing so was to enable people to understand the software they were using. But soon Affero faced the situation in which another company had downloaded the source code and developed its own version without revealing the source code of the modifications. Affero did not mind people or companies copying the software and making a commercial version of it, but it felt it had a right to receive and review the modifications. The 'Affero GPL,' also known as the GPL version 3.0, was released to address this issue.

Facilitating competition

Another reason why authors want to rewrite parts of a license or even the entire license is competition. Consider the LGPL, which is based on the GPL. The FSF decided to create a new license because it had developed an open source library that needed to compete with proprietary libraries. The library was called the GNU C library and could have been licensed under the GPL. In that case, however, the library would hardly have been used. The reason was mentioned earlier; all software linked to the library would then have to be licensed with the GPL. Most software companies, however, spend sizeable sums developing proprietary software and do not want to license their software with the GPL. If a company can choose among a number of libraries, they are unlikely to choose one licensed with the GPL. So GPL-licensed libraries are likely to attract fewer users than those licensed differently. Ultimately this could lead to a scenario in which fewer people participate in the development and maintenance of the source code; and too little usage would result in the death of the project.

The LGPL was created to deal with this problem. The vice-president of the FSF explained:

> When we developed the GNU C library, there were a lot of alternative libraries. No proprietary software would use this library if it were protected by the GPL. Therefore we decided to create the LGPL, which does not have the viral character. Proprietary software using libraries under the LGPL can remain proprietary.

Signs of convergence: widespread adoption of the GPL

The choice of open source license is usually made at the beginning of a development project, and in nine out of ten cases the license remains the same during the lifespan of a community.[38] Typically, the formation of an open source community starts with an individual or a company wanting to make source code available on the Internet. The Linux community, for instance, started when Linus Torvalds created a simple program and posted it on a mailing list. Or consider the Mozilla community, which began when Netscape decided to make its Web browser and the corresponding source code available. The company or individual who decides to make software available is usually the one to choose the open source license. As described in the previous pages there are many – more than 40 – to choose from. Moreover, one can always decide to create a new license, if the existing licenses do not satisfy the desires and ideas of the initiator of the community.

An example of someone creating a new license is Bram Molenaar, who started the Vim project. When Molenaar made his software available he decided to develop an entirely new license, the so-called 'Vim license.' One of his reasons for doing so was that he considered the GPL too restrictive. He took time to develop the new license and discussed the exact contents with Richard Stallman of the FSF and with developers from the Debian community.[39]

In short, the creator of a project has a lot of leeway in the choice of license. Combined with the observation that a license is never finished but needs to be regularly updated in response to new technology or market forces, it is not surprising to witness the enormous variety of open source licenses that is currently available. Yet, despite this trend of divergence, there is also evidence of a surprising level of convergence. The best proof of this convergence is undoubtedly the large number of open source communities that have licensed their software with the GPL. Statistics show that of a collection of more than 30,000 open source projects, 65 percent had adopted the GPL to protect their software. Runner-up is the LGPL, which is adopted in 6 percent of the projects.[40]

A number of explanations can be put forward for the widespread adoption of the GPL and the LGPL. For instance, a number of important projects, of which Linux is the best known, use the GPL. For many individuals and even companies this could very well be considered a sign that the GPL is a good license. Another explanation could be that many, especially individuals, do not make a conscious choice among the licenses. Rather, they choose the one that seems most chosen by others. In other words, there could be a 'bandwagon' mechanism in play. The bandwagon mechanism refers to a process in which individuals follow the choices of others in an attempt to reduce the complexity they are facing (Egyedi & Van Wendel de Joode 2004, Farrell & Saloner 1988). On a collective level this results in convergence.

News sites to protect boundaries and educate developers

Many websites on the Internet deliver news about open source communities. Examples are sites like Freshmeat and Slashdot. Also, community mailing lists perform a similar function, namely, bringing members the latest news and developments. What is striking about these sites and mailing lists is that anyone can write an article and request that it be posted. Most of these posts are merely summaries with links to articles and opinions posted elsewhere on the Internet. Thus, volunteers worldwide summarize news items they find interesting and then submit them to sites like Slashdot or Freshmeat. The sites have a large number of reviewers who read the summaries and post them on the site. Although the news on these sites is frequently biased toward open source, this research argues that these sites act as *boundary spanners* for the communities and as such perform a crucial function in protecting open source software development and educating people about other protection mechanisms, like licenses.

Boundary spanners are said to perform a pair of tasks related to demarcating the boundaries of organizations, namely, *representing the organization externally* (Aldrich & Herker 1977) and *acting as a buffer* for other parts of the organization. The boundary spanner scans and interprets information and channels relevant items to the rest of the organization as seen fit (Russ et al 1998). News sites like Slashdot perform both tasks. They channel relevant information and in so doing teach people about all sorts of issues. Furthermore, they represent the communities externally. This is particularly evident in situations that touch upon the boundaries of open source communities, for instance, when companies or individuals act in a way that reaches the limits of what is allowed by the licenses or try to appropriate the software.

The case of SCO (revisited)

Consider the legal claim presented earlier in this chapter, namely that of SCO against IBM and others involved in Linux kernel development. SCO claims that the Linux kernel consists of Unix code on which it owns IPRs which it intends to enforce. This claim has caused upheaval in the community as evidenced on, for instance, the Slashdot news site. For months Slashdot has featured articles about new developments, reactions, opinions and official press releases on the 'SCO Case.' The Groklaw site is another example. This site was created in 2003 and is entirely dedicated to the SCO lawsuit.[41] It provides many types of information, like a timeline of the origin of the Linux code and comments and links to the latest activities and press releases relevant in the case.

The articles and links on both sites serve the two goals mentioned above. The first is to inform and educate. The websites have many references to articles that present and discuss the latest developments in the case. These explain the exact details of the SCO claim and why the claim might hold in court and why not. The discussions deal with both the nature of the claim, and with the nature of, for instance, the GPL and why the GPL might provide a defense mechanism against the claim. An example of the type of information conveyed is an article written by two well-known developers. They respond to an open letter by the CEO of SCO. Point for point they explain why the CEO's claims are "not merely both false and slanderous, but contradictory with SCO's own previous behavior."[42]

The second goal is to represent the Linux community externally. In its claim SCO tries hard to convince people that its goal is not to destroy the community, but to "protect SCO's intellectual property and contractual rights" and that it is "open to ideas of working with the Open Source community."[43] At the same time SCO has repeatedly threatened companies and individuals that they should either (i) stop using Linux or (ii) buy a license from SCO.[44] SCO's strategy has had some results. Gardner, for instance, issued a statement which advises companies to wait for the court to decide on the issue before making a switch to Linux.[45]

The Slashdot website and other news sites represent Linux externally and try to convince potential users that the SCO case is invalid and void. They do so primarily with a strategy of *naming and shaming*. In many instances SCO is depicted as an evil company: "The thieves and liars at SCO."[46] Furthermore, SCO is accused of having changed its "business model and now they sue companies [and] people"[47] as a last resort to make money. Other articles are somewhat more refined, many can be found on the Groklaw website.

Summary: the importance of news sites

Summarizing, there are many news sites that play an important role in the protection of open source software development. They do so in two ways, namely by (i) informing and educating developers in the communities about open source licenses and threats that the communities face and (ii) representing the communities externally, explaining why certain allegations pose no threat and should not cause companies to decide not to use open source software.

Foundations to protect the boundaries

More and more open source foundations have been established in recent years. O'Mahony (2003) writes that of the six communities she analyzed, five had created a legal entity to secure IPRs and protect individual contributors. She writes, "Incorporation allows projects to protect volunteer contributors from individual liability, enter into agreements collectively, and protect their code, trademarks, licenses, and brand" (O'Mahony 2003, p. 1190). The foundations serve as beachheads; they 'protect' individual contributors from all sorts of external and legal issues (Van Wendel de Joode et al 2003). Furthermore, many foundations enforce open source licenses. Enforcement in this case means that the foundation might formally announce the release of a new version of the license and summon those infringing on an open source license to change their behavior and act according to the rules stipulated.

Many of the foundations are dedicated to a single software project or group of software projects. The Free Software Foundation (FSF) is the oldest foundation and is host of the GNU project. Other foundations are much younger. The foundations introduced and discussed here are the FSF, the Apache Software Foundation (ASF), the KDE Free QT Foundation and Software in the Public Interest (SPI). These foundations are the dedicated host of one or more software projects. A foundation that has no direct tie to a software project is the Open Source Initiative (OSI).

The Free Software Foundation

"The Free Software Foundation started in 1985 as a legal umbrella for the GNU projects, mainly for the legal implications of copyrights."[48] Richard Stallman founded the FSF as a charity foundation that would earn its money through the sale of books, printed materials and charitable donations. The FSF hosts many free software projects under the GNU name. Projects include Emacs, GNU Compiler Collection (GCC) and Lilypond.

The FSF employs ten people.[49] Two are president Richard M. Stallman and vice-president Bradley M. Kuhn. Of the other eight employees only one is actively involved in boundary maintenance activities. The task of this staff member is to surf the Internet and spot companies or individuals that infringe the GPL.[50] When a violation is discovered the staff member typically performs background research and composes a letter that is sent by either Stallman or Kuhn to inform the company or individual that they have infringed on the license and should change their policy. The other FSF staff perform various activities: raising money, creating awareness of free software and maintaining the software infrastructure on which the GNU projects are hosted.

The president and the vice-president of the FSF are responsible for creating new versions of the GPL, and they publicly respond to attacks against open source communities. Both Stallman and Kuhn, for instance, sat at the table with Affero and a lawyer to construct the new version of the GPL. They have also been interviewed by various media to give their opinion on the SCO attacks.[51]

Finally, and most importantly, the FSF owns the GPL and GNU trademark; and it holds the copyright on all software that is part of the GNU project. The vice-president of the FSF explained, "We encourage people to assign their copyrights to the FSF." Over the years many developers have signed their copyrights over to the FSF, which are stacked in a big filing cabinet. "Richard Stallman has a closet full of signed agreements making the FSF the licensee of the software," one respondent remarked.[52] In other words, the FSF is the collective copyright owner of many of the projects that fall under the GNU umbrella.

The Apache Software Foundation

The Apache Software Foundation (ASF), like the ASL, was created in 1998. At that time, IBM had decided to adopt the Apache software, but wanted to have a single entity it could approach and correspond with. IBM largely funded the erection of the ASF,[53] which is a membership foundation, with membership based on invitation. In August 2003, the ASF consisted of 100 members, nine of whom were on the board of directors.[54]

The ASF serves Apache developers by protecting them against possible lawsuits from companies:

> We are developing open source software for free, but that does not mean that we want to get poor out of doing that. Therefore we started a legal entity. We decided to create a not-for-profit organization that owns the copyright… It is a signal to other parties and companies that it is safe to use the code, because there is a serious and legal party behind it. It is a level of formalization and legalization we as a project had to go through.[55]

Those who want to contribute code to the Apache server must first sign an agreement in which they state that (i) they own the copyright on any piece of source code they contribute to the project and (ii) they transfer to the ASF the copyright on every piece of source code they contribute to the project.

The KDE Free Qt Foundation

Qt, which is licensed under the GPL, lies at the heart of one of the most popular Linux desktop environments, namely KDE. Qt is a graphical toolkit created and maintained by the company Trolltech. A graphical toolkit is a library used by developers to create a graphical environment. Other programs are linked to this library.

Qt used to be a source of much controversy. When Trolltech decided to provide an open source version of its library it initially issued the library under the QPL license. This license states that no entity is allowed to use Qt for commercial purposes. Many developers felt that this was not in the open source and free software spirit. One of the basic principles in the open source definition is that a license should not discriminate against 'fields of endeavor,' which means that a license may not restrict the use of a program in, for instance, business.[56] However, basically the license did intend to prevent such use. Eventually Trolltech capitulated and decided to re-license Qt under the GPL.[57] However, many developers still mistrusted Trolltech and its intentions. To express its alliance with open source communities, Trolltech and the KDE community decided to create a special foundation, namely the KDE Free Qt Foundation.

The KDE Free Qt Foundation defines its mission as follows:

> [to] control the rights to the Qt Free Edition and ensure that current and future releases of Qt will be available for free software development at all times. All changes to the Qt Free Edition license will have to be approved by the KDE Free Qt Foundation… Should Trolltech ever discontinue the Qt Free Edition for any reason including, but not limited to, a buy-out of Trolltech, a merger or bankruptcy, the latest version of the Qt Free Edition will be released under the BSD license.[58]

Thus, the foundation owns the copyright on Qt and it provides a safety measure that there will always be an open source version of Qt. The foundation extends insurance to developers and users that they can use Qt without having to worry about the strategies and actions of Trolltech. Finally, the erection of the foundation improved Trolltech's image in the community.

Software in the Public Interest[59]

The foundation Software in the Public Interest (SPI) was erected in 1997. The SPI serves as an umbrella for a number of communities. One of these communities is Debian. The SPI is officially the copyright owner of Debian. According to a package maintainer in the Debian community, the SPI was erected to minimize the chance of lawsuits against contributors in the Debian community. The foundation has a ten-member board of directors with four board members also serving as SPI officers.

The Open Source Initiative

In 1998 Bruce Perens and Eric Raymond decided to use the term 'open source' to promote free software in companies. They registered the term as a certification mark, with a binding legal definition that enabled them to defend it against abuse.[60] At first, the term was connected to one particular community, which was Debian. However, open source was supposed to be a general term for the entire community and not tied to a single project. To make this clear the pair decided to establish a new nonprofit organization called the Open Source Initiative (OSI).

The OSI consists of five people who together form the board. Two others are affiliated with the foundation; one is lawyer and the other is webmaster. What is striking is that the five board members and the two affiliated people seem to spend a minimal amount of time on OSI, as most have full-time jobs and are project leaders in various free software communities.[61]

The foundation has a number of goals:

> to own and defend the Open Source trademark, to manage the www.opensource.org resources, to develop branding programs attractive to software customers and producers, and to advance the cause of open-source software and serve the hacker community in other appropriate ways.[62]

The most interesting aspect of the OSI is its certification program. The basis of the program is the open source definition, which describes the exact requirements of an open source license. Companies and individuals who decide to create their own license can submit the license to the OSI, which then decides whether it conforms to its definition. If it does, the license becomes officially 'OSI certified' and the software protected with the license is considered to be open source. According to the OSI, "the community needs a reliable way of knowing whether a piece of software really is open source."[63]

Differences and similarities between the foundations

The foundations protect the open source development process (O'Mahony 2003). There seems to be a link between the size and age of a community and incorporation. When the communities of developers grow and when the software is used by an increasing number of people and companies, then foundations seem to emerge. Each of the foundations protects the open source development process.

Table 4.3 presents an overview of the foundations. The overview is structured along the activities that the foundations perform. The first activity is *to host a community of software developers*.

The FSF, ASF, SPI and the KDE Free Qt Foundation host one or more communities. The second activity performed for most communities is *to collect copyrights*. Of the five communities, only the KDE Free Qt Foundation and the Open Source Initiative do not collect the copyrights from the developers who contribute source code to the community. The third activity is *to maintain a particular license*. Only the ASF and the FSF are copyright holders of a license and as such in name they are responsible for the maintenance of the license. The fourth activity is *to evaluate the openness of licenses*. Both the FSF and the OSI have created a website on which they maintain a list of licenses which they believe are open or, in the case of the FSF, free. Compared to the FSF, the OSI has a more official standing. It is the owner of the open source trademark and it determines whether a license complies with the open source definition and can be called an open source license.

Table 4.3 – The main activities of the foundations

Foundation	**To host at least one community**	**To collect copyrights**	**To maintain a license**	**To evaluate openness of licenses**
FSF	x	x	x	x
ASF	x	x	x	
SPI	x	x		
KDE Free Qt Foundation	x			
OSI				x

Foundations protect the open source development process

The first way in which the foundations protect the open source development process is by reducing the liability of individual developers. In communities without a foundation, companies can sue each developer individually. The developers are copyright holders of the source code they contributed. If their contribution infringes an IPR, the holder of the IPR can sue them. Foundations reduce individual liability because the individual contributors have handed their copyright over to the foundation. Now the foundation is the copyright holder and as such is the entity sued when the software is claimed to infringe a patent or copyright. Thus, foundations provide developers with some insurance against legal action.

At least one foundation also tries to reduce the liability of users of the software. According to an ASF board member, the Apache Software License includes a clause which states that the ASF remains responsible for the software. The ASL thus tries to eliminate any liability of the user by placing full responsibility at the ASF.

The fact that foundations can act as one entity is another way in which they protect the development process. Open source licenses depend on copyrights and according to copyright law, only the copyright holder is allowed to sue for copyright infringement. "Without the copyright we can do nothing when someone infringes the GPL. We collect copyrights because we are the custodians of the public interest."[64] Collecting the copyrights on software enables a foundation to go to court. This does not mean, however, that the foundations go to court frequently. The FSF, for instance, claims that it has never needed to go to court to enforce the GPL. Usually a simple e-mail is enough to change the behavior of the infringing party.[65] The mere presence of a foundation that owns the collective copyright and is able to file a lawsuit is said to provide sufficient protection against infringement of the license.[66]

Finally, foundations protect the open source development process because, according to a number of respondents, companies are unlikely to sue a nonprofit organization. According to a member of the ASF, suing a nonprofit organization makes a company look bad. Furthermore, there is not much money to be gained from suing nonprofits.

Foundations are lean legal institutions

Foundations are an efficient legal means to enforce compliance with open source licenses and to protect the open source development processes. They are efficient because they are surprisingly small when measured in staff and expenses. The *Free Software Foundation* is probably the best known. Its president and vice-president appear in the news regularly. Furthermore, the FSF is the copyright holder and maintainer of the GPL, which is the most used license. It also hosts the GNU C Library, which is vital in the development of open source software. Still, the foundation itself is small. "Remember we are a charity organization with just ten employees."[67]

The FSF is the only foundation that has a headquarters. It is located in a relatively old building in Boston, where it rents two small offices on different floors. Upon arrival one cannot help noticing the disorder of the building, the elevators and the two offices. One of the offices is only used to stack promotional material, like leaflets, books, T-shirts and caps. The other is used as an office. The physical location of the FSF is not what one might expect of the headquarters of an organization that is considered to be one of the most important institutions for software used by millions of people and an increasing number of large multinationals.

Not only is the physical location small and the number of staff limited, the money available to the foundation is also relatively scarce. The FSF raises money "through book sales of printed material about GNU projects. Next to these revenues it depends on charitable donations."[68] Both the president and vice-president continuously stress the fact that there is little money to spend on trials, salaries and other promotional activities: "Microsoft's budget is 5,208 bigger than ours."[69]

The foundation *Software in the Public Interest* is also small. It has no staff and, most striking, it has made no public statements in the past three years. Neither has it been in the news more than two or three times during the last years.[70] The *KDE Free Qt Foundation* is comparable to the SPI in that it has no staff and has hardly been in the news. This indicates the limited scope of activities undertaken by both foundations.

The final two foundations, the *Apache Software Foundation* and the *Open Source Initiative*, are also small. They have no staff and no headquarters. Compared to the SPI and the KDE Free Qt Foundation both are, however, much more active. The main activities of the ASF are to announce new releases, maintain contacts with companies like IBM and check the origin of new additions to verify that they are free of IPR held by a third party.[71] The OSI regularly registers new licenses and warns about licenses that do not comply with its open source definition.

Institutions or individual choice: the importance of individual choice

Can the observations in this chapter be understood based on the presence of institutions? Or are behaviors in the communities based on individual choice? At first glance one might be tempted to conclude that this design principle, *clearly defined boundaries*, emanates primarily from institutions. One example of an institution discussed in this chapter is the open source license. However, open source licenses are largely governed by individual choice. This is evidenced in a number of ways. First, the open source licenses are "not about the absence or irrelevance of intellectual property rights" (McGowan 2001, p. 244). On the contrary, the licenses depend on copyright law. Copyright is a right that, by definition, is connected to individuals, namely to authors, artists and composers of original works (Van Wendel de Joode et al 2003). Upon contributing source code to a specific community, contributors retain their copyright. It is thus the individual who builds the defense line. It is a patchwork of individual copyrights that together form a boundary. Second, though the licenses are collectively available on websites, the choice of license is an individual one. Many communities start with a company or an individual deciding to make software available on the Internet. At that point a license is chosen. This choice is made by the individual or company that posts the software. Once the license is chosen it is taken as given[72] and is frequently perceived as just another characteristic of a community.

A second institution is the foundation. The foundations serve as a boundary against appropriation attempts and lawsuits. They are institutions that protect the open source communities. In that sense the foundations would seem to suggest that part of the design principle is addressed through institutions. Yet these institutions do leave ample space for individual choice. For instance, even though foundations collect copyrights, it is still the choice of the individual developers to sign the copyright agreement. "The GNU and Apache projects are probably the most active foundations in encouraging contributors to assign their copyrights, but even they are generally reluctant to make this a condition of contribution" (O'Mahony 2003, 1191). Furthermore, while the foundations protect the boundaries they typically refrain from interfering with actual activities in the communities.[73] This observation is also in line with the observation that most of the foundations are lean and efficient.

Finally, it is predominantly individuals who defend the boundaries of the communities. Individuals perform the "task of monitoring and identifying license violations" (O'Mahony 2003, p. 1187). Consider the news sites. They are an important mechanism for demarcating and protecting the boundaries of open source communities. Collection of news items, however, is performed by individuals who stumble upon interesting, informative or blunt items which they then decide to submit to the news site. The same is true for the identification of license violations. Foundations like the FSF rely mainly on the efforts of volunteers in the communities to inform them of license infringements (O'Mahony 2003).

Conclusion: redundancy of boundaries

Boundaries in communities are generally needed (i) to create a feeling of belonging, (ii) to determine who is outside or inside, and (iii) decide who has the right on the fruits of the efforts of the community members. Research on common pool resources stresses that not only organizational boundaries must be created; boundaries of the resources are equally important.

Boundaries are also needed in open source communities, as the resource is susceptible to depletion. Yet what is striking is that the communities in general lack a boundary to limit the size of their membership. In other words, organizational boundaries tend to be absent. The boundaries that are present contribute to the protection of (i) the source code, (ii) the community, (iii) the participants in the communities and (iv) the users of the software. Open source licenses are probably the most important boundaries of open source software. The licenses depend on copyright and essentially state that others may use and modify the source code. Participants from many communities have created a redundant system of many different licenses. The system of licenses is not static; instead, it is developing and continuously modified to meet changing requirements. Enforcement of the licenses is supported by a great number of mailing lists and news sites. These mailing lists are used by a large portion of the participants in open source communities to collect articles and distribute interesting news items. Both the licenses and news sites are embedded in a system of foundations.

Combined, these mechanisms effect a system of redundant protection mechanisms, which make the software produced in the communities highly resilient to appropriation. If a particular license fails to protect software from appropriation, then the mailing lists and possibly even a foundation can serve as a back-up for influencing public opinion and starting legal action.

Supporting the observations: individual behavioral rules

One of the questions that remains is what explains the great variety in licenses and their dynamics. This is especially interesting when we consider that the choice of license occurs typically only once in the lifetime of a community and when we realize that this selection is frequently based on the choice of an individual developer or company. How do they choose their license and why do they so frequently select the GPL to protect their software?

A potential answer to these questions, especially those concerning the licenses, is that individuals in the communities act according to a similar underlying logic. From interviews with developers and from secondary literature a small number of individual behavioral rules[74] can be deduced that are surprisingly simple and yet they are sufficient to understand many of the observations on a collective level. This section proposes three rules, which are the result of a quest to understand the underlying logic of the observations. The rules do not provide an ultimate answer, but are postulated as propositions. Furthermore, the claim is not that every individual actually acts the same as other individuals. The rules are formulated in such a way that they reflect trade-offs. Different individuals make different choices and thus act differently, based on these trade-offs.

"*Participants adopt a license that maximizes the guarantee that they will benefit from the participation of others.*" The original creators of a software development project or a new software module typically select a license only once, which is when they place the first version of the software on the Internet. They then have the option of selecting an existing license or creating a new one. The license they choose should maximize the likelihood that they will benefit from other people's use of the software and changes and improvements made to it. The license should thus maximize the chance that the creator receives something in return.

"*Participants adopt licenses that minimize the barriers for others to participate.*" Typically, individuals and companies make source code available on the Internet in order to attract others to use and participate in the maintenance and improvement of the software. To achieve this they adopt existing licenses or create new ones that they believe to be sufficiently simple and provide sufficient motives for others to become involved in the communities.

"*Participants spend limited time analyzing licenses.*" Time is the scarce resource in open source communities (see also Hertel et al 2003). Therefore, most participants want to spend as little time as possible on activities related to licenses. They want to spend their time on other activities, for example, writing new source code.

How the individual behavioral rules support the observations

The first two rules reflect a trade-off. The wish to use a license to motivate others to participate is in tension with the wish to build in as much of a guarantee as possible to profit from other people's adoption, use and modification of the software. This trade-off is different for each participant, which explains the rise of divergence in licenses. The BSD license, for instance, is claimed to offer much incentive for companies to participate in a community (e.g. Bonaccorsi & Rossi 2003b),[75] whereas the GPL provides software creators a greater chance of benefiting from other people's participation in development (Perens 1999). Deviations from these two licenses arise due to small differences in individual preferences. The creator of the Vim license, for instance, diverged from the GPL by including a clause obliging others to "make the changes to the source code publicly accessible, or send them to me."[76]

The desire to spend only a limited amount of time analyzing and/or creating licenses constitutes a counterforce against the rising tide of ever more varied licenses. In most cases participants prefer to download software with a familiar license over software with a new and relatively unknown license. This means that, for comparable software, developers would rather download GPL-licensed software than software licensed with a relatively unknown license like the Vim license. This provides an additional break on the rise of divergence, as adopting a popular license – instead of creating a new one – increases the likelihood that others will adopt the software and contribute to it.

The fact that the licenses are dynamic can be understood by the desire of individuals and companies to have as much security or guarantee as possible that they will benefit from the adoption of the software by others. If a license, for whatever reason, turns out to have loopholes that allow others to appropriate the software or to use it in ways that are not intended in the license, then there is a big chance that the software will become unpopular and attract few new participants. To continue to attract new participants the license thus has to be updated.

Finally, the foundations and the mailing lists complement the function of the licenses in open source communities. Together they educate potential users about the licenses and stimulate compliance with the licenses. Chapter eight, focuses on monitoring and sanctioning, addressing in more detail the reason why individuals use mailing lists to stimulate compliance. The foundations are the only institutions that cannot be understood in terms of individual rules. They are an example of incorporation in the communities. The fact that foundations are created does not, however, conflict with the conclusion that individual choices govern the open source licenses.

Notes on chapter four

[1] The ASF is a foundation created to protect the source code and the participants in the Apache community from all kinds of legal threats. The foundation will be presented and discussed in more detail in another part of this chapter.
[2] The name 'infected code' is adopted from an article on the Internet: http://www.crn.com/Components/printArticle.asp?ArticleID=43781 (September2003).
[3] Cited from an interview with the president of the Free Software Foundation (FSF).
[4] The FSF is one of the best known foundations in the realm of open source and free software. Its exact role and activities will be discussed in a later part of this chapter.
[5] In the interview the respondent even took one step further and argued that it is hardly possible to write any kind of software without infringing a patent.
[6] From an article written by Matthew Anslett, from the Internet: http://www.cbronline.com/latestnews/62cbf9d13b40711e80256d880018c80f (August 2003).
[7] From an article on the Internet: http://www.theregister.co.uk/content/4/31938.html (July 2003).
[8] From an article on the Internet: http://www.eweek.com/article2/0,3959,1224877,00.asp (August 2003).
[9] From the Internet: http://armedndangerous.blogspot.com/2003_08_17_armedndangerous_archive.html (September 2003).
[10] This point was also made by Daniel Egger from Open Source Risk Management (OSRM) in a talk given at the open source and open source standards conference, September 12-14, at Scottsdale, Arizona.
[11] From an interview with a former project leader of the Debian community.
[12] Many researchers have addressed the positive effects of redundancy on the resilience of systems and organizations, e.g. Bendor JB. 1985. *Parallel Systems: Redundancy in government*. Berkeley: University of California Press, Chisholm D. 1989. *Coordination without Hierarchy: Informal structures in multiorganizational systems*: University of California Press, Landau M. 1969. Redundancy, Rationality, and the Problem of Duplication and Overlap. *Public Administration Review* 29: 346-58
[13] From an interview with a member of the Board of Directors of the Apache Software Foundation.
[14] The respondent is a developer in the Linux community.
[15] From an interview with the vice-president of the FSF.
[16] Adopted from the Internet: software.freshmeat.net/stats/ (September 2003).
[17] Cited from an interview with the president of the FSF, Richard Stallman.
[18] The GPL states that distribution of the source code is not obligatory, as long as the software is kept private, i.e. not distributed. It is unclear, however, when source code is actually kept private and when it is distributed. Consider the example in which a developer in a company modifies GPL-protected software and the source code is distributed throughout the company. It is generally accepted that distribution within a company does not infringe the GPL. In other words, in such a case the source code would not have to be made available. What if the company consists of business units or franchises? Are the franchises still part of the company?
[19] From an interview with the creator of the GPL, Richard Stallman.
[20] From the license, which is available at: http://www.gnu.org/copyleft/gpl.html (July 2003).
[21] Based on an interview with the editor in chief of the Linux journal.
[22] The license and the preamble can be found on the Internet: http://www.gnu.org/copyleft/lesser.html (July 2003).
[23] Based on the preamble of the LGPL.
[24] The website http://www.gnu.org/philosophy/license-list.html (August 2003), which is hosted by the FSF, explains that 'X11 license' is a more appropriate name than MIT license. The latter is confusing, as MIT has used many licenses for its software.
[25] From an interview with the editor in chief of the Linux journal.
[26] This argument was heard in many interviews, for example, during an interview with two members of the ASF.
[27] This section is largely based on interviews with two members of the ASF Board of Directors. Both were actively involved in the development of Apache at the time of IBM's decision to become involved.
[28] Cited from the license. The text of the license can be found on different places on the Internet, e.g. http://www.opensource.org/licenses/apachepl.php (last visited August 2003).

[29] Currently, the second version of the ASL has been created. This license contains more protection mechanisms, which are primarily intended to safeguard the individual participants in the Apache community and ensure the continuous improvement of the software.
[30] Indeed, the two projects, Mozilla and OpenOffice.org, that use the MPL and SISSL have suffered from lack of participation. Respondents also mentioned other reasons why both projects initially had a difficult time, but uncertainty about the license is among them.
[31] From the Internet: http://www.ososs.nl/index.jsp?page=809 (August 2003).
[32] From the Internet: http://www.gnu.org/philosophy/license-list.html#TOCGPLIncompatibleLicenses (September 2003).
[33] The ruling does not allow distribution of GPL-covered code when it is governed by additional conditions that conflict with the conditions of the GPL.
[34] The concept of fair use is only relevant in American copyright law.
[35] From an interview with the vice-president of the FSF.
[36] Based on an interview with the vice-president of the FSF.
[37] This example is based on an interview with the vice-president of the FSF.
[38] One reason why licenses are not changed is because every contributor of source code has to agree to such a change. However, there are examples of communities in which the software license was changed in the course of time. One is the Qt library, which used to have a specially created license but later became licensed with the GPL.
[39] The information presented here is based on an interview with the author and maintainer of Vim.
[40] Freshmeat is a website that provides a summarized overview of many open source software projects (http:// freshmeat.net, September 2003). Their website contains statistics on the number of projects that adopt a particular license. These show that the GPL is used in 65 percent of the projects (http://software.freshmeat.net/stats, September, 2003). The statistics do not correlate the license and the adoption rate of the software; it could very well be that the distribution of licenses is much more equally spread were one to consider only the bigger projects. However, the enormous differences between the licenses do not suggest that this is true. An interesting proposition would be to try to understand the relationship between 'success,' as for instance, measured in lines of code or number of participants, and type of license.
[41] See the interview with the creator of the website on the Groklaw website (http://www.groklaw.net/staticpages/index.php?page=20031004190519196, September 2004).
[42] From the Internet: http://linuxtoday.com/news_story.php3?ltsn=2003-09-10-016-26-OS-CD-CY (September, 2003).
[43] Both citations are from an open letter by the CEO of SCO: http://linuxworld.com/story/34007.htm (September, 2003).
[44] Based on numerous articles available on the Internet, for instance, an article featured in Wired: http://www.wired.com/news/business/0,1367,59701,00.html (September 2003). Even Bill Gates has commented on the issue, stating that SCO's case will harm Linux's commercial prospects: http://crn.channelsupersearch.com/news/crn/43532.asp (September 2003).
[45] From the Internet: www.theregister.co.uk/content/4/31938.html (September 2003).
[46] From an article on the Internet: http://www.eweek.com/article2/0,4149,1258642,00.asp (September 2003).
[47] From an article on the Internet: http://crn.channelsupersearch.com/news/crn/43781.asp (September 2003).
[48] From an interview with the vice-president of the FSF.
[49] Based on an interview with the vice-president of the FSF.
[50] Based on an interview with the staff member of the ASF. His official position is called 'free software licensing guru.'
[51] For example, Computerworld, ZDNet and Linux Today.
[52] Based on an interview with two respondents who are active members of the Debian community.
[53] Based on an interview with a member of the ASF Board of Directors.
[54] A list with ASF members and members of the ASF Board of Directors can be found on the Internet: http://www.apache.org/foundation/ (August 2003).
[55] From an interview with a member of the ASF Board of Directors.
[56] The definition can be found in many places on the Internet, for instance: http://www.oscommerce.com/about/opensource (September 2004).
[57] From two interviews with three package maintainers from the Debian community.
[58] Taken from the Internet: http://www.kde.org/kdeqtfoundation.html (August 2002).
[59] This information was taken from the Internet: http://www.spi-inc.org/ (August 2002).

[60] From the Internet: http://www.opensource.org/pressreleases/osi-launch.php (August 2002).
[61] From the Internet: http://opensource.org/docs/board.php (August 2003).
[62] From the Internet: http://www.opensource.org/pressreleases/osi-launch.php (August 2002).
[63] From the Internet: http://www.opensource.org/docs/certification_mark.php (August 2002).
[64] From an interview with the president of the FSF.
[65] This finding came up in a number of interviews.
[66] Partly from an interview with two package maintainers of the Debian community.
[67] From an interview with the president of the FSF.
[68] From an interview with the vice-president of the FSF.
[69] From an interview with the vice-president of the FSF.
[70] According to the Slashdot website, which is a popular news site for all kinds of open source news, the last time SPI was in the news was on November 25, 2002 and even this news was only about an internal change in the foundation.[70]
[71] Based on a presentation by a member of the Board of Directors of the ASF, in Rotterdam, September 15, 2004.
[72] This excludes the adoption of a new version of the license. The point is that no *different* license is chosen. The license itself might be subject to chance.
[73] Based on a personal interview with a member of the Board of Directors of the ASF and on a talk given by the current president of the ASF at an informal open source meeting in Rotterdam, September 15, 2004.
[74] The term 'individual behavioral rule' is adopted here. The rules are 'behavioral' because they explain the actual behavior (see also chapter two) and they are 'individual' because they explain the behavior of individuals in the communities, and not of actors, which could include companies or groups.
[75] This point is also based on an interview with two members of the ASF, who argue that they felt the BSD license to be a very reasonable license. A company that invests time and effort in the improvement of a software program, they said, should have enough freedom to make a return on their investment and therefore the license should have fewer restrictions than imposed by the GPL.
[76] From an interview with the author and maintainer of Vim.

CHAPTER FIVE

PROVISION AND APPROPRIATION

The focus of this chapter is the second design principle: *congruent appropriation and provision rules*. First, the design principle is presented and discussed. Then it is argued that (additional) appropriation rules are hardly needed in open source communities, for two reasons: (i) The boundaries in the communities already provide some protection against the threat of appropriation. (ii) Usage of software, like downloading and installing software on a local computer, does not diminish the amount of software that is available to others.

The communities do need provision rules to structure the development and maintenance of the software. The chapter argues that provision in the communities is based on individual choice and is hardly structured by any form of central planning or formalized task identification and allocation. Instead, participants in the communities have created and adopted a great number of mechanisms that enable them to create software without much collaboration and personal interaction.

The second design principle: provision and appropriation rules

This design principle addresses two distinct problems that face common pool resources (Ostrom 1990, Ostrom et al 1994). They are the problem of appropriation and the problem of provision. Regarding appropriation, "the problems to be solved relate to excluding potential beneficiaries and allocating the subtractable flow" (Ostrom et al 1994, p. 9). The appropriation problem concerns the resource flow. Somehow, people in the communities that govern common pool resources must reach consensus about the flow of resources from the resource stock. The question is *when* is *who* allowed to appropriate *what* part of the common pool resource and with what *technology*? The primary aspect of the appropriation problem is the assignment problem: Who is assigned to what piece of the common pool resource? Another aspect of the appropriation problem is the presence of technological externalities, which relate to the means by which people appropriate from the common pool resource (Ostrom et al 1994). For natural resources like fishing grounds this externality can have serious impacts on the appropriation problem. It makes a big difference whether someone appropriates fish from the fishing grounds with a hook and line or with a fishing net.

The provision problem can be described as the difficulties "related to creating a resource, maintaining or improving the production capabilities of the resource, or avoiding the destruction of the resource… In provision problems, the *resource facility or resource stock* of the CPR is problematic" (Ostrom et al 1994, p. 9). Provision problems are related to the construction and maintenance of the resource. One of these is that people are tempted to free ride; they are not intrinsically motivated to contribute their time and effort to construct and maintain a resource if they can enjoy the benefits for free. *If* people are motivated to contribute, a second problem arises, namely, 'How should the resource be constructed and maintained?' Motivated people who want to contribute is not sufficient. Their contributions must benefit the resource.

The design principle 'appropriation and provision rules' prescribes that people must formulate regulations that determine (i) how much and when people in the community are allowed to appropriate and (ii) how much people should contribute to the provision of the common pool resource.

Little need for appropriation rules in open source communities

In open source communities appropriation rules are not needed for two reasons. First, boundaries in the communities constitute a protection mechanism against appropriation. Second, most types of usages of open source software do not affect the amount of the good available to others.

The boundaries as appropriation rules

Appropriation rules regulate who is allowed to extract what portion from the common pool resource. In open source communities 'extraction' means to appropriate a piece of source code from a software program and disallow others from using it. Earlier chapters argued that intellectual property rights like copyright and patents enable this type of appropriation. In response to this threat, chapter four argued that the communities have created open source licenses. These licenses are an important part of the boundaries of the communities, as they intend to prevent extraction of source code from a software program.

Open source licenses are different from the boundaries described for community-governed common pool resources. In the latter, boundaries serve two goals. They demarcate the borders of the resource and they identify the members of the community. One reason to separate members from non-members is that only members are allowed to appropriate from the common pool resource. Furthermore, appropriation rules are needed to ensure that members do not appropriate too much of the resource. The rules protect the continuity of the resource.

Comparing the two types of boundaries, the conclusion must be reached that open source communities need no additional appropriation rules that apply to members of a community. The reason is that the goal of open source licenses is to prevent appropriation by anyone.

Use of software does not affect the amount of software available

In many 'traditional'[1] community-governed common pool resources, community members depend on the appropriation or consumption of the resource. Yet their consumption of the resource affects the amount available to others. Consider the consumption of fish or beef. In both situations the stock is affected.

This is different for open source communities. Previous chapters argued that appropriation of software is possible and that this is enabled through intellectual property rights. This is one type of 'consumption' of software. However, most types of consumption – or rather most forms of use – of software do not affect the amount of software available to others. Copying of software is basically unlimited. There is no restraint. To simply download software from a website or install it from a compact disk does not affect the amount of software left on that website or CD. No regulation in the form of appropriation rules is therefore necessary, and such rules are even irrelevant with respect to the use of software.

The need for provision rules in open source communities

The development and maintenance of software in open source communities can be compared to provisioning in community-governed common pool resources. The development of source code is the actual writing of source code and the invention of new and innovative ideas. Yet development is not confined to innovation and novelty; it also involves refinement of the software. If someone writes a certain piece of source code, other people analyze that code and make improvements where necessary. These improvements can take a variety of forms. People encounter, report and possibly fix bugs, write new features and rewrite existing architectures.

Maintenance of source code is difficult to separate from the actual creation or development of the code. Software is never finished and requires continuous improvement, which blurs the distinction between maintenance and development. Yet certain activities are more oriented toward maintenance than development. Many communities, for instance, maintain an old version of the software while continuing to modify it to keep it compatible with the latest hardware. The older version, however, is not otherwise improved and no new features are added. Other maintenance activities are the translation of a program into foreign languages, creation of manuals on how to use the software, maintenance of a server on which the software is stored and hosting mailing lists to discuss the software.

Why are provision rules in open source communities relevant and important?

Most open source software development projects start with an individual or company that decides to make software available on the Internet. Others are invited to download the software and install it on a local computer. The creator usually becomes the project leader or maintainer of the source code. This implies that users send to the software creator their comments, suggestions, ideas, bug fixes or anything else. The creator is expected to act on these contributions. A response could be that the creator explains why a contribution of code was not accepted or an announcement of a bug fix.[2]

As the community grows in size it becomes clear that many different activities need to be performed and that they must somehow be managed. Consider this statement:

> When more than one programmer started working on a project together… everyone needed to work on coordinating their work with each other. One person couldn't start tearing apart the menus because another might be trying to hook up the menus to a new file system. If both started working on the same part, the changes would be difficult if not impossible to sort out when both were done (Wayner 2000, p. 195).

Developers in the BlueFish community faced a similar problem. At first, they simply e-mailed each other at the start of the development project. In the beginning this worked fine.

Soon, however, they realized that this system did not scale. Each morning and afternoon they needed to send around e-mails explaining what they had done and what they were going to do next. In the end they were spending so much time communicating their activities that less and less time was left to actually develop and maintain the software. And every time communication failed the developers had to spend time to "attune two different versions of a part of the program."[3] In other words, one-to-one communication no longer worked; the community had grown too large.

To summarize, when communities grow and start to involve many participants rules appear to be needed to govern how and when people perform a certain activity. The need for such 'provision rules' becomes even more evident when we realize that boundaries to control the entry of new developers are absent. As discussed in the previous chapter, membership in the communities is fluid. This not only applies to the users of the software. It also applies to participants who want to add new contributions to the software. Provision rules are needed to deal with these contributions and to structure the development and maintenance activities.

Note: the presumed presence of many motivated individuals

The remainder of this chapter identifies a great number of mechanisms that it argues support coordination in the communities or reduce the need for coordination. It is important to note that a commonly made assumption is that popular communities like Apache and Linux consist of many individuals who are motivated to become involved and want to contribute their time and effort.

The introduction of this book already demonstrated the wide variety of motives that drive participants to become involved in open source communities. Participants want to learn or build a reputation, they may have a personal need for the software or consider it fun to be involved. In a sense, the influx of these people reduces the need for provision rules. In community-governed common pool resources, the rules are also intended to ensure that people perform a particular task or activity. Thus, rules in these communities are also needed to compensate for a potential lack of motivation.

In this research the presence of motivated individuals is to some degree accepted as given.[4] Therefore, this chapter focuses on the question, given that people are motivated; 'How do people coordinate their activities and ensure that they result in a software program that works?' How are activities divided among individuals? Who or what determines what activity a participant is supposed to perform?

Individualism dominates the provision of open source software

Theory would predict that developers are mutually dependent and must collaborate to develop a complex product like software (De Bruijn & Ten Heuvelhof 2000, De Bruijn 2002). However, many participants in open source communities claim to hardly collaborate and coordinate their activities. Instead, they claim to develop and maintain the software in a very individualistic way, based on their individual choices and preferences:

> Well, you have a pool of people who do what they want to do. Nobody gives me an assignment. Instead, I think, 'hey how weird, this process is very slow'. Well, then I myself will

> start to work to solve it… This is of course a very selfish approach, but I think that most people work like that. They work on what they run into.[5]

Developers decide for themselves what they find interesting and what they would like to work on. "The nice thing about open source is that you can make what you want to add." One respondent claimed that individual choice results in an anarchistic community: "Linux is anarchy! Everybody does what he feels like doing."[6]

The fact that open source developers behave individualistically has three consequences. First, participants in the communities have no way of ensuring that other participants will perform the activities they are asked to perform. Participants work independently and no one can force anyone to do anything, especially when they do not want to do these activities. The vice-president of the FSF claimed that few free software developers work together. "A company like Microsoft hires developers, puts them in a room and makes them work together. We don't because we do not have walls… We cannot make them do anything, we can only suggest."

The second consequence is that things only get done when someone wants to do them. According to an ASF board member, "That is actually what the entire open source philosophy is about. Things only get done if at least one person feels that they are important. That person makes sure that it works." In other words, an activity will not be performed unless someone in the community considers it important.

The third consequence of individualism is that participants in the community do not keep track of what others are doing. "We try and keep things simple. We like to spend more time getting stuff done instead of tracking everything."[7] The developers in the communities can be compared to ants in "a big ant colony. You don't know what someone else is doing."[8]

The reliance and importance of individual choice appears to contradict the previous observation, namely, that provision rules are needed. The previous section claimed that individuals need to divide activities and require ways to coordinate who does what. However, this section claims that most individuals simply do what they please. The sections below identify mechanisms that are argued to relieve the need for formal planning and coordination and support a very decentralized and individualistic development process.

Mechanisms to relieve the need for collaboration and coordination

If software becomes complex and difficult to understand the need for collaboration increases, as does the need to define and maintain provision rules. In such a situation, changes to one part of a program would likely have the effect that "something else does not work anymore"[9] and therefore the software development process "is no longer controllable."[10] To prevent these problems and to relieve some of the need for rules and structuring activities, developers in open source communities have two guidelines, or rather, two mechanisms. These allow the number of lines of source code to grow without proportionally increasing the overhead needed to collaborate and coordinate. The mechanisms are *elegance* and *modularity*.

Elegance of software: making the software easier to understand

Elegance is a technical characteristic of software:

> [It combines] simplicity, power and a certain ineffable grace of design… The French aviator, adventurer, and author Antoine de Saint-Exupéry, probably best known for his classic children's book *The Little Prince*, was also an aircraft designer. He gave us perhaps the best definition of engineering elegance when he said "A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away."[11]

Donald Knuth is often seen as the founder of the concept of elegant code.[12] According to Ellen Ullman, elegance is structured and reductive. It is a term used to indicate that software works and, at the same time, a notion of beauty. "So from the standpoint of a small group of engineers, you are striving for something that is structured and lovely in its structuredness."[13]

Elegance is sometimes claimed to be an indisputable characteristic of source code. Source code is either elegant or it is not. The more experienced and skilled software programmers are claimed to be best judges of whether source code is elegant. "If you narrow your circle to a small group of good developers then it is easy to decide what software is elegant and what is not."[14]

From an organizational perspective, elegance performs a role in enabling coordination and collaboration in social groups. Compared to inelegant source code, elegant code is effectively easier to understand and likely to better express "what it is doing while you are reading it."[15] It implies that the software will not be doing something "in a non-intuitive, ineffective way."[16] The fact that elegant source code is easier to understand has a number of advantages. First, it is easier to make changes to it. "You can only work when the software is beautiful and elegant, to be able to implement changes easily."[17] Thus, although the number of lines of source code is bound to increase when the functionality of software is enriched, an elegantly written piece of software provides some counterforce to complexity and to a certain degree ensures that the code remains relatively easy to understand and to change.

Another advantage of elegance is that participants have to invest less time and effort to understand what the source code is trying to accomplish. From reading elegant source code, programmers claim to be able to understand what the source code aims to achieve and to judge whether the source code indeed can accomplish the task. The relative ease with which developers can understand a certain piece of code effectively lowers the time needed to decide whether a certain piece of source code is good. This enables them to make decisions without paying more attention to reading and understanding source code than is strictly needed. Elegance, for instance, enables maintainers or others in the community to decide whether a patch should be included. It also allows people who were not previously involved in the community to improve code without spending much time and effort deciphering the source code.

Elegance also relieves the need for coordination and collaboration. Because the code is elegant it is easy to understand what the effects of a change in one part of the software will be for other parts. Elegance allows developers to either adjust other parts of the software or to ask others to take a look at it. If the source code were inelegant, the chance is much higher that a minor change in one aspect would have dramatic effects for others. In that case, developers must collaborate, simply because they cannot oversee the consequences of their changes. They need the help of others to discover and solve the mutual dependencies in the software.

Modularity: untangling the complexity of software programs

The second mechanism, modularity, is based on the idea of 'divide and conquer' (Dafermos 2001). Modular software is divided into smaller pieces, building blocks, which together create a software program. Consider the following statement from a Linux developer: "Of course, Linux software is very complex… The answer is to divide and conquer. When you have a complex piece of software, you cut it into ten pieces and if you manage to provide them with good interfaces then you only need to understand the separate pieces."

Thus, modularity depends on two aspects, namely, the modules and the interfaces that connect the modules. Essentially, the modules are the different parts of a software program, which perform separate tasks and which can act independently of one another. Software that is divided into modules has the big advantage that each module performs a limited set of activities or tasks, which "individuals can tackle independently from other tasks" (Lerner & Tirole 2002b, p. 28). An ASF board member explained how the Apache software came to be an entirely modular software program and the advantages of this modularity:

> In 1995 Robert Tow rewrote the entire NCSA server to make it entirely modular. The server became much easier to maintain. You could work on part of the server, without having to worry that you would damage the rest… [Developers] could work in parallel without stepping on each other's toes.

Thus, modularity is said to reduce the costs of coordination (Benkler 2002a, Bonaccorsi & Rossi 2003c, Dafermos 2001, Garzarelli 2003, Kogut & Metiu 2001, Lerner & Tirole 2002b, p. 28, Narduzzo & Rossi 2003).

For modularity to work, however, the interfaces between modules must be well defined and changes to the interfaces kept to a minimum. Modularity allows a developer to work independently "as long as she gets the communication interfaces right" (Weber 2004, p. 173). The editor in chief of the Linux journal explained, "Free software maintains complexity with such a loose structure because the interfaces are well defined." Hence, clear interfaces are important to ensure that modules can successfully be integrated into one software program. The need for clearly defined interfaces that remain relatively unchanged leads to at least one question: How are open source developers who base their decisions on their individual preferences capable of keeping the interfaces of software modules constant? Isn't it likely that a developer will want to make changes to a module that require a change in the interface?

This is not the case, according to the editor in chief of the Linux journal. He argues that free software developers tend to be "almost perverse" in their strive to implement development perfectly along the standard interfaces. Does this imply that developers build difficult and unnatural solutions to a problem? If so, do developers value modularity above elegance, simply because they would rather write a piece of code that is counterintuitive than change the interface? According to the president of Linux International this is also untrue. He said that modules can always be changed in ways that have no effect on either the elegance of the code in the module or the interface that connects the module to other modules. "If you have a glass of beer and drink out the rim, then I know how to model my mouth and lips. If the content under it changes but the rim stays the same I would still know how to hold my mouth. So the code or technique doesn't matter, as long as it fits in with the interface."

No matter how much space is available for developers to change the modules without having to change the interfaces, there are situations in which the interfaces must be changed.[18] In that case, the developers must somehow coordinate their changes to ensure that the modules remain compatible. Thus, modularity does relief some of the need for coordination, but it also raises a whole set of new questions.

Summarizing: modularity and elegance reduce some of the need for coordination

Irrespective of how changes in an interface are coordinated, it is by now clear that elegance and modularity do relieve some of the need to coordinate changes made to software. Furthermore, if the software is modular, developers need to invest less time to understand the consequences of a change they might implement.

Mechanisms to coordinate massive amount of individual efforts

Participants in open source communities claim that their efforts are uncoordinated. This is, however, not completely true. It is true that they spend little time coordinating their activities. For instance, they do not send e-mails explaining to other developers that they are currently developing software module X or translating software program Y. Generally, developers do, however, use a large and rather sophisticated infrastructure that supports their activities and, more importantly, coordinates their efforts with those being invested by hundreds if not thousands of other developers. As communities start to grow and attract more and more participants, they tend to support their activities with an increasingly sophisticated technical infrastructure. “These drastic changes were possible, mainly because of the technical conditions that have improved” (Bauer & Pizka 2003, p. 172). This infrastructure consists of mechanisms that computerize coordination. Next to the technical infrastructure, participants have adopted a number of devices that are related to a specific way of working. Many communities have further adopted methodologies and standards to coordinate their individual efforts.

The next pages identify a number of these mechanisms and explain for each mechanism (i) what it is, (ii) what it does, (iii) how it leads to the coordination of individual efforts and (iv) why it allows participants to spend as much of their time as possible on the actual development and maintenance of source code.

Furthermore, the description of the mechanisms demonstrates why fluid membership hardly poses a problem for the quality of software, which is a concern addressed by a number of researchers (e.g. Markus et al 2000). Indeed, fluid membership has limited potential to negatively affect the quality of open source software because many of the mechanisms described take away the reasons to know who the other developers are and whether they are able to write high-quality source code. Hence, they constitute part of the explanation of why boundaries are hardly needed to restrict access to the communities.

The Concurrent Versions System

Most open source communities support their development and maintenance activities with a concurrent versions system (CVS).[19] A CVS is a client-server repository. It is an automated system that allows remote access to source code and, according to the firm that develops and markets the CVS, it enables multiple developers to work on the same version of the source code simultaneously.[20] According to a number of interviewees, one of the advantages of using a CVS is that it becomes unimportant to know whether participating developers are able to write high-quality source code, since they cannot cause major damage to the source code. Older versions of the source code are automatically stored in the CVS and can be used as a backup. An ASF director explained, "It actually doesn't matter whether the committers are any good. In a CVS you cannot cause a lot of damage to the software. First, the older versions of the software are saved and you can return to these versions."

Basically anybody can download source code from the CVS. Some communities, however, have restricted access to their CVS. In these restricted communities, only participants with *committer status* can upload source code. Developers without committer status must send their patch to someone in the community who has that status. The way in which people gain committer status depends on the community. In most communities if you write and submit a few good patches of source code you will soon receive an invitation to become a committer: "To become a committer is really easy. You can become a committer with only one e-mail."[21]

An ASF board member explained how the CVS supports the software development process in the Apache community:

> In a CVS you start with a new version. Then something gets added and a new version is created. Then again something new is added. This way the line grows longer and longer. This line is called the "head": the most recent version. When someone disagrees with something, he can take an earlier version from the line and implement his changes to that version… For every new version you have to explain what you changed, why and in which way. You write this information in a log, which enables other developers to understand what you have done.

Much literature has addressed the importance of a CVS to support software development and maintenance (German 2002, Hemetsberger & Reinhardt 2004, Von Krogh et al 2003a, Scacchi 2004, Shaikh & Cornford 2003). Basically, a CVS supports the decentralized development process (Himanen 2001) in a number of ways. First, participants can access the CVS simultaneously. They do not have to wait until another developer has finished working on the source code. Second, the presence of logs is important. The log files provide participants with an explanation of how the source code works and what it intends to accomplish. The third reason is that with every new commit a new log is created, which is also sent to a mailing list. This way, other participants in the community are notified each time a new patch is added.[22] Finally, the CVS allows participants to move back in the development line and take an older version of the source code. This enables them to take out a commit that at a later stage of development proves to be bad code. This last option effectively reduces the need for participants in the community to monitor and analyze the value of every new commit. It also explains why most communities make it fairly easy to become a committer.

Mailing lists

The presence and use of mailing lists is another part of the infrastructure that supports collaboration (Bauer & Pizka 2003, Edwards 2001, Kogut & Metiu 2001). Every community has a number of mailing lists on which different issues are discussed. The Debian community, for instance, hosts 157 mailing lists.[23] These lists serve different purposes and target different audiences. Certain lists focus on users, providing them with a forum to ask questions and receive answers. Other lists host discussions about specific programs like KDE, the Linux kernel and Python. A respondent from the Debian community considered these mailing lists to be one of the most important tools for supporting the development of software in open source communities, because "people on these lists dare to say things and really want to hold their ground."

In short, the mailing lists provide the developers with a forum to exchange and discuss their ideas, and they also give users a forum to ask questions and receive answers.

Bug-tracking systems

Bugs are basically mistakes or flaws in a software program. Many software bugs are discovered while the software is actually in use. Users of open source software often write a 'bug report' when they come across a mistake or when they find that something does not work (e.g. Von Hippel 2001). The challenge of a bug report is to precisely describe the problem the user encountered. Such a report involves a description of, for instance, the kind of hardware being used, the other software installed on the computer, the particular configuration of the software and the error message encountered.[24]

To manage the processes of writing a bug report, solving a bug and communicating the solution to fix the bug, communities have created and adopted different systems. "Sometimes bug reports are handled through mailing lists. Other projects use bug-tracking systems."[25] Yet the underlying principles are basically the same.

More advanced bug-tracking systems provide a format in which a bug report should be written. Such a format includes a number of questions that need to be answered. The report of the bug is then stored in the system, where it awaits someone to fix the bug. Once the bug is fixed, the report is removed from the list. At this time the system will, and this is only true for the more advanced systems, automatically send an e-mail to the person who wrote the report to let that person know that the bug is solved.

Effectively, such systems eliminate the need for people in the communities, first, to contact others and explain about the bug they found and, second, to convince another developer to solve the bug. Instead, the bug-tracking system allows people anywhere in the world to report and describe a bug whenever they like. Also, anyone can access the repository of bug descriptions and can decide to fix a bug that has not yet been solved. Once the bug is solved the system ensures that an e-mail is sent to the person who filed the report. All these processes are automated and relieve the participants of the 'burden' of communicating and collaborating with others.

Manuals and coding style guides[26]

The Debian distribution is a collection of different software packages. This has at least one advantage, namely, that participants are able to work rather independently. The packages in the Debian community are maintained by so-called 'package maintainers.' They are responsible for ensuring that 'their' software program becomes part of the Debian distribution. The programs in the distribution must be packaged according to a certain standard that ensures that users can easily install the complete set of packages on their computer. To achieve this standardization, the participants have created and adopted a policy manual and a package manual. Two developers in the Debian community described the policy manual as a practical tool. "It tries to ensure that one program does not destroy others. It does not describe what to do, but how to do it." A maintainer in the Debian community continued, "The policy manual enforces that kind of integration in Debian. The manual simply says, okay this is the way in which we put packages together. These are very simple rules saying that we will obey the Linux file system hierarchy standard."

One thing the manuals prescribe is that each package maintainer must classify the software. There are three classifications: 'provides,' 'depends' and 'conflicts. The classification *provides* is used to explain what the software is supposed to do, for example, Netscape provides Internet browser functionality. The classification *depends* explains which other packages are needed to make the software work. The classification *conflicts* indicates that the software will not work properly if a certain package is already installed on the computer. According to one respondent from the Debian community, "In this system you assign relationships and that is what makes the system as a whole work." The policy and package manual "ensure that the organization as a whole is very flexible."[27]

Communities like Apache and Linux also have manuals, but they are called *coding style guides*.[28] The coding style guides are different from the Debian community's manuals, but they aim to achieve a similar goal. The guides prescribe how a piece of source code should be laid out and the way the source code should be styled. Each community makes its own choices about the style and layout of the source code. Using different styles in one software program makes the program difficult to understand. A developer in the Linux community explained how one participant "changed the entire source code to the GNU style. Although, this is just a minor change, it does make it almost impossible to compare the programs, because the indentation is completely different."

Thus, the coding styles, like the policy and package manuals, aim to achieve unified definitions and a single style of writing software among all developers in a community. This uniformity or standardization reduces the time needed to understand source code written by other participants and diminishes the need to communicate about a piece of software. It thus increases independency in the communities.

To-do lists

Participants typically have many ideas about how a software program should work or what new features should be added to a program. The only way these ideas are transformed into actual lines of source code is by someone writing the source code. The problem is that not everyone is able to do so, for instance, because they lack the skills and time or because they are simply not motivated to do the job. Yet the ideas might nonetheless be valuable and very much needed by a large part of the community.

To ensure that new ideas are actually transformed into source code, participants have created to-do lists. As the name implies, a to-do list is an inventory of things that at least one person considers important or wants to have. One community that has adopted a to-do list is the PostgreSQL community. Typically, an idea is transferred to this list when someone sends "an e-mail with a suggestion."[29]

The to-do list is a coordinative mechanism because it signals developers as to what others in the community find important. Participants do not have to discuss and explain why they find the ideas important. Instead, they just put the item on the to-do list. Others can take a look at the list and judge for themselves what they find interesting and what they would like to work on. As such, the to-do list serves as a marketplace in which demand, for example, for a certain feature, meets supply, namely participants who have the knowledge, time and motivation to develop the feature. To-do lists are another example of a mechanism created and adopted with the goal of structuring the efforts of individuals in the communities.

Orphanage: packages in need of a new maintainer

In many communities, especially ones without a CVS, one participant is responsible for the maintenance of a module or even an entire software program. In the first case the participant is referred to as the 'maintainer,' for instance, Debian has its package maintainers. Participants who are responsible for an entire software program or for a community are usually called 'project leaders.' Examples of project leaders are Linus Torvalds in the Linux kernel community, Olivier Sessink in BlueFish and Bruce Perens in Debian. Typically, the maintainers and project leaders collect the contributions and incorporate them in the latest version of the software. In this process, they are the one who decides whether to include a contribution. Linus Torvalds has described his role in the Linux kernel community on numerous occasions. Once he wrote, "Think of me as CVS with a brain and with some taste. Nothing more, nothing less."[30]

A problem that could occur is that a project leader or maintainer loses interest in maintaining the software and decides to quit. What happens then? This is especially relevant in a situation in which a project leader or maintainer does not communicate the decision to quit to the participants in a community. Participants will then continue to send their patches of source code as if nothing had changed. One respondent explained what would typically happen: "Usually we take a week. In general this is enough time for him to respond. Look, a couple of days [of no response] that happens." After this period it is assumed that the maintainer or project leader has stopped his activities and is said to be *missing in action*.[31]

Even without a maintainer, participants are almost always able to continue development and maintenance of the source code and, maybe more importantly, they seem to do so without much disruption. The availability of the source code on the Internet is one of the reasons why participants face relatively few problems when a maintainer or project leader quits.[32] The availability of the source code enables participants to continue their work. The only problem is to decide how to (re)arrange the process to ensure that the contributions find their way into the latest version of the software.

To support this transition period, the Debian community has created a special website called "packages in need of a new maintainer."[33] This website distinguishes two categories. The first category lists packages for which the maintainer has announced a desire to no longer maintain the software. The package is now referred to as an orphan[34] and is "up for adoption."[35] The second category lists packages that no longer have a maintainer. This category is called "orphaned packages."[36]

For every package, irrespective of whether it is orphaned or put up for adoption, Debian's website lists two things. The first is a link to the official message in which either the maintainer indicates the desire to cease activities or in which someone else reports that the maintainer is missing in action.[37] The second is a link to the latest stable, unstable and test release of that package. Both the website and the two links make it easy for anyone to become the new maintainer of a software package in the Debian community. By performing a number of fairly simple tasks the orphan is taken off the website and the new maintainer can continue the development and maintenance of the source code.

What the orphanage effectively does is to ensure that participants in a community like Debian have a quick reference and explanation as to why their patches are not included in the source code. Furthermore, it eases the process of becoming a new maintainer of an existing development project. In other words, the website simplifies the process of stopping or starting maintenance of a project.

Added text: an important signaling function

Many participants in open source communities try to write source code that is elegant. Sometimes, however, they cannot. Take for instance software in which a security hole is discovered. In such a situation the first and foremost goal is to fix the hole to ensure that the software is secure again. Whether the code of the solution is elegant is less important.[38] Another example in which elegance is less important is where developers write software that is compatible with proprietary software. Typically, the participants have no access to the actual source code of that software and therefore they have to guess how the source code looks based on the way the software functions. To write software that is compatible with proprietary software is very difficult and frequently results in inelegant code. "Sometimes code has to look inelegant to work around something. Most of the time we see curse words accompanying these posting, explaining why it is not elegant. It is inelegant because it has to deal with other [proprietary] software."[39]

Thus, in some situations it is almost impossible for developers to write elegant code. Yet when other developers read the inelegant code, they have a hard time understanding how the software works and why that particular piece of source code is needed. They may even be tempted to remove the code or think that the author is a poor programmer. Why else would someone have written such inelegant code?

Interviewees indicated two general ways for an author to communicate the presence of inelegant code and the fact that circumstances forced them to write the inelegant code. One way is the use of curse words written in the sidelines of the source code. These curse words signal three things to other participants who read the source code. First, the curse words communicate that the author of the code knows that the code is inelegant as written, but that no other way has as yet been found to work around a certain problem. Second, the curse words explain what the source code accomplishes. Hence, the curse words try to reduce the time that a participant needs to understand the inelegant source code and they are a way to make it as easy as possible for others to understand and possibly modify the code. Third, the number of curse words serves as an indication of the elegance of the code and thus provides a warning to others that the source code is considered to be inelegant and thus difficult to understand. The editor in chief of the Linux journal remarked, "In some architectures you see more curse words than in others. If you take a piece of code for the IBM mainframe, there you would see no curse words. In code for the Spark on the other hand you see a lot of curse words… because the IBM mainframe is elegant."

A second way to indicate the presence of inelegant code is by placing the inelegant code between brackets and surrounding the code with #ifdef. "This makes it easier to delete and replace the code with an elegant solution in the future."[40] Adding this text to the code performs a similar function to the curse words added in the sidelines of source code.

Adding text, like curse words or #ifdef, not only communicates the inability of the developer to write an elegant solution for a certain problem. It also has a clear function in neutralizing some of the negative effects of inelegant code, namely helping other developers to understand the code and modify it.

Names attached to improvements

In every open source community one or more mechanisms are adopted to relate participants to their contributions. One such mechanism is the credits list, which contains the names of developers who have contributed to the development of software. The credits list also specifies what they have contributed. "My name is attached to every KDE program."[41] In most programs the name of the person that fixed a part of the software is put "next to the resolved item."[42] The Apache community "makes a point of recognizing all contributors on its website even those who simply identify a problem without proposing a solution" (Lerner & Tirole 2002b, p. 27). For many participants, having their names attached to the contributions is an important acknowledgement of their work and part of the reason why they contribute their time and effort. "I am actually quite proud to see my name on a piece of code."[43]

Another way in which a contributor is related to a certain piece of source code is the voting system that is adopted in the Apache community. For every new commit[44] a vote is held. "Don't expect too much from [the vote], if no one is against the patch then the patch will remain in. If you explicitly voted that you wanted the patch to remain then you commit yourself to help clean up the software if the patch turns out to be less good."

Connecting participants to their contributions also fulfills a coordinative function. The contributor of the source code is known and thus feels responsible. "If something turns out to be wrong it is my mistake. Usually the one who made the last change is the most appropriate person"[45] to fix the problem. If a user of the source code discovers that it does not work, for whatever reason, then the user knows whom to e-mail. Chances are, the contributor will feel responsible, if only because the contributor's name is attached to the source code that does not work. Typically, the contributor fixes the problem. "Iterations work like this; someone will comment on a patch and the author will make the changes and re-submit the patch. It rarely happens that someone ignores the comments. If this happens and it is valuable to me I would go in and do it myself, or I would tell the author that we are waiting."[46]

Small and incremental patches

Another coordinative mechanism is the norm that developers should only contribute source code in small and incremental patches. Keeping the patches small is important, as it makes it easier for others to understand what the source code aims to achieve and how it intends to do so. "This system has too many improvements. I have to be able to understand it. Please, only send small patches."[47] An example is an incident in the Apache community in which the developers rejected a patch from a company because "it was simply too big. The developers [in the Apache community] could not unwrap it to understand how it worked."[48]

Thus, small and incremental patches make it easier for others to understand what the source code aims to achieve. This is because (i) they lower the amount of time participants need to invest to understand what the source code intends to do; (ii) they make it easier for other developers to make modifications to the patch; and (iii) they lower the value of resources that are destroyed when a new addition is not accepted or is removed. This last point can be explained as follows. A large patch of source code can be viewed as a collection of small patches. Consider a participant who combines the small patches into one large patch and adds this source code into the code base. One small mistake in the collection of patches could compromise the integrity of the entire program. In this case, participants might have to remove the entire collection of patches. If the patches had been contributed in small pieces, the participants could remove just the small patch of source code that created the problem. In that case the destruction of value, that is, the time and effort spent to create the rejected patch, would have been relatively small, as it involved just one change. In general, the bigger the patch, the more time and effort is involved in developing and maintaining the software.

For these reasons, Linus Torvalds is unlikely to add large patches. "I will repeat my rule: I do not apply large patches with many separate changes. I apply patches that do one thing."[49]

Conclusion

This chapter began by observing that open source communities do not need additional appropriation rules. First, the rules are laid down in the boundaries of the communities and, second, most types of usage do not affect the stock available to others and are therefore not in need of restriction. Furthermore, the chapter showed that provision in the communities is based on individual choice and action. It identified a number of mechanisms that support and enable activities in the communities. Table 5.1 lists these mechanisms.

The paradox: voluntary standardization by professionals

The effectiveness of mechanisms like elegance, modularity, use of a CVS, a credits file, a bug-tracking system, manuals and an orphanage depend on their rate of adoption. A minimal number of participants should adopt and use the mechanisms. If most contributors decide to ignore the mechanisms, they will likely become worthless. What, for example, would be the use of having a coding style guide if no one used it? What would be the value of having a log file in which only a small percentage of contributors documented what they had added to the source code? If this were the case, the mechanisms would be less and less valuable and would soon become obsolete. In other words, the mechanisms in the communities are effective as long as many contributors use them.

Table 5.1 – Mechanisms that structure software development and maintenance and allow individual choice

Mechanisms to relieve the need for coordination	**Description**
Elegance	Writing source code that is obvious and intuitive, which makes it easy to understand
Modularity	Dividing the software in small and independent parts, which increases independence
Coordination mechanisms	**Description**
Concurrent versions system (CVS)	Database system allowing distributed software development
Mailing lists	Impersonal and one-to-many communication channel
Manuals	Documents that describe how certain source code should be written
Bug-tracking system	Process to post bugs on a list to be addressed and solved by anyone
Coding style guides	Description of the standard layout of the source code
To-do lists	Inventory of activities that are wanted and are open for work
Orphanage	A place to 'park' software development projects that lack a maintainer
Added text	Used to explain and signal problems experienced when writing the source code
Names attached to improvements	Means to enable people to contact the author; also creates a level of responsibility and recognition
Small and incremental patches	Norm that changes be kept small in size so they remain easy to understand and improve on

The mechanisms do succeed in supporting many activities because they are used and because they lead to a certain level of standardization in the communities. For instance, a style guide encourages all developers to use the same indentation and thus makes the source code easier to understand. Similarly, a CVS forces every download and upload to be done in a similar fashion. Standardization, however, also restricts contributors' autonomy. For instance, the fact that source code must be elegant generally precludes contributors from making a quick, short-term and less elegant fix to a problem. Or consider the coding style guides; they restrict participants' freedom because contributors must write source code in a style that is documented in the coding style guide.

Most of the contributors are typically *professionals*. They have access to a rich skill set and possess a high level of intrinsic knowledge (De Bruijn 2002). To deliver a good job, professionals need and strive for a high level of autonomy and flexibility (De Bruijn 2002).

Therefore, they generally resist all attempts to in some way restrict their autonomy and flexibility. Thus, the voluntary adoption and use of mechanisms is counterintuitive, because it restricts their autonomy and flexibility. The *paradox* is that this is exactly what professionals in open source communities have done; they voluntarily restrict their autonomy and flexibility. The question is 'Why?'

Three individual behavioral rules

Software development and maintenance in the communities are governed by individual choices. This does not mean that individuals are always efficient in their actions and that they always contribute to the creation of software. On the contrary, many of the actions of individuals do not result in better software. Most open source communities consist of frantic messes of people who act in seemingly uncoordinated and random ways. People act according to their own preferences and based on local stimuli. It is this behavior that resulted in the metaphor of open source communities as bazaars (Raymond 1999b). The fact that people act according to their own preferences contributes to a high level of redundancy and overlap. It is, for instance, not uncommon for people to simultaneously and independently create multiple solutions to one problem (e.g. Bekkers 2000). Open source communities are thus confronted not only with redundancy, but also with conflicts over whose alternative is the better one. The next chapter discusses how the communities deal with conflict.

Yet despite the high amount of redundancy, the software created does frequently work. The reason why the software works and why participants in the community voluntarily create and adopt the mechanisms identified in this chapter can be understood with three individual behavioral rules. These three rules lie at the base of the choices the participants make in the communities.

"*Participants want to increase the chance others will accept and adopt their contributions*." Participants who expend time and effort developing software want to increase the chance of their contributions being accepted and adopted by others. There are many reasons for this desire, for instance, to improve their reputation (Lerner & Tirole 2002b, McGowan 2001). The reason could also be more pragmatic. Participants whose contributions are not accepted will have to maintain and update their contribution themselves with every new official release. But if the contribution is accepted and adopted, others will take on some of these activities.

"*Participants spend limited time searching for software and analyzing contributions from others*." Like one of the rules put forward in the previous chapter, this rule asserts that participants want to spend their time as efficiently as possible. As a result, they want to minimize the time they spend analyzing and searching for other people's contributions.

"*Participants replace a contribution, if another contribution is easier to understand*." Related to the previous rule, other things being equal, participants will replace existing contributions when there is an alternative that is easier to understand.

How the individual behavioral rules support the observations

The first rule describes the importance for participants in the communities that their contributions be adopted. A participant invests time and effort to, for instance, create a new piece of source code. The contributor receives benefits like recognition and reputation once the contribution is accepted. However, at the same time the contributor knows that other participants – like himself or herself – want to reduce the time they spend searching for contributions or trying to decipher a contribution. To substantially increase the chance of adoption, participants thus have to make their contributions easily accessible and understandable. To write source code according to the mechanisms identified in this chapter, that is, to write elegant code, to write modular code and to write source code according to a coding style guide, does involve additional investment. But the chance of receiving greater benefits is also high. In other words, to create and adopt the mechanisms maximizes the chance of reaping greater benefits at relatively low cost.

The third rule is very much in line with the second rule. Its added aspect is that existing source code can be removed and replaced with source code that is easier to understand, in other words, source code that is more elegant or modular. This rule explains why the bigger and more popular open source communities typically have a great deal of source code that is elegant and modular, whereas many of the smaller communities have programs that consist of less modular and elegant source code.[50]

Notes on chapter five

[1] The word 'traditional' is used here to refer to the communities that are described in most research on community-governed common pool resources (e.g. Ostrom E. 1990. *Governing the Commons; The Evolution of Institutions for Collective Action*. Cambridge: Cambridge University Press)
[2] In some communities, e.g. Linux, development starts this way. In other communities the creator could decide to upload the software in a support tool.
[3] From an interview with the author and maintainer of BlueFish.
[4] This research does include the idea that certain activities, structures or mechanisms can result in less participation, as they could, for instance, lower the expected motivation of potential participants.
[5] From an interview with a maintainer in the Linux community.
[6] From an interview with a maintainer in the Linux community.
[7] From an interview with the project leader of the PostgreSQL community.
[8] Cited from an interview with a contributor in the KDE community.
[9] From an interview, which was held with one of the two maintainers of Lilypond.
[10] From an interview with one of the two project leaders of Lilypond.
[11] Cited from the jargon file: <http://www.cnam.fr/Jargon/jargon.html?559> (August 2002).
[12] From an Internet magazine called *Salon.com*, written by Mark Wallace in 1999: <http://www.salon.com/tech/feature/1999/09/16/knuth/index1.html> (August 2002).
[13] Cited from an interview with Ellen Ullman by Scott Rosenberg in 1997. The interview is published on *Salon.com*: <http://archive.salon.com/21st/feature/1997/10/09interview.html> (August 2002).
[14] Cited from an interview with a maintainer in the Linux kernel community.
[15] Cited from an interview with the editor in chief of Linux journal.
[16] Cited from an interview with the editor in chief of Linux journal.
[17] Cited from an interview with a maintainer in the Linux kernel community.
[18] Respondents provided two examples in which the interfaces had to be changed. These examples are discussed in some detail in the next chapter, which focuses on collective choice.
[19] Actually one of the communities that does not use a CVS is the Linux kernel. Allegedly Linus Torvalds does not like the tool. Until recently he rejected every kind of support tool, but in 2004 he started testing another tool called Bitkeeper (based on the Linux kernel developer mailing list).
[20] Based on information from the CVS home page: http://www.cvshome.org/ (August 2002).
[21] Cited from an interview with an ASF board member.
[22] Based on an interview with one of the board members of the PostgreSQL community.
[23] This is the total number of mailing lists listed on the Debian website (http://lists.debian.org/completeindex.html) on April 10th, 2003.
[24] This information is based on multiple interviews and observation of a number of, particularly user-oriented, mailing lists.

[25] Based on an interview with the president of Linux International.
[26] This section is partly based on a previous publication: Egyedi TM, Van Wendel de Joode R. 2003. *Standards and coordination in open source software.* Presented at 3rd IEEE Conference on Standardization and Innovation in Information Technology, Delft, The Netherlands
[27] Cited from an interview with a package maintainer in the Debian community.
[28] See: www.apache.org/dev/styleguide.html (May 2003) and www.linuxjournal.com/article.php?sid=5780 (May 2003).
[29] Cited from an interview with a member of the steering committee in the PostgreSQL community.
[30] Cited from the website: http://www.kerneltraffic.org/kernel-traffic/kt19991101_41.html (September 2004)
[31] Missing in action refers to maintainers who have stopped their activities without communicating this to other developers and users in the community (based on an interview with two package maintainers in the Debian community).
[32] Based on an interview with a package maintainer in the Debian community.
[33] The website can be found on page: http://www.debian.org/devel/wnpp/work_needing (April 8th, 2003)
[34] Cited from a developer in the Linux community
[35] The website literally states "packages up for adoption", apparently to indicate that there is a similarity with a parent who puts his child up for adoption.
[36] Cited from the website: http://www.debian.org/devel/wnpp/orphaned (September 2004)
[37] A description of the way in which a maintainer can be reported as "missing in action" see the discussion on: http://lists.debian.org/debian-qa/2002/02/msg00111.html (September 2004).
[38] Based on an interview with a member of the ASF.
[39] Cited from an interview with the Editor in Chief of Linux Journal.
[40] Cited from an interview with the maintainer of BlueFish.
[41] Cited from an interview with the head of Dutch translation in KDE.
[42] Cited from an interview with a member of steering committee of the PostgreSQL community.
[43] Cited from an interview with one of the translators in the BlueFish community.
[44] A commit in the Apache community is the act of adding source code to the CVS.
[45] Cited from an interview with a developer in the Linux kernel community.
[46] Cited from an interview with the maintainer of the PostgreSQL community.
[47] Cited a from an interview with a developer in the Linux kernel community.
[48] Cited from an interview with a fellow PhD researcher on open source software development.
[49] Cited from the Internet: http://kt.zork.net/kernel-traffic/kt19991101_41.html (September 2003) The role of Linus Torvalds in the community and his ability to "enforce" a rule on other contributors in the Linux community will be addressed in more detail in a next chapter.
[50] This observation was made in a number of interviews with maintainers of software packages that are less popular and attract fewer developers compared to communities like Linux and Apache. In particular the maintainers of BlueFish and of Lilypond were interviewed on this aspect. They claimed they would replace the software with source code that is easier to understand, but they have few contributors who can help them in this process. The Apache community has also seen a transition of the software. The structure of the source code was completely rewritten and made more modular in the beginning of the project.

CHAPTER SIX

CONFLICT

This chapter addresses the third design principle, which is the presence of *conflict resolution mechanisms*. This design principle is relevant in open source communities. One of the reasons is rather extensively addressed in this chapter, namely, the fact that open source communities have a relatively high potential for conflict. The high potential for conflict can be recognized and understood when examining the backgrounds of participants in the communities. Many open source communities connect thousands of individuals who have hundreds of different nationalities, who have different types of education and different motives to participate. All these individuals are interdependent, as they need each other to create and maintain open source software. However, the fact that most open source software is modular reduces the interdependencies between participants and thus dissipates some of the potential for conflict. Still, the differences and the remaining interdependencies are likely to result in quite a number of conflicts.

Theoretically, conflicts can give rise to overt hostility and inertia, and they can threaten the continuity of communities. The fact that open source communities have a great potential for conflict implies that they need some way to manage conflict and to prevent negative outcomes. But there is another reason why the management of conflict is important. Research demonstrates that the ability of a team of programmers to deal with conflicts is a more important predictor of the performance of that team than the skills and abilities of the individual software programmers in that team (Elliot & Scacchi 2002).

Two conflict resolution mechanisms that are discussed in the literature are mediation and arbitration. Open source communities consist of actors who could potentially fulfill the role of mediator or arbitrator. However, the ability of these actors to actually influence conflicts, let alone solve them, is relatively limited. This is because many participants appear to ignore decisions made by an arbitrator. Instead, they prefer to do what they think is best. Many respondents argued that they dislike discussions and conflict; they prefer to actually *do*. This observation is referred to as a culture of 'doing.'

This chapter argues that the culture of 'doing' is a key aspect of open source communities. The culture of doing, combined with mechanisms that allow anyone to create multiple development lines, minimizes the potential negative consequences of conflicts in open source communities.

The third design principle: conflict resolution mechanisms

Literature on conflict is characterized by a great variety of definitions of what a conflict is and when we can say that a conflict is actually present. According to Pruitt (1998) the definitions of conflict found in literature can be divided into two categories. The first is "conflict behaviour – opposing action taken by two or more parties" (p. 470) and the second is "one or another source of conflict behaviour, most commonly divergence of interest and annoyance that is attributed to another party" (p. 470).

The definition adopted in this chapter is that given in Jehn (1995). She writes, "Conflict has been broadly defined as perceived incompatibilities (Boulding, 1963) or perceptions by the parties involved that they hold discrepant views or have interpersonal incompatibilities" (Jehn 1995, p. 257). Thus, this chapter speaks of a conflict when parties have conflicting ideas or personal characteristics that collide. This definition focuses on the sources of conflict, which falls in the second category identified by Pruitt.

Most literature on conflict distinguishes types of conflict. One type is a *task conflict*. A task conflict is "an awareness of differences in viewpoints and opinions pertaining to a group task" (Jehn 1995, p. 238). Much research has been performed to understand the relationship between task conflicts and team performance. Somewhat paradoxically, most research has identified a positive relationship between task conflicts and team performance (e.g. Amason 1996). Empirical research has shown that teams with conflicts about how a specific task should be performed have a better general understanding of the task at hand. This contributes positively to the performance of that task (Jehn 1995).

The second type of conflict is *affective conflict*. Such conflict is emotional and focuses on interpersonal incompatibilities. It includes "affective components such as feeling tension or friction" (Jehn & Mannix 2001, p. 238). Empirical research demonstrates that affective conflicts are dysfunctional and are a threat to group performance (Amason 1996, Jehn 1995).

Sources of conflicts

It is generally acknowledged that conflicts have two important sources, the first being *diversity* (Gefu & Kolawole 2002, p. 9, Smith 1999). Studies show that differentiation and specialization in organizations increase the level of diversity among members and thus give rise to more conflicts (Dipboye et al 1994, Morgan 1986).

Diversity that leads to conflict can be present on different levels: the level of opinions, of values or of interests. This means that more diversity in opinions, interests and values is likely to lead to more conflicts. Deutsch explained that conflicts stemming from a difference in values "concern what 'should be…' It is not the differences in values per se that lead to conflict but rather the claim that one value should dominate or be applied generally, even by those who hold different values" (Deutsch 1973, p. 15). A difference of opinion or belief can lead to a conflict about what *is*. It is a conflict about the interpretation of facts, information, knowledge or reality (Deutsch 1973). A conflict of interests is about achieving different, conflicting, goals. In this situation the parties cannot achieve their goals simultaneously, which is likely to lead to a conflict.

A second important source of conflict is *interdependency* (e.g. Dipboye et al 1994). The more people depend on each other to achieve a certain goal or perform a certain task, the more likely that conflicts will occur.

Advantages and disadvantages of conflicts

As early as 1976 Thomas argued that conflict can have positive effects. "More and more, social scientists are coming to realize – and to demonstrate – that conflict itself is no evil, but rather a phenomenon which can have constructive or destructive effects" (Thomas 1976, p. 889). Currently, the advantages of conflicts are primarily associated with task conflicts (e.g. Amason 1996, Jehn 1995). Research shows that conflicts can have at least four advantages. First is a *higher level of productivity*. According to Likert and Likert (1976), groups that regularly have conflicts achieve a higher level of productivity than groups that do not have conflict. Second is to *prevent groupthink*. Task conflicts stimulate group members to thoughtfully consider criticism and alternatives (Jehn 1995). Third is increased *creativity*. Task conflicts stimulate interest and curiosity and are therefore likely to lead to more creativity (Deutsch 1973). Fourth is *increased vitality*. According to Rosenthal (Rosenthal 1988), stimulating conflict and competition in an organization creates checks and balances and introduces countervailing forces. In this view, to have conflicts means that no party can easily enforce their interests, opinions and values on others, which increases the stability of the collective. In other words, the presence of conflict – and this too seems paradoxical – leads to more stability and balance in the collective (see also Pondy 1967).

Next to advantages, conflicts can have disadvantages. Research demonstrates that affective conflicts can be particularly dysfunctional. First, conflict can become *destructive* and produce "strong negative feelings, blindness to interdependencies, and uncontrolled escalation of aggressive action and counteraction" (Brown 1984, p. 378). Conflict can thus lead to overt hostilities, which is obviously unbeneficial to organizations. Second, conflict can lead to *inactivity*. Everybody actively pursuing their own interests and protecting their own values obstructs the decision-making and implementation process (Jehn & Mannix 2001). This situation can best be understood as a 'trench war' (Rosenthal 1988) in which no one is able to progress and the collective as a whole is at a stalemate.

Management of conflict

Empirical research shows that the performance of a team highly depends on the way it is able to manage its conflicts (e.g. Lovelace et al 2001). Montaya-Weiss et al. (2001) argue, "[W]e focused on conflict management because it is a fundamental issue for effective virtual teams' performance, given the inherent communication and coordination challenges they face" (p. 1252). The management of conflicts is no simple activity, however, for a number of reasons. First, task conflicts and affective conflicts are related (Amason 1996). Under certain conditions, group members are reported to perceive comments related to tasks as personal criticism, which gives rise to affective conflicts (Mannix et al 2002). If members continue to perceive task conflicts as affective conflicts then "the result may be a steady rise in both task and relationship conflict and a performance loss rather than gain" (Jehn & Mannix 2001).

The second reason is that there is an optimal level of task conflict, below or beyond which it ceases to be beneficial to group performance (Pondy 1967). Similarly, Brown (1984) states that too many task conflicts can result in poor group performance. Neither are task conflicts always beneficial to group performance. Jehn (1995) argues that groups in which routine tasks are performed do not benefit from task conflicts; on the contrary, task conflicts are likely to reduce their performance.

Thus, conflict management is needed (i) to prevent task conflicts in groups from giving rise to affective conflicts and (ii) to prevent the level of task conflict from falling below or rising above the optimal level to successfully complete a certain task.

The role of conflict in research on community-governed common pool resources

According to Ostrom (1990) the rules that govern people's behavior vis-à-vis common pool resources are often not as straightforward as one might like them to be. This results in ambiguity, which in turn can give rise to conflicts about whether someone's actions should be considered an infraction of a rule. To ensure a sustainable common pool resource, Ostrom argues that people should have rapid access to low-cost arenas to resolve these conflicts. She calls these arenas 'conflict resolution mechanisms.' She explains, "Such mechanisms sometimes are quite informal, and those who are selected as leaders are also the basic resolvers of conflict. In some cases – such as the Spanish *huertas* – the potential for conflict over a very scarce resource is so high that well-developed court mechanisms have been in place for centuries" (Ostrom 1990, p. 101).

Three aspects attract attention when analyzing the work of Ostrom and colleagues on the subject of conflict. First is the implicit negative connotation of conflict. The fact that conflicts must be resolved implies that Ostrom focuses on its negative aspects. Conflict is believed to give rise to major upsets in social groups and, as such, must be resolved. Other researchers have taken a similar stance. Compare, for instance, a statement by Likert and Likert: "[F]or many if not all conflicts, the need is to find a way to resolve them" (1976, p. 5). Second is the idea that people must be able to vent their opposing ideas and interests (Ostrom 1990). Therefore, the argument is made that communities must have some arena where people can actually make their point and deal with the conflicts that arise. Third, there is an implicit focus on conflicts concerning the content and explanation of rules, especially those related to boundaries, appropriation and provision. Ostrom (1990) writes, "If individuals are going to follow rules over a longer period of time, there must be some mechanism for discussing and resolving what constitutes an infraction" (p. 100). Conflict resolution mechanisms must be in place to deal with such conflicts.

Other research on common pool resources has shown that conflicts can also arise on aspects other than the operational rules (Bennett 2000) of a common pool resource and that resolution mechanisms are needed to solve these conflicts as well. Gefu and Kolawole (2002), for instance, describe how diverse types of land use are destined to lead to conflicts that somehow must be resolved.

The remainder of this chapter is not limited to conflict regarding rules and the infraction thereof, but extends to analyze the conflict that surrounds the development and maintenance of a resource; that is, the source code. In some cases this involves conflict about procedures and rules that govern how and who should contribute source code; in other cases it involves a different kind of conflict.

A high potential for conflicts

Open source communities face many sources of conflict. For example, how should a certain piece of source code be developed? Should a piece of source code be included in the latest version of the program? Is a piece of source code elegant? How should a software program be divided into modules? Which open source license is best? These types of conflicts are very different in nature and address a wide variety of topics. One type of conflict that is particularly interesting is that surrounding decisions on whether a piece of source code should be included in a software program. Consider this statement:

> Developers occasionally get into conflicts where they overwrite changes that they disagree with, only to find that they are over-written again by developers advocating competing approaches. These 'commit wars' are… an expected part of a project where "you have 300 people changing files all over, millions of files" (McCormick 2003, p. 19).

This section argues that the two sources of conflict described in the literature are also present in open source communities. Recall these are high levels of diversity and interdependency.

A high level of diversity

A frequent observation made about open source communities is that there is a high level of diversity among the contributors. One source of this diversity is that the communities are truly global. "The community of developers that contributes to Linux is geographically far flung, extremely large and notably international" (Weber 2004, p. 66). With such a global distribution, the communities harbor people from totally different cultures, with different values and with different beliefs. These differences are a source of conflict.

Contributors also differ in their professional backgrounds. "[T]here are many different types of contributors to open-source projects: organizations and individual initiators and those who help with other people's projects; hobbyists and professionals" (Markus et al 2000, p. 15). The creator and maintainer of the Python programming language explained how diversity is present in the Python community: "There are more than half a million users. Everybody wants different things; there are programmers who want the language to remain easy, some use it for Web server development, others for numerical calculations for simulations, they all use Python scripts and they all have different requirements for what the language is supposed to do." These differences in interests are likely to lead to conflict about how the software should be developed and maintained.

A growing number of companies is getting involved in the communities. Companies typically hire people to participate in the communities, which gives rise to even higher levels of diversity and hence more frustration and conflict. Two ASF members explained, "One of the problems is that individuals who develop on Apache and who come from Sun have deadlines. This creates frustration, when they need to work together with people who do it for fun." People hired by Sun Microsystems to participate in open source software development thus have interests that differ from those of contributors who take part for fun. The contributor from Sun wants or needs to meet the deadlines imposed by the employer, whereas the independent contributor is more likely to explore new and innovative solutions to existing problems. One of the creators of the Apache Web server and member of the ASF Board of Directors explained the difference between a developer from, for example, IBM and a private developer who works at it for a hobby: "The first will be working more structurally towards deadlines and with a certain approach, whereas the latter is more creative and more risky in trying things and generally has a higher payoff-risk ratio." Again, this difference is likely to result in conflict.

Some of the differences are structural ones, and can be compared to the differences between the departments of an organization, as identified by Morgan (1986, Chapter 6). He explained that the nature of any job in an organization includes elements that, by their nature, contradict each other and inevitably result in conflict. This sets the actors in an organization on a "collision course" (p. 157). To a degree the different types of contributors in open source communities are also set on such a collision course in which conflicts are almost inevitable. Contributors paid to develop software and contributors who develop software for their own personal reasons are destined to frustrate each other's interests and are likely to run into conflicts (see also Van Wendel de Joode 2004b).

Interdependency in open source communities

A second source of conflict is interdependency, which is always present in open source communities. The project leader of Python highlights the existence of mutual dependencies when he states, "The last couple of years it has become increasingly difficult to get agreement in the community as to how things should be done." This statement refers to the presence of mutual dependencies; why else would it be important to get an agreement?

The reason why developers are mutually dependent is that no single developer can ever alone write complex software like an operating system or a Web server that includes millions of lines of code. The development of this kind of software requires the input and expertise of many different professionals. Professionals have their own expertise, yet they are also required to work together. They are dependent on the input of others (e.g. De Bruijn & Ten Heuvelhof 2000, p. 12). But it is not only individual participants in the communities who are dependent on each other; companies are also dependent on the input of others. Take, for instance, the previous example of Sun developers who need to collaborate with volunteers. Apparently, developers from Sun, which is truly not a small enterprise, cannot develop all software on their own. They must depend on the input of others.

Taking away some of the interdependencies: modularity

The previous chapter introduced the concept of modularity. The basic rational that underlies modularity is to divide a complex product like software into separate and relatively independent modules. For two reasons development of modular software is likely to result in fewer conflicts than development of software that is not modular.

First, modularity increases *independence* of participants and hence dispels a potential source of conflict. Participants work on specific and relatively isolated parts of the software. Without modularity, changes made in one part of the software would likely have the effect that "something else does not work anymore."[1] Without modularity, participants would have to make many decisions collectively and would be likely to frustrate each other's development efforts. Modularity decreases the number and complexity of interdependencies and hence reduces the chance that conflicts will occur.

Second, modular software enables the *localization and isolation of conflicts*. Consider any random conflict between two participants about software that is not modular. Even if the conflict was about a small part of the software, it would likely affect the entire software program and hence the entire community. The modularity of software enables participants to break most conflicts down and attribute them to a small part of the software, that is, to one module. Most conflicts can thus be localized to a specific module and discussed in isolation from others in the community.

Some examples of modularity

Most of the communities analyzed in this research have created software that is highly modular. Consider the Debian distribution. All packages in the distribution are basically different programs that are connected through a limited number of interfaces. These interfaces are laid down in the manuals, which prescribe how participants are supposed to define relationships between the packages. The participants in the community operate fairly independently of one another and they focus on the packages of which they are the maintainer.

Another example is the Linux community. The Linux kernel has many parts that are modular. Consider the Journal File System in the Linux kernel. Currently there are at least five different file systems that can be used in the Linux kernel. They are separate modules and every user of the Linux kernel can decide which system they would like to install.[2] Another example is the way in which the drivers are organized. Drivers basically create the interface between hardware and the Linux kernel. In the Linux kernel, for every different type of hardware different drivers are created. These drivers are modular and they can be created and maintained in isolation of other drivers.

The limited role of mediation and arbitration

Although the presence of modularity reduces some of the potential for conflicts, conflicts do arise regularly in open source communities (e.g. Elliot & Scacchi 2002). To prevent conflicts from endangering the continuity of the communities, they must somehow be dealt with. Raiffa (1992) lists a number of ways in which actors can influence and mitigate conflicts. He argues that they can (i) bring parties together, (ii) establish a constructive ambience, (iii) collect and wisely communicate confidential information, (iv) help parties to clarify their opinions, interests and values, (v) deflate unreasonable claims, (vi) seek joint gains, (vii) keep negotiations going and (viii) articulate rationales for agreement (pp. 108, 109).

Literature commonly identifies at least two important types of third-party intervention.[3] One type is *mediation* and the other is *arbitration*. Mediating in conflicts means that a third party intervenes between the conflicting parties. This third party is "an impartial outsider who tries to aid the negotiators in their quest to find a compromise agreement" (Raiffa 1992, p. 23). The role of a mediator is somewhat comparable with that of an arbitrator, but differs on one important respect. Whereas the mediator has no authority to impose a solution the arbitrator does have such authority.

The sections below investigate the role of mediation and arbitration in the way the communities deal with conflicts. Three communities are presented and discussed. They are Apache, Debian and Linux.

Mediation and arbitration in the Apache community

The Apache community basically consists of two institutions. The first is the Apache Software Foundation (ASF). Potentially, the ASF could act as a third party when conflicts in the community occur. However, in an interview an ASF board member disagreed that the ASF would take up such a role. He explained, "The board of the foundation essentially limits its attention to financial and judicial aspects of Apache. We enforce the project management committee and we support them in their activities. The board does not influence the direction of Apache and we do not interfere with coding." Thus, the respondent explicitly disagrees that the board of the foundation would interfere with the process of coding, which means that it is unlikely to interfere when conflicts do arise surrounding software development and maintenance.

In the statement this board member refers to a second institution, the project management committee. The Apache community is divided into many 'parent projects.' Examples are the HTTP Web server and the Jakarta project. Each of these projects has its own project management committee, which is responsible for the project's technical direction. As a management committee responsible for the technical direction it is not unthinkable that this committee would take on a mediating or arbitrating role when conflicts occur, especially when the conflict involves technical issues. One respondent from the ASF argued, however, that it does not: "Nowadays, the PMCs [project management committees] are too far away from the actual coding. The PMCs do little. They coordinate licensing issues with the board. They also discuss if they want to start something new or cease something old." In other words, according to this respondent, a project management committee does not interfere much in the actual processes in the projects. Conflicts do occur regularly, though it would seem that the management committee is not involved in resolving each and every one of them.

Both the ASF and the project management committees are claimed to interfere little with the actual software development and maintenance processes. Yet many conflicts are likely to surround these processes. Take, for example, users' many different, often conflicting, requirements. How does the Apache community manage these conflicts? The project management committees and the ASF might mediate or arbitrate some of the conflicts, particularly those that surround legal and financial issues. However, there must be something else. The interviews with members of the ASF left the impression that conflicts are dealt with otherwise.

Mediation and arbitration in the Debian community

The Debian community has an elected project leader. This person is potentially suitable to mediate and/or arbitrate when conflicts occur. One of the former project leaders interviewed in light of this research characterized most conflicts as "technical." He continued, "Internally, the project manages itself. Every now and then the community derails and then I will interfere, predominantly via e-mail… I ask a question like why doesn't it work, how can we change it, or how can we make it work?" In other words, the project leader claims to perform some activities that fit the profile of a mediator. The project leader asks questions and tries to move the opinion of the conflicting parties in the direction of a solution.

Yet project leaders also realize that they can only try to convince people to look in a certain direction. "You can try to make people look in a certain direction." But even when the project leader interferes in the conflict it can, according to the respondent, take "months of discussion" before some form of consensus is reached.

Mediation and arbitration in the Linux community

Much research addresses the role of Linus Torvalds in the Linux community. Some argue that he is a clear leader (e.g. Lerner & Tirole 2002b, Moon & Sproull 2000), while others feel that his role is more moderate (e.g. Von Hippel & Von Krogh 2003, Van Wendel de Joode et al 2003). Whatever, his role, one thing that appears to be lacking is the possibility that he would take on the role of mediator or even arbitrator.

Consider the example of a conflict about virtual memory. For a long time, the Linux kernel had two alternatives. One version was written and maintained by Rik van Riel, the other by Andrea Arcangli. The conflict first manifested in May 2000 and continued at least until December 2001. The conflicting parties had many clashes over this period. Zack Brown described one of these clashes:

> He [Rik] gave his own technical explanation, at which point Andrea accused him of going off-topic. Rik said he'd just been arguing against Andrea's point; and added, "Unfortunately you seem to ignore my arguments, so lets close this thread." Andrea replied, "I've not ignored them, as said they were either obviously wrong of offtopic." To which Rik quipped, "Without giving any arguments."[4]

Many more such examples of eruptions of the conflict can be found on the Linux kernel mailing list.

Occasionally, Linus Torvalds interfered in these discussions. When he does, he is an active participant displaying his preferences just like any other participant. Take for example the following statement by Linus Torvalds: "I still don't like some of the VM [virtual memory] changes, but integrating Andrea's VM changes results in (i) better performance and (ii) much cleaner inactive page handling in particular."[5]

Thus, the role of Torvalds does not seem to be one of a *mediator*. He interferes and displays his preferences, but does so as just one of the people involved in the conflict. This is just one example of many in which Torvalds is unafraid to make his opinion heard. Therefore, the title mediator appears to be inappropriate. Neither does Torvalds appear to be an *arbitrator*. There is one rather straightforward reason why Torvalds is not a stereotypical arbitrator: If he were an arbitrator, then the conflict between Andrea and Rik would not have continued for so long. In his eyes, Andrea's system has almost always been the clear favorite and he has made his preference heard on many occasions. Yet the conflict lingered on. In other words, he could do little to stop it.

Obviously, this is just one example of a conflict. In other conflicts, Torvalds could have played a more important role in finding a solution. The president of Linux International disagrees. He argues that Torvalds should not be seen as a mediator or arbitrator of conflict. Instead, according to him, Torvalds is known to stimulate conflicts rather than try to solve them. "Linus always says that there are ten solutions to each problem. Sometimes Linus may not know that there is a better design, but he is willing to bet there is a better design based on this principle."

The fact that the role of Torvalds was limited in the conflict described above does not imply that he can be compared to any other participant in a conflict. It is, for instance, very likely that his opinions, interests and values are better listened to than those of other participants in the conflict. It is not at all clear how this affects the resolution of conflicts; it could lead to conflict resolution but in certain instances his visibility could just as well give rise to more conflicts.

Conclusion: mediation and arbitration have limited influence on conflict resolution

In sum, many conflicts in open source communities do not appear to be solved by the interference of a third party. Two observations lead to the conclusion that mediation and arbitration cannot be said to be the primary source of conflict resolution: (i) There are many conflicts in which no project leader or institution interferes, which implies that these conflicts are managed or solved in a different way. (ii) There are many examples of conflicts in which a project leader or institution gets involved but where there is no clear indication that these interventions resulted in a quicker or easier resolution of the conflict. On the contrary, a former project leader in the Debian community said that the conflicts in which he had interfered typically lingered on for months.

These findings are similar to the observation by McCormick (2003), who analyzed the role of one particular institution regarding conflicts in the Star Linux community. He writes, "A Technical Review Board exists in Star Linux to resolve technical issues, but it is almost entirely inactive and no formal action has been taken by this board to resolve a conflict between members for over three years, despite a few petitions for them to do so" (McCormick 2003, p. 30).

Ignoring the open display of conflicts: a culture of doing

Participants in open source communities generally have a low tolerance for conflicts about how things should be done. Participants "hate discussions."[6] Consider this statement made by a developer in the Linux community, who was clearly fed up with the conflict between Andrea Arcangeli and Rik van Riel about virtual memory in the Linux kernel: "I'm also less than thrilled by the whole situation with VM – all sides of it… I'm taking no part in your merry 5-way clusterfuck – sort that mess out between yourselves… I do _not_ appreciate being enlisted into anyone's holy wars."[7]

Participants in open source communities appear to agree that 'doing' is better than 'talking.' Participants can have great ideas and arguments for why a certain solution is better, but most appear to agree that this needs to be demonstrated or 'proven' with actual source code.

Without source code you have no real voice. "Look, that is the way… it works, when you make it then it is there. Factually, you do not have a voice when you do not program."[8] Basically, participants in the communities need to "put their money where their mouth is." They are free to discuss ideas and concerns, but eventually they have to produce proof; they must show the source code. This aspect of open source communities is perhaps best viewed and understood as part of the technical rationality, which is claimed to be an important norm in open source communities (Weber 2004).

One of the people in the Linux community who actively propagates and defends the culture of doing is Linus Torvalds. The next statement is an example of Torvalds openly criticizing the conduct of another developer, because he keeps bringing up arguments and ideas without actually showing proof: "And to Donald's statement about having ten modifications outstanding," Torvalds went on, "Again, THIS is the problem. They shouldn't be outstanding. They should be out there, in the standard kernel. If they work, they work. If they don't, we find out. And if they don't work, then others can at least help."[9] The statement demonstrates how, in Torvalds' opinion, only the actual source code can prove the claim that source code is good or better than another piece of source code. This can, however, only be done when the developer (Donald in this case) places the source code in the open and effectively shares it with other developers in the community. This allows other people to judge whether the claims made are indeed valid.

Another statement by Torvalds reiterates the importance of factual proof: "I'll take numbers over talk any day. At least Mike had numbers, and possible explanations for them… In short, please don't argue against numbers."[10] Again, this statement elicits how actual proof is the only way to convince others that certain ideas are better than other ideas. If there are developers who still feel that their ideas are better, then they should either improve the existing code or come up with other 'proof' to support their claim.

Ignoring conflicting ideas and arguments

A side effect of this culture of doing is that participants can, and will, ignore conflicts that are heavily debated on the mailing lists. They argue that they want to focus on the actual coding and maintenance of the source code, rather than listen to participants who have different perspectives or ideas and believe that things should be done differently. Consider this statement by the head of a team translating KDE software into Dutch: "So if I think something should be changed then I will do it myself, particularly when the group disagrees." The interesting aspect of this statement is not that the programmer understands that the group disagrees; but that he implements his ideas anyway, even though he knows that others in the group will not like the change. Clearly, he does not consider the interests or opinions of others as an obstacle or problem preventing him from doing what he believes should be done. Furthermore, implementing changes without consulting the other members of a community diminishes open display of the conflict. Others can criticize ideas, opinions and changes to the software, but the developer ignores them. The same respondent had this to say about another person in the community: "In the discussion I can say 'no' as loudly as I like, but if they decide to change it, I cannot do anything to prevent it." Again, this is an example in which a conflict about how things should be done is bypassed. The respondent can tell another person that they should not make an intended change, but if that person decides to ignore the advice and continue with the changes anyway, there is nothing the respondent can do to stop it.[11]

Thus, not only are developers able to ignore the opinions of the group, the group cannot prevent participants from implementing changes. "If something requires a lot of work and you do that work and someone else wants something different that's okay, but that person has to write it."[12] Finally a statement from a member of the Debian community: "That is typically something where you as a package maintainer say, I will do it or I won't. And then other people can jump high or low, but you don't need to be bothered with it."

To summarize, many participants in the communities ignore other people's ideas and interests. They do what they feel is the right thing to do.

The use of archives

A number of communities have created archives. An example of such an archive is the *to-do list* that is used in the PostgreSQL community. This is a list that names all the functions and ideas that people deem important enough to be implemented in the software.[13] There are a number of action items on the to-do list that are linked to previously discussed conflicts on the PostgreSQL mailing list. Consider, for instance, the action item 'update.' The link redirects the reader to a discussion between two participants on how this idea should be implemented. There are many more such links on the to-do list. Some link to 'friendly' conflicts, that is, conflicts that end in agreement. Others link to conflicts that are clearly not resolved. They just die out, because people stop discussing them.

A document that is similar to the to-do list in the PostgreSQL community is the *Python Enhancement Proposal* (PEP), used in the Python community. The project leader of the Python community explained how certain conflicts reccur every now and then:

> At the end of the nineties some discussions recurred time and again, even a couple of times a year. "Why don't we do this or that with Python?" In the first round there is discussion about why something should be changed. After that there is frequently a second round of discussion on whether it can be done, and then the interest is gone. Half a year later someone else comes along and suggests the same and then a new discussion with new people starts.

The project leader explained why they created the PEP:

> The old timers created the PEP… People are recommended to write down their ideas, as long as their ideas are not ridiculous… Then the PEP is placed in a newsgroup… The result of the discussion is added in the PEP. The PEP remains in the archives. When the same question comes up, the old timers can now say, "See PEP no. x."

The to-do list and the PEP serve a similar goal; they allow participants to ignore conflicts respectfully. It is respectful, because participants are not saying that they are ignoring other people's arguments and ideas. On the contrary, the arguments have been taken seriously, and their item is moved to a to-do list or a PEP. The items in these archives are considered to be important by at least one person. In a sense, the archives constitute a tool for managing the 'losers' in a conflict (see De Bruijn & Ten Heuvelhof 2000). The fact that it takes little time and effort to place the arguments in such archives, makes them efficient tools for ignoring conflicts. Participants can move a conflict to the archives and then continue with other activities. Furthermore, every next time a similar conflict arises participants can again ignore the conflict and simply refer the involved parties to the archives.

Deflecting conflict: creation of parallel development lines

The culture of doing can give rise to conflicts. Participants might have created alternative solutions and subsequently disagree as to which solution is better and should thus be implemented. An extreme example of such a conflict is known as a 'commit war.' "We talk about a commit war when people working on the same parts of code keep overwriting each other's changes" (Bauer & Pizka 2003, p. 174). Basically, a zero-sum game is created in which either of the two solutions is accepted and adopted.

One way to end such conflicts is the creation of two or more parallel development lines. The editor in chief of the Linux journal explained, "There are many projects where two groups are taking on the same issue and are basically doing the same… It seems like there are often two different ways of tackling a problem and people try both. The good thing about it is that a lot of software migrates back and forth between the two groups." The basic premise is that parties with conflicting ideas, interests or values can start different lines of software development. By allowing them to start a new line the conflicting parties can actually write code the way they deem most appropriate. These two or more lines compete and the participants working on them can learn from each other and adopt their ideas. An ASF member explained the process of parallel development lines in the Apache community: "It is more a competition kind of a thing. Someone says, 'we should do it this way' and another does something else. They will form little projects. Each project will compare its software with the software from the other project, if they like a feature they will add it to their project as well."

Parallel development lines do not resolve a conflict. The conflicting parties can still disagree with each other. Importantly, however, the potential negative consequences of a conflict, namely inactivity and overt hostility, are reduced. Parallel development lines "ensure a project will continue to evolve and improve."[14] Conflicts are not a problem or a threat to the organization of an open source community and to the continuity of that organization, because activities can continue.

There are different ways in which the mechanism of parallel software development lines is institutionalized in the communities. The three most observed ways are (i) new 'heads' in the CVS, (ii) institutionalization of a 'stable' and an 'experimental' line and (iii) the presence of a separate commercial development line. These are explained in more detail below.

New heads in the CVS

Many communities use a CVS to support software development and maintenance processes.[15] Contributors themselves can commit their source code to the CVS. But it is possible that others will disagree with the modification. In that case there are a number of options open to the participants who disagree. One is to remove the latest addition from the

CVS. Yet this would likely lead to controversy, especially if some members like the modification or even see the modification as necessary. How to decide who is right?

In Apache the usual solution is to start what is called a new *head*.[16] A head is the most recent version of a particular piece of source code in the CVS. Usually there is one head. Creating a second head, or in other words, a second development line, allows people to develop and implement source code the way they consider best. One member of the ASF Board of Directors explained that the developers in the community then monitor the development of both lines and judge for themselves which alternative they consider best. Often, after a while the community reaches a consensus on which alternative they agree.[17]

For now, the key aspect of the creation of a second head is not the way in which the two lines are reintegrated.[18] Rather, the interesting fact is that the second head does not solve the conflict but rather diminishes one negative effect and promotes the positive aspects of conflict. A potential threat or disadvantage of a conflict is the rise of inertia or inactivity. The creation of a second head reduces this threat, as participants can continue their programming and maintenance activities. An advantage of the second head is that it stimulates creativity and active participation. Participants are lured into competition with one another and are stimulated to demonstrate that their solution is the better one. To make their point they have to create new and better source code. If they do not, their arguments are likely to be ignored, as explained earlier in this chapter.

The stable and the experimental development line

Many communities harbor a wide variety of participants. This diversity is one of the reasons for the occurrence of many conflicts. One source of diversity is the difference between non-technical users of open source software and highly skilled and trained software engineers. Typically, non-technical users want a product they can download and install as easily as possible. Furthermore, they want to use software that does not change much and thus remains the same over a longer period of time. Typically, however, highly skilled and trained professionals would rather work on challenging new tasks.

The literature appears to focus on one possible problem this difference might lead to, namely, motivating the highly skilled developers to work on tasks that they consider mundane, but which are important to non-technical users. Examples are activities like solving bugs and writing manuals (Benkler 2002a, Lakhani & Von Hippel 2003). Much less attention is given to another consequence, namely, the conflicts that are bound to derive from this difference. The creator and maintainer of the Python programming language explained:

> There is demand for a conservative release, especially from the business forum. You can never agree about releasing more or less often. They do not want the language to change except for bug fixes. Personally I would lose my interest if I could not enhance the language. I have too many things in my head that I want to add. This is a tense relationship.

This statement stresses the way in which the different interests are bound to lead to conflict. Furthermore, the statement demonstrates the respondent's belief that it is impossible to get people to agree on these issues. Therefore, conflicts between non-technical users and professionals are destined to surface regularly.

Practically all communities have institutionalized a mechanism to deal with some of the conflicts between non-technical users and professionals. This mechanism is maintaining two versions of the software, namely, the stable and the experimental software line. Similar to the previous mechanism (creation of a new head), the aim of this mechanism is not to resolve the conflict. The goal is rather to facilitate satisfaction of the diverse needs of the participants. The founder of Linux International explained:

> Linux consists of a production version and development version. The even numbers are production versions; the odd numbers are for development. The development version is for trying out new things and testing. The even or production versions are the versions that are used in the distribution and indicate that the version will remain stable for a reasonable amount of time.

This mechanism is not limited to Linux. The presence of two 'official' versions can also be found in communities like Debian, PostgreSQL, Apache and Python.

Having two versions ensures that conflicts surface less often. Non-technical users and professionals who favor a stable software package download the stable version of the software and are ensured that their interests, less changes, are at least partly satisfied. Highly skilled developers, on the other hand, are likely biased to use and participate in the development of the experimental or development version of the software. This version challenges them to apply their skills and test new ideas.

The commercial development line

There is an inherent tension between voluntary participants and contributors employed by companies for which the goal is to market a product based on free software. The latter have to meet deadlines and predefined performance levels, and their products must meet the requirements of their customers. How can they keep these deadlines and performance levels when the communities consist of many participants who do not have such clear-cut goals to meet? To address this issue and prevent conflicts between voluntary and paid contributors, at least one company decided to create a separate commercial development line. The commercial development line allows the programmers from the company to work independently from the other participants in the community and thus prevents many of the conflicts from rising to the surface.

The company Covalent created its own Apache software development line, enabling it to improve the software almost entirely independent of the community. In its version, Covalent has the option of optimizing and reconfiguring any part of the latest version of the Apache software and ensuring that its version meets its customers' requirements. This also allows the company to shield its customers from the community. "The ASF may release four new versions in one week… our customers don't notice it though, because we isolate them completely from the ASF."[19] For Covalent, the advantage of disconnecting the development lines is that it no longer needs to pressure the community. "We just don't care if the ASF releases a new version or not."[20]

The dual development lines enable Covalent to disconnect its development efforts from the efforts in the Apache community. Covalent focuses primarily on creating a product that meets the wishes of its customers. Therefore, it runs a lot of tests and performs a lot of bug-fixing. These bug fixes are added in Covalent's version and are sent to the Apache community. Participants in the Apache community can decide whether they want to add the fix in their version as well.

The dual development line method also has drawbacks. The most obvious is the continuous need to integrate the improvements made by Covalent into the official Apache version and vice versa. Covalent must continuously monitor developments in the Apache community. The community might find and fix a bug after Covalent has downloaded a version of the software. In that case, Covalent must integrate the bug fix into its own version as well. In other words, the commercial development line raises the costs of the development and maintenance of the software.

Voting with your feet: the exit option

Chapter four argued that open source communities are relatively fluid; participants can join and exit at practically anytime they like. For instance, mailing lists allow any one to join at any time and leave at any time. Furthermore, there are relatively few communities in which participants need to sign a contract before they can actually contribute source code. In the communities that do have contracts, like Apache, participants are still free to determine how much time and effort they devote to the community and how or when they want to leave the community. In short, individuals can exit a community whenever and however they like.

The exit option

One of the first researchers to address the option of individuals and/or companies to exit a relationship and the consequences of this exit option is Hirschman (1970). He writes, "Some customers stop buying the firm's products or some members leave the organization: this is the *exit option*. As a result, revenues drop, membership declines, and management is impelled to search for ways and means to correct whatever faults have led to exit" (Hirschman 1970, p. 4). According to Hirschman, the exit option is the underlying principle of all economic thought. The idea is that exit is a way to signal the management of an organization that it is doing something wrong, at least in the eyes of its customers and members.

Hirschman compares and contrasts the exit option with the voice option. "The firm's customers or the organization's members express their dissatisfaction directly to management or to some other authority to which management is subordinate or through general protest addressed to anyone who cares to listen: this is the *voice option*" (Hirschman 1970, p. 4). Both the voice option and the exit option are ways for people to ventilate dissatisfaction and disagreement, the difference being that the voice option is bound to lead to discussions and conflict, as it is a direct confrontation of opposing ideas. Exit, however, does not lead to conflict. One could say that it is a way of communicating dissatisfaction by 'doing' or 'voting with your feet.'

The exit option and the voice option, Hirschman argues, are each other's substitutes. In his book he makes the following claim: "[T]he presence of the exit option can sharply reduce the probability that the voice option will be taken up widely and effectively. Exit was shown to drive out voice, in other words" (Hirschman 1970, p. 76). Given that the voice option gives rise to overt conflict, this implies that the exit option results in fewer conflicts. When people have the possibility to exit, they use their voice less and thus do not mobilize their dissatisfaction directly. Instead, they exit by leaving the organization and ceasing use of the organization's products.

The exit option in open source communities

The fact that open source communities have very few restrictions to exit a community creates a powerful tool for participants to ventilate their opinion. If they disagree with a certain decision they can leave. Or, if they dislike the discussions on a mailing list, they can unsubscribe to that list and thus no longer receive the e-mails. The low costs attached to leaving a community ensure the institutionalization of the exit option and thus provide a way to defer many of the (potential) conflicts.

The exit option also promotes and in a way institutionalizes the culture of doing. Guido van Rossum, project leader in the Python community, explained why certain contributors feel the need to exit the community: "Sometimes developers have the idea that they can make a tool that can work with Python, but there might be no place for it in Python. Sometimes they start their own project in which they can specify their solution or idea, like Phyrex or Phycom."

Psycho is an example of such a project. A Swiss PhD student named Armin Rigo developed Psycho. Rigo wanted to develop software that would make Python, which is a higher level programming language, faster than lower level languages. Rigo knew, however, that this idea contradicted popular belief, which presumes that a higher level programming language is always slower than a lower level language.[21] Of course he could have stayed in the community and tried to develop the tool there. This, however, was likely to lead to resistance and hence conflict, as people did not believe this to be possible. His decision to exit the community and develop the tool outside the Python community enabled Rigo to develop Psycho without running into these conflicts. In other words, his decision to exit prevented many potential conflicts. In an interview with Python's project leader, he stated that there is a good chance that Psycho will be included in an official Python release in the near future.

The fork

One of the most extreme forms of the exit option is the *fork*. According to the jargon file, a dictionary for open source developers, a fork "occurs when two (or more) versions of a software package's source code are being developed in parallel which once shared a common code base, and these multiple versions of the source code have irreconcilable differences between them."[22] The fork is a process in which two different and separate software programs are created based on one software program.

Most forks start with a conflict: a difference in opinion, value or interest between two or more parties. Then one of the parties decides to take the source code and start a new development project based on that code. Hence, that party exits the community. The difference between a fork and the exit discussed in the example of Psycho is that Rigo developed a tool on top of Python. He did not start a second Python development project. In the case of a fork, however, the exiting party takes the source code from the original community and tries to create a new and competing community. The software in the newly erected community is a true fork once both programs have irreconcilable differences between them.

Most research on open source communities appears to associate the fork with something negative. Narduzzo and Rossi (2003) state, for example, "These forks are an expression of a coordination failure" (p. 29). Kuwabaru takes a similar stance regarding the act of 'forking,' meaning to deliberately create a fork. Kuwabaru writes, "Forking… and other behaviors become taboos" (Kuwabara 2000, p. 48). These researchers focus on one element of forking, which is that it can lead to a destruction of value. It effectively results in the presence of two communities. Initially these communities have few differences between them. Having two communities spending resources on more or less the same source code would seem to be a waste of resources.

However, the destruction of resources is not a sufficient reason to conclude that forks are inherently bad or an expression of coordination failure. One could also consider them to be a dramatic way of exiting a community and hence an effective means of deflecting the potential negative effects of a conflict. "Forking can often be good for society because it prevents one person or clique from thwarting another group" (Wayner 2000, p. 211). The point here is not that forks are necessarily good, but to view forks as inherently bad is equally incorrect.

The fact that forks do result in a destruction of resources explains why participants in open source communities stress that one should refrain from using this mechanism for as long as possible. They also recognize, however, that it is sometimes impossible to continue working together. In that situation, forking a project is justified and perhaps even recommendable. The former project leader of Debian stated, "It is essential that you can decide to leave if you do not agree. There is a rule that you should not do it if you do not think it is strictly necessary. But sometimes it is necessary!"

Conclusion

This chapter presented a number of observations. It started with a discussion of why many open source communities face a high potential for conflicts. The argument was that the communities harbor participants who have different backgrounds, interests and motives to participate. Yet the participants are mutually dependent. At the same time it argued that the modularity of open source software reduces the conflict potential, as it increases the independence of participants.

This chapter further identified and discussed mechanisms that handle conflicts. They work differently, however, then one might expect. Mechanisms like parallel development lines, the exit option and the fork do not solve conflict, they deflect at least one potential negative effect of conflicts; that is, the risk that conflicts will threaten the speed and continuity of processes in the communities. In the long run, this could endanger the continuity of a community. How and why the mechanisms defer the negative consequences of conflicts can be understood with, again, three rules that drive the individual behavior of participants.

"*Participants want to demonstrate that they are right.*" Individual and corporate participants become involved in the communities for a wide variety of reasons. One thing that unites both groups is that they are highly skilled and in many ways behave like professionals. Many of the participants enjoy participating, but they do want to see their ideas accepted (Hertel et al 2003). This last point is in line with one of the motives for participants to invest their time and effort in the communities, namely, to build a reputation (Lakhani & Von Hippel 2003, Raymond 2000). However, the only way for participants to ensure that their ideas are implemented is to actually create solutions. Participants must do. This behavior is related to the second rule.

"*Participants want to minimize the time they spend in conflicts and discussions.*" Participants by and large ignore conflicts when no solution or alternative is available. They might at first enjoy a discussion, exchanging ideas and learning from others. However, many of the respondents claim they are easily bored and annoyed by conflicts. Conflicts without a working solution are considered hypothetical and frequently ignored. Such conflicts can easily become a waste of time since the communities see the 'proof of the pudding to be in the eating.' In other words, to prove that an idea actually works an idea must be implemented. Without implementation, participants soon become bored and disregard the conflict.

"*Participants remove or overwrite contributions if they deem them inappropriate.*" This third rule is not associated with the culture of doing. A previous chapter argued that software development is based on individual choice. This also means that anyone can remove or overwrite other participants' contributions if they feel the contribution is inappropriate (McCormick 2003). This rule is especially true for communities that have adopted a CVS. In the Linux community this is somewhat different.

How the individual behavioral rules support the observations

Consider the first and the third rule. Combined, both rules could give rise to many conflicts. Two or more participants might want to prove that their solution is best and therefore continuously remove each others' contributions. Indeed, a number of researchers have described how such 'commit wars' can arise. These are conflicts in which participants continuously remove and/or overwrite each other's work (Bauer & Pizka 2003). Commit wars are especially relevant in communities in which software development is supported by a CVS. In the Linux kernel community, the source code cannot be removed by just anyone, but Linus Torvalds can decide not to include a certain piece of source code. This suggests the presence of conflicts between participants and Linus Torvalds.

This chapter argued that if participants want to prove they are right, they must create a working solution. This was referred to as a culture of 'doing.' This culture of doing can be explained with both the first and second rule governing individual behavior. Many individuals in the communities participate because they want to create solutions to complex problems or because they simply enjoy creating software that works. Conflicts can be interesting, but they do consume time, which participants can better use to implement some of their own ideas. Conflicts that lack competing solutions are frequently considered a waste of time, because there is no proof that the idea will actually work. Such conflicts are frequently ignored by many of the community members.

This chapter also introduced three mechanisms that have one thing in common, namely, they result in multiple development lines. These mechanisms were a stable and development version of the software, a new head in the CVS and a commercial development line. These mechanisms influence the way in which open source communities deal with conflicts. The mechanisms all give participants space to prove their ideas. The option of creating a parallel development line provides participants the option of writing source code and demonstrating that their solution is better than the existing solution(s). This also works the other way around: conflicts in which only one solution is available can be ignored. If participants truly believe there is an alternative that is better, they must stop arguing and demonstrate their point by actually creating the source code.

The question then is what happens when two solutions to one problem have been created? How is such a conflict resolved? Again, the answer lies in the parallel development lines. The fact that parallel development lines are created implies that two or more solutions can coexist. Everyone in the community can then make their own informed choice and decide which of the solutions they like best. This does not mean that the conflict is resolved; instead it is deferred, meaning that everyone is free to choose the alternative they want to work on.

With or without alternatives, the culture of doing combined with the mechanisms to create multiple development lines, ensures that conflicts are unlikely to threaten the continuity of software development and maintenance. First, many conflicts can be ignored. Second, the conflicts that cannot be ignored, that is, those in which more than one alternative has been created, do not threaten the continuity of processes. Everyone can make their own choice and continue to perform their activities as if nothing has changed.

The mechanisms described appear to provide an effective means to prevent conflicts from threatening the continuity of processes in open source communities. This does not mean, however, that they are efficient. The fact that anyone can ignore conflicting arguments and can create a parallel development line does imply a continuous rise of solutions and software programs. It also suggests high levels of redundancy and overlap. In many communities this has led to a rise of alternative solutions that appear to have similar functionalities.

The next chapter describes some of the forces countering this continuous rise of divergence and describes how a certain level of convergence is achieved.

Notes on chapter six

[1] The interview was held with one of the two maintainers of the Lilypond software.
[2] Based on an interview with the executive director of Linux International.
[3] Raiffa adds two other forms of third-party intervention, namely, intervention by a facilitator and intervention by a rules manipulator (see Raiffa H. 1992. *The art and science of negotiation.* Cambridge: The Belknap Press of Harvard University Press). These two forms of intervention are not further analyzed in this section. The reasons are relatively straightforward. First, the facilitator is used to get the relevant parties to talk to each other. The problem, however, in open source communities is not that the involved parties do not talk to each other. On the contrary, most conflicts are heavily discussed and debated on the mailing lists. Intervention by a rules manipulator is also not further analyzed, but is considered to be a specific form of arbitration.
[4] From the Internet: http://kt.zork.net/kernel-traffic/kt20010112_102.html#10 (April 14th, 2003).
[5] From the Internet: http://kt.zork.net/kernel-traffic/kt20011001_135.html#4 (June 27th, 2003).
[6] From an interview with a programmer at CNet who is also member of the ASF.
[7] From the Internet: http://kt.zork.net/kernel-traffic/kt20011001_135.html (July 4th, 2003).
[8] From an interview with the project leader of BlueFish.
[9] From the Internet: http://kt.zork.net/kernel-traffic/kt19991101_41.html#6 (July 9th, 2003).
[10] Taken from the Internet: http://kt.zork.net/kernel-traffic/kt20010316_111.html (April 2nd, 2003).
[11] Most groups of developers do have the means to prevent others from adding code or making other changes to the code base they collectively manage. But, as shown in the first parts of this chapter,

developers who want to make changes have leeway to implement changes in other ways. They could, for instance, start a new head in the CVS, create a fork, or start a new project.
[12] From an interview with a maintainer of a number of modules in the Linux community.
[13] Based on the TODO list website: http://developer.postgresql.org/todo.php (April 2nd, 2003).
[14] From an interview with the editor in chief of the Linux journal.
[15] Linux is probably one of the best-known examples in which no CVS is used. Most other communities do use a CVS. Examples are Apache and PostgreSQL.
[16] Based on two interviews with members of the ASF Board of Directors.
[17] How the community decides which solution is best will be discussed in the next chapter.
[18] The question of how a community decides what alternative to choose is addressed in the next chapter.
[19] From an interview with a member of the ASF Board of Directors.
[20] From an interview with a member of the ASF Board of Directors.
[21] Based on an interview with the creator of Psycho, Armin Rigo.
[22] From the Internet: http://jargon.watson-net.com/jargon.asp?w=fork (April 7, 2003).

CHAPTER SEVEN

COLLECTIVE CHOICE

This chapter discusses the fourth design principle, namely, the *presence of collective choice arrangements*. Basically, collective choice arrangements are decision-making processes that involve a collective of individuals.

Collective choice is relevant in open source communities. This chapter focuses on one reason in particular, namely, the large potential for divergence in software development. The previous chapters identified a great number of mechanisms, like parallel development lines, the fork and modularity that provide an opportunity for participants to implement their own ideas and to diverge. Combined with the large diversity of the participants, which results in a wide diversity of interests and ideas, the huge potential for diversity is obvious. To some extent this large potential for divergence is a threat to open source communities. Divergence gives rise to fragmentation, which in turn could lead to incompatibility of software, to inefficiency and hence to a destruction of resources. Clearly some form of convergence is needed, which suggests the presence of collective choice.

Two ways in which collective choice can be achieved in open source communities are through a voting system and through project leadership. This chapter argues that the outcomes of a voting system and the choices of project leaders hardly restrict the actions of participants in the communities. The actions of individuals are primarily based on individual choice. This is not only true for the way in which convergence is achieved in software development; individual choice is also relevant when an open source license needs to be chosen and when a coordination mechanism is adopted.

One might expect that these individual choices result in randomness. However, this chapter argues that this is not the case. Five mechanisms are identified and described, which are said to influence the choices of individuals in the communities. These mechanisms result in a process of 'swarming' in the communities. Participants move in swarms when they select an open source license or when they select a limited number of software programs from a wide variety of potential alternatives.

The fourth design principle: collective choice

The outcome of a collective choice binds individuals

Ostrom (1990) distinguishes different levels of rules. At the lowest level are operational rules. Individuals in the communities are bound by the operational rules that restrict their behavior. Operational rules specify what people in the community are allowed, required and forbidden to do with the common pool resource. Operational rules include boundary rules, appropriation rules and provision rules (Ostrom et al 1994). They regulate the "timing, technology used, purpose of use, and quantity of resource units harvested" (Hess & Ostrom

2001, p. 61). The operational rules are 'managed'. They need to be formulated, communicated and adapted when the situation calls for it:

> So property rights and other so-called operational rules of conduct are made subject to wider societal processes for adaptation. The wider societal processes involve collective choice rules within which stakeholders, possessing varying bundles of property rights, will articulate and aggregate their interests. Decisions will emanate in the forms of revised operational rules and other outcomes (Sproule-Jones 1998).

The operational rules are formulated, reviewed and changed in the 'collective arena' or 'collective setting.' Collective choice arrangements, then, are the rules or procedures that organize the process of formulating, changing and adapting operational rules to changing circumstances (e.g. Ostrom 1990). Individuals who participate in a collective choice thus determine when and how they and others are allowed to appropriate from the common pool resource and when they must contribute to the provision of the resource. Oakerson writes, "Individuals are no longer entirely free to decide for themselves how to make use of the commons, as in a private-property arrangement, but participate in a process of collective choice that sets limits on individual use" (Oakerson 1992, p. 47). Thus, a collective choice binds and restricts the individuals in the collective. Outcomes of such a choice affect the actions of individuals. "Decisions made in collective-choice situations *directly* affect operational situations" (Ostrom 1990, p. 192). Individuals cannot make decisions based solely on their personal discretion, but are bound by the restrictions imposed on them. These restrictions are not limited to rules. Other types of decisions that bind and restrict individuals can also be called a collective choice (Schwartz 1986).

To summarize, a key aspect of the definition of collective choice is that it is a decision process of which the outcome *binds* and *restricts* the members of the collective.

Collective choice in community-managed common pool resources

According to research on community-managed common pool resources, a collective choice is a decision-making process in which individuals affected by the outcome are able to participate (Ostrom 1990). The reason is that the involved individuals possess knowledge about the resource and about the effects of certain types of usage on the resource. This knowledge must be used to ensure a proper fit between the rules and the actual use, and the effects they have on the sustainability of the resource. In a similar line of reasoning, Buck (1998) writes that if users are not included in a collective choice, those who do not participate are likely to be ignored and therefore their particular type of usage will be ignored as well. Thus, according to Buck, it is important to include all the appropriators (i.e. users with appropriation rights) in the collective choice. Analyzing forest management in India, Ghate (2000) argues that most efforts fail. One of the reasons she claims is that "there has been no conscious effort to give representation to members from different social and economic strata. Therefore there is a likelihood of formulation of rules more suitable to particular class within a community" (Ghate 2000, p. 16). Ignoring particular classes in a community could be destructive as it creates inequality and disregards types of usage that must be included to fully understand the consequences of certain restrictions.

Collective choice arrangements

This chapter introduces two mechanisms that could function as collective choice arrangements in open source communities. They are *project leadership* and *voting systems*. The decisions made by a project leader are typically decisions in which those affected do not participate. In that sense they are different from the collective choice arrangements described in research on community-managed common pool resources. The reason for including project leadership in the analysis is that many publications on open source communities attribute much value to the role of project leaders (Bonaccorsi & Rossi 2003c, Fielding 1999, Markus et al 2000, Moon & Sproull 2000). "As a rule, open-source projects have well-structured governance models with clearly identified leadership" (Markus et al 2000, p. 21). A second reason is that project leaders are potentially able to make decisions that restrict the individuals in a collective and in that sense they are a collective choice arrangement.

The voting system is probably the most frequently cited mechanism to reach a collective choice (e.g. Schwartz 1986, Walker et al 2000). A central question in research addressing voting and collective choice is how voting systems can lead to a fair aggregation of individual choices. This type of research is not confined to common pool resources. It extends into many disciplines (Sen 1970). The origins of voting and collective choice theory can be traced back to economics and specifically to a publication by Kenneth Arrow (1951). He defines collective choice as a function responsible for aggregating the preferences of many individuals into a collective preference (Ferscha & Scheiner 1999). According to Arrow, there are a number of conditions such a choice should adhere to. At the same time he argues that it is impossible to develop a voting system that can adhere to each one of these conditions simultaneously. Research building on Arrow's conclusions has focused primarily on the development of voting systems that result in a 'better' aggregation of individual preferences. To achieve this goal, research has looked into a wide variety of voting systems and their influence on the aggregation of individual preferences. Systems analyzed are, for example, majority rule, plurality voting and Borda counting.

Convergence in open source: the need for collective choice?

Previous chapters presented and discussed a great number of mechanisms and reasons for divergence in open source communities. Mechanisms like parallel development lines, forks and modularity allow participants to create their own version of existing software, and to do so they need to invest relatively little time and effort. The previous chapter also argued that participants in the communities are highly diverse; they have different nationalities, enjoy different levels of education and participate in the communities for a wide variety of reasons. Some developers are paid to participate, while others are volunteers who simply enjoy creating new and innovative source code.

The differences in backgrounds combined with the mechanisms, create both motive and opportunity to modify source code. Therefore, "one would expect many branches ('forks') and variants in open source software to emerge. Such diversity more likely leads to incompatibility and fragmentation" (Egyedi & Van Wendel de Joode 2004, p. 2). This problem is also recognized by Kogut and Metiu (2001, p. 257) who write, "the danger of open-source development is the potential for fragmenting the design into competing versions." Too much divergence results in many different versions of software programs, modules and interfaces, and is likely to result in incompatibility. The software becomes too complex and too much time is needed to constantly adapt it to new and changing interfaces and to new versions of the software programs.

To prevent incompatibility and fragmentation and avoid an extreme waste of resources, a certain level of convergence is needed (see also Egyedi & Van Wendel de Joode 2003). One would expect a need for choices to determine which source code becomes the standard and which development trajectory is abandoned. Participants might be expected to sit together and mutually agree on these questions. Consider the following statement by an ASF board member:

> The projects can split into different branches, one that is more conservative and one that is more risky and creative. For example, we will give some project six months and after that period we will get together and try to make them work together and get a stable version out of it again, this we do by taking the good things out the creative version and scrapping out the bad things.

The most interesting aspect of this statement is the use of the word 'we': "*we* will give" and "*we* do." Apparently, the respondent feels that there is a body or collective that can restrict the behavior of individuals in the community and that is able to force some level of convergence. This statement triggers many questions: Who is *we*? How are individuals affected by decisions made by *we*? How do *we* enforce decisions?

Limited influence of leadership and voting systems

Voting systems are present in many open source communities. The Apache, Debian and PostgreSQL communities have all created and use voting systems. O'Reilly even claims that voting systems are an essential element of an open source community (O'Reilly 1999). The next pages present the voting systems in the three communities. The main conclusion is that the systems are hardly used and when they are used, they barely restrict the participants in their future choices and actions.

Many of the communities analyzed in this research have a project leader. This discussion here on project leaders centers on the Debian and Linux communities, and demonstrates that in both communities the participants are only bounded or restricted to a limited extent by the choices made by project leaders. Participants are free to make their own informed decisions and act as they feel is most appropriate.

The voting system in Debian

According to a Debian website the Debian voting system is based on the Condorcet Voting System.[1] Because the Condorcet system is fairly complex, an example is first given to explain how it works. The use of the system is discussed thereafter.

Consider an example in which four people can vote on five alternatives. They all indicate their preference by ranking the alternatives. Let's assume that these four people give ranks as shown in table 6.1. One possible voting scheme could be to add the individual ranks of each alternative. The alternative with the highest score is then chosen. In this example alternative W is chosen, as it has the highest overall score. However, alternative V is the choice preferred by the majority of the people. Only person 4 preferred W over V. The alternative preferred by the majority is called the *Condorcet alternative* (Schwartz 1986). To ensure that the majority-preferred alternative is chosen, alternative V in the example, the French mathematician Condorcet developed a pair-wise method in which people vote between pairs of alternatives. In the Condorcet voting system alternative V would turn out as the best alternative.

Table 6.1 – Ranking the alternatives (adopted from Schwartz 1986)

	Person 1	**Person 2**	**Person 3**	**Person 4**	**Total**
V	4	4	4	0	12
W	3	3	3	4	13
X	2	2	2	3	9
Y	1	1	1	2	5
Z	0	0	0	1	1

Most striking about the voting system is not the way it works, but rather the way it is used by Debian. On first glance the voting system seems to be used in situations that concern the collective and thus require a collective choice. According to the Debian website, the voting system is used in three situations: (i) to elect a new project leader, (ii) to select logos for the license and the community as a whole and (iii) to make a constitutional change.[2] Observations, however, lead to the conclusion that the voting system has quite a limited role in the Debian community. This is evidenced by the fact that it is hardly used.

Since 1999 the voting system has been used ten times. Five of those times were to elect a new project leader. Obviously, many more choices were made in that period that affected the collective of individuals. However, for these decisions a different arrangement must have been used to come to a decision. Furthermore, the voting system is not used in the software development and maintenance processes in the Debian community. Issues that arise on these topics must be solved in a different fashion. Therefore, the voting system appears to hardly restrict or bind the participants in the community. In the many situations that are not addressed by the voting system, the participants are not affected by the system.

The voting system in Apache

The Apache community uses a simpler method than that of Debian. In Apache, the voting system is used every time a new patch of code is committed to the source code. One of the first Apache developers and current member of the ASF Board of Directors had the following to say about the voting system: "A vote is held at every commit. The vote usually does not amount to anything much. The patch will stay in if no one votes negatively."[3] The contributors participating in the voting procedure have three options, namely, +1 if they agree that the code should be included, 0 if they want to abstain and –1 if they disagree. Patches remain included only if at least *three* people vote +1 and *no one* votes –1. Otherwise the patch is removed from the source code.[4] If you as a participant give a +1 to a patch then you automatically commit yourself to the patch. You become partly responsible for it. If, after a while, the patch turns out to be faulty then you have the responsibility to remove it and clean up the source code.[5]

The inclusion or exclusion of patches of software is a choice that is important to every individual in the community. In that sense the voting system could be said to be a mechanism to make important decisions that affect the individual. Still, this chapter argues that the voting system does not result in a collective choice. First, individuals can ignore the vote. If, for instance, a participant discovers that the source code does not work or has many bugs, then they are free to remove the source code even if the outcome of the vote was to include it. Furthermore, participants are allowed to create a second head if they disagree with the vote.[6] Obviously, the creation of second head is a way to evade the outcome of the vote, just like the removal of the source code. Thus, the vote can be ignored.

Second, the voting system does not intend to bind the individuals in the community. Rather, it serves as a first check or control to ensure that the source code has a certain level of quality. [7]

Third, the voting system resolves only a minor selection of the choices that need to be made in the community. The voting system is, for instance, not used to select between two heads that could be created. Neither is it used to decide on issues that concern more than one module. The voting system is used only to decide whether source code should be included and even then participants are not restricted by or bound to the outcome of the vote.

Voting in the PostgreSQL community

Another example of a community that uses a voting system is the PostgreSQL community. The project leader of PostgreSQL explained how the voting system is used: "We just vote and that will force a decision. Everybody on the mailing list can vote. When you ask for a vote to decide on a matter, usually a few are going to vote because it is their concern." According to this respondent, the voting system is used to reach a collective choice. A little later in the interview he explained what happens with the outcome of the vote: "It gets transferred to the 'to-do' list, so we know what has been decided on."

The question, however, is whether the outcome of the vote really affects the individuals in the community. One steering committee member thought not. "That list contains… features that at least some percentage of developers wants to have. Not everyone agrees on it… people are still free to implement whatever they think is best." In other words, irrespective of the outcome of the vote, individuals can still decide for themselves what they want to work on and how. Again, the voting system has not lead to a collective choice.

Leadership in the Debian community

The Debian community has a project leader. According to the Debian constitution[8] the project leader has a number of privileges that other participants in the community do not have. The project leader may, for instance, lead discussions among developers or make any decision that requires urgent action. One respondent described the position of project leader as follows: "There is a lot of respect for this person. He has proven that he is very good."[9] Next to the project leader there are other bodies with special privileges: (i) the technical committee, (ii) delegates appointed by the project leader and (iii) the project secretary. Finally, the Debian community has many package maintainers. Although, they are not mentioned in the constitution, they too have a substantial role in the community. Each of the maintainers is responsible for importing 'their' open source software packages into the Debian distribution. They are accountable for their packages and have the power to make decisions concerning them.[10]

The Debian constitution identifies the tasks, activities and privileges of the project leader.[11] Irrespective of how the role of the project leader is formally described in the Debian constitution, many of the interviewees, including former Debian project leaders, view the project leaders' role in decision making very limited. Consider an often-cited statement by former project leader Bruce Perens: "Trying to lead Debian is like trying to herd cats."[12] Another former Debian project leader makes a claim along the same lines: "You can try to steer, but you cannot force it. You can try and make people look in another direction." In both statements, the project leaders describe a perceived inability to restrict the actions of participants. According to them, their decisions as project leader do not bind the individual participants. One respondent explained, "Debian is a bazaar of 500 mini-leaders. In the end they are the ones who have responsibility for their projects."[13]

These statements highlight the notion that participants are not restricted by the decisions of the project leader and therefore the project leader in the community is not a substitute for a collective choice arena.

Leadership in the Linux community

In the Linux community the leadership role is fulfilled by Linus Torvalds. "He rapidly moved to writing less code and coordinating the software development project, assessing contributions and arbitrating disputes" (Lerner & Tirole 2002b, p. 15). Much research has acknowledged the leadership role of Torvalds in the Linux community. It claims that the development process is centralized and that Torvalds is the one with decision authority (Kogut & Metiu 2001, McGowan 2001, McKelvey 2001a, Moon & Sproull 2000). Torvalds, however, does not manage the entire community alone. So-called 'lieutenants' have authority over subsystems of the Linux kernel. Torvalds does have a number of sources of power. To start with, he is the trademark owner of the name Linux.[14] He also owns the collective copyright of the Linux kernel.[15] Finally, he maintains the most popular and most used version of the Linux kernel, referred to as the 'standard version.'

Yet, like the project leader, even decisions made by Torvalds himself do not bind the participants in the communities. Wayner writes that Torvalds' greatest trick was his decision to avoid the mantle of power. Torvalds wrote in 1992, "Here's my standing on 'keeping control,' in 2 words…: I won't" (Wayner 2000, p. 62). If Linus Torvalds really were to exercise his potential power and behave as a leader, one might question whether he could keep the developers in the Linux community together. Especially if we remember that many developers are professionals with a lot of skills, knowledge and a strong, outspoken opinion as to how things should be done.[16] Furthermore, observing the Linux kernel mailing list it becomes evident that in many instances Torvalds' preferences and ideas are not adopted by other developers in the community. Often Torvalds adopts the decision that is preferred by many developers in the community but which is not his own preference (e.g. Kuwabara 2000). A developer in the Linux community agrees that Torvalds cannot be understood as a dictator who enforces his decisions on the individuals in the community. "But that means there is no boss, who tells us which way to go. It is completely anarchistic."

One of the better analyses of leadership is that done by McCormick (2003) in two open source communities. He conducted many interviews with members to understand the role of project leaders and ascertain whether they have more influence and power than other individuals in the community. The conclusion is that they do have more power. "These groups are not quite non-hierarchical… in both projects there are several 'non elected' leadership roles with varying degrees of control over the project" (McCormick 2003, p. 23). The question is 'What can they do with this control?' To which the answer is 'Nothing much.' "Unless you are paying someone, you can't do much of anything to officially delegate tasks. All you can do is cheerleading, or simply debating some ideas until the other person gets excited about it" (McCormick 2003, p. 11). If the project leader tries to impose too much control and exercise power, this "would be resisted by the group" (McCormick 2003, p. 27).

Although McCormick did not analyze the Linux community, his findings are comparable with most of the statements made by the respondents interviewed for this research. They are also in line with the processes observed on the Linux mailing lists, namely that participants are not bound in their actions by the decisions of the project leader.

Choices are made on the individual level

There are many situations that are potentially in need of collective choice arrangements. Yet we are left with an impression of communities in which decisions are made on the individual level. The developer makes an individual choice. This observation is partly based on the previous section, which argued that voting systems and leadership bound and restrict the choices and actions of individuals only to a limited extent. This observation is further strengthened by findings in earlier chapters. Remember how open source licenses are chosen? The discussion on boundaries argued that a single individual or company at the beginning of a project decides what license to adopt. Most communities keep this same license throughout the life of the project.

Chapter five identified and discussed the presence of a great number of coordination mechanisms. It argued that the creation and use of the mechanisms could be understood from individual choices. "I don't even remember who created the website. It was not planned. Someone wrote an e-mail with the message, 'I have a password there.'" Or consider the statement of a developer as to when and how the community started to use a CVS: "Then somebody came and said if you are developing open source software, you could get a CVS account somewhere." Both statements indicate an absence of collective choice and the presence of individuals who simply undertake action: people who *do*.[17]

Architectural changes that affect multiple modules are also largely based on individual choices and actions. A developer in the Linux community described an example in which he made an architectural change to the drivers that became the standard communication layer for Linux drivers. He felt that the architecture was a mess and wrote a layer "nested between the system and the hardware in a standard way. This layer was reasonably well adopted and it has now become an important part of Linux."

Finally, convergence is achieved through individual choice. Software development depends on the efforts invested by individuals, and software development and maintenance is governed by individual choice. This is true for the way in which convergence is achieved as well. The general principle is always that "decisions are made 'by those who are willing to do the work' and that in the long run, developers tend to support the best technical solution for a problem" (McCormick 2003, p.31). Individual developers "choose and select among" (McKelvey 2001b, p. 26) the open source programs available.

Positive feedback

In 1990 *Scientific American* published an article by W.B. Arthur (1990) about increasing returns and positive feedback. The argument laid out in the article is that in certain markets and under certain conditions self-reinforcing mechanisms ensure that the company that gains a head start is most likely to become the market leader.

In open source communities positive feedback is also present. Positive feedback is based on the idea that popular communities will become more popular and unpopular communities will become even less popular. Consider the statement, "programmers will want to work on software projects that attract a large number of other programmers" (Lerner & Tirole 2002b). The fact that a project with many developers or programmers attracts more developers means that popular communities will become even more popular. This increasing popularity is supported by the growing number of users that the community will attract. "It is like a chain reaction, popularity leads to more users, more users leads to more developers, to more applications and thus to more users."[18]

The fact that positive feedback occurs is highly relevant, because it results in an aggregation of choices made by individual developers. These aggregated choices result in a selection of communities, of licenses, of software development lines and of coordination mechanisms on a collective level. Thus, choices in open source communities are not made collectively but rather individually. These individual choices are then aggregated into a dominant preference or choice on the collective level.

Aggregation of individual choices: the role of tags

Aggregation in the communities is not coincidental, but is to some degree institutionalized through the presence of a number of self-reinforcing mechanisms. These mechanisms ensure that certain communities and certain software development lines become more dominant than others, which leads to a choice on the collective level. But most of the mechanisms do more than just result in aggregation; they also ensure that with a minimal amount of effort other developers can make choices about which software they will download and use. Consider a statement by a developer in the Debian community: "You prefer to use applications that are reliable and where you know what it does without having to 'check under the hood.'" In open source communities a number of mechanisms are present that enable developers to form an opinion about the quality of software with relatively little time and effort.

Self-organization literature refers to these mechanisms as *tags*. Axelrod and Cohen (1999) define a tag as "an initially arbitrary property of an agent (say, a number between 0 and 1) that is detectable by other agents and that can be copied by other agents. Examples of tags might include accents and styles of clothing" (p. 95). Holland (1995) further explains that tags are not necessarily connected to individuals; they are artifacts. He uses the example of flags and banners, which are used to "rally members of an army or people of similar political persuasion" (p. 13). The fact that other individuals are triggered by these tags and will largely base decisions on these tags leads to an aggregation of individual choice (Holland 1995) and thus to selection on the collective level.

The next sections identify five tags and demonstrate how these tags (i) lead to aggregation of individual choices, (ii) diminish the time and effort needed to search for and analyze the source code of software and (iii) stimulate the selection of software that is of relatively high quality. These tags exist alongside one another, and it is difficult if not impossible to judge which of the mechanisms is more dominant and has more effect on individuals' choices. Neither is there a logical sequence that determines which of the tags plays a bigger role.

Next to these tags are more tags in open source communities. One example is the logo of a community. "In other cases, people don't know which to pick and they just close their eyes and join the one with the cutest logo" (Wayner 2000, p. 207). Other tags are, for instance, the license, the trademark and the design of the website of the community. What they have in common is that they influence individuals' choices. These, however, do not result in the collective selection of software that is of high quality. The discussion here focuses on tags that influence individual choice and are likely to result in a selection of software that is of a relatively high quality.

Elegance of source code

The first tag is the elegance of the source code. Participants deciding which source code to download are partly guided by the principle of elegance (Van Wendel de Joode et al 2003).[19] The concept of elegance has been explained in some level of detail in chapter five. It was explained that elegance is a term used to indicate that the software works. At the same time elegance is a notion of beauty.

Elegance is a 'tag' of source code and it has a signaling function. Respondents have argued that experienced programmers can usually determine whether source code is elegant. "If you narrow your circle to a small group of good developers then it is easy to decide what software is elegant and what is not."[20] Furthermore, experienced developers partly base their choice of software on the level of elegance of the source code, because it is a measure of the quality of the source code. "Compare it to tightening a screw with a pipe wrench, instead of a screwdriver. Obviously that would be totally wrong."[21]

The reputation of developers

The second tag is the reputation of the developers involved in an open source project. Reputation is, in fact, one of the most cited motives for individuals to become actively involved in the development of open source software (e.g. Benkler 2002a, Von Hippel 2001, Lakhani & Von Hippel 2003, Markus et al 2000, McGowan 2001). Based on the quality and the quantity of the code that developers contribute, their reputation as a software programmer is either bettered or worsened. This motive has a number of consequences for participants' expected behavior. One consequence is that participants tend to prefer communities in which they have a large audience, since as they gain credits they will improve their reputation among more people (e.g. Lerner & Tirole 2002b).

Writing good source code improves a programmer's reputation. Therefore, participants with a good reputation are expected to be better software programmers than programmers who are relatively unknown and those who have a poor reputation. Consider, for instance, a statement about the quality of participants: "Reputation within a well-informed and self-critical community becomes the most efficient proxy measure for …quality" (Weber 2004, p. 142). Thus, reputation is a way to signal competence and trust in the skills and knowledge of a participant in the development of software (for a more elaborate discussion on the interplay between trust and reputation see Nooteboom 2002).

The first effect of reputation is that participants with a good reputation, like for instance, the project leaders in large communities, are generally better listened to than others. If a participant has little knowledge about a certain issue he or she will be especially inclined to accept or agree with a decision made by a participant with a good reputation. "So then there is someone who has more knowledge about these things. You will automatically respect that person. You see these names everywhere and therefore those people are likely to be right when they say something."[22]

The second effect is that people are attracted to communities in which many participants are present who have a relatively high reputation (see also West & O'Mahony 2005). Good reputation may be a "primary reward in itself, but it is also a way to attract attention from others" (Ljungberg 2000, p. 212). A former project leader of Debian claims that he partly bases his decision of whether to get involved in a community on the presence of participants with a good reputation. "Reputation does help. If you see a good name, then you will take a look."

To summarize, reputation of participants is an indication of ability to write source code or perform other types of activities. Participants who have a good reputation are likely highly skilled and have contributed much to one or more communities. Furthermore, other developers are attracted to developers with a good reputation. Developers tend to accept their decisions more easily; they are more likely to join communities that harbor highly reputable developers.

The level of activity in a project

The third tag is the level of activity in a project. It seems logical that most developers and users would want to download and install the best-engineered product or module. They would do so to be sure that the product works and is reliable. A fascinating fact, however, is that many developers do not base their choice of software on the actual source code, but primarily on other indicators. One measure often referred to is the level of maturity or activity of a project.[23] "Sourceforge and Freshmeat indicate a maturity level for each project, so you know that the software listed on there is stable and works."[24] This level of 'maturity' is a measure that combines the age of the project (i.e. the community or development group) with the number of releases. Neither are direct indicators of the quality of the source code. But the argument is that if the community has been able to make many releases and has continuously attracted new developers, then the software is likely to be good.

> Thus the *number of developers* involved in a project could be an indicator of success. The number of developers can be measured in at least two ways. OSS [open source software] development systems such as SourceForge list developers who are formally associated with each project. Examination of the mailing lists and other fora associated with projects can reveal the number of individuals who actively participate (Crowston et al 2003a, p. 6).

Besides maturity the websites also list statistics about the level of activity. Freshmeat, for instance, contains a so-called 'popularity index'[25] and SourceForge, which hosts a large number of open source projects,[26] uses a variety of statistics to measure activity in projects (e.g. the number of downloads). The level of activity is a frequently mentioned indicator of quality and supports decision making by individual participants. One developer, for instance, said that he bases his decisions on "how actively the community works on the software, improves it and implements it."[27] Another respondent explained, "SourceForge has information on activity as well. If there are 20 different versions of a library for a certain purpose, and one has been downloaded 10,000 times and another one 20 times, it is clear which you choose first."

The fact that participants decide to get involved in communities that are active does stimulate positive feedback. The more active the community is, the more likely it is that participants will download and use the software, which further increases the level of activity. Also, the level of activity and maturity provide an efficient and quick reference to estimate the quality of the software. The higher the activity and maturity level, the greater the chance that the software is of high quality.

The role of distributions

The fourth tag is inclusion in a distribution. There are many open source communities, and an unprecedented variety of software is being developed in them. This means it can be quite difficult for users and developers to choose among applications that have similar functionality. Consider, for instance, the Linux.org website. It lists an enormous number of open source applications that are available via the Internet. According to the website, there are 116 MP3 applications and 78 emulators.[28] Obviously, it is not an easy task for a developer or user to decide what applications to download and install. Indeed, in most cases this is not what is actually done. "There are only a few users who choose between programs… Usually the decisions are made by the distributions."[29]

A distribution is a collection of applications bundled together to create a sensible whole. Popular distributions are, for instance, Red Hat, SuSE and Debian. That last is the only one of the three that is actually created in an open source community; the other two are created commercially.

For a community it is quite important that their software be included in a distribution. "It means a lot to us to be in the distributions, because we get more exposure."[30] Exposure is achieved because many developers and users simply install the applications chosen by the manufacturer of the distribution. Thus, in most instances it is not developers or users who choose among, for instance, the 116 MP3 players. The manufacturer of the distribution makes the choice for them. Inclusion in a distribution is considered "an independently important success measure" (Crowston et al 2003a) of the software developed and maintained in an open source community.

Distributions stimulate positive feedback, because the most widely used applications are the ones usually chosen. Users and developers use the applications selected by the distribution and thus make popular applications even more popular.

Sites with directory services

The fifth and final tag is a mention of the open source software on a website with a directory service. Similar to the distributions, an important part in the process of selecting software takes place "when more or less official websites – and code depositories – choose which to make available" (McKelvey 2001b, p. 26). An effective means to aggregate individual choices for a development project is to get listed on, for instance, Tucows or Freshmeat, both of which include a directory of open source software applications (Nakakoji et al 2002). The creator of BlueFish described what happens when he sends an announcement of a new release to one of these sites:

> Usually, we have 600 to 800 hits per day, but after an announcement this number tends to increase. The craziest that I have seen so far has been in the order of 30,000 hits per day. That is quite a lot. Usually the number of hits is pretty consistent and then after a new release wham the number rises again.

Another strategy to get more exposure is to locate the activity of a community on a certain website or as part of a bigger project. The GNU project, for example, serves as an umbrella for a great number of projects. One of these is Lilypond. The creator of Lilypond explained why they decided to affiliate with GNU: "At that time our primary concern was to get exposure; it is better if more people know who you are."

One problem with this tag is that sites might list and promote poor quality software, with source code that is inelegantly written or which is not modular. One way to prevent this is through a system of rating the software. Tucows, for instance, rates the software that it lists. The scale ranges from one cow for software of low quality to five cows for software that is high in quality. Typically, software with a low rating is downloaded less than software that is highly rated and of a higher quality.

The sites are much used by open source participants and users, as they provide a quick overview of the applications available and they immediately gain an impression of whether the software is any good. Furthermore, the sites stimulate positive feedback, as they point to communities which have a relatively high level of activity.

Beyond collective choice and individual choice

Individuals in the communities are led by their individual choices. This does not mean that these individual choices are random; they do result in a selection. The individual choices are guided and influenced by the tags. These tags cause an aggregation of individual choices and result in a selection on the collective level. How this is done is explained below.

Graphical representation of the aggregation of individual choices

The five tags aggregate the individual choices of developers and make popular communities even more popular. From this emerges a selection on the collective level. Consider figure 6.1. In the figure small circles represent the participants.[31] The small *dark* circles represent developers with a high reputation. The bigger circles are clusters of participants. The clustering of participants occurs when a software module, an operating system or an application becomes popular. As more developers cluster around the software the level of activity of the community is likely to increase and the software might be included in a distribution. Websites might even include the software in a list. Hence, even more individuals and companies are attracted to the community. They download the software and a percentage start to participate in development and maintenance of the source code. If we assume that the clusters of participants represent communities, then the gray colored circle represents a popular community with a high level of activity.

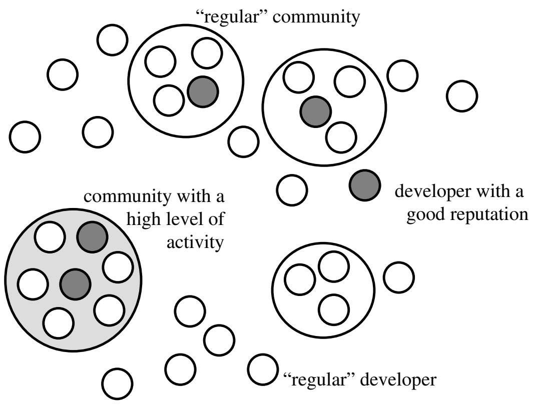
"regular" community
community with a high level of activity
developer with a good reputation
"regular" developer

Figure 6.1 – Clusters of developers

In the figure one developer with a good reputation (the small dark circle in the right of the figure) is not currently a member of one of the communities. Let us assume that this developer has created a new application. On the strength of the developer's good reputation, he or she will attract other developers to take a look at the source code. This is graphically represented in figure 6.2.

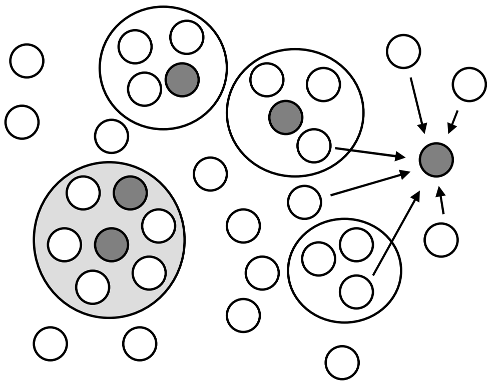

Figure 6.2 – A developer who receives a lot of attention

Now, there are roughly two scenarios, which again are quite simplified. The first is that the source code is inelegant. Experienced developers detect this, because they can read the source code and judge whether it is elegant. They agree that the source code is inelegant and if they have no pressing need for the application or if there are other, more elegant alternatives available, then chances are they will not install and use the source code. If every experienced developer makes the same choice the level of activity surrounding the new application is likely to remain low. Furthermore, creators of distributions are unlikely to choose the application, because the level of activity is low and the source code is inelegant. The only reason why the new application would still attract some developers and users is due to the high reputation of the developer and perhaps due to other tags. The developer might, for instance, have created a cute logo. Still, these tags are unlikely to compensate for the lack of elegance and activity and for the fact that the application is not included in any of the distributions.

The other scenario is that the source code is elegant. In that case there is a much greater chance that other developers will like the software. They might start to use it and establish a community by commenting on the source code and adding to it. In this scenario the community gains a higher level of activity, which attracts even more developers. The software might catch the attention of a distribution like Red Hat, SuSE or Debian. It might be included in one or more of these distributions and be listed on websites. This creates an even bigger amount of activity. At the same time, other communities are likely to lose some of their participants to this new community. Their level of activity will slow down. As such, the new software and the new community could contribute to the death of a previously popular community. Figure 6.3 illustrates these processes. The new community is colored gray to indicate that it has many members and is thus attractive to new developers. The circle with the dotted line indicates the near death of another community.

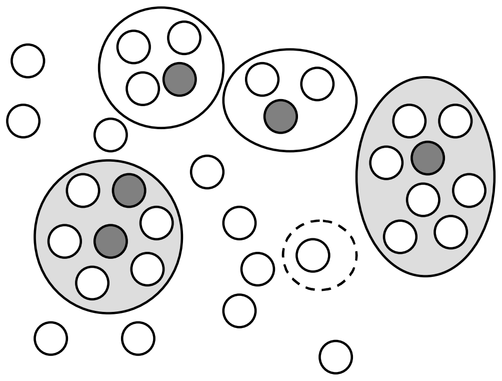

Figure 6.3 – The growth and decline of communities

Many developers in the community are triggered by the mechanisms called 'tags' that were described in the previous section. Consider, for instance, users that have no experience in software development. How should they choose which software to download? In many cases they use the applications that are included in a distribution, that have a high level of activity, that include people whose names they have heard and that are listed on a website like Tucows. Not only users display this behavior. Developers do too. They simply do not want to invest time in choosing what application to download.[32] In many cases it is much easier for both developers and users to make a choice based on the choice of others. It is easy to understand how this behavior, led by the five mechanisms, can lead to a situation in which one or two competing applications become the most popular among a wide variety of applications. Other, less popular applications might continue to receive attention from developers, but they receive less attention than the popular applications.

Choices in open source communities: the concept of swarming

The selection process described in open source communities resembles a process known as *swarming*. Swarming is governed and guided by individual actions, the collective pattern of which is important (Kelly 1994). The observation that is central to the swarm model is adopted from research on ants and their colonies:

> Without centralized control, workers are able to work together and collectively tackle tasks far beyond the abilities of any one of the individuals. The resulting patterns produced by a colony are not explicitly coded at the individual level, but rather they emerge from myriads of simple nonlinear interactions between individuals or between individuals and their environment (Theraulaz et al 2003, p. 1265).

The point is that individual ants have no picture of the colony. A similar thing is true of birds that are blind to the beauty and efficiency of a flock in flight (Kelly 1994). Like the selection process described here for choices among open source applications, a swarm is the result of an aggregation of individual choices. Aggregation is ensured through mechanisms that stimulate positive feedback (Bonabeau et al 1999).

In a sense the individual and corporate participants in open source communities form swarms. They act based on simple individual behavioral rules and copy the behavior of other participants through tags, like level of activity and inclusion in a distribution. As a swarm they select a limited number of modules or software programs. These swarms are not static. On the contrary, they are highly dynamic and change in response to even small fluctuations in the environment. In swarms, participants select and decide which software program becomes most popular among a wide variety of available programs. Similarly, they select which open source license becomes popular and which coordination mechanisms are adopted. Other options might continue to attract small groups of participants but might also die.

Project leaders and voting systems revisited

Project leaders play an important role in the aggregation of individual choices. Their decisions can also be thought of as tags. They influence the choices of others, because they are believed to have oversight of the project and perhaps are the most suited to make certain decisions. It would be interesting to understand what happens when a participant with a high reputation disagrees with a choice made by the project leader. Whose choice will the participants copy?

This chapter argues that the outcomes of a vote or the choice of a project leader is not binding for participants in the communities. They make their own decision, though obviously they include the outcome of the vote in that decision-making process. But they can also decide to do something else. They are free to do what they believe is best.

Much literature conveys a different image of the role of project leaders in general and of Linus Torvalds in particular. It tends to portray them as hierarchical leaders who actually steer software development and enforce decisions on other developers (e.g. Bezroukov 1999, Bonaccorsi & Rossi 2003c, Lerner & Tirole 2002b, McGowan 2001, Selz 1999). Furthermore, many participants in the community seem to support this image. One example of this is the term that some communities use to refer to the project leader, namely 'benevolent dictator.'[33] Why do developers insist that project leaders have such an important role and why do so many people refer to communities as being hierarchical? Why do they create and protect the myth that communities are organized hierarchically and that project leaders are so important? There are two reasons why this myth might be functional and perhaps even vital.

The first reason is that the presence of a benevolent dictator is a comforting thought. Many developers in, for instance, the Linux community are volunteers and they spend much time and effort in the development and maintenance of the Linux kernel. Would they be willing to spend so much time creating software if they understood that sometimes it is pure luck or coincidence that their source code finds its way into the standard Linux kernel? Much more comforting is belief in the sagacity of one person, in this case Linus Torvalds, who is responsible for the entire project and makes the right choices about which source code to include. One developer in the Linux community, for instance, argued that Torvalds is an accepted leader, simply because he always makes the right decisions. Not only is this impossible, it is also an excellent example of the trust of developers in project leader Torvalds and the comfort they get from believing that he will make the right and objective decisions.

The second reason is that companies need someone they can speak to and whom they think they can hold accountable if anything goes wrong.[34] Companies must explain to their shareholders and other stakeholders why they are investing in open source software. Would their stakeholders accept a million-dollar or even billion-dollar investment in, for example, Linux[35] if its development and maintenance processes could not be explained? Obviously, companies want to strengthen the myth that the development and maintenance of open source software has nothing to do with luck. Therefore they strengthen the myth of a benevolent dictator. This idea is further fortified by the straightforwardness and ease of communicating the notion of a benevolent dictator to others.

Like the project leader, the use of voting systems is paradoxical. No evidence supports the idea that the voting systems play an important role in reaching a collective choice. Yet most communities do have some kind of voting system, which they have ritually decorated with all kinds of rules and procedures. The voting systems probably have a role similar to that of the project leaders; they serve as a tag and are used to create and uphold the idea that decision making in the communities is structured and organized.

A note on consensus: its role in decision making

Many respondents interviewed for this research argued that consensus is important in decision-making processes in the communities. Consider this statement by the vice-president of the FSF: "One person cannot do everything. Consensus is needed." A similar view was expressed by a member of the steering community of the PostgreSQL community: "The technical discussions are held on the mailing list and we usually come to consensus." Yet what exactly is consensus and how is it reached? The answer put forward here is different from the ones that dominate much of the discourse on the role of consensus in decision making in open source communities.

From the interviews it became clear that respondents had very different ideas about what is meant by 'consensus.' One respondent said it "means that everyone has to agree [which] differs from a majority vote."[36] An ASF board member however argued, "[We] would ask the members for their input and based on that we would come to a decision." This idea of consensus is quite different from the former definition. At the same time, people tend to talk about consensus as if it is a commonly understood notion. "Decision rights are primarily invested in individuals and most decisions are reached by consensus" (Sharma et al 2002, p. 10). Sharma and colleagues do not take the trouble to explain what they mean by consensus. They treat it like it is common knowledge and universally understood.

According to the literature, consensus means something like "any reasonable and accepted means of aggregating individual interests" (Arrow 1974, p. 69). Consensus is thus not a term that says anything about the actual content of the decision. Instead, it is a characteristic of the decision-making process. Consensus means that everyone agrees with the decision-making process. Furthermore, consensus has a very positive connotation; it implies agreement and satisfaction. But it is unlikely that all decisions in communities full of 'stubborn' programmers and voluntary hobbyists are made by consensus.

One participant in the Linux community explained how consensus is reached concerning the question of whether source code is elegant: "If you narrow your circle down to good developers, then you will find much mutual agreement about which software is elegant and which is not."[37] According to this respondent, consensus is easily reached within a small group of good developers. At the same time, he dismisses people who disagree as unknowledgeable. This has two effects. First, developers are less inclined to disagree with a decision reached by the developers who are generally accepted as 'good developers.' Second, developers who do disagree are no longer among the group of good developers. In other words, the group of knowledgeable developers will always reach consensus, simply because people who do not agree are either not knowledgeable or simply do not make their counterarguments heard.

Conclusion

A collective choice is a particular type of choice, namely one that binds and restricts individuals in the social group. The most commonly described mechanism to reach a collective choice is through voting systems. Many open source communities have voting systems. However, even in the communities that have adopted such systems, the actual influence and binding character of the votes is limited. The binding character of a choice made by a project leader is similarly limited; individuals remain free to make their own informed decisions. Individuals are also free to remove source code once it is voted in. When the outcome of a vote is that a piece of source code should not be included, individuals are free to create a new development line to which the source code is added.

Yet individuals do not make decisions randomly. There appears to be some form of intelligence at work. The reason for assuming intelligence in decision making is that software developed in bigger open source communities has a relatively high overall quality and market share (e.g. Bauer & Pizka 2003, Mockus et al 2002). Somehow participants are able to select software they perceive to be of relatively high quality. The basis of this intelligence was argued to lie in five mechanisms that influence participants in their choices. These mechanisms or 'tags' are (i) level of activity in a community, (ii) elegance of the software, (iii) reputation of the participants involved in a certain project or community, (iv) inclusion of the software in a distribution and (v) listing of the software on a website with a directory service.

Individual behavioral rules

To understand the observations in this chapter we need only two individual behavioral rules, which underlie the choices and actions of participants in the communities. Combined with the observation that the five mechanisms or tags are proxies for the quality of the software, they support the observation that selection processes in open source communities result in a selection of software that is of relatively high quality.

"*Participants select software that meets their specific user needs.*" Participants' primary selection criteria must be based on their own specific needs. They are related to, for instance, the perceived quality of the software, the expected continuity of the software and its functionality.[38]

"*Participants spend a limited amount of time analyzing software and base their choice of software on tags.*" Time is the scarce resource. Therefore, participants want to choose software for which they "know what it does without having to 'check under the hood.'"[39] To reduce the time needed to choose software, individuals base their choices on tags.

How the individual behavioral rules support the observations

Individual choices are not random. This observation is primarily related to the second rule, namely, participants try to limit the time they spend analyzing alternatives. They therefore base their choices on tags like level of activity or reputation of the developers involved. These two tags, and the other three, stimulate participants to mimic the behavior of others and at the same time increase the chance that high-quality software is selected.

First, the mechanisms stimulate mimicking. The level of activity is rather straightforward in this respect. A community with a high level of activity attracts more participants, which further increases the level of activity. Also the reputation of the developers involved results in mimicking, because a community with participants who have a good reputation will attract other participants.

Second, the software programs, licenses, or whatever else is selected, are likely to be the options that are comparatively higher in quality. There are a number of reasons why this is so. Elegance of software is an indicator of high quality. A high level of activity in a community among other things means that many bugs are resolved and thus the quality of software is relatively high. The developers' reputations is a proxy for the ability of participants to write qualitatively high software; and the fact that software is included in a distribution means that more programmers check and improve the software, which is especially true for distributions like Red Hat and SuSE, for which programmers are hired to improve the software.

Notes on chapter seven

[1] http://www.mathematik.uni-kl.de/~wwwstoch/voss/comp/vote.html (March 2003).
[2] See http://www.debian.org/vote/howto_follow (December 2003).
[3] The commit refers to the act of a developer to include a piece of source code (patch) into the existing source code.
[4] http://httpd.apache.org/dev/voting.html (March 2003).
[5] Based on an interview with an ASF board member.
[6] This point was addressed in the previous chapter. It argued that the second head is one of the forms in which parallel development lines are institutionalized. The parallel development lines are a way to defer the negative effects of conflicts.
[7] From an interview with an ASF board member.
[8] www.debian.org/devel/constitution.
[9] From an interview with a package maintainer in the Debian community.
[10] From an interview with a package maintainer in the Debian community.
[11] The status of this constitution, however, is not at all clear.
[12] From interviews with Bruce Perens and other members of the Debian community.
[13] From an interview with a member of the Debian community.
[14] http://linux.omnipotent.net/article.php?article_id=6475 (July 2003).
[15] http://news.com.com/2100-1016_3-1018057.html (July 2003).
[16] From an interview with two members of the Debian community.
[17] According to a respondent from IBM, *doing* is the only way to get things done in open source communities.

[18] From an interview with the creator of BlueFish.
[19] See chapter five for a more elaborate discussion on the notion of elegance.
[20] From an interview with a Linux developer.
[21] From an interview with a developer from the Apache community.
[22] From an interview with the head of Dutch translation in KDE.
[23] A project can, for instance, be a community, but could also be the group of developers that creates and maintains one specific module.
[24] From an interview with the coordinator of one of the New Hamsphire Linux user groups.
[25] The index is based on the number of subscriptions, URL hits and record hits. (software.freshmeat.net/stats/p.popularity, May 21, 2003).
[26] sourceforge.net/top/ (May 21, 2003).
[27] From an interview with a developer from the Debian community.
[28] http://www.linux.org/apps/all.html (October, 2003).
[29] From an interview with a Linux developer.
[30] From an interview with one of the project leaders of Lilypond.
[31] The figure is an extreme simplification of the real situation. One simplification is, for instance, that a participant can only be a member of one community at a time. Nonetheless, it does explain how participants select between enormous amounts of variation.
[32] This point was also made in an interview with a developer from the Linux community.
[33] From interviews with developers from the Linux, the PostgreSQL and the Python community. See also: http://lists.debian.org/debian-devel/1998/debian-devel-199810/msg00193.html (October 2003) or http://en2.wikipedia.org/wiki/Linus_Torvalds (October 2003).
[34] See chapter ten for a more elaborate discussion of the role of foundations and project leaders as a point of contact for organizations.
[35] In 2000, IBM announced that it would invest one billion dollars in the development and marketing of Linux. http://www.crn.com/Sections/BreakingNews/dailyarchives.asp?ArticleID=22304 (October 2003).
[36] From an interview with the head of marketing in the OpenOffice.org community.
[37] From an interview with a Linux kernel developer.
[38] Consider this statement by a respondent. "You judge: the maintainability of the code, the level of activity of the community and whether or not the code is understandable."
[39] From an interview with a package maintainer in Debian.

CHAPTER EIGHT

MONITORING AND SANCTIONING

This chapter focuses on the fifth design principle: *monitoring and graduated sanctioning*. Ostrom (1990) presents monitoring and sanctioning as two separate yet related design principles. The reason for combining the principles is that they both serve a common goal, namely, to ensure that individuals and companies act according to the rules created in the community and to ensure that members adopt the values and norms of the community. The first section of this chapter discusses in more detail what monitoring and sanctioning are and identifies some of the problems facing both activities. It then continues with some examples of violations or infractions in open source communities that suggest a need for monitoring and sanctioning. The remainder of the chapter is devoted to observations concerning the way in which the communities deal with infractions.

The first observation is that the communities deal with many small violations and infractions almost automatically. Also, many actions form a potential threat to the continuity of the communities. Yet, as discussed in previous chapters, a redundancy of mechanisms creates slack and makes the communities resilient to individuals and companies who ignore one mechanism or violate one rule.

Although the need for monitoring and sanctioning is not great, this does not mean it is absent. This chapter demonstrates that many monitoring and sanctioning activities are performed at hardly any cost to the participants in the communities. The argument is that individuals who actively participate in the communities monitor the actions of other participants and in some instances they also impose a sanction almost as a matter of course.

The fifth design principle: monitoring and graduated sanctioning

Ostrom (1990) found that in all sustainable communities time and effort is invested in monitoring and sanctioning activities. "[A] self-organized group must solve the commitment problem without an external enforcer. They have to motivate themselves (or their agents) to monitor activities and be willing to impose sanctions to keep performance high" (Ostrom 1990, p. 44). In this statement, Ostrom highlights two issues that surround monitoring and sanctioning.

The first is that monitoring and sanctioning are needed to create a certain level of commitment. Commitment is important to ensure that members and non-members of a community comply with the formulated rules and other accepted conduct. For instance, monitoring and sanctioning are needed to ensure that everyone complies with the appropriation rules formulated in a community. Such rules are worthless in themselves. Rules start to have value if people actually behave and act in accordance with them. Previous research, not restricted to common pool resources, suggests that the commitment to collaborate and act according to norms and rules can only be achieved when monitoring and sanctioning are in some way institutionalized in the community (e.g. Ostrom 1990, Ostrom 1999, Pruitt 1998). In other words, noncompliance needs to be monitored and sanctioned.

The second issue highlighted in Ostrom's statement is the challenge to motivate individuals and ensure that they are willing to invest some of their time to actually perform monitoring and sanctioning activities. This task is not straightforward, as both activities can be costly (Ostmann 1998). Theoretically, both monitoring and sanctioning are said to be public goods.

Monitoring and sanctioning as public goods

The actual performance of monitoring and sanctioning activities in a collective is a public good (e.g. Orr 2001, Osterloh 2002) for two reasons: (i) People cannot be excluded from the benefits of monitoring and sanctioning. (ii) Consumption of the benefits of monitoring and sanctioning is joint.

Consider, for example, the appropriation rules in a community in which common pool resources are governed. Suppose the community has been able to motivate a number of its members to invest time and effort in performing monitoring and sanctioning activities. The other members benefit from their efforts, as there is a larger chance that everyone will adhere to the rules. It is difficult to exclude the other members from these benefits. Furthermore, the benefits enjoyed from monitoring and sanctioning are not affected by a new member entering the community.

The fact that monitoring and sanctioning are public goods implies that many members in the community are tempted to free ride (Osterloh 2002, Ruttan 1998). There is little incentive for individuals to participate and invest time and effort in performing monitoring and sanctioning activities. This raises the question of why "community members cooperate to sanction offenders when they themselves could free-ride on their own duties as enforcement agents" (Ruttan 1998, p. 44). The answer lies in the costs and incentives involved in monitoring and sanctioning. The lower the personal costs and the higher the personal incentives, the more likely it is that individuals in the communities will perform monitoring and sanctioning (Lazega 2000).

Examples of monitoring and sanctioning in common pool resources

In her book, Ostrom (1990) provides examples of sustainable common pool resources. She argues that in sustainable communities members have devised systems to keep the personal costs of monitoring and sanctioning to a minimum. At the same time these systems raise the personal incentives for members to perform both activities. The relatively high incentives and low costs explain why profit-maximizing actors perform monitoring and sanctioning activities.

One example is an inshore fishery in Alanya, Turkey. The members of this community devised a rotation system to assign the fishers to designated areas. The system works as follows. First the fishing area is divided into separate locations. Then a lottery is used to assign each fisher to one of the fishing locations. The fishers are only allowed to fish at their designated location. The next day every fisher is assigned another location. The system is termed rotating because every day the fishers move up one location, either eastward or westward, depending on the season. Regarding the rotation system, Ostrom (1990) writes:

> The process of monitoring and enforcing the system is, however, accomplished by the fishers themselves as a by-product of the incentive created by the rotation system. On a day when a fisher is assigned one of the more productive spots, that fisher will exercise that option with certainty (leaving aside last-minute breakdowns in equipment). All other fishers can expect that the assigned fisher will be at the spot bright and early. Consequently, an effort to cheat on the system by traveling to a good spot on a day when one is assigned to a poor spot has little chance of remaining undetected (pp. 19/20).

Thus, the example demonstrates how people can design systems that lower the personal costs of monitoring and sanctioning. The reasoning is basically that a fisher who is assigned a location will actually be in that location to fish and will therefore monitor others and protect the boundaries and appropriation rules without incurring significantly more costs.

A second example is that of an irrigation rotation system. Ostrom explains that in many of the long-enduring communities systems are created which "place the two actors most concerned with cheating in direct contact with one another" (Ostrom 1990, p. 95). One design could be a rotation system based on time, in which one appropriator has to finish activities before another can start to appropriate water from the system. The first appropriator will want to extend the time and the second will want to start early. However, the "presence of the first irrigator deters the second from an early start, the presence of the second irrigator deters the first from a late ending. Neither has to invest additional resources in monitoring" (Ostrom 1990, p. 95). In other words, monitoring has been transformed into a natural by-product of their activities and the costs needed to monitor are brought to a minimum.

Next to low costs, sustainable communities have created systems that raise the personal incentives to monitor and sanction infractors (i.e. people who violate a rule, see Lazega 2000). In the example of Alanya, Ostrom (1990) writes:

> Cheating on the system will be observed by the very fishers who have rights to be in the best spots and will be willing to defend their rights using physical means if necessary. Their rights will be supported by everyone else in the system. The others will want to ensure that their own rights will not be usurped on the days when they are assigned good sites (p. 20).

Why sanctioning should be graduated

According to Ostrom (1990), in communities that are sustainable the sanctioners use graduated sanctions to penalize infractors. She reasons that in certain settings people who usually comply with the rules might feel forced to break a rule. Infraction is, for instance, more likely and should be sanctioned mildly during a depression. Mild sanctions could for instance take the form of "unobtrusive and unsolicited advice and the spread of gossip" (Lazega 2000, p. 194). The importance of graduated sanctions is indicated by the observation that "a large monetary fine imposed on a person facing an unusual problem may produce resentment and unwillingness to conform to the rules in the future" (Ostrom 1990, p. 98).

This effect described in Ostrom (1990) can be linked to the concept of *crowding-out* (Frey 1997). Crowding-out is relevant in situations in which an individual is intrinsically motivated to perform a certain activity. Frey (1997) reasons that in such a situation the presence of excessive monitoring and sanctioning *can* have a reverse effect and can result in a decline in motivation. Crowding-out then is the idea that monitoring and sanctioning can result in agents feeling either (i) a diminished sense of self-esteem concerning their ability or (ii) that they are not trusted to take action without monitoring or incentives (Orr 2001). Crowding-out refers to a situation in which monitoring and sanctioning actually reduce motivation and create less incentive to collaborate. Research suggests that crowding-out is more likely to occur when (i) the person being monitored and sanctioned perceives these activities to be excessive and (ii) the person has no idea why they are being monitored and sanctioned (Orr 2001).

The first reason suggests that monitoring and sanctioning should not be excessive. It should be designed to take into account the situation with which the person is faced. The presence of graduated sanctions reduces the chance that sanctions are excessive and result in crowding-out.

The need for monitoring and sanctioning in open source

There are a great number of reasons why monitoring and sanctioning are needed in open source communities. "There is a whole spectrum of things that are not likable."[1] Consider, for instance, the lack in boundaries to regulate who is allowed to enter a community. The fact that anyone can join a mailing list and write, for example, offensive, off-topic or ridiculously long e-mails has been empirically proven to be a serious threat to the continuity of virtual communities (Kollock & Smith 1996, Smith 1999, Turkle 1995). Indeed, respondents interviewed for this research frequently referred to this type of behavior as resulting in annoyance and being counterproductive. "Bad behavior is mainly rude behavior, swearing, disrespecting consistently other people's opinion."[2] It is considered to be undesirable and a threat to the processes in the community if a participant is, for instance, "dogmatic and doesn't work well with the rest."[3]

Counterproductive behavior is not restricted to the mailing lists. The communities use many more tools and mechanisms to support and structure individuals' activities. In all of these, counterproductive behavior could become a threat to the mechanisms' intended use. Consider, for instance, the voting system in the Apache community. What happens if the community consists of members who continuously vote negatively, effectively putting a stop to the improvement of the source code? Chances are the bigger projects in the Apache community consist of one or more participants who are conservative. These participants will tend to vote negatively and by doing so could prevent much source code from actually being added to the latest version of the software. If they did so, implementation of new ideas would become difficult. Furthermore, chances are that other participants in the project would become frustrated, no longer enjoy participating in the project and might consider leaving.[4]

Another example of a coordination mechanism is the credit file, which is used in many open source communities. Next to a coordinative role, the credit files are an important mechanism to motivate participants to contribute time and effort to the development of the software. The credit files provide a means for participants to earn a reputation and demonstrate to others how much they have contributed. This might be the reason why the removal of someone's name from a credits list is said to be a serious crime. "Surreptitiously filing someone's name off a project is, in cultural context, one of the ultimate crimes" (Raymond 2000).

A different type of offense and potential threat is to waste the time of participants in open source communities. "The worst thing you can do is to waste people's time, especially when it is something that is already dealt with."[5] There are a number of reasons why respondents consider wasting someone's time disruptive. One is that many participants contribute their time and effort because they want to be challenged and enjoy developing new source code. Yet their time and effort is limited; they can only spend so much time in the communities. If much of that time is consumed by people who are constantly asking stupid questions, making stupid contributions or writing inelegant or crooked source code, chances are that their motivation to continue to participate in the communities will rapidly diminish. Yet every user of and every contributor to open source software depends on the presence of a sufficient number of motivated contributors.

Robustness to survive many small infractions

The previous section argued that there is a need for monitoring and sanctioning in open source communities. The reason is that many infractions are conceivable against which the communities need protection. However, open source communities are fairly robust systems, which protects them against small infractions. Further, open source communities have redundant sets of mechanisms which transform many actions that could potentially harm the continuity of the communities into relatively harmless events. Two examples are provided below.

First, chapter five demonstrated the presence of a redundant set of coordinative mechanisms. These mechanisms provide developers with many easily accessible options to remove patches that might harm the integrity of a software program. Mechanisms like elegance, modularity, small incremental patches, the CVS and coding style guides ensure that developers need expend relatively few resources preventing the addition of bad source code. The mechanisms are redundant, which lowers the communities' dependence on a single one and strengthens the system as a whole. Consider, for instance, a contributor who writes inelegant code. If the contributor uses the coding style guides and contributes the code in small patches, then other participants need invest relatively little time to understand the source code, even if it is inelegant. Now consider a community in which the elegance of source code is the only mechanism to reduce the time needed to understand new patches. In such a community a contribution of inelegant source code would require high investments by the participants to understand the source code. Not only is the source code inelegant, the lack of a coding style guide could also imply that the source code is structured differently and therefore relatively more time and effort is needed to determine how it is written and whether it is actually inelegant. Thus, communities that adopt fewer coordinative mechanisms have a lower level of resilience to nonuse of a single coordinative mechanism. In such a situation, monitoring and sanctioning would become more relevant.

Second, chapter six described the presence of conflict resolution mechanisms. It argued that conflicts are a potential threat to the continuity of open source communities, as they could slow decision-making processes and hinder implementation. This is especially relevant when we consider that the communities have a high potential for conflicts. The chapter identified many mechanisms that were said to mitigate conflicts. Conflicts are mitigated through parallel development lines, which can take the form of a second head in a CVS or the presence of both a stable and development version of the software. Next to parallel development lines, participants have the option of exiting the communities or creating a fork. Combined, these mechanisms are redundant; there is not just one mechanism to mitigate the potential negative consequences of a conflict. Instead, many mechanisms are available to every participant at any time. These mechanisms build in resilience against the potential threats of conflicts.

Infractions by participants, such as continuously seeking conflicts, ignoring coding style guides, writing inelegant source code or composing long and abusive e-mails, are generally relatively easily dealt with. The redundant set of mechanisms described in each of the chapters ensures that many of these actions are relatively unimportant and harmless. The fact that developers do not mention these events as counterproductive and potentially harmful further strengthens the idea that they are simply considered to be part of the processes in the open source communities.

More formal sanctioning mechanisms are hardly used

In the interviews, respondents reported the presence of two, somewhat more formal, mechanisms available to them to sanction participants. They are (i) to remove the right of a participant to directly upload source code into the CVS and (ii) to remove a participant from a mailing list.

As discussed earlier, many communities have adopted a CVS to support their software development and maintenance activities. Not just anyone can upload source code into the CVS. Before participants can actually upload source code they need to be granted the right to do so. In the Apache community, a participant with access to upload source code directly into the CVS is referred to as a 'committer.' The procedure to become committer might appear strict and a real barrier of entry. But in reality it hardly is. One respondent from the Apache community explained, "I often think, if this person has added a few good patches, let them join. These people will then get an e-mail which says, 'Congratulations, you have become a committer, you now have access. Please respond if you accept this invitation.'"

Participants with committer status, however, could potentially cause much frustration and even endanger the continuity of software development and maintenance activities. They could, for instance, upload horrible patches of source code or continuously remove source code that they believe is bad, but which is actually good.

There is a relatively easy and efficient mechanism to sanction participants who abuse their committer status. If participants abuse their right to upload source code to the CVS too often, they risk having their right rescinded. "Abuse of …privileges is easy, but if abuse occurs regularly, then you are out."[6]

Yet the surprising finding from the interviews was that although the use of this sanction mechanism is considered to be easy it is hardly used in the communities. A member of the ASF Board of Directors, for instance, explained, "We could kick out people, but even that we have never done." Another member explained, "It has never happened, but he could be removed." Apparently, in many communities and according to many respondents there is hardly any need to actually remove participants' rights to commit source code directly into the CVS.

The same is true for participants on mailing lists. Many mailing lists have a maintainer who can remove participants who write long and abusive e-mails. Yet this mechanism too has hardly been used.

Two remarks need to be made here. First, the fact that someone's write access is hardly ever removed does not automatically imply that it is not an important sanctioning mechanism. Neither should the limited use of the mechanism be taken to mean that it has no effect. The mechanism's mere presence could be sufficient. It could be that simply the threat of removal of committer status or the right to send e-mails to a mailing list is sufficiently powerful to prevent or stop participants from writing stupid e-mails or crooked source code. Second, the actual claim here is not that the sanctioning mechanisms are never used. The claim is that the mechanisms are used only to a limited extent, much less than one might expect.

The costs of monitoring are low: development is monitoring

The observation that many activities are relatively harmless does not mean that monitoring in the communities is entirely absent. On the contrary, participants in open source software communities are likely to monitor the behavior of other participants. They do so almost automatically; primarily because the costs of monitoring are extremely low. Three factors explain the low costs: (i) transparency, meaning that the behavior and actions of every developer are highly visible to others (Sharma et al 2002, in particular table 2); (ii) the presence of tools that automatically notify other participants in the communities of a new activity, such as a new addition of source code; and (iii) the nature of software, which enables instant feedback about the quality of the source code.

The communities are highly transparent

Osterloh (2002) writes the following about monitoring in open source communities: "[I]n the open source community monitoring the behavior of users is easy because the Internet gives full transparency" (p. 15). This "full transparency" refers to the fact that many of the tools and forums in open source communities are open and can be reviewed by practically anyone. For instance, the mailing lists, the repositories in which the source code is stored, the credit files and the open source licenses are completely open and visible, that is, they are transparent. This transparency is formally enforced by at least one open source license:

> The GNU GPL also includes provisions ensuring that the code includes information about the programmers who wrote it. The license allows licensees to modify the licensed code so long as they "cause the modified files to carry prominent notices stating that [the files were] changed and the date of any change (McGowan 2001, p. 256).[7]

Transparency enables participants in the communities to monitor the activities of other developers. Every time a participant or user downloads source code from the Internet, they have access to the history of the source code, they can examine who changed what and when and they can review the content of the open source licenses. This enables anyone to monitor the activities of participants in a community. Even actors who do not participate, who are not actively involved in the communities, can have a glimpse at the actions of participants.

The automatic notification of new activities

At least two coordinative mechanisms work in such a way that participants are automatically notified when something happens that is in line with their interests. The two mechanisms are the mailing lists and the CVS. Both share the fact that they significantly lower the costs needed to monitor the behavior of other participants in the communities.

The primary goal of a mailing list is to enable participants to exchange knowledge and ideas and to inform one another of new developments. Most mailing lists have an easy subscription process. Generally they are computerized. To become a member, one needs only to create a password and provide a mailing address. The mailing list automatically distributes new messages to all members subscribed to that list. Members need make no investments to receive messages from a mailing list they are subscribed to.

Typically, the more popular mailing lists generate a lot of traffic, which means that many e-mails are sent and distributed on these lists. Furthermore, participants generally subscribe to a number of mailing lists. One respondent claimed to spend an hour each day reading and answering the e-mails received. Obviously, not every participant is able to invest an hour or more in the communities each day. Thus, it is practically impossible for participants to thoroughly read each and every e-mail they receive. Yet they do have the option of reading the e-mail correspondence between participants on a mailing list, because they automatically receive every e-mail in their in-box.

Most communities store the source code in a CVS. According to an ASF board member, the participants on a particular module of the Apache software automatically receive a written explanation (called a *log file*) of every change made to that module. The log file provides information like who contributed the code, why the code was submitted, when it was submitted and how it is written. In much the same way as the mailing lists, this log file allows contributors to share knowledge and ideas and understand what and why others have made modifications (Shaikh & Cornford 2003). The log file also turns participants into monitors, as it allows them to monitor and evaluate the actions of the participant who contributed the source code. Doing this monitoring requires little additional investment; monitoring in the CVS is almost automatic.

Instant feedback

The mechanism of instant feedback refers to the fact that the use of software almost automatically results in testing of the software (see also Raymond 1999b). This characteristic is one reason why Von Hippel and Von Krogh (2003) claim that free riding in the communities is hardly a problem. Free riders in open source communities use the software and by doing so bring in something else that is valuable. According to the authors, free riders, for instance, report the bugs they encounter using the software.[8] A similar point was made by a respondent interviewed for this research. This respondent, who worked for a company that uses Apache software, argued how their experience with the software was invaluable to the community, as they brought in knowledge about how the software works in a business situation. Even if they only report bugs, they are contributing to the community. They also identify aspects of the software that need improvement.

Users of open source software have downloaded a specific version of the software and run that version on their computers. The reason why they automatically monitor the activities of the contributors is as follows. When using software, there are two possibilities: either the software does what it is supposed to do or it does not. Assume that it does not work like it is supposed to, for example, the software crashes or certain functionalities fail. This implies that something is wrong. The user has found a bug and could report this to the community. In itself this is already an act of monitoring. Participants in the communities are notified of a problem and can try to discover the source of the problem and fix it.[9]

Another option available to the user who discovers a bug is to go back a version of the software and try that version (e.g. Shaikh & Cornford 2003). If this older version of the software works, then the conclusion must be that certain changes were made to the older version of the software in which something has gone wrong. The user has essentially spotted source code that should be changed or removed from the software. Or the user could simply have stumbled on a flaw. The discovery of a flaw is an automatic by-product of using the software. In other words, active users of open source software are to some degree automatically monitors.

The presence of many mild sanctioning mechanisms

Many participants in open source communities automatically monitor the activities in these communities. The claim made here is that these participants also, almost automatically, sanction infractors. Participants who sanction others incur little additional cost and, in some instances, they are not even aware of the fact that they have sanctioned someone; their mere presence and participation is sufficient. One reason why these light sanctioning mechanisms work is that many of the infractions in open source communities have no serious effect. They hardly threaten the continuity of the communities. Therefore, the infractors need no serious and harsh redress.

The sanctioning mechanisms discussed below have a mild effect; they primarily influence the reputation of the infractors. Five such mechanisms were mentioned in at least one interview: (i) the hall of blame, (ii) flaming, (iii) spamming, (iv) shunning and (v) forking.

The hall of blame

Essentially, the hall of blame[10] is the reverse of what Markus et al. (2000) refer to as the scoreboard of open source projects. "Our scoreboard is the 'credit list' or the 'history file' that's attached to every open-source project" (This statement is cited from an open source developer who is cited in Markus et al 2000, p. 15). This statement is similar to one made by a respondent interviewed in this research: "The free software developers can almost be considered professional athletes, everyone can see their statistics. You can see what they did and what they are good at."[11]

Not only are the good and valuable contributions stored and made available; the less productive contributions are also stored and remain visible for all to see. Therefore a respondent argued, "Everything you write can be used against you."[12] The Internet can thus turn into a hall of fame, but also a hall of blame. "Online, a record is kept of who did what and when. Once you make a big mistake then it becomes a 'hall of blame.'"[13] To prevent this from happening, two respondents said that they make sure to check every piece of source code before they add it to the repository. They do so because they want to prevent stupid mistakes to avoid the laughter of the community.

The hall of blame is an automatic by-product of the presence of people who participate in open source communities. The hall of blame is an instant sanctioning mechanism, which requires no investment from participants in the community. It is a modern day version of a pillory; everyone can make fun of a person who does something stupid or obviously wrong.

Flaming

Flaming, alongside spamming and shunning, is one of the sanctioning mechanisms that has received the most attention in the literature (e.g. Maggioni 2002, Markus et al 2000, Osterloh 2002, Sharma et al 2002). Flaming is "the public condemnation, over e-mail lists, of people who violate norms" (Maggioni 2002, p. 8). According to the Jargon File dictionary version 4.4.5, flaming has several meanings. One is "to post an e-mail message intended to insult and provoke."[14] An extension to this definition is also given, namely, "directed with hostility at a particular person or people."[15] Essentially, flaming is an act of naming and shaming; it is writing negatively about someone who supposedly did something wrong. Mailing lists and news sites are used to communicate this to others within the community or even to people outside of the community. Though the number of readers of a single e-mail can be quite high, the costs of flaming are relatively low. The only costs are those associated with writing the e-mail. The sanction mechanism is available to everyone who is subscribed to the mailing list.

Spamming

Spamming means "flooding someone with unsolicited e-mail" (Osterloh 2002, p. 14). Thus, a person or a company is spammed when they are sent huge amounts of e-mail. This is a form of sanctioning, because the receiver is distracted and has to sort through the e-mails. It is likely to lead to annoyance. The content of the e-mails is usually not very friendly either.

Shunning

Shunning means ignoring or "deliberately refusing to respond" (Osterloh et al 2003b, p. 16). The editor in chief of the Linux journal, for instance, describes how most participants have actually created customized e-mail systems to manage their incoming mail from open source mailing lists. The participants can configure their systems in such a way that it automatically filters e-mails from people and companies from which they do not want to receive mail. The shunned people and companies are added to the so-called 'kill list.' "If the noise becomes too bad, people will attach that person or sender to their kill list and they don't read messages from them anymore."[16] The kill list is a way to ignore (i.e. shun) people on a mailing list.

Shunning is not limited to mailing lists. It is visible in a great number of forums. Maggioni (2002) acknowledges this and therefore defines shunning in more general terms. He writes, shunning is the act of "refusing to co-operate with someone" (p. 8). Consider a contributor who has written inelegant code or created a new functionality that is deemed useless by the other participants. According to Wayner (2000) these contributors are automatically penalized because, in the case of inelegant code, "others come along and try to use their code. If it's inscrutable, sloppy, or hard to understand, then others will ignore it… That is a strong incentive to do it right" (Wayner 2000, p. 118).

The fork

Chapter six described the fork in some detail. The fork was defined as two versions of software that have the same origin, but which, in the course of time, become irreconcilable. The fork was argued to be a mechanism to defer some of the potential negative effects of a conflict.

Essentially, the fork is available to everyone. Furthermore, the primary goal of a fork is not to sanction others. A fork is an inherent element of open source communities, since it affords individuals the freedom to create a competing project if they want to. Anyone can create a fork for any reason. The actual act of creating a fork starts with the creation of a new forum, typically a new community, in which the source code is maintained. This act is simple and requires little investment. However, applying a fork to truly sanction another party or an entire community requires a lot of investment and is much less simple.

The actual costs of creating a successful fork are relatively high. This is true for the party who created a fork as well as for the community from which the software has been copied. The party responsible for creating the fork needs to invest time and effort in building the new community (community A) and in attracting new participants to contribute time and effort to maintain and improve the software. A lack of participants will make the software in community A less popular for reasons described in the chapter on collective choice. For the community from which the software was forked (community B) the costs can also be high. The more popular the software in community A becomes, the fewer people will participate in the maintenance and improvement of the software in community B. Finally, on a macro-level the costs of forks are also high, since it is said to result in a destruction of value. The fork gives rise to two communities, which at least at first need to perform many of the same activities.

Nonetheless, no matter how high the costs, forks are used in open source communities and they can be a powerful sanctioning mechanism. They are a last resort for any party who strongly disagrees – or is annoyed – with the general direction of a community, with its atmosphere or with individual decisions. A well-known example of a fork used as a sanctioning mechanism is a split in the GNU Emacs community.

The Emacs community is a relatively old community. In the nineties the project leader was Richard Stallman. In 1994 participants became annoyed with the way Stallman led the project. They felt he did not maintain the program well. "The programs had not been maintained for quite some time."[17] Many participants wanted to contribute code, which they sent to Stallman. However, he frequently refused to include the patches in his version of Emacs. This meant that the product did not improve as much as it could. Furthermore, many people got frustrated, as they spent time improving the program and writing patches, which were not included in the official version. The only option for them was to include the patches in their own privately maintained versions of Emacs. This resulted in inefficiency, as users needed to regularly change and update their own version. In the end, a group of developers, most of whom worked for the company Lucid, created a new version of Emacs. They were fed up with the way things were going in the community, "upset by the constant delay… a group of people decided to fork off GNU Emacs and start a new project intended to gather all the latest technologies that were picking up steam fast."[18]

In this example the fork proved to be an efficient sanctioning mechanism, as "it resulted in a lot of activity in the GNU community and they started to make new versions again."[19] The fork appears to have provided a way to change the state of affairs in the GNU Emacs community.

The impact of sanctioning mechanisms is proportional

The impact of most of the sanctioning mechanisms discussed above appears to be proportional. The question of whether sanctioning is proportional is relevant, because if it is that means participants who commit small infractions are not excessively penalized. This, in turn, minimizes the chance of driving out intrinsic motivation. Participants are likely to cease their activities if they believe they are being sanctioned too excessively or too frequently. There is thus a preference for sanctions that are related to the seriousness of the offence.

This section argues that proportional sanctioning mechanisms are quite feasible in open source communities. This claim is based on two assumptions: (i) The impacts of many sanctioning mechanisms depend on the number of participants who actually use them. (ii) The more serious the infraction the higher the number of participants who are affected by it and thus the more people who will use the sanctioning mechanism. If both assumptions are true, then many of the open source communities' sanctioning mechanisms are indeed proportional.

For instance, the impact of the hall of blame is clearly proportional. Every participant now and then writes crooked source code or an irrelevant e-mail. The frequency with which this happens directly influences the 'size' of the hall of blame, simply because there is more 'evidence' available. Effectively, the more nonsense a participant writes the bigger the hall of blame becomes.

To shun, flame and spam are also proportional sanctioning mechanisms. The impact of the three mechanisms depends on the person who actually performs the sanction. If, for instance, that person is a project leader then the impact is rather large compared to an unknown developer. The impact of shunning, spamming and flaming also and maybe more importantly depends on the number of people that use the mechanisms. If only one or a few participants use the mechanism then the impact is relatively low. How different this is if an entire community becomes annoyed with a person and decides to ignore or pummel that person with questions or unfriendly remarks. It is plausible that the greater the number of people affected by the infraction, the more people will want to use the sanction mechanism and, for instance, ignore the infractor. Therefore the higher the effect of the sanction will probably be. To summarize, the impact of flaming, spamming and shunning is likely to depend on (i) the number of people using the sanction and (ii) the relative importance of these people in the community.

Forking is also a proportional sanctioning mechanism. The reason is similar to the previous. The more people who concurrently decide to fork a project, the more impact it has. If just one participant decides to fork software developed in a community made up of, say, 1,000 members, then the impact will probably be small or even negligible. Depending on the importance of the developer engineering the fork, the participant might hardly be missed. Furthermore, the speed of software development in the community remains high, much higher than a single software programmer can generate. This is different in a situation in which a larger number of participants decides to fork a project. In this scenario the communities might become serious competitors and the forked community could lose participants to its new competitor. The impact of a fork as a sanctioning mechanism thus depends on the total amount of effort that is left in the community that was forked versus the total amount of effort mobilized to participate in the development of the fork.

Discussion: is use of the sanctioning mechanisms gradual?

The fact that sanctioning mechanisms are proportional is important to lower the chance of driving out motivation. But what happens when a person continues to infract rules and norms? According to Ostrom (1990), more serious forms of sanctioning are needed to impose harsher sanctions on recurrent infractors. Some evidence suggests that developers in the communities use the mechanisms gradually.

One important source of evidence is the way in which participants use their ability to remove someone's committer access to the CVS. It is a sanctioning mechanism, but at the same time, as a previous section in this chapter argued, it is seldom used. It appears to be primarily used as a deterrent against serious infractions. The maintainer of the Python language explained that they "never had to throw someone out." Earlier we read the same respondent arguing that they would only remove someone's access to the CVS if a violation occurred regularly. In other words, access to the CVS will not be removed if a participant contributed inelegant source code only once. It is not used to sanction first-time infractors. Instead, it is likely to be used only in cases in which a participant continues to write inelegant source code or continues to remove good source code written by other participants.

There are examples of communities that rescind people's right to directly include source code in the code tree. One well-known example is the NetBSD community. Allegedly, Theo de Raadt's access was removed because he gave rise to many conflicts. Many developers in the community felt that his behavior was simply too obtrusive and abusive. Allegedly, "De Raadt's behavior and abusive messages had driven away people who might have contributed to the project" (Wayner 2000, p. 214). The developers in the community addressed this issue with De Raadt on many occasions, but apparently this had no effect. Therefore, they believed they had to remove his access to include source code directly in the source tree.

The exact details of this example, which is still surrounded with confusion and debate (Wayner 2000), are less important. What is interesting is that it provides some evidence to support the claim that this sanctioning mechanism is gradual. The participants in the NetBSD community first cautioned Theo de Raadt and used other sanctioning mechanisms like shunning. In the end, however, they decided to sanction his behavior by rescinding his committer status and thus his right to directly upload source code into the CVS.

Conclusion

This chapter discussed the fifth design principle: monitoring and sanctioning. It argued that the principle is important to ensure that participants adhere to the rules and norms of a collective. The challenge of this design principle is to get people in a collective motivated to actually perform monitoring and sanctioning activities despite the costs, which can be quite high. Furthermore, monitoring and sanctioning are public goods. This means that the benefits of performing these activities must be shared with others in the collective. There is thus a collective action problem: 'How to get people in a collective motivated to perform these activities themselves?' In many of the sustainable community-governed common pool resources, this problem is solved by creating systems in which the costs are minimized and the personal benefits maximized.

The chapter continued with an argument of why open source communities are in need of monitoring and sanctioning. It proposed a number of reasons why the communities need to monitor the behavior of individuals and companies and need to ensure that they adopt coordinative mechanisms like the coding style guide and the log files in the CVS. Another type of offense was to waste the time of participants in the communities.

Although there is a need for monitoring and sanctioning, many types of infractions are dealt with almost automatically. Due to a redundancy in mechanisms to deter conflicts, to support the development processes and to prevent appropriation of open source software, many actions that could become a serious threat are actually automatically resolved. This redundancy in mechanisms reduces the potential threat of infractions.

The remainder of this chapter presented three observations. The first was that two of the more formal mechanisms to punish infractors are hardly used. Participants in open source communities might decide (i) to exclude individuals from a mailing list and (ii) to rescind an individual's right to commit source code directly into the CVS. Yet respondents claim that both mechanisms are seldom used. One explanation could be that these sanctioning mechanisms are considered harsh and only for use in cases of serious infractions.

The second observation is that participation in the communities automatically involves monitoring. There are three reasons why: (i) The communities are highly transparent. (ii) Participants are automatically notified of new activities. (iii) Use of the software automatically results in testing of that software.

Third, developers in open source communities have access to many mild sanctioning mechanisms. The sanction mechanisms the hall of blame, flaming, spamming and shunning are almost automatic by-products of participation in the communities. These sanction mechanisms cost little and sometimes occur without sanctioners even knowing that they are actually sanctioning someone. The fork does not come without costs, but these costs are the costs involved in development and maintenance of the software. They are thus not considered a new or different type of expense.

The underlying 'system' of monitoring and sanctioning

In the sustainable communities analyzed by Ostrom (1990) the participants devised systems that minimize the costs and personalize the benefits of monitoring and sanctioning. This system is created, implemented in the communities and adopted by the community members. This is different from the way in which monitoring and sanctioning in open source communities is organized. There is no macro system that explains why and how individuals perform these activities. Neither is there an authority to decide whether a certain participant should be sanctioned. The sanctions are performed by the individuals themselves.

Another striking characteristic, which is not necessarily different from the sanctions in the communities described in Ostrom, relates to the type of sanctions. Apparently, individuals in the communities consider the presence of evidence that they have written bad code or stupid e-mails, a sanction in itself and a reason to do things right. Similarly, being ignored by other participants or receiving many negative e-mails is sufficient motive for individuals to stop certain types of behavior. The underlying logic of these sanctions and why they are viewed as a sanction relates to the logic that determines why and how individuals participate and act in the communities.

Making sense of the observations: two individual behavioral rules

The reason why being ignored is considered a sanction can be understood with just two individual behavioral rules. In fact, many of the observations related to monitoring and sanctioning can be understood with these rules.

"*Participants want to increase the chance that others will accept and adopt their contributions.*" This rule was previously introduced in chapter five. It argued that there are a number of reasons why participants want others to adopt their contributions: to increase their reputation and reduce the effort they need to expend to maintain the source code.

"*Participants base their choice on tags, like the level of reputation of participants.*" This rule is a somewhat modified version of the rule introduced in chapter seven. That rule basically stated that participants want to minimize the time they spend analyzing software and therefore base their choices on tags. Relevant in this chapter is the tag *reputation*. Chapter seven argued that participants prefer to use and download software from communities or projects in which participants with a high level of reputation are involved; and participants listen better to participants who have a relatively higher reputation. "Jeremy Allison of Samba enjoys so much respect that he can talk to the Linux people and the Linux people will listen."[20]

With every contribution, whether it involves an e-mail or source code, the name of the contributor is connected. "My name is attached to every KDE program I have ever translated."[21] This system allows other participants to judge the reputation of other participants in the communities. The reputation of participants is likely to improve when they make positive contributions, for instance, when their names are attached to highly sophisticated and elegantly written source code or when they have solved many problems of end users. In a similar line of reasoning, their reputation will diminish when they write stupid e-mails or inelegant source code. The level of reputation is also affected by others writing negatively about them, since participants' opinions are affected by these negative e-mails.

The fact that other participants base their choices on tags, as argued in the second rule, implies that participants in the communities have an incentive to want to increase their reputation. The higher their reputation, the greater the chance that others will take a look at their source code and the higher the chance the source code will be accepted and adopted.

The first rule also explains why the act of shunning, that is, ignoring, is viewed as a powerful sanction. Less people will download and adopt a contribution made by a shunned developer.

Notes on chapter eight

1 From an interview with a member of the ASF Board of Directors.
2 From an interview with the project leader of PostgreSQL.
3 From an interview with a member of the ASF Board of Directors.
4 From an interview with two members of the ASF.
5 From an interview with the editor in chief of the Linux journal.
6 From an interview with the project leader of Python. The project leader of the PostgreSQL community more or less stated the same.
7 McGowan cites this part directly from the GPL.
8 One can wonder whether we should still speak of a free rider in this case. There are two lines of reasoning. On the one hand, these people are no longer free riders as they contribute time and effort to writing a bug report. On the other hand, they are still acting as a free rider; in writing a bug report free riders hope others will solve the problem for them.
9 This process is not as simple as portrayed in the text. In reality to write a good bug report, which can help others to fix the bug, is quite a difficult task, which requires some level of knowledge and skills.
10 This name was adopted from an interview in which two members of the ASF explained how their actions remain traceable and how the Internet could thus turn into a hall of blame.
11 From an interview with the editor in chief of the Linux journal.
12 From an interview with a developer at IBM who is actively involved in the Linux kernel.
13 From an interview with a member of the ASF Board of Directors.
14 From the Internet: http://people.kldp.org/~eunjea/jargon/?idx=flame (January 2004).
15 From the Internet: http://people.kldp.org/~eunjea/jargon/?idx=flame (January 2004).
16 From an interview with the coordinator of the Greater New Hampshire Linux User's Groups.
17 From an interview with a developer in Linux kernel and based on an article on the Internet: http://www.beust.com/weblog/archives/000014.html (January 2004).
18 From: http://www.beust.com/weblog/archives/000014.html (January 2004)
19 From an interview with a developer in Linux kernel.
20 From an interview with the editor in chief of the Linux journal.
21 From an interview with the head of Dutch translation in KDE.

CHAPTER NINE

LAYERS OF NESTED ENTERPRISE

The design principle *multiple layers of nested enterprise* is the topic of this chapter. This principle is especially relevant in communities in which larger and more complex resources are governed. Such resources are likely to consist of many subsystems, which must be managed differently. Yet they cannot be managed in total isolation from each other. The subsystems are interconnected and interdependent. To deal with this interconnectedness, communities that govern common pool resources sustainably have created a structure that is based on multiple layers of nested enterprise.

This chapter argues that open source software programs are complex products that consist of many different yet interdependent subsystems. Previous chapters already demonstrated one way in which the complexity is approached, namely, by dividing the software into smaller and relatively independent modules. Another way the complexity is diminished is introduced in this chapter, namely, through the creation of roles; that is, by dividing complex activities into clusters of activities. This division of labor is argued to be emergent, as the process is not managed by a project leader or by any other type of formal authority.

Three other observations are introduced and discussed in this chapter: (i) The division of labor in open source communities results in task specialization. (ii) It raises the level of efficiency in the communities. (iii) The simpler activities are more visible and are relatively easily accessible, which creates a learning environment for new participants.

The sixth design principle: multiple layers of nested enterprise

The design principle multiple layers of nested enterprise is primarily relevant in larger and more complex common pool resources (Ostrom 1990). The reason is that complex common pool resources consist of many diverse subsystems which can be quite different. But, at the same time, these subsystems are interconnected and cannot be governed in complete isolation from each other. Establishing mechanisms and rules in one subsystem of the common pool resource is said to "produce an incomplete system that may not endure over the long run" (Ostrom 1990, p. 102).

The design principle basically addresses two problems. The high level of diversity that characterizes many complex common pool resources fuels the first problem. That is, it is impossible to create one set of rules that applies and is relevant throughout the entire resource. Instead, different and localized rules need to be created. According to Ostrom (1990), the presence of diversity means that communities face many problems, which need to be tackled in different ways. She uses the example of a Philippine irrigation system consisting of different types of canals. The main artery is a so-called 'secondary canal.' In this canal water is transported to various parts of the irrigation system to then be divided among the tertiary canals. The main problem is to decide how the water should be transported and divided among the tertiary canals. Users of the irrigation system appropriate water from the tertiary canal. The problems facing this type of canal are very different from those facing the secondary canal. In the tertiary canal the challenge is to decide how the water should be divided among a given number of appropriators. Ostrom (1990) writes, "The problems facing irrigators at the level of a tertiary canal are different from the problems facing a larger group sharing a secondary canal. Those, in turn, are different from the problems involved in the management of the main diversion works that affect the entire system" (p. 102).

The second problem is managing the interdependencies between the subsystems in the common pool resource. This need is fueled by the dependencies that exist between the subsystems and the sub-communities. In the example, irrigators at the level of the tertiary canal are dependent on the decisions made in the secondary canal. For instance, at the secondary canal irrigators decide how much water will flow to the tertiary canal. No matter how good and robust the institution managing the tertiary canal, if they receive no water from the secondary canal they have no water to divide. Therefore, Ostrom (1990) observes, "Establishing rules at the one level, without rules at the other levels, will produce an incomplete system that may not endure over the long run" (p. 102).

Thus, the design principle multiple layers of nested enterprise addresses the need for localization and at the same time for managing the interdependencies between localized efforts.

A need for multiple layers of nested enterprise in open source

Typically, large and complex common pool resources consist of many interdependencies. This section demonstrates that most open source software programs are indeed complex, extremely diverse and have many interdependencies. Therefore, the design principle is relevant and attention needs to be given to managing the interdependencies that exist between the participants in the communities *and* the interdependencies that exist in the technical artifact.

A first indicator of the complexity of open source software is its sheer size. Consider, for instance, the number of lines of source code in a Linux distribution. David A. Wheeler estimated the actual number of lines of code in a Linux distribution back in 2001 to be approximately 30 million, which would take 8,000 men years to build.[1] Obviously, this figure already indicates some of the complexity that faces open source communities.

The level of variety in the Linux distribution is also enormous. The Linux.org website presents a list that provides access and/or links to a large selection of applications that can be used in combination with the Linux kernel. The website lists 116 MP3 applications, 65 libraries and 36 file managers.[2] Next to the wide variety of applications are many different versions of the Linux kernel itself.[3] First, there is the stable version and the development version of the Linux kernel, both of which are maintained in the Linux kernel community. Second, all the maintainers of a major part of the Linux kernel manage a version of the Linux kernel in which their improvements are integrated. To ensure compatibility, the locally maintained versions have to be regularly updated with the development version of the kernel maintained by Linus Torvalds.[4] Third, most Linux distributions use a different version of the kernel. Companies like

Red Hat or SuSE pay programmers to select a version of the kernel, improve it and make it ready for commercial use. Fourth, many Linux end users can make modifications and not all of these modifications are included in the versions maintained in the community. End users maintain these versions on their own computer, giving rise to an even greater variety of the Linux kernel. Another indication of the variety in Linux can be found on the Debian website. Debian lists all of the software programs in the latest stable release of the Linux distribution. According to the site Linux users have a choice of 412 software administration utilities, 752 libraries and 476 games.

A final indicator of the level of complexity in open source communities is the number of individuals and companies that participate in the more popular communities. It is difficult to exactly determine the number of contributors, because the boundaries of the communities are fluid and no up-to-date records are kept of the number of members. However, some statistics are available, which provide an idea of the size of the communities. Consider, for instance, the Debian community. On its website 787 people have listed themselves and claim to be Debian developers.[5] Another example is the Linux kernel community. One estimate is that the community has 250 maintainers.[6] These are not just people who post one message or ask a question; instead they are developers recognized as being the maintainer of a certain part of the Linux kernel. Yet another indication of the size of the Linux kernel community is found on a website called 'kernel traffic.' This website produces a weekly newsletter which summarizes the activity on the Linux kernel mailing list. They have been doing this for a number of years and thus far have quoted 1,600 different people, 352 of whom have been quoted five times or more in the weekly summaries.[7] Considering the fact that these latter two numbers are based on the summaries and not the actual Linux kernel mailing list, it is safe to say that the number of people who have posted messages on the actual mailing list is much larger.

Decomposing the complexity: the presence of multiple layers

It would be undoable if all participants collectively had to decide on each and every change in a software program. Equally problematic is a situation in which participants cannot improve certain parts of a software program because someone else is already improving another part of that program. The dependencies between participants in the communities must be limited to prevent such problems and annoyances. Chapter five demonstrated a number of mechanisms that enable participants to contribute and participate relatively independently from one another. One important mechanism identified in that chapter was the modularity of software. Next to modularity, this section argues that the complexity of the activities facing the communities is also tackled through the creation of different roles. These roles are essentially collections of activities.

Modularity

As a reminder, *modularity* refers to a set of general principles in which a complex problem is divided into separate pieces. Modularity intends to "eliminate what would otherwise be an unmanageable spaghetti tangle of systemic interconnections" (Langlois 2002, p. 19). Chapter five cited a respondent who argued that many potentially complex problems in the communities are much less complex, because they are divided into smaller and relatively more independent parts (i.e. modules). This modular structure raises the level of independence among participants.

However, modularity also creates some problems. From the interviews two types of problems with modularity were identified. The first problem is to decide how to divide the software into smaller modules and how to decide what part of the software should belong in what module. In other words, it is difficult to exactly define the content of a module and to maintain the borders of a module. One respondent was asked whether he believed a certain patch of source code belonged in the Linux kernel. He responded by saying, "If it has something to do with hardware, or if it concerns the communication between processes, and if it should be dissected from the process itself, then it belongs in the kernel." Apparently, participants in the Linux community have an idea of what they believe belongs in the kernel and what not. However, the definition, as provided by the respondent, does create much room for interpretation and ambiguity. For instance, when should communication between processes be dissected from the process itself? And when does software have to do with hardware and when not?

The second problem is that of coordination between modules and across communities. The president of Linux International presents what he feels is a good example of this problem, namely, the lack of coordination between the Gnome and the KDE community. Gnome and KDE are the two most popular open source desktops for Linux. One of the differences between the two desktops is that they are based on different libraries. Gnome is based on GTK+ and KDE on Qt. This creates all sorts of problems, for instance, 'Should I base my graphical Web application on GTK or Qt or both?'[8] Or, 'What happens when I switch laptops and log in with KDE and then with Gnome?'[9] The problem of coordination not only exists between communities, within communities modularity has created the same type of problems. A core member of the Gnome foundation explained that one of their biggest problems is "that every application in the Gnome desktop uses different fonts."[10] This results in a program in which a document has different fonts on screen than it has when printed.

The creation of roles

A second mechanism to decompose the complexity in open source communities is through the creation of specialized roles. The activities that need to be performed in the communities are different. Examples are the actual creation of new source code; testing the software in all sorts of settings; finding, reporting and fixing bugs; translating software into different languages; writing manuals and other documents; and creating and maintaining support tools, like a website, CVS and mailing list. Most of these activities are separated from each other and similar activities are clustered in roles, which again are clustered in projects or even a sub-community. An example of these is the Apache sub-community that translates a wide variety of documents into languages other than English. Another example is the huge number of mailing lists in the Debian community. For almost every activity a separate website has been created, such as the list on which a broad spectrum of legal issues is discussed.[11]

The activities are clustered in a wide variety of roles, some of the most prevalent and frequently cited of which are *credited developer*, *maintainer*, *contributor*, *user* and *release manager*.

According to Edwards (2001), most open source communities have a *maintainer* who is the central person of the project and, as such, is responsible for it. According to Raymond (1999b), there are three ways to become the maintainer in a community. The first is to start a community. By starting a project one is automatically recognized as the project's maintainer. This is how Linus Torvalds became the maintainer of the Linux kernel. The second is through title transfer. There are many examples in which maintainers hand over their title to another participant. Usually they do so because they have lost interest in maintaining that particular piece of software. The third way is through a lapse by the current maintainer. This was referred to in an earlier chapter and works as follows. A participant improves the original code and asks the maintainer to include the improvements in the software. If a minimum period elapses without the developer receiving a response from the maintainer, the participant can decide to take over the project and announce himself or herself the new maintainer of the software.[12] The new maintainer announces the transition in a number of different forums to ensure that everyone in the community (i) is aware of the transition and (ii) has the opportunity to voice an objection. There is also a fourth way to become maintainer of a project, which Raymond did not mention, which is to *fork* an existing software project. The fork can be compared to a hostile takeover (McGowan 2001). In this scenario an individual or company makes a copy of the source code and starts a new project.

The *credited developers* are those developers who are named in the credit file, the project list or the maintainer's list available in a particular open source community (Moon & Sproull 2002). Credits are intended to signal the reputation of developers, both within a particular community and to people outside the community. The credited developers and the *contributors* provide improvements to the project. The difference is that the contributors are not mentioned in the credit file of the software. However, their names can be found in the archives of mailing lists and in the history files of the software.

Users in open source communities come in many types. The least skilled users do not develop source code or make improvements. They just use the code and are therefore on the outskirts of the communities. Skilled users are likely to take a more active role. They might solve the problems of other users and report bugs to the community. They could gradually move up the ranks and become a contributor or credited developer.

Some communities have a *release manager* (also known as 'release coordinator,' 'release dude' or 'head beekeeper').[13] The release manager develops a schedule and defines deadlines for the release of a new version of the software and is responsible for getting that schedule accepted by the other community members. Furthermore, the release manager facilitates coordination and communication within the community to ensure that the deadlines, as defined in the schedule, are met.

An example: Apache

The Apache community provides an example of how complexity is decomposed in the communities. One of the first things that can be observed from the Apache community website is that there is not a single Apache community. Instead, the Apache website lists 19 projects.[14] Each of these projects has a separate website and behaves as a relatively independent unit. The projects are equivalent in scope and activities.[15] The modular design of

Apache is one of the reasons for the high level of decomposition. This structure enables developers who, for example, would like to have a tighter integration with say Perl or Java, to create a new and relatively independent project.[16]

The 19 projects themselves also consist of subprojects. Jakarta is an Apache project that "creates and maintains open source solutions on the Java platform."[17] The Jakarta project consists of many subprojects, one of which is Tomcat. Tomcat is a Web server based on Java. The Tomcat community consists of a number of relatively independent software modules.

The oldest Apache project is the HTTP server. Like the other projects, the HTTP server consists of many subprojects. These sub-communities are not divided based on the software, but rather, based on the activities that need to be performed. The sub-communities are Docs, Test, Flood, Librapreq and Modules.[18] The Docs sub-community is short for the documentation project. This project creates and maintains the Apache documentation.[19] The documentation project is again split into a number of subprojects, one of which is translation. Currently, there are seven languages into which the Apache software is translated. These translations differ in maturity: some have their own website, a mailing list and a downloadable translation of the Apache documentation.[20]

The emergent division of labor

The separation of activities and the modular design of most open source software contributes to a *division of labor* in open source communities.[21] This division of labor is not created by project leaders in communities or, for instance, the project management committees in the Apache community. Instead, the division of labor is spontaneous and unplanned. It is emergent, based on self-selection. Individual participants claim to create their own activities and select freely among the activities that need to be done.

The head of marketing in the OpenOffice community, for instance, explained how he wanted to contribute time and effort to the community, but did not know what he could do. He was never really good in programming. He did know that he was good in marketing and he felt that most communities, like OpenOffice, did not actively perform marketing functions. Marketing, however, is important to attract new participants to improve and maintain the software. Therefore, he took on marketing, creating a marketing sub-community in which other volunteers began to participate.[22]

A member of the BlueFish community provided another example. He described how he downloaded BlueFish from the Internet for free. In return he wanted to contribute something to the community. He lacked programming skills so contributing source code was not an option. He was skilled in reading and writing in English, and he realized that he could put this knowledge to good use. He picked up the idea of translating software from English into Dutch. He tried his hand at translating different software packages and BlueFish was the first program for which he felt successful. "I tried some programs and BlueFish was the first at which I succeeded. I translated a little part of the software and then joined the BlueFish mailing list. There I asked whether someone already translated BlueFish into Dutch. No one had." Other participants in the community thought the translation would be a good thing, so this participant started translating the software. Essentially, he created his own role, contributing time and energy to translate the software into Dutch. As of the time of the interview he was still the only person to maintain a Dutch version of the software.

In the above examples, both individuals created their own set of activities. They possessed certain skills, which they felt they could put to good use. In both cases they added new activities to those already being performed in the community. No one created these activities for them. In short, such creation of roles and division of labor are emergent. Individuals decide for themselves what activities they want to perform. If necessary, they create their own chores. "[A]gents choose freely to focus on problems they think best fit their own interest and capabilities" (Bonaccorsi & Rossi 2003c, p. 1247).

The division of labor ensures specialization and improves efficiency

The division of activities in open source communities into smaller chunks has two advantages. It promotes specialization and it results in an efficient allocation of individuals' time, effort and capabilities.

Specialization

Participants in open source communities tend to become specialists in a limited number of software modules or a limited number of activities (Von Krogh et al 2003b). A number of respondents, for instance, explained how they specialized in translating software strings into their native language.[23] A core member of the Dutch KDE translation group explained how he became a specialist in translating software: "I wanted to gain some experience in HTML and I have always been good in English and German." Furthermore, he explained that specialization is needed to properly translate the software. Many terms can be translated in more than one way. However, the translations need to be consistent. For instance, the term 'application' can be translated as 'applicatie' or as 'programma.' It is not very important which of the two is chosen. It is, however, important to use the same translation throughout the software program. Specialization ensures that if people participate for a longer period of time, they come to know what has been agreed upon and will be more inclined to use the 'standard' translation.

Other respondents described how they truly dislike translating software. They would never do such a thing. "Translating software is an incredibly dull job!"[24] These respondents' different background partly explains why they enjoy working on other tasks. The project leader in the Python community, for instance, explained how he would lose interest if he were no longer allowed to program and improve the software. His background lies in software engineering and he wants to be challenged to create new and improve existing software.

A growing number of programmers in open source communities are being paid to participate (Hertel et al 2003). They are employed by organizations to make sure that certain open source programs work. The software needs to be stable and system crashes must be kept to a minimum. Ideally, the software should be up and running 24 hours a day, 7 days a week. This is especially true for companies like Yahoo and CNet, which base their business model on providing content on the Internet. Both companies use Apache to run their websites. One programmer at CNet, who is also an active member in the Apache community, explained, "We are not experts in coding, but we bring in something else… We contribute in giving cases or certain situations where Apache can be improved, based on real business situations." Later in the interview he explained that this is why he himself specialized in writing software bug reports and fixing them. Other communities have similar contributors: "My main interest is to make PostgreSQL stable, as I need it for my work. So I devote most of my time to fixing bugs."[25] Both respondents have specialized in finding and fixing bugs.

The presence of specialization is also supported by a finding in Mockus et al. (2002). In their research, they performed a quantitative analysis of members in the Apache community. They found that of the top-15 bug reporters in the Apache community only three were also core programmers. Thus, in general, the participants who report bugs are different from those who contribute code. This again supports the claim that specialization is present in the Apache community. Participants focus on a certain group of activities.

Efficient allocation of time, effort and skills

Closely related to the advantage of specialization is the efficient allocation of time and effort. Many participants become specialists. This means that they can perform their activities more efficiently. For communities as a whole this results in an efficient allocation of time and effort. "I could start programming, which is totally new. I would do very simple things… I would not contribute anything truly valuable. I stick to translating, which is more valuable because I have a feel for languages."[26] As the statement illustrates, this respondent feels it would be inefficient to start learning how to program software. He wants to work on activities that best fit his interest and capabilities. Not only is this efficient for him, but it is also efficient for the community as a whole; the activities in the communities are matched with the capabilities of the individuals in question. If someone possesses language skills, that person should use those skills and preferably not start to perform a different kind of activity for which others are more highly skilled.

The observation that time, effort and skills are efficiently allocated is in line with the earlier-mentioned finding of Mockus et al. (2002) regarding the top bug reporters in the Apache community. Presumably, the core programmers in Apache are highly skilled programmers. For the community, it is most efficient if these participants spend as much of their time as possible creating new and maintaining existing source code. Therefore, from a community-level perspective, it is most efficient if they spend as little time as possible on other activities, like reporting bugs, translating software or answering dumb questions on a user mailing list. The reason they should refrain from these activities is because (i) their time and effort is better spent elsewhere and (ii) other, less skilled, programmers can also perform these activities. The fact that only three of the top bug reporters were also core programmers seems to indicate that core programmers indeed spend less time reporting bugs and more time writing new source code.

A Linux developer's portrayal of highly skilled programmers as snobs also supports this observation: "There are people who know practically everything. They won't react to trivial matters."[27] The respondent uses the term trivial matters to refer to users who ask easy questions on mailing lists. Such questions could be called trivial because users and less skilled programmers are able to answer them. And indeed, they frequently do: "Users also have many questions. Other users frequently solve them."[28] Again, the fact that highly skilled programmers refrain from answering these questions is efficient. It leaves them more time to improve the software, an activity that is better performed by them than by users and less skilled programmers.

Open source communities and learning

The structure of open source communities is frequently typified as an onion. Graphically this structure is depicted in figure 10.1 (based on Crowston et al 2003b, Nakakoji et al 2002, Van Wendel de Joode et al 2003). Roles are distributed among the layers of the onion. This does not mean that individuals are restricted to just one particular role and thus to one layer of the onion (Nakakoji et al 2002). The roles do, however, provide a structure for understanding how activities are divided among the individuals in open source communities. Furthermore, it helps explain how and why new entrants in the communities learn.

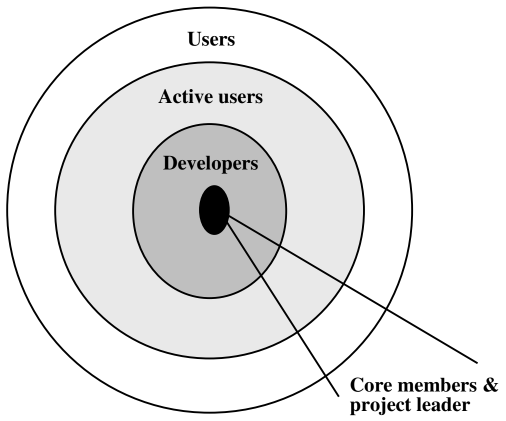

Figure 10.1 – The onion model in open source communities

On the outside layer of the onion are people who *passively use* the software (Crowston et al 2003b). They like the software and have downloaded a particular version of it. They might find certain bugs but do not go so far as to write a bug report. They might become a member of the mailing list because they want to be notified about important happenings in the community. Users in this layer are typically not active participants (Van Wendel de Joode et al 2003).

Moving inward we find *active users*. These users perform fairly simple activities like reporting bugs. Sometimes they also fix them. Other tasks performed here are, for instance, answering questions on mailing lists and helping people install open source software on their computers.

Developers form the next layer. Many different tasks are performed in this layer, but it "typically includes people who have implemented several patches and would know how to solve certain bugs."[29] The main activities in this layer are the development of new code and the maintenance of old/proven code.

The *core members* and the *project leader* are at the center of the onion. In many communities the core members and the project leader are responsible for the biggest part of the lines of code. Research has shown that the top-15 developers in the Apache community wrote more than 80 percent of the Apache software (Mockus et al 2002).

The outside layers as learning environment

One of the most important aspects of the onion model is that the users[30] of open source communities are not considered to be outside the community. In many communities users can, and they actually do, perform activities, like reporting and fixing bugs, asking and answering questions on mailing lists and helping other users solve their problems. The fact that users in open source communities are considered part of the community and that they perform useful activities makes them a valuable source of input in the development of open source software (e.g. Von Hippel 2001).

Users become involved in the communities for two main reasons. The first is that the organizational boundaries are highly permeable. It is, for instance, easy for users to join a mailing list and they need little knowledge to do so. Furthermore, users need not be a member of a mailing list to, for instance, report a bug. The second reason is the presence of many activities that are highly visible. Anyone can become a member of a mailing list and merely monitor the e-mails sent to that list. The e-mails provide users with an easily accessible learning environment which enables them to passively absorb knowledge from other participants (Edwards 2001).

One of the respondents in this research is currently a member of the Apache Software Foundation. He still remembers how he became involved in the community. "I started reading the mailing lists and slowly I started reporting bugs when I spotted them and a little later I was contributing small bug fixes that grew bigger and bigger." Fairly simple activities allowed him to get acquainted with the practices and processes in the Apache community: "That way I also picked up the etiquette used in Apache."

Mechanisms to achieve coordination between activities and modules

The fact that open source communities are divided into many separate modules results in a question about how activities are aligned across the modules. Obviously, some form of alignment and coordination is needed, because the activities performed in each part of a community should not result in software that is incompatible. Consider, for instance, an example given by a maintainer in the Linux community. In an interview this maintainer explained that Linux must be localized to operate on different systems, like the Power PC, Spark and Intel. The Linux kernel differs in the parts written for a specific type of hardware. These parts are maintained by relatively small groups of programmers. These groups have to ensure that their part of the program remains aligned with the other parts of the Linux kernel. The question is 'What ensures that the software works?' In other words, how is coordination achieved across the sub-communities and subprojects? A number of mechanisms are reported to have a role in creating and maintaining these linkages. Two are mentioned and briefly discussed here. These are the presence of the distributions and of participants who are known to have a good reputation.

Distributions

Distributions perform an important role in ensuring that the activities and modules are aligned. They do so in a number of ways. Companies like Red Hat and SuSE create a commercial distribution from open source software. To create the distribution they hire programmers from a wide variety of communities. Within open source communities they might not collaborate or coordinate their activities. However, since they are working at the same company they are forced to collaborate. They become the linking pins in open source communities. The vice-president of the FSF argued, "Red Hat also does a good job of ensuring communication and collaboration between projects, since they hire people from many different projects and make them work together."

Another way distributions create linkages between projects is with their primary activity of collecting software developed in different projects and combining it and ensuring that it operates and is compatible on one system. This means they have a task to ensure that programs within one distribution run smoothly. The programmers hired by a company like Red Hat install multiple applications on one computer and run many tests to verify that applications do not conflict. If they do they make patches to solve the problem.[31]

In the Debian distribution the programmers have written a policy manual to ensure that the programs within the distribution are compatible and easy to install. Essentially, the policy manual prescribes relationships between programs in the Debian distribution. There are five such relationships: *depends*, *recommends*, *suggests*, *enhances* and *pre-depends*.[32] For every program, Debian maintains lists of programs on which the program depends, which are recommended, etc. This categorization provides users an overview of software that is needed or recommended to assure the proper functioning of a software program.

The level of reputation

Another mechanism that ensures collaboration between modules and activities is the presence of highly reputable programmers. In most cases, the reputation of highly respected participants is not limited to a small fragment of a community. Communication media like mailing lists and news sites frequently report on highly skilled programmers who have earned credits. They are frequently interviewed, which creates visibility across the boundaries of communities. The opinions of these respected developers "are often sought on planned work or changes, as these developers are expected to be familiar with what's going on in the community as a whole."[33] A good reputation works across communities. It allows highly respected participants to get things done, not only in the community they are active in, but also in other communities. "Jeremy Allison of Samba enjoys so much respect that he can talk to the Linux people and the Linux people will listen to him. He can ask them to develop a piece of code… they understand that he knows a lot on how these systems should work together."[34]

This statement does not mean that others automatically listen to everything a well-respected participant has to say and that they will perform everything they ask them to do. Rather, participants pay more attention to what highly respected developers have to say and to what they contribute. Therefore, such developers are able to get things moving, even when they are not a part of that particular community.

Conclusion

This chapter first observed that open source communities face a high level of complexity. The larger communities consist of thousands of active contributors and many more inactive users. They have to align their activities with a great number of software programs, most of which consist of millions of lines of source code. Moreover, there are usually many different versions of one and the same software program. One strategy to deal with this complexity is by dividing the software into relatively independent modules and by dividing the work into clusters of activities (i.e. roles).

The division of activities and the modular design of the software contribute to a division of labor. The division of labor is emergent: individuals choose for themselves what they want to work on. This division of labor results in a high degree of task specialization, which combined improves the overall efficiency of the communities.

However, there is also a downside to the division of labor and the high number of modules. The structure promotes independence, but coordination is needed at some point. The activities of the participants need to be coordinated and the interfaces between the modules managed. Two coordination and management mechanisms were identified in this chapter, namely, the presence of respected participants and the coordinative role of the distributions. Nonetheless, the overall image that remains is one of redundancy and waste. Individuals work on the things they enjoy and interaction with other members is frequently absent.

Finally, the chapter argued that the structure of open source communities can be compared to the layers of an onion. The easier tasks, like reporting bugs or solving user problems, are located on the outside layers, and the more difficult and challenging tasks are closer to the center. The outside layers of the community constitute a learning environment. In these layers participants learn about the software and the rules and norms of the community. As they learn they are quite likely to move inward and start to undertake more complex and challenging activities (Ye et al 2002).

Individual behavioral rules

This chapter presented a number of observations based on individual choices. For instance, the way in which activities in open source communities are identified and undertaken is based on the actions of individuals. Furthermore, their level of specialization is based on personal preferences, which in many cases means that participants stick to what they are good at. On the other hand, Ye et al. (2002) argue that in the course of time participants will undertake more complex and challenging chores. How can we understand these observations? What is the underlying logic that drives these individual actions?

A possible explanation is found in Hertel et al. (2003). They claim, based on extensive quantitative research, that a participant's desire to work on a certain activity is determined by three factors. Two of the factors are "the perceived importance of their own contributions for the subsystem" and "the perceived personal ability to accomplish the tasks" (p. 1175). These two factors can be translated into two rules that underlie individual actions.

"*Participants perform activities that they perceive they are able to complete successfully.*"

"*Participants perform activities of which they perceive their contribution to be important.*"

How the individual behavioral rules support the observations

The two rules create a trade-off. First, the simpler the task the more certain participants may be that they are able to perform the task successfully. The downside is that the simpler the task the lower the importance of that contribution, as perceived by other participants (see also Ye et al 2002). Participants are therefore triggered to perform the most difficult tasks which they perceive themselves able to perform. The differences in knowledge and skills among participants explain why they create[35] and select different activities to work on.

The trade-off also explains the other observations in this chapter. Highly skilled participants generally want to work on the more difficult activities, because they are considered to be more important. For the same reason, they ignore the simpler tasks. This is a first reason to presume the presence of specialization. A second reason is that specialization reduces uncertainty. Participants know they have previously completed a similar activity. Performing the same activities over longer periods of time reduces the chance of failure. It also minimizes the time and costs needed to perform the activity; that is, to create new functionality or maintain a module (Von Krogh et al 2003b).

In the course of time, as they have performed similar activities in the past, participants' knowledge and skills improve. When reassessing their personal ability, they might perceive their skill to solve tasks differently and decide to work on more complex tasks. This is a potential explanation as to why participants move inward in the communities.

Notes on chapter nine

[1] From his website: http://www.dwheeler.com/sloc/ (April 2004).
[2] From the Internet: http://www.linux.org/apps/index.html (April 2004).
[3] Based on an interview with a maintainer in the Linux kernel community.
[4] Based on an interview with a maintainer in the Linux kernel community.
[5] From the Internet: http://www.debian.org/devel/developers.coords (April 2004).
[6] From the Internet: http://shell.n.ml.org/n/OSS/ (April 2004).
[7] From the Internet: http://www.kerneltraffic.org/kernel-traffic/quotes.html (April 2004).
[8] From an interview with the executive director of Linux International.
[9] From an interview with the executive director of Linux International.
[10] From a presentation by Nat Friedman, June 2002 at Boston, user meeting.
[11] Based on an interview with a maintainer in the Debian community.
[12] Respondents indicated that the norms require developers to wait a certain period of time. This can vary from a weekend to more than a month, depending on the importance of the improvement.
[13] Adopted from an interview with the release manager of Gnome:
http://www.zdnet.com.au/news/software/0,2000061733,39118923,00.htm (July 2004).
[14] There are two more projects, namely the Apache Software Foundation and Conferences. They are, however, not really subprojects (also based on an interview with the president of the Apache Software Foundation).
[15] From an interview with a member of the ASF Board of Directors.
[16] From the Internet: an interview with two members of the ASF Board of Directors.
[17] From the Internet: the Jakarta website: http://jakarta.apache.org/ (April 2004).
[18] From the Internet: http://httpd.apache.org/ (April, 2004).
[19] From the Internet: http://httpd.apache.org/docs-project/ (April 2004).
[20] From the Internet: http://httpd.apache.org/docs-project/translations.html (April 2004).
[21] Koch reports that open source communities have a high division of labor. Koch S, Schneider G. 2002. Effort, co-operation and co-ordination in an open source software project: GNOME. *Information systems Journal* 12: 27-42

[22] This account is based on an interview with the head of marketing.
[23] From interviews with two respondents. One translated KDE into Dutch, the other translated BlueFish into Dutch.
[24] From an interview with the maintainer of BlueFish.
[25] From an interview with a core member of the PostgreSQL community.
[26] From an interview with one of the core members of the Dutch KDE translating group.
[27] From an interview with a Linux kernel developer.
[28] From an interview with one of the project leaders of Lilypond.
[29] From an interview with a member of the ASF Board of Directors.
[30] In itself the term 'users in open source communities is misleading and inaccurate, as almost every participant in open source communities also uses the software. This means that everyone can be viewed as a user, not just the participants who are on the outside layers of the onion. For now, we will use the term, as it is in line with the terminology used by respondents and indicates a difference in participation between programmers who spend more than 10 hours a week on the development of the software and others who irregularly answer questions on a mailing list.
[31] From an interview with an employee from LinuxCare.
[32] From the Internet: http://www.debian.org/doc/debian-policy/ch-relationships.html (April 2004).
[33] From an interview with a programmer from one of the three BSD communities.
[34] From an interview with the editor in chief of the Linux journal.
[35] A respondent from the BlueFish community explained how he created his own new task, which is to translate BlueFish into Dutch. "I tried some programs and BlueFish was the first at which I succeeded. I translated a little part of the software and then joined the BlueFish mailing list. There I asked whether someone already translated BlueFish into Dutch. No one had."

CHAPTER TEN

EXTERNAL RECOGNITION

This chapter presents and discusses the seventh design principle, which is *external recognition*. The first section discusses what is meant by external recognition and why it is important. The second section looks at the wide acknowledgement of open source communities and the fact that open source software is adopted by a large variety and number of organizations. However, it is unclear whether external authorities acknowledge the processes and mechanisms in open source communities. It is said that in some respects the communities are organized completely different from many software development companies. The question is whether authorities will support and enforce the fundamentals in open source. This question is complicated by the presence of metaphors and stories that have given rise to two major debates, both of which ignore many subtleties.

The first debate is whether open source software is more or less secure than proprietary software. The second debate concentrates on the question of whether open source is a stimulus for innovation or whether it kills innovation. The debates are characterized by two extreme positions with a middle ground appearing to be lacking. Furthermore, it is unclear what the outcomes of the debate will be and what resulting actions authorities will in the end undertake.

In a response to some of the critique from their opponents the developers in open source communities appear to have created and adopted a number of mechanisms that solve some of the perceived downsides of open source. The chapter identifies four of these mechanisms and discusses how they counter some of the critique.

The seventh design principle: external recognition

Communities are never completely isolated from external authorities. On the contrary, most communities analyzed in research on community-managed common pool resources have a physical location and are thus subjected to the rules of one or more countries. However, Ostrom argues that in many situations external authorities have difficulties in acknowledging and thus supporting self-organizing communities. In an article published in 1999 she presents an example from the Chitwan Valley in Nepal. She argues that an engineering design team recommended building a dam across a river, because this would allow farmers to irrigate their crops. This recommendation totally ignored the presence of 85 irrigation systems that already existed in the valley and that were successfully managed by local farmers (Ostrom 1999). What would have been the consequence if the dam had actually been built? It probably would have destroyed the 85 irrigation systems that already existed. It is this threat that is addressed by the design principle external recognition. The principle focuses on the fact that without external recognition the rules and mechanisms that are devised in self-organizing communities, and thus the capacity of these communities to achieve and sustain coordination, become very fragile. A lack of external recognition is likely to result in the end of a community and could lead to a depletion of a common pool resource.

The fact that a lack of external recognition is not uncommon was highlighted by Berkes: “The commons literature is full of examples of destructive state intervention which eliminates or stifles existing local institutions and prevents self-organization” (Berkes 2000, p. 3). The thing is that no community, no matter how robust, can solve “all the collective action problems they face, without drawing on external resources and facilities” (Tang 1992, p. 125).

One of the reasons why external authorities fail to acknowledge and hence support local and self-organizing communities is that communities use local knowledge and lack central direction (Berkes 2000, Ostrom 1999). Central, that is, external, authorities are much more inclined to use ideas and knowledge that are generally accepted and scientifically proven. Therefore, it is not unlikely that the processes and mechanisms used in self-organizing communities, which are based on local and unproven knowledge, will conflict with generally and scientifically accepted ideas (Berkes 2000).

The lack of recognition of locally devised processes and mechanisms and its potential disastrous effects on the future of a self-organizing community relates to each of the design principles in research on common pool resources. Tang (1992), for instance, argues that in water irrigation systems penalties like “loss of rights to water and incarceration” (p. 31) can be efficient and effective sanctioning mechanisms. They are, however, worthless if external authorities do not support them. Another example concerns the boundaries erected to protect a resource and community against ‘outsiders.’ What is the value of a fence when, for whatever reason, federal authorities do not recognize the location of the fence or the entire concept of a fence? Does this mean that anyone can now enter the common pool and appropriate resources?

From the outside looking in

There is one major difference between this and the other design principles, namely a difference in focus. The previous design principles were clearly internally focused, on the community. They resulted in mechanisms and processes used by community members. The design principle external recognition focuses on external authorities and questions the ability and willingness of external authorities to acknowledge and support the mechanisms present in the communities. The idea is that an absence of external recognition will make a community fragile, no matter how robust its internal structure may be (Ostrom 1990).

This difference in perspective also becomes visible when we consider the type of conclusions that are drawn from the analysis of this design principle. The conclusion when external recognition is absent is not that the communities should change their structure. Instead, research on common pool resources appears to suggest that external authorities should begin to understand and acknowledge the processes and mechanisms and institutionalize them in laws and other forms of regulation. Obviously, communities do have mechanisms and tools available to influence and change the perceptions of external authorities.

External recognition for open source: two extremes

Currently, open source software is gaining an important position in the software market. This means that more and more companies and governments are adopting open source to support their internal processes. Furthermore, they pay programmers to develop and improve open source software. Essentially, open source software and open source communities are moving into the physical domain.[1] This has the consequence that they are – if they ever were – no longer isolated from existing legislation and competitive pressures. This also has the consequence that they can no longer be dismissed as nerdish hobby clubs unworthy of attention. The increasing embedment of open source communities in the physical domain, in a commercial market governed by legal rules and profit-seeking entities, leaves both companies and governments one unavoidable fact: 'Their presence cannot be ignored.' And indeed they are not, as the introduction to this book argued. More and more organizations are deciding to adopt open source software and participate in the development of the software. In that sense, open source communities are receiving external recognition. Contrary to many of the communities studied in research on common pool resources, open source communities are highly visible and their presence does not go unnoticed.

The fact that companies and governments cannot and do not ignore open source communities means that they have to decide how to deal with them. They must choose whether or not to support the processes and mechanisms present in open source communities. Organizations face many dilemmas in this, because there are many situations in which the processes and mechanisms in the physical domain are different and sometimes even in conflict with those in open source communities. Should, for instance, source code be something that can be kept private? Should an organization download and adopt source code that is not bought from a company, but rather is assembled by people whom the organization will never meet and who live around the world? Will an organization follow through and write and send a bug report when it encounters something that does not work? In short, will organizations recognize, accept, preserve and protect the processes and mechanisms of open source communities? Basically, two scenarios are conceivable.

The first is that organizations acknowledge and support many of the mechanisms and processes in open source communities. This could result in their institutionalization and an improvement of the robustness of the communities. Governments could, for instance, acknowledge the presence of open source licenses like the GPL and the BSD license and they could codify the licenses. They might even make sure that the licenses are enforced. This would substantially increase the impact of the licenses and thus improve the robustness of the boundaries of open source communities. However, codification of open source licenses could also result in a software market dominated by open source software, the outcome of which is difficult to predict. Is a market dominated by open source software better than a market consisting of large companies and proprietary software? The answer to this question cannot be predicted either (e.g. Van Wendel de Joode et al 2003).

The second scenario is the complete opposite. In this, not unlikely, scenario, organizations undertake actions and make choices that render open source communities extremely fragile institutions. Regulations concerning software patents is one of the most serious threats facing open source communities. Software patentability could destroy the future of open source communities. Yet the codification of software patents is not irrational. In fact, it is believed to stimulate innovation (Cowan & Harison 2001, Harison 2002, McGowan 2001). Liability is another such threat. The question is whether programmers in open source communities can be held liable for any damages related to open source software downloaded from the Internet for free. Should a user be able to sue the programmer who wrote the software or the company the programmer works for? If so, this could pose a serous threat for the future of open source communities.

The current debate

The problem facing open source communities is not that they are not recognized, as many governments and companies acknowledge their presence. Neither is there a lack of opinions about open source and its effects on the software market and society at large. The problem is to decide how to act on the current situation and debates surrounding open source. This problem becomes even more evident when we realize that the current debate about open source software and open source communities is riddled with metaphors and stories (see for a more elaborate discussion on the pros and cons of metaphors and stories Van Eeten 1999, Van Eeten & Dicke 2004). The problem with these metaphors and stories, as we will see, is that they try to present a reality in which there is only an 'either or' choice: either adopt strategy A or strategy B. Either open source software is inherently good or it is inherently bad. There is no middle ground. Governments and organizations should support open source software or they should refrain from using open source software and warn other organizations of the dangers of open source.[2] Both the proponents and opponents of open source have tried to paint this picture of reality, effectively distracting attention from the underlying logic and opportunities and strategies available with regard to open source.

The next sections present two debates that receive much attention and are addressed and discussed on many news forums on the Internet. They are the security of open source software and the level of innovation in open source communities. For each debate the two extremes are presented. The aim of the sections is not to argue that these are the only ideas and opinions, as there are also more refined and sophisticated views. The claim is that the debates and the extremes in these debates are very visible and receive a lot of attention. As the number of quotes demonstrates, they do dominate much of the discourse on open source and as such they are relevant and in need of further analysis.

Open source and security

One of the debates addresses the level of security in open source software: Is it more or less secure than proprietarily developed software? Both the opponents and proponents focus on the development process in open source communities, yet their conclusions are radically different. The opponents argue that the development process in open source is an open invitation for anyone to include backdoors in the software and that the software is thus inherently vulnerable and insecure. Proponents focus on the same characteristics but reach an entirely opposite conclusion, namely that open source software is inherently more secure than proprietarily developed software.

Is the development process of open source software inherently insecure?

The opponents of open source software claim that open source is vulnerable and insecure. "The vulnerabilities are there. The fact that somebody in the middle of the night in China who you don't know, quote, 'patched' it and you don't know the quality of that, I mean, there's nothing per se that says that there should be integrity that comes out of that process."[3] This quote is from an interview with Steve Balmer from Microsoft. In the interview Balmer explains why companies and individuals should refrain from using open source software. One of his arguments is that open source software is insecure and that the development process is inherently vulnerable. In contrast, Microsoft has a methodology, an approach and a testing process, which results in a "sustained and predictable level of quality."

Dan O'Dowd wrote a series of white papers on security in open source software. According to him, "It is ridiculous to claim that the open source process can eradicate all of the cleverly hidden intentional bugs when it can't find thousands of unintentional bugs left lying around in the source code."[4] It is thus irrational to believe that a community can discover all mistakes in software. He feels there is proof for this claim, because "despite the 'many eyes,' new security vulnerabilities are found in Linux every week in addition to dozens of other bugs."[5] The claim is that open source communities have adopted a random and uncontrolled development process, in which anyone can intentionally hide bugs to sabotage critical infrastructure systems. "The chance of someone infiltrating a backdoor into Linux is close to 100%."[6] Until Linux has been certified at a certain level of security, it should not become part of any US defense systems, because "our soldiers should not be asked to trust their lives with it."[7]

Is open source software per definition more secure?

The proponents of open source software claim that open source software is inherently secure. The claim is that the "generally accepted notion among IT professionals [is] that Linux is more inherently secure than Microsoft's professional operating system platforms."[8] Open source software is claimed to have fewer security breaches and security problems than proprietarily developed software, particularly the software from Microsoft. One of the open source software programs that is said to be very secure is Linux: "Security Breaches [are] Rare in Linux Environment."[9]

Security in Linux and open source is based on the idea that "'[w]ith enough eyeballs, all bugs are shallow.' This Linux axiom points to the fact that when a bug becomes an issue, many people have the source code, and it can be quickly resolved without the help of a vendor."[10] Thus, the availability of the source code in open source communities is said to enable many people to analyze the source code and fix it where appropriate. The openness of the source code explains why in open source communities problems are fixed more rapidly: Open source software and free software systems "fix problems more rapidly, reducing the time available for attackers to exploit them."[11] Problems and security breaches in, for instance, Windows are argued to take much longer to fix, as evidenced by a security flaw in Internet Explorer. "This particular vulnerability has been known about for more than 9 months, said David Endler, director of incident response for security company Tipping Point."[12]

Another difference between open source software and proprietarily developed software, is the monetary goal. Companies need money to be able to improve the software. In a quest to earn money, companies are likely to downplay the importance of security. "Many companies are strapped for cash, and they omit the security review entirely. They figure that security is something they can go back and fix after they have enough customers for it to matter, and when people start reporting problems."[13] In contrast, participants in open source communities are said to have little interest in money and attracting customers. They can and will spend the time needed to create software that is secure, which explains why "developers ranked Linux's security roughly comparable …to Solaris and AIX, two very secure operating systems long trusted by large enterprises, and above any of the Windows platforms by a significant margin."[14]

The core of the debate: openness

Analyzing the debate it becomes evident that both sides focus on openness and on transparency. The question is 'Does openness contribute to security or is it detrimental to security?' Both the proponents and the opponents focus on this aspect to claim that open source software is either secure or insecure.

The opponents claim that openness is nice, but it provides an opportunity for anyone to contribute bad source code. The focus is on two aspects, namely (i) the absence of organizational boundaries and (ii) the possibility to intentionally add source code with traps and backdoors, so-called 'Trojan horses.' There is no real check on who contributed the code and where it came from. Contributors are from all over the world and some contributors could have bad intentions. How can one be sure that no one has included source code that includes backdoors? Surely the availability and openness of source code is not a sufficient protection. Too much openness is unwanted. Protection measures are needed to rigorously analyze each piece of source code that is added to the program and to verify the institutions and backgrounds of the persons contributing source code. Essentially, the claim is that too much openness results in vulnerability. The development process in open source communities is basically labeled naïve.

The proponents of open source software seem to ignore this potential danger. They focus on the advantages that openness of the source code offers. They say that because the source code is open every backdoor can be discovered. The idea is that the 'proof of the pudding is in the eating.' Organizational boundaries or verification of contributors' intentions is of less importance because the contributed source code can be analyzed and removed if foul play is discovered. The fact that openness of the source code allows anyone to analyze it is said to result in two advantages. First, popular open source programs are able to mobilize thousands of bug spotters, because every user is a potential spotter of a mistake or security breach. In companies, only a limited number of people have access to the source code and only they can actually discover security breaches. Second, transparency in the open source development process stimulates participants to do things right. Everything is out in the open and so reputations can easily be damaged. The openness creates a 'hall of blame' for contributors who make mistakes.[15]

Untangling the debate: what are the consequences of openness?

What can we say about the openness of software and the development process, and its claimed inherent consequences for the resulting software? Is it possible to claim that open source software is inherently more or less secure than proprietary software? Both opponents and proponents seem to suggest that there is a direct link between openness and security and that definite conclusions can be drawn.

When analyzing the debate one could say that each camp focuses on a different aspect of openness. In any case, the debate is much more complex than suggested and it could be held in a more constructive manner. Arguably, open source software can be extremely secure, as it is able to mobilize a huge development effort. On the downside, it is unclear whether the communities are always able to motivate people to truly verify every piece of source code, especially in large and complex software programs. Who is to say that there are no backdoors in popular open source software programs that contain thousands of lines of source code? It is not unlikely that they do have backdoors. Just as it is not unlikely that there are backdoors in proprietarily developed software. If we accept backdoors as given, then the question becomes which is more secure, open source or proprietary software.

Perhaps in certain situations and for certain applications more control over who is who is preferable. It is quite likely that openness is not the answer to every problem. For instance, an ASF member explained that in certain situations participants in the communities practically invite others to break into the software:

> If they have spotted a bug that can cause security problems they will give it back and say that there is something wrong but they will not tell you how to fix it, nor will they give the exploit. I don't need a 15 year old to own my site because he abuses a bug, when I haven't had time to upgrade my website to a newer version of Apache… If you are running a business you may not have the time to upgrade. The open source community doesn't always realize that.

Furthermore, it is just as likely that in certain situations companies should open up their source code and introduce more transparency. Attention might therefore be better focused on understanding what conditions and circumstances influence the choice of openness. This could prove to be a more constructive exercise and result in a better grasp of the actions and strategies that companies should undertake given their circumstances.

Open source, commercial endeavors and innovation

A second debate focuses on innovation. The question is whether open source communities are able to be innovative and whether the software is truly state of the art. Related to this debate is the issue of intellectual property rights. Are patents and copyright needed to create incentives for companies and individuals and to ensure innovation? Yes or no? Do patents on software stimulate innovation or are they a hindrance to innovation and should they be abolished? The opponents of open source software argue that companies are needed to create innovation and that patents are an essential part of any viable business model.

Open source as communism and a threat to innovation

"Linux is communism"[16] reads the title of an article on Microsoft's stance towards Linux and open source. The president of the SCO Group holds a similar position when he argues that open source software "is some sort of communistic plot against America."[17] Steve Ballmer from Microsoft explained why he compares Linux to communism: "And it had, you know, the characteristics of communism that people love so very, very much about it. That is, it's free."[18] Thus, one reason to call open source 'communistic' would be that it blocks commercial endeavors. It is free.

A second and perhaps more important reason is that open source software is said to undermine and attack software patent laws and copyright laws. "'Open source is an intellectual-property destroyer,' Allchin said. 'I can't imagine something that could be worse than this for the software business and the intellectual-property business.'"[19] The president of the SCO Group made a similar claim. He argued that the FSF, Red Hat and others in the open source world consider private benefits a hindrance, an impediment to the improvement of software. A statement made by him provides an explanation as to why SCO started its lawsuits against IBM and two commercial users of Linux: "'We're fighting for the right in the industry to be able to make a living selling software,' McBride told the audience. He compared this right to the ability 'to send your children to college' and 'to buy a second home.'"[20] He argued that open source is strictly opposed to the idea that companies should be able to earn a profit from the development and distribution of software. Furthermore, so his argument goes, leading figures in the open source world have "spent great efforts, written numerous articles and sometimes enforced the provisions of the GPL as part of a deeply held belief in the need to undermine or eliminate software patent and copyright laws."[21]

Intellectual property rights are considered by many to be essential for companies to stay in business and make a profit. Companies must therefore protect themselves against infringement and theft. "Any software or hardware vendor out there that owns intellectual property knows that they have to protect it. If you let others steamroll your intellectual property, then why or how are you going to stay in business?"[22]

The opponents of open source software continue the argument by stating that companies are needed because they are the ones who spur innovation. It is they who generate the funds to create new and innovative software. Open source software is said to prevent commercial endeavors and therefore kill innovation. "[O]pen-source software kills software innovation because it effectively, over time, kills the funding for it. Much of the innovation we have today comes from proprietary companies."[23] One of the reasons frequently used to explain why open source software will never be innovative is because innovation requires investments. And only companies can make such investments. "Innovation requires a level of risk, and the returns will never justify the risk when the playing field has been leveled by an open source philosophy. Even the most successful open source products that already exist tend to be imitations of successful commercial products."[24]

Open source as an endless source of innovation

Obviously, the proponents of open source software hold an opposing view. They argue that open source software does provide room for innovation. They even say it stimulates innovation. "[T]he recent rise of the Internet and Open Source software has created the most innovation any of us have seen in our life times."[25] There are a number of reasons why open source software is said to result in more innovation than proprietary software development.

First, open source software is said to create a level playing field in which every company has the same opportunities to make a profit (e.g. Markus et al 2000). Open source software is said to support "a free market economy where consumer choice is based on full information about competing suppliers."[26] Therefore, in a market dominated by open source software, companies must work harder to attract new customers. They can no longer sit back and relax, but are forced to continuously innovate and add value. "[T]hat's one reason that Linux on the desktop makes sense. It'll be good for Microsoft… They won't like it, but it will force them to innovate."[27] In other words, the proponents contest the idea that open source software prevents commercial endeavor and innovation. On the contrary, "Open source software adds even more competition to the software industry and thus increases the need for innovation."[28]

Second, intellectual property rights and especially software patents are said to hinder and prevent innovation rather than to stimulate it. "There is a risk that patents will strangle the life out of software innovation."[29] Patents are thus argued to stifle innovation. They are said to only be good for large companies and to form an impediment to individuals and small-sized companies that are creative and have innovative ideas. "Software patents stifle the innovation that the open source movement has helped foster by taking power away from the individual inventor and putting it into the hands of a few large or specialized companies that have the most patents and the most lawyers."[30] Richard Stallman, for instance, claims that new software must always be based on existing ideas, ideas created by others. He writes, "Nobody is so brilliant he can re-invent all of computer science, completely new… If you develop a nice new innovative word processor, that means there are some new ideas in it, but there must be hundreds of old ideas in it. If you are not allowed to use them, you cannot make an innovative word processor" (Stallman 2002, p. 84).

Untangling the debate: the effects of intellectual property rights

Both extremes in the debate focus on intellectual property rights: 'Are intellectual property rights on open source software a necessary incentive to ensure and stimulate innovation or are they detrimental to innovation?'

According to the opponents of open source software, intellectual property rights are needed to ensure innovation. The claim is that 'radical' innovations require high investments and therefore can be achieved only by companies. However, companies will not invest if they have no way to protect their inventions from third parties. The goal of IPR is to give inventors property rights on their technological advances (Nordhaus 1969) and to give them a right to exclude competitors from using and exploiting their invention (Arrow 1962). Opponents argue that open source software is bad and a threat to innovation because the participants in the communities are said to ignore and violate the property rights of companies. They are portrayed as thieves, as people who steal property belonging to companies.[31] Open source software is said to remove the incentive for companies to innovate. The SCO Group, for instance, argues that its intellectual property is an essential part of the Linux kernel, which means that its property is being used on millions of desktops worldwide. However, it receives no compensation for this use. It receives no licensing fee. SCO claims that open source participants have taken its property and illegally distributed it to millions of users.

The proponents of open source software focus on the same issue, but reach a completely different conclusion. They argue that property rights are no longer used as a means to stimulate new inventions and to protect innovative ideas. Instead, companies use property rights, particularly patents, to protect their market share, and as a result they file a patent on every idea, whether it is innovative or not. Many respondents in this research mentioned the incompetence of the US Patent Office to judge whether an idea is truly new and innovative.[32] The recent run on patents (Harison 2002) indicates some of the triviality of the patents being filed. Popular examples of trivial patents are the one-click ordering system of Amazon.com[33] and IBM's patent on multi-tasking.[34] The fact that innovative software programs are always based on the ideas of others means that individuals and small companies are unable to create innovative software. They must pay a fee to build on the patented ideas, which they simply cannot afford. This is claimed to block and prevent a major source of innovation.

Who is right?

The debate about patents, copyrights and their role in innovation is not limited to the software market. In a wide variety of sectors debates are raging about whether patents and copyright actually contribute to innovation or hinder it. The debate ranges from the music industry to biogenetics to the software market. Apparently, there is no clear-cut answer to the question of whether patents are good. There is probably no right or wrong. However, the proclaimed obvious relationship between companies, patents and innovation can be doubted. Companies like Trolltech are able to earn a profit from open source software and their software is innovative (Van Wendel de Joode et al 2003). Also, the huge investments made by companies like IBM and Sun in the improvement of open source software do imply that businesses can exist in a market dominated by open source software. Finally, many researchers of open source communities claim that the software is quite innovative (e.g. Von Hippel 2001, Kogut & Metiu 2001, Shah 2003). They needed no patent regime to make innovation possible.

It is also clear that patents do threaten the continuity of open source communities. The reason is the following: "The uncapitalized open source development model simply has no means to bear the transaction costs, licensing fees and risks that pervasive patenting entails" (Kahin 2002, p. 3). Individuals in the communities cannot be expected to be able to pay the licensing fees for patents if they develop and distribute open source software. They simply lack the money.

Mechanisms to solve problems and counter critique

One of the things that unites the two debates about open source is the way in which the opponents portray the communities and its participants. They are represented as thieves and communists who create software in a random, unstructured and uncontrolled fashion. No one should consider open source software as a viable alternative for proprietary software, because "[t]here is no company called Linux, there's barely a Linux road map."[35] The critique is focused on the absence of a viable business model. The claim is that there is no incentive or room for businesses in a market dominated by open source software. There is no business model in open source communities.

The absence of a business model is claimed to cause many problems. The previous sections discussed the alleged negative effects on innovation, but also more implicitly on security. Other negative effects are said to be lack of user support and quality control. If, for instance, something is wrong with Windows then users can blame Microsoft and pressure the company to solve the problem. However, there is no entity to which users of open source software can turn if they have a problem. "If there's problems and people do have security issues, I'm SteveB@microsoft.com, they know where to send e-mail and give somebody a hard time about it, and to the very best of our ability to get a response. None of that is true in the other world."[36]

To protect themselves against these accusations and to continue to attract new users participants in open source communities have copied the strategy of their opponents. Attempting to influence public opinion about open source, the proponents and especially the participants themselves work to discredit the proprietary software development model. The two debates summarized here demonstrate how proponents of open source software counter many of the arguments through a process of naming and shaming. Proponents of open source software, for instance, claim that intellectual property rights are merely instruments for companies to make money. They argue that closed source code results in qualitative inferior products; that companies' only goal is to make money and that openness is the only way to create secure software. Furthermore, open source communities provide much better user support compared to commercial companies, like Microsoft, in which problems take a seemingly endless amount of time to solve.

Neither do participants in open source communities limit themselves to naming and shaming. Instead, they appear to have adopted another strategy. They have created and implemented mechanisms to resolve some of the concerns in the debate. In other words, they have created mechanisms to deal with some of the potential negative effects of, for instance, openness. They have also created a number of mechanisms to deal with the potential threat of violating intellectual property rights. Many of these mechanisms display a certain level of institutionalization and aim to make the open source software development process and structures more formal and structured.[37] For a number of these mechanisms it is relatively obvious and straightforward that they were created in response to problems surfacing in open source communities. For other mechanisms this is less obvious. Some of these mechanisms are introduced below. The observed problem is explained and then the way the mechanism addresses, or tries to address, the problem.

Developer's Certificate of Origin

SCO's lawsuit 'against'[38] open source and Linux was first introduced in chapter four. SCO's claim is that Linux is based on source code that it, SCO, owns. Linus Torvalds has responded that the entire process of the creation of Linux is open and he claims to be sure that the origin of all the source code in the Linux kernel can be traced back to individual contributors. This is a direct result of the openness of the source code in the communities.

Linus Torvalds does acknowledge at least one problem, namely, that searching the archives for the origin of source code requires a lot of time and effort. "People have been pretty good (understatement of the year) at debunking those claims, but the fact is that part of that debunking involved searching kernel mailing list archives from 1992, etc. Not much fun."[39] He proposes a solution to solve this problem and ease the process of data retrieval. The solution is to "explicitly document not only where a patch comes from...but the path it came through."[40] This document is called the Developer's Certificate of Origin (DCO). A copy of it is presented in appendix D. According to Torvalds, "We've always had transparency, peer review, pride and personal responsibility behind our open-source development method. With the DCO, we're trying to document the process. We want to make it simpler to link submitted code to its contributors. It's like signing your own work." [41]

The document does more than simplify the process needed to discover where a certain piece of source code came from. It is quite likely that another goal of the document is to convince companies to use and adopt the Linux kernel, as the document provides an overview of the origin of the source code. "It's an attempt to address industry concerns over code origination within the Linux kernel… The DCO is a CYA [cover your ass] measure for Linux, and it may have a feel-good benefit for organizations that are deploying or planning Linux in their IT infrastructures."[42]

Similarly, the Apache Software Foundation asks prospective contributors to sign a document in which they state that they own the copyright on any piece of source code they contribute to the project. At the same time, they transfer their copyrights to the ASF.[43] The goal of this document is similar to that of the DCO in the Linux community.

Linux Standard Base and the Free Standards Group

One of the claims of opponents of open source is that it provides no opportunity for companies to make a profit. The absence of companies and the seemingly unorganized and random development process also raise another concern, namely, that open source software is unlikely to be compliant with software standards. Individual programmers are said to create software in the way they consider best or most appropriate. Allegedly, there is no incentive for them to develop source code that complies with prescribed standards. Furthermore, there is much potential for participants to diverge away from standards, which could result in all sorts of compatibility problems (this line of reasoning is similar with the line of reasoning in the introduction in Egyedi & Van Wendel de Joode 2003).

Companies that favor open source have responded to this critique by, essentially, creating a standard for Linux distributions, which is called the Linux Standard Base (LSB) and they have created the Free Standards Group (FSG). The LSB is the name of the organization as well as the resulting standard. The aim of the organization is "to develop and promote a set of standards that will increase compatibility among Linux distributions and enable software applications to run on any compliant system."[44] The standard was first released in June 2001. It "provides a way to ensure behavioral compatibility across Linux distributions and version releases. An application written to the standard will function the same across all LSB certified platforms."[45] To stimulate adoption of the LSB, the FSG was created in April 2000. The FSG is a nonprofit group that is supported by industry and which hosts the LSB workgroup (Egyedi

& Van Wendel de Joode 2003). The FSG also facilitates a LSB certification program. To provide greater confidence in the program, they have trademarked the name "LSB Certified." To become LSB Certified a company must run a number of tests on the software (Claybrook 2004).

Both the LSB and the FSG were created to ensure compatibility between Linux distributions and, probably equally important, to increase the credibility of Linux software. Essentially, software labeled LSB Certified is software which users can be sure has passed a rigorous testing procedure and complies with a certain set of standards. Thus, both the LSB and the FSG are efforts to increase the credibility of open source software and to counter some of the critique and doubts opponents have concerning open source software and its development model.

What is striking is that a number of respondents claim that open source communities have no problem with adhering to standards. Despite the huge potential to diverge and ignore standards, open source software programs like Apache comply with almost all of the standards that are relevant to them. The claim is that open source programmers "tend to take an almost perverse way of doing [software development] perfectly along the standard interfaces."[46] Apache is a well-known example of a community in which the software is said to comply with standards. The president of the FSF claimed:

> We wanted Apache to be a reference implementation of HTTP. We wanted to deliver a high-quality product: it should be fast and relatively bug free. The open protocol and open reference implementation helped by enforcing HTTP to become the standard Internet protocol. In a way this shows that we feel that compatibility with standards is the most important goal for the Apache project.

The presence of open source software like Apache, which adheres to many standards, could quash the need for certification programs like the LSB.

External representation: project leaders and foundations

One point of critique in the two debates is that open source communities have no room for companies: There is no business model. This means that "there is no company called Linux."[47] This lack of a company is claimed to result in a market in which users have no one to turn to if their software does not work. The validity of this claim can be doubted. Still, open source communities have implemented three strategies to deal with this issue. The underlying idea of each of the strategies is to create a certain level of institutionalization and to establish a visible contact point.

One of the strategies is the creation of foundations. Consider the Apache community and its historical development. At a certain point in time IBM decided to adopt the software and approached the core members of the community. Before IBM would adopt Apache, however, it wanted to make sure it would have a "contact point," "someone they could talk to."[48] To create this contact point and for other reasons, which were addressed in more detail in chapter four, the community in a combined effort with IBM created the Apache Software Foundation. Companies like IBM can now contact the foundation if they have questions or problems. The presence of the ASF also has another effect, namely, it gives the Apache community the appearance of a more formally organized and more structured entity compared to communities that lack such a foundation, despite the fact that the ASF hardly interferes with the actual development and maintenance of the Apache software.[49]

A second strategy is to appoint a project leader. Again, the goal of the project leader is not solely to act as a contact point for companies. In practice, however, some of the project leaders do spend a lot of their time communicating with companies. Consider the Debian community. A former project leader explained how it surprised him to be so little involved in technical issues. Instead, contacts with companies were one of the things that took much of his time. "For instance, with IBM I had contacts on shows. With Corel I talked about distributions, with IBM about support." Although the influence of project leaders on the actual processes and activities might be limited, the project leaders are easy-to-identify and approachable spokespersons for the communities. A project leader of the PostgreSQL community also described staying in close contact with companies wanting to adopt the software. He described one of his activities as marketing. "It is marketing in the sense of doing interviews for magazines… come up with slogans, feed questions from companies and give speeches… I answer technical questions for [companies] and keep them up to date with the latest headlines of what happens in the community."

A third strategy is the actual creation of a marketing department. One of the communities with such a department is the OpenOffice community. Its marketing department was created in a process similar to that described in chapter nine, where certain roles and activities are created based on the strengths of participants. In an interview, the head of the marketing department explained how he started to market the product and the community. He believed himself to be good at it and no one else was doing it at the time. Slowly other people began to show interest and wanted to do something similar, which resulted in the creation of the department. About his role, he explained, "I also coordinate and process all the requests we get from people who want to know more about OpenOffice." One of the marketing department's activities is thus to communicate with companies and encourage them to adopt the software. According to the respondent, companies also turn to the department if they experience problems or have complaints.

Preventing problems of liability: the use of disclaimers in open source licenses

Some believe that participants in open source communities should be liable and accept the consequences for the source code they write. This means that people and companies would be liable if it turns out that their contribution causes the software to be defective. The idea is that "software should be measured just like any other product. If it does harm, you should be held responsible. If you have not given it enough care then you are responsible."[50] This would mean that participants in open source communities are "accountable for negligent behavior. We are responsible for everything we do, also as citizens. If you step into the car and you cause a car accident then you are responsible."[51]

The question now is whether liability is truly applicable to open source contributors. Should users of open source software be able to sue programmers who participate in the development and maintenance of the software? Certainly the programmers purposefully contribute source code to the software. In many cases, however, they do not earn money for their efforts (Hertel et al 2003). Should they be viewed as software producers or are as mere hobbyists, who enjoy participating in a joint effort, sharing knowledge and demonstrating their skills. In other words, 'Should voluntary programmers be viewed as producers who enter into a transaction with users of the software?' Or, should they be viewed as hobbyists who enjoy creating software and do so with the best intentions and with no interest in others adopting and using the software?

It is unclear what the courts would decide if this issue were brought forward. In the meantime, participants in the communities have created a mechanism to minimize the chance of a lawsuit. They have added liability disclaimers to open source licenses. The disclaimers aim to release the producer of the software from liability. Almost all open source licenses include such a disclaimer at the end of the document. Consider the GPL. The license states that "parties provide the program 'as is' without warranty of any kind."[52] If the program turns out to be defective then the costs of all "necessary servicing, repair or correction"[53] must be assumed by the user and not the producer.[54]

Conclusion: are the communities under pressure?

The central question of the design principle external recognition is whether external authorities recognize and acknowledge the processes and mechanisms adopted in the communities. This chapter argued that companies and governments do acknowledge the existence of open source software as a viable alternative to proprietary software. But it is unclear whether they acknowledge the mechanisms underlying the open source development model. For instance, do governments and courts acknowledge the contents of the GPL? Does openness of the source code and the lack of organizational boundaries pose threats to the security of the software?

Irrespective of the outcome of the debates, participants in the communities have created mechanisms to protect against the perceived lack of external recognition and to counter some of the pressures being exerted on the open source development model. Four mechanisms were identified. The first mechanism is the Developer's Certificate of Origin. This document was created by Linus Torvalds to counter some of the threat of intellectual property claims and to take away some of the perceived negativity of the so-called 'random process' of software development in the Linux community. Every contributor now has to sign the document and attach it to source code they contribute. This way, the origin of the code becomes more visible and some of the concerns of companies are alleviated.

The second mechanism is the creation of the Linux Standard Base and the Free Software Group. The LSB aims to ensure compatibility between the various Linux distributions. Many distributions are currently available, for example, Red Hat, SuSE and Debian. One concern is that the distributions are not interoperable. To dispel this concern, the FSG created a procedure to test the distributions. Distributions that pass the tests become LSB Certified.

The third mechanism is the appointment of project leaders, the creation of foundations and the building of marketing departments. This set of mechanisms accomplishes a variety of goals. One is to create a contact point, a point of reference, for companies that want to get involved in open source. A critique of open source is the absence of a 'Linux company.'

Basically, the idea is that companies have no one to turn to if they have a problem or question. This set of mechanisms creates a visible and recognizable point of reference.

The fourth mechanism is the use of disclaimers. This last mechanism is perhaps less a response to the arguments used in either of the two debates. Instead, it is more a means to protect contributors from lawsuits if software turns out to be crooked.

The four mechanisms have two things in common. Whether the mechanisms were all created with this goal in mind is unclear, but each mechanism does counter some of the arguments made by open source software's opponents. Another commonality is certainly that they leave intact the basic principle, which was identified in many of the previous chapters, namely, that open source development processes are based on individual choices and actions. A certain degree of institutionalization is certainly taking place. But it does not interfere with the actual decisions and actions of individuals. The transaction costs have perhaps increased somewhat, but they are still relatively low. Whereas everyone used to be able to contribute source code to the Linux kernel by simply sending it to Linus Torvalds, they now have to sign a standard document and include it with their contribution. The costs involved in this process are relatively low. The same is true for the other mechanisms identified in this chapter. If anything, they strengthen the processes that were identified in the previous chapters in this book.

Notes on chapter ten

[1] The distinction between the physical and the virtual domain is adopted from a brochure for new subsidies published by IT*e*R (a Dutch research program on information technology and law) in April 2001 and produced by Paper Handling in The Hague.
[2] A white paper available at http://www.ghs.com/linux/manyeyes.html (October 2004) warns governments of the potential for terrorist attacks when open source software is used in sectors critical to national defense.
[3] From an article on the Internet: http://www.microsoft.com/presspass/exec/steve/2003/10-21Gartner.asp (July 2004).
[4] From an article on the Internet: http://www.designnews.com/article/CA435615.html (July 2004).
[5] From an article on the Internet: http://www.designnews.com/article/CA435615.html (July 2004).
[6] From the website: http://www.ghs.com/linux/manyeyes.html (October 2004).
[7] From an article on the Internet: http://www.designnews.com/article/CA435615.html (July 2004).
[8] From an article on the Internet: http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2907749,00.html (October 2004).
[9] This is the title of an article on the Internet: http://www.businesswire.com/cgi-bin/f_headline.cgi?bw.040802/220982285 (October 2004).
[10] From an article on the Internet: http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2907749,00.html (October 2004).
[11] From an article by David Wheeler: http://www.dwheeler.com/oss_fs_why.html#security (October 2004).
[12] From an article on the Internet: http://news.zdnet.com/2100-1009_22-5256297.html (October 2004).
[13] From an article on the Internet: http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2859555,00.html (October 2004).
[14] From the Internet: http://www.businesswire.com/cgi-bin/f_headline.cgi?bw.040802/220982285 (October 2004).
[15] See chapter eight for a more elaborate discussion of sanctioning in open source communities in general and on the hall of blame in particular.
[16] From an article on the Internet: http://www.theregister.co.uk/2000/07/31/ms_ballmer_linux_is_communism/ (October 2004).
[17] From an article on the Internet: http://news.bbc.co.uk/1/hi/technology/3537165.stm (July 2004).

[18] From an article on the Internet: http://www.theregister.co.uk/2000/07/31/ms_ballmer_linux_is_communism/ (October 2004).
[19] From the Internet: http://linuxtoday.com/news_story.php3?ltsn=2001-02-15-008-06-PS-MS (October 2004).
[20] From the Internet: http://news.zdnet.co.uk/business/legal/0,39020651,39115782,00.htm (October 2004).
[21] From an open letter by Darl McBride, available on the Internet: Mhttp://www.newsforge.com/trends/03/12/04/2024240.shtml?tid=85 (July 2004).
[22] From an article on the Internet: http://searchenterpriselinux.techtarget.com/qna/0,289202,sid39_gci931259,00.html?track=NL-301 (October 2004).
[23] From an article on the Internet: http://www.technewsworld.com/story/32154.html (October 2004).
[24] From an article on the Internet: http://www.embedded.com/internet/0009/0009ia1.htm (October 2004).
[25] From the Internet: http://www.linuxmednews.com/linuxmednews/969514607/index_html (October 2004).
[26] From the Internet: http://news.bbc.co.uk/1/hi/technology/3537165.stm (July 2004).
[27] From an article on the Internet: http://news.com.com/2102-7344_3-5195651.html?tag=st.util.print (October 2004).
[28] From the Internet: http://lpf.ai.mit.edu/Patents/chaining-oss.html (October 2004).
[29] From an article on Internet: http://news.zdnet.com/2100-3513_22-5342291.html (October 2004).
[30] From the Internet: http://www.alwayson-network.com/comments.php?id=P5141_0_3_0_C (October 2004).
[31] From a presentation by Lawrence Lessig in Amsterdam, June 7, 2002.
[32] One of the respondents was the editor in chief of the Linux journal.
[33] The patent number is 5,960,411. Its triviality is discussed on many places on the Internet. For instance, www.gnu.org/philosophy/amazon.html (October 2004).
[34] See, for instance, http://www.linuxjournal.com/article.php?sid=5087 (October 2004).
[35] From an article on the Internet: http://www.theregister.co.uk/2000/07/31/ms_ballmer_linux_is_communism/ (October 2004).
[36] From an article on the Internet: www.entmag.com/news/article.asp?EditorialsID=6004 (October 2004).
[37] An anonymous reviewer commented on a paper written by Egyedi and Van Wendel de Joode entitled "Variety: the tension between adaptability and interoperability of open source software." The article was sent for review to the journal *Computer Standards and Interfaces*. The reviewer wrote, "it might be useful to look at situations where the discursive process does not scale to solve the problems at hand and explicit processes, institutions, and instruments come into play." The reviewer named a number of examples, like the FSG, which is the Linux standardizing body, and a new process developed for Linux to track the originator of the source code.
[38] To be accurate, SCO has not filed a lawsuit against open source or Linux. Instead, it has filed suits against IBM and a number of corporate users of Linux.
[39] From an article on the Internet: http://news.com.com/Linux+contributors+face+new+rules/2100-7344_3-5218724.html (October 2004).
[40] From an article on the Internet: http://news.com.com/Linux+contributors+face+new+rules/2100-7344_3-5218724.html (October 2004).
[41] From an article on the Internet: http://news.com.com/Linux+contributors+face+new+rules/2100-7344_3-5218724.html (October 2004).
[42] From the Internet: http://www.nwc.com/showitem.jhtml?docid=1511buzz1 (October 2004).
[43] This document was also described in chapter four of this book.
[44] From the Internet: www.linuxbase.org (May 2003).
[45] From the Internet: www.linuxbase.org (May 2003).
[46] From an interview with the editor in chief of the Linux journal.
[47] From an article on the Internet: http://www.theregister.co.uk/2000/07/31/ms_ballmer_linux_is_communism/ (October 2004).
[48] From an interview with a member of the Board of Directors of the ASF who was involved in Apache at the time when IBM became interested.
[49] See chapters four and six for a more elaborate discussion of the actual role of the ASF in the development and maintenance processes.
[50] From an interview with Lawrence Lessig.

[51] From an interview with Lawrence Lessig.
[52] From clause 11 of the GPL, which is available on the Internet: http://www.gnu.org/copyleft/gpl.html (February 2004).
[53] From clause 11 of the GPL, which is available on the Internet: http://www.gnu.org/copyleft/gpl.html (February 2004).
[54] With the liability clauses the question shifts but essentially remains the same, namely: 'Do courts accept the liability clauses?' The answer to this question will depend on many variables.

CHAPTER ELEVEN

CONCLUSION

This chapter draws conclusions from this research on the organization of larger open source communities – in terms of number of contributors and size of the source code. Two of the most extensively analyzed communities are the Apache community, which consists of a great number of subprojects like the Apache HTTP Web server, and the Linux kernel community. Open source communities are virtual organizations in which individuals, for a wide variety of reasons, develop and maintain open source software. Many of these individuals are volunteers who are not paid to participate in open source communities (Hertel et al 2003). Furthermore, they seldom or never see each other in real life; they meet virtually, on the Internet.

Open source communities are scientifically interesting because they appear to lack many mechanisms that are frequently said to be essential to ensure coordination. For instance, the communities lack labor contracts or more general contractual relationships that tell participants what they should do and how (Franck & Jungwirth 2003). Participants decide what they want to work on and how they want to do it. "Work is not assigned; people undertake the work they choose to undertake" (Mockus et al 2002, p. 310). Furthermore, open source communities lack clearly defined organizational boundaries (Fielding 1999, Raymond 1999b). "Membership in the community is fluid; current members can leave the community and new members can also join at any time" (Sharma et al 2002, p. 10). The absence of these mechanisms results in a puzzle, namely, 'How are the communities organized?'

This puzzle is further complicated by the internal and external pressures that face open source communities. The external pressure emanates from the fact that the communities are intertwined with the software market. The communities are confronted with intellectual property rights which can be used to appropriate software from the communities (Benkler 2002b, Bollier 2001b, Bollier 2002, Boyle 2003). Patents and copyrights, but also the commercialization of open source software, poses external pressures to open source communities and their continuity (Vemuri & Bertone 2004, Van Wendel de Joode et al 2003). The internal pressures, on the other hand, are closely related to the organization of open source communities, and includes, for example, social dilemmas like free riding[1] (Von Hippel & Von Krogh 2003, McGowan 2001) and cascading conflicts.

Despite the above, individuals in the communities are able to collectively develop software that is highly complex and successful,[2] as evidenced by the increasing number of organizations and governments that are turning to open source software to facilitate their critical business processes.[3] Furthermore, programs like Sendmail (Lerner & Tirole 2002b) and Apache[4] dominate their respective segments of the software market. Open source software like Apache[5] and Linux[6] have proven to be of comparable quality to proprietarily developed software.

The fact that open source communities face different types of pressures and yet the software has been able to gain a large share of the market gives rise to our research question:

How are open source communities organized and how do they sustain themselves?

Two terms in this question require explanation. *To organize* refers to the fact that having motivated individuals is not enough. Much research on open source communities has concentrated on the question of why individuals are motivated to participate (e.g. Hars & Ou 2002, Hertel et al 2003, Lakhani & Von Hippel 2003). Clearly, however, an organization is more than a collection of motivated individuals. An 'organization' implies the presence of coordination, which calls attention to processes like gathering and mobilizing resources, allocating resources, and negotiating and structuring responsibilities and activities among individuals. *To sustain* refers to the fact that the existence or continuation of open source communities over time is far from obvious or logical. Research on open source communities appears to take their presence as a given, as if it needs no further inquiry. Such research concentrates on many relevant questions, but not on the question how a collective of individuals survives over a longer period of time.[7] How do the communities protect their continuity in light of internal and external pressures? This research aims to fill both gaps in current research on open source communities.

The research framework

Two explanations from the state-of-the-art research: self-organization and institutionalization

State-of-the-art research on the organization of open source communities can be roughly divided into two groups. The first group is made up of researchers who argue that open source communities are self-organizing systems (Axelrod & Cohen 1999, Bekkers 2000, Kuwabara 2000). The communities are not created according to a grand design and centralized control is absent. Lanzara and Morner write, "Open-source software projects are made of heterogeneous components that keep a dynamic balance with one another. The balance does not come from *ex ante* or centrally planned design, but rather emerges out of unplanned, decentral interaction" (2003, p. 11). Coordination in a self-organizing system is the result of the actions of agents and their interactions with other agents and local conditions. Research on self-organizing systems typically uses rules to represent and describe agents' actions (Bonabeau et al 1999, Holland 1995, Kelly 1994, Resnick 1994). Such rules are behavioral, as they are said to describe the behavior of agents. Bonabeau et al. (1999), for instance, focus on how social insects create nests. Though these nests can take a variety of the most amazing shapes, they show that each shape can be explained with a limited number of behavioral rules.

The second group is comprised of researchers who analyze specific elements of the organization of open source communities. This type of research has identified a great number of collective mechanisms and institutions. Institutions are defined here as the factors and forces that constrain or support the behavior of individuals *and* that give rise to regularities in the patterns of human behavior (Crawford & Ostrom 1995). Examples of the institutions that can be found in open source communities are project leadership, foundations and open source licenses (e.g. Bonaccorsi & Rossi 2003c, Hann et al 2002, McGowan 2001, O'Mahony 2003). The implicit claim in this line of research is that institutions explain how the communities are organized, as they determine how individuals will behave.

The two groups of researchers forward explanations that are contradictory in important respects. One group claims that the communities are collections of interacting agents. Agents make their own informed choices based on local information and any form of centralized control is absent. This explanation conflicts with the second explanation, which is based on institutions. Quite a number of researchers have identified and discussed the role of institutions in open source communities. The claim is that these institutions influence and determine the behavior of individuals. Therefore, the institutions explain how (parts of) the communities are organized. Open source communities should not be thought of as collectives of interacting agents because such an explanation is overly simplistic and metaphorical (Kogut & Metiu 2001, Weber 2004).

Reconciling the differences: lessons from community-managed common pool resources

The framework adopted in this research to analyze the organization of open source communities is based on the lessons from research on community-managed common pool resources. The first and foremost reason for adopting the lessons from research on community-managed common pool resources is that it incorporates explanations based on self-organization *and* institutions. They are not treated as mutually exclusive, but rather as complementary. The second reason is that much of the research on community-managed common pool resources is developed and tested through empirical research – as opposed to the metaphorical models dominant in the open source literature. A third reason is that research on common pool resources and the resulting framework are widely accepted and have been tested in many different settings and by a variety of researchers.

Ostrom (1990, 1999) and many of her colleagues, most of whom either work at or are associated with the Workshop in Political Theory and Policy Analysis at Indiana University, have performed extensive research on self-organizing communities in which common pool resources are managed. One important outcome of this research is the identification of eight *design principles*. These principles are said to explain why and how communities are able to organize and sustain themselves, and the resource they manage (see Ostrom 1990 and chapter two of this book for an overview of the design principles). The design principles are basically generalized descriptions of institutions that communities craft to organize and sustain themselves over longer periods of time (Ostrom 1993). Individuals in the communities are affected by the institutions and change their behavior because they are aware of the rules and because they expect nonconformance to be monitored and sanctioned.

The design principles were used in this research in a slightly revised form as a heuristic to understand the organization of open source communities. The communities were analyzed and observations structured according to the design principles. The underlying assumption is that the presence of the institutions described by the design principles would provide a conceptual explanation of how open source communities are able to organize and sustain themselves.

Looking ahead to the findings: the importance of individual behavior

The goal throughout this research has been to understand how the design principles and the functions addressed in them are implemented in the communities. A first step was to understand how institutions present in open source communities provided the functions, as they are described in the design principles. One of the important findings is that institutions in the communities are surprisingly light. The influence of most institutions on the actual behavior of individuals appears to be marginal. Thus, we are confronted with a puzzle: 'The functions described in the design principles are not addressed or implemented through institutions.' Organizational ordering in open source communities can not be understood with a focus on institutions alone.

A next step was to look for alternative ways in which the functions described in the design principle might be fulfilled. The analysis was explicitly focused on the individual level; on the level of individual participants. Are the functions described in the design principles addressed emergently?

The next pages identify a set of individual behavioral rules that are comparable to the rules formulated in research on self-organizing systems. The claim is that these rules, combined with the mechanisms in open source communities, support many observations for each design principle – complemented by less important, light-handed institutional structures.

The individual behavioral rules identified are based on the assumption that individuals want to maximize their utility. This is in line with state-of-the-art research on people's motives to participate in open source communities. Also, the aim is to search for a set of rules that is as limited and simple as possible. Finally, it is important to note that the claim is not that the rules can predict the behavior of every individual in the communities. Rather, the rules provide support for most of the observed collective patterns in open source communities (see for instance Bonabeau et al 1999, Resnick 1994). Therefore, they support the main finding of this research, which is that the organization of open source communities must be understood through the behavior of individual agents.

Boundaries

The first design principle is the presence of clearly defined boundaries. Boundaries in communities are needed to create a feeling of belonging, to determine who is outside and inside and to decide who has a right to the fruits of the efforts of the community members. Research on common pool resources stresses that not only must organizational boundaries be created; boundaries delineating the resource are equally important (e.g. Ostrom 1990).

Boundaries are also needed in open source communities, as the resource is susceptible to depletion. Yet what is striking is that the communities in general lack a boundary to limit the size of their membership. In other words, organizational boundaries tend to be absent. The boundaries present in open source communities are primarily intended to demarcate the boundaries of the source code.

The most important boundary in open source communities is undoubtedly the open source license (see for instance Benkler 2002b, Dalle & Jullien 2003, Lerner & Tirole 2002a, McGowan 2001). The licenses protect open source software against appropriation (O'Mahony 2003).[8] A number of observations can be made concerning the boundaries. First, each individual or company can always create a new license, which has resulted in an wide variety of open source licenses. The Open Source Initiative (OSI), which is a foundation established to promote the concept of open source, lists 48 licenses that comply with its definition of open source.[9] The differences between most of these licenses are small, and yet individuals and companies have taken the time and effort to create these variations to suit their specific needs.

The licenses are dynamic and responsive to emerging threats and challenges. The environment of open source software development is not static. Technology is constantly changing, and companies and individuals are constantly exploring the limits of what the licenses allow them to do with the software. The creators and maintainers of the licenses must anticipate and respond to these changes and new challenges. An example of a license that needed change is the GPL. "When writing a license, one can only see a limited number of years ahead. Currently the GPL 2.0 has trouble addressing certain issues."[10] The GPL version 3.0 or the Affero GPL was written to address some of these issues.

There is convergence in the selection of open source licenses. Most open source communities, as if it were a collective decision, are governed by a limited number of 'favorite' licenses. For instance, the GPL is by far the most used license.[11] Furthermore, there are many relatively obscure and unknown licenses that are largely ignored and which are used by only a very small number of communities.

Next to open source licenses, many communities have erected a legally constituted foundation to protect the individual participants (O'Mahony 2003). These foundations provide a powerful protection mechanism against the appropriation of software; they create a nonprofit tax status and they enable legal representation of the community and individual contributors.[12] But they are also non-intrusive, as they hardly interfere with the actual processes within the communities. They also leave ample space for individual decisions, flexibility and change.

A third boundary in open source communities consists of the large set of mailing lists that participants actively use. These mailing lists serve as 'boundary spanners'; they educate about the licenses, they filter relevant external information to the participants and they provide a means to organize pressure on people or entities that infringe on the licenses.

Of the three boundaries, the open source licenses are the most important. They define what users are allowed to do with the software and what not. The licenses can be said to be embedded in a system of foundations and mailing lists. The latter two educate about the licenses and provide a vehicle to enforce them when needed. One could say that the foundations are formally erected institutions to, among other things, legally enforce the licenses, and that the individuals on the mailing lists informally punish organizations and individuals who try to exploit loopholes in the licenses.

Individual behavioral rules

Three individual behavioral rules are sufficient to support most of the above-stated observations.

"*Participants adopt licenses that maximize the guarantee that they will benefit from the participation of others.*" Open source software creators aim to select an existing license or create a new one that maximizes the chance that they will benefit from the contributions of others who use the software and make changes and improvements to it.

"*Participants adopt licenses that provide sufficient motivation for others to participate*." The creators or initiators of a software development project or a new software module typically select a license only once, when the first version of the software is made available on the Internet. Programmers elect to make the source code available so as to attract others to use and participate in the maintenance and further improvement of the software. Therefore, they adopt the license that they perceive will provide sufficient incentive for others to participate.

"*Participants want to limit the time they spend analyzing the licenses*." Time is the scarce resource in open source communities (see also Hertel et al 2003). Therefore, most participants want to spend as little time as possible on activities related to licensing. They would rather spend their time on other activities, such as writing new source code.

Boundaries as an emergent property of individual behavioral rules

According to the first rule, participants want to create and adopt licenses that maximize the guarantee they will benefit from the participation of others. To accomplish this goal, they are tempted to devise restrictive and complex licenses. To others these licenses are generally more difficult to understand and less attractive. A company for instance will by and large be more inclined to contribute to a community with an open and unrestrictive license, as this provides more room to make a future profit. Thus, the licensor needs to make a trade-off between an unrestrictive license, which is likely to attract many contributors, and a restrictive license, which might attract fewer contributors but provides a greater chance that changes to the source code will flow back to the community.

Each participant may strike a different balance for this trade-off, which supports the observation that there is a wide variety of licenses. The BSD license, for instance, is claimed to offer a lot of incentive for companies to participate in a community (e.g. Bonaccorsi & Rossi 2003b),[13] whereas the GPL provides the creator of the software a greater chance of benefiting from the participation of other individuals (Perens 1999). Deviations from these two licenses arise from minor differences in individual preferences.

The desire to spend as little time as possible in analyzing and/or creating a license constitutes a counterforce against the rise of ever more variation in licenses. In most cases participants will prefer to download software with a familiar license rather than software with a new and relatively unknown license. This means that – for comparable software – potential users would rather download GPL-licensed software than software licensed with a relatively unknown license. This acts as a counterforce to the rise of divergent licenses, since adopting a popular license instead of creating a new one increases the chance of attracting participants.

The fact that licenses are dynamic can be understood by the wish to have as many guarantees as possible to benefit from the contributions of others. If a license enables actors to disallow others from benefiting from improvements to the software, fewer new participants will be attracted to the project. To continue to attract new participants in such a case the license will likely have to be revised.

The rules, however, do not shed light on the establishment of foundations. These are formally erected and provide protection against outsiders, but cannot be understood in terms of individual behavioral rules.

Appropriation and provision rules

To ensure the continuity of the source code, theory would suggest that the communities need both appropriation and provision rules. Appropriation rules aim to determine how much the members of a community are allowed to consume. Provision rules regulate how and when members are supposed to contribute their time and effort to the development and maintenance of the resource.

Open source communities have no appropriation rules other than the licenses mentioned in the previous section. This lack of additional appropriation rules is partly explained by the fact that many types of software usage are non-subtractive. This means that use does not reduce the amount available to others. The other part of the explanation is that there are boundaries that restrict the types of usage that do result in a reduction of software available to others. For this reason, attention is best focused on the way in which provision is organized. It is important to note that the mechanisms identified and described apply only given the presence of a sufficient number of individuals motivated to contribute. Thus, only in the larger communities that attract many participants will these mechanisms be sufficient to ensure the provision of software.

First, software development and maintenance in open source communities is characterized by an apparent lack of body or platform which actually defines or delegates activities among the participants. For instance, many communities have project leaders and/or maintainers. However, these positions carry little real authority and lack the ability to assign tasks to individual participants (see also Markus et al 2000, p. 21). Furthermore, there is no body that actually monitors progress on tasks and activities.[14] Development of software in the communities is instead based on individual choice and a 'culture of doing', which is explained in some more detail under the heading of *conflict resolution mechanisms*.

Second, elegance[15] and modularity[16] lower the need to coordinate activities of participants. Elegance reduces the investment required for participants to understand software written by others, as the software itself is able to explain "what it is doing while you are reading it."[17] Obviously, different people will interpret software differently. However, elegance makes the software easier to understand. Furthermore, elegance eases the implementation of changes.[18] The big advantage of modular software is that each module performs a limited set of tasks, which "individuals can tackle independently from other tasks" (Lerner & Tirole 2002b, p. 28). Therefore, for modular software the need to formally divide and coordinate activities among participants is reduced[19] and the testing of new patches is simplified (Torvalds 1999).

Third, many participants voluntarily use a great number of primarily technical tools and mechanisms to structure their activities and enable others to find and understand software. The mechanisms are (i) a concurrent versions system, which many communities use to structure the actual development process (Bauer & Pizka 2003, Shaikh & Cornford 2003); (ii) mailing lists, which allow one-to-many communication to discuss ideas and exchange knowledge (Bauer & Pizka 2003); (iii) bug-tracking systems, which enable participants to report bugs that can be addressed and solved by others; (iv) manuals and coding style guides,[20] which prescribe how software should be written; (v) to-do lists, which provide an inventory of functionalities that are wanted and are open for work; (vi) the 'orphanage,' as in the Debian community, where software that is in need of a new maintainer is stored;[21] (vii) text added in the source code to signal and communicate source code characteristics; (viii) names attached to improvements, which provides a means for others to contact the author; and (ix) small and incremental patches, which make changes relatively easy to understand and which ease improvements or reversals to these changes.

Many participants are professionals who enjoy doing what they think is best and who want to retain as much autonomy as possible (De Bruijn 2002). Yet they adopt mechanisms that limit some of their autonomy. One interesting question is: why?

Individual behavioral rules

Three individual behavioral rules are needed to understand why participants have created the mechanisms and why they actually use them.

"*Participants spend a limited time searching for software and analyzing others' contributions.*" Again, participants want to spend their time as efficiently as possible. They therefore want to minimize the time spent analyzing and searching for contributions from others.

"*Participants replace a contribution if another contribution is easier to understand.*" Related to the previous rule, other things being equal, participants will replace an existing contribution when there is an alternative that is easier to understand.

"*Participants want to increase the chance others will accept and adopt their contributions.*" The previous two rules are not sufficient to understand the observations. This third rule basically states that participants who spend time and effort developing software want to increase the chance that their contributions will be accepted and adopted by others. There are many reasons why this is so, such as to improve their reputation (McGowan 2001). There are also more pragmatic reasons. Participants whose contributions are not accepted will have to maintain and update their own version of the software with every new update from the community. But if their patch is accepted others may (partially) maintain the patch for them and they will be able to benefit from the contributions of others.

Provision rules as an emergent property of individual behavioral rules

According to the first rule, participants spend a limited amount of time searching for software and analyzing other people's contributions. This poses a risk to contributors. They want to increase the chance that others will accept and adopt their contributions (this is the third rule). To increase the chance that their contributions are accepted and adopted, they create new tools and use existing ones that make their contributions visible and understandable. Thus, they implement and use tools like a CVS, a bug-tracking system and mailing lists. Every contributor uses these tools, since not using them would reduce the chance of their contributions being accepted. Both the first and the third rule also explain participants' drive to create software that is elegant and modular, as this increases the chance that other participants will understand and accept the software. Finally, the rule that participants replace contributions with alternatives that are easier to understand suggests that most open source software becomes more elegant and modular over time.

Conflict resolution mechanisms

Conflicts can have both negative and positive effects on organizations. Some of the advantages are said to be a higher level of productivity, a rise in creativity and an increase in vitality (Jehn 1995, Rosenthal 1988). However, conflicts can also have negative effects. They can become destructive and result in inactivity (e.g. Jehn & Mannix 2001). Communities need some way to manage and resolve conflict, to prevent conflict from threatening the continuity of the community (Ostrom 1990).

Open source communities have a high potential for conflict. The literature describes at least two sources of conflict: *interdependency* (Dipboye et al 1994, Jehn 1995) and *diversity* (Deutsch 1973, Gefu & Kolawole 2002). In open source communities the participants are mutually dependent. Participants cannot develop and maintain the software alone; they need contributions from others, as well as maintain compatibility with other pieces of software. There is also a high level of variety among open source developers (Markus et al 2000) and they differ widely in what they want the software to do. Thus, many decisions may invoke conflicting positions. For instance, consider a situation in which two participants have created a different solution to the same problem. Both spent time to create their solution. They will be tempted to defend their solution and explain to others why their solution is the better one.

The high potential for conflicts is tempered through the software's modular design. Modularity increases the independence of participants. It ensures that changes made in one part of the software are less likely to have the effect that "something else does not work anymore."[22] Modularity also enables conflicts to be localized and isolated. Indeed, programmers can break down most conflicts into smaller parts and attribute them to a part of the software, that is, to one module.

In the communities, conflicts are plentiful, but their negative consequences are relatively minor. A number of reasons can be given for this. First, conflicts are transformed into a competition between two or more alternatives. Thus, conflicts are not so much resolved, but translated into parallel development lines. In this way, conflicts do not disrupt the activities of individuals in the communities. Second, participants in the communities have easy access to the exit option. The exit option dampens the emergence of conflicts (Hirschman 1970), because people can vote with their feet. If participants disagree strongly, they can exit the community. An extreme example of the exit option is the *fork*. At the heart of a fork are two parties that disagree, for instance, about the direction of a software development project. One of the parties could decide to exit the community, but still continue the development of software in the way they consider best. At a certain point in time the two programs are likely to develop irreconcilable differences, at which point the fork is complete.

The underlying principle of both the parallel development lines and the exit option is the 'culture of doing.' Quite a number of respondents argued that they frequently ignore conflicts on mailing lists and in other community forums. They just do what they think is right. "So if I think something should be changed then I do it myself, particularly when the group disagrees."[23] The basic principle is that if participants want to convince others in the community that they are right, they have to provide the proof. They do this by producing the source code and demonstrating that their solution is better. In other words, participants must put their code where their mouth is. They quickly lose interest and ignore conflicts surrounding hypothetical solutions.

Individual behavioral rules

"*Participants remove or overwrite contributions if they deem them inappropriate.*" In many communities participants can remove or overwrite contributions made by other participants if they feel the contribution is inappropriate (McCormick 2003).

"*Participants want to demonstrate that they are right.*" Individual and corporate participants become involved for a variety of reasons. One thing that unites both groups is that they are highly skilled and in many ways behave like professionals. Many of the participants enjoy writing source code and want to see their contributions accepted in the communities (Hertel et al 2003). They are motivated to prove their point and demonstrate that they are skilled programmers. This last point is in line with one of the motives participants cite for investing time and effort in the communities, namely, to build a reputation (Lakhani & Von Hippel 2003, Raymond 2000).

"*Participants want to minimize the time they spend in conflicts and discussions.*" This rule is related to the second rule, namely, that participants want to spend their time creating new source code. They are easily bored by hypothetical discussions.

Conflict resolution as an emergent property of individual behavioral rules

The drive to create modular software was discussed in the previous section. The thing we need to understand here is why and how conflicts are 'managed' through parallel development lines.

Consider the first and the second rule. Combined, both rules could give rise to many conflicts. Indeed, a number of researchers have described how 'commit wars' arise, that is, conflicts in which two or more participants continuously remove or overwrite each other's contributions (Bauer & Pizka 2003). Commit wars are especially relevant in communities in which software development is supported with a CVS. In the Linux kernel, Linus Torvalds makes the decision on whether to include the contributed source code.

In both cases the solution is to allow participants to create a parallel development line or to otherwise exit a community. A parallel development line can be created by one of the conflicting parties, but it can also be initiated by a third party who is annoyed with the conflict. With the creation of the second line, the conflict no longer obstructs the development process. The option of creating parallel development lines is functional, as it prevents inertia. Participants can continue to develop and maintain the software. "It ensures that a project will continue to evolve and improve."[24] However, this also results in additional costs, as resources are now spent on two competing software development trajectories.

The third rule supports the observation that open source communities have a culture of 'doing,' instead of talking. They want to spend their time and effort creating new source code, on new solutions and maintaining existing source code. Discussions and conflicts distract too much attention away from these activities.

Collective choice

A collective choice is a particular type of choice, namely, one that binds and restricts the individuals in a collective. The "decisions made in collective-choice situations *directly* affect operational situations" (Ostrom 1990, p. 192). The most commonly mechanism to reach a collective choice, as described in literature on collective choice, is through voting systems (Arrow 1951, Walker et al 2000).

Voting systems are present in open source communities. But in the communities that have adopted such systems the actual influence and binding character is rather limited. In the PostgreSQL community, for example, though the voting system is regularly used, the outcome is merely "transferred to the 'to-do' list, so we know what has been decided on." Yet participants are still free to do what they want, irrespective of the outcome of the vote. "Not everyone agrees on [the vote]… people are still free to implement whatever they think is best."[25] The voting system of the Apache community is used for every new addition of source code. Yet, as in the PostgreSQL community, the vote is not used to reach a collective choice. Individuals are still free to remove the source code once it is voted in. When the outcome of the vote is that the source code should not be included, individuals are still free to create a new development line of which the source code is a part. In short, the voting systems hardly seem to result in a collective choice, that is, 'a choice that binds the collective.' Individuals in the community are still free to do something else.

Despite the relatively weak influence of the voting systems, some form of collective choice is needed. Participants in the communities have opportunity and even incentive to diverge and create variety (Van Wendel de Joode 2004b, Van Wendel de Joode et al 2003). Consider, for instance, the mechanism of parallel development lines, the openness of source code and the diversity among participants. In light of this potential to diverge, threats like inefficiency, incompatibility and fragmentation would seem to emerge (Egyedi & Van Wendel de Joode 2004, Kogut & Metiu 2001). Yet software developed in bigger open source communities has an overall quality and market share that is relatively high, which hardly seems possible if the communities were unable to come to some degree of convergence and focusing of resources (e.g. Bauer & Pizka 2003, Kogut & Metiu 2001, Mockus et al 2002). The question is 'How, among the huge variety of software, are participants able to decide what software to adopt and what to ignore?' How is convergence achieved?

Part of the answer lies in the fact that individuals base their choices on *tags*. Tags create positive feedback. Individuals in the communities appear to mimic each other's behavior, which on a collective level gives rise to 'swarming' (for a discussion on swarming, see Kelly 1994). In swarms the participants select one patch of source code over another, or one open source license over many others. Many tags facilitate this process of mimicking and swarming. Important tags are, for instance: (i) the presence of statistics indicating the level of activity in a project. Such a statistics stimulate participants to select software that is also selected by many others. One respondent explained, "SourceForge has information on activity as well. If there are 20 different versions of a library for a certain purpose, and one has been downloaded 10,000 times and another one 20 times, it is clear which you choose first." Another mechanism is (ii) the level of reputation of the participants. Reputation may be "a way to attract attention from others" (Ljungberg 2000, p. 212). Other mechanisms are (iii) elegance of the source code, (iv) inclusion in software distributions like Red Hat and Debian and (v) listing on an Internet directory site like Freshmeat.[26]

The empirical data in this research do not provide a clue as to which of these five tags is most dominant and important in attracting individuals. Conceptually, it is possible that each individual mechanism constitutes a sufficiently strong tag to attract many new participants and make certain communities popular. However, in line with an observation by West and O'Mahony (2005), with the rise of the number of open source communities the competition for attention becomes greater. It is conceivable that it is increasingly difficult for new projects to attract new participants and users. This could mean that for new projects to become popular, they need to have a combination of tags.

Individual behavioral rules

Two rules shed light on the dynamics as to how participants are able to select among the huge variety and why the resulting pattern resembles the patterns of swarms.

"*Participants select software that meets their specific user needs*." Participants' primary selection criteria are based on their own needs; for instance, the perceived quality of the software, the expected continuity of the software and its functionality.[27]

"*Participants spend a limited amount of time analyzing software and base their choice on tags*." Time is the scarce resource. Therefore, participants want to select software without having to "check under the hood."[28] To save time, individuals base their choices on tags.

Collective choice as an emergent property of individual behavioral rules

Individual choices are not random. This can be understood with the second rule, namely, that participants will limit the time they spend analyzing alternatives and therefore base their choices on tags like level of activity or reputation. These two mechanisms and the others stimulate participants to mimic others' behavior and increase the chance that they will swarm around high-quality software.

First, the mechanisms stimulate mimicking. The level of activity is rather straightforward. A community with a high level of activity attracts participants, which further increases the level of activity. Also the level of reputation results in mimicking, because (i) a community with participants who have a good reputation attracts other participants and (ii) participants tend to participate in communities with many other participants, as this increases their visibility and thus provides more opportunity to increase their reputation (e.g. Lerner & Tirole 2002b).

Second, the selected software will be of comparatively higher quality. There are a number of reasons why this is so: elegance of software is an indicator of qualitatively high software. Communities with more activity will, among other things, find and resolve more bugs, meaning the quality of software will be relatively good. The level of reputation is a proxy for the ability of participants to write qualitatively good software. And the fact that software is included in a distribution means that more programmers check and improve the software, which is especially true for distributions like Red Hat and SuSE for which programmers are hired to improve the software.

Monitoring and graduated sanctioning

Monitoring and sanctioning in a community are needed to ensure that everyone adheres to the rules and codes of conduct that are agreed upon by the participants. In open source communities monitoring and sanctioning ensure that participants adopt the coordination mechanisms, act according to the licenses and treat other participants respectfully. "Bad behavior is mainly rude behavior, swearing, consistently disrespecting other people's opinion."[29]

A first observation concerning monitoring and sanctioning is that many types of infractions are dealt with almost automatically. Due to a redundancy in mechanisms to deter conflicts, to support the development process and to prevent appropriation of open source software, many actions that could become a serious threat are actually automatically resolved. This redundancy in mechanisms reduces the potential threat of infractions and lowers some of the need for formal sanctioning mechanisms.

Participants in open source communities have access to a limited number of more formal sanctioning mechanisms, but these are hardly used. In the Apache community participants could be "kicked out of the community." This means they lose their right to commit source code directly into the CVS or they are banned from a mailing list. However, this mechanism has never been used. "We could kick out people, but even that we have never done."[30] The fact that these sanctioning mechanisms are hardly used does not automatically mean that their role is limited. Their mere presence may be a sufficient deterrent for individuals to display grave counterproductive behavior.

The third observation is that the mere act of participating in open source communities automatically leads to monitoring. In other words, participants incur few additional costs to monitor others, because (i) the processes and activities of participants in the communities are highly transparent (Osterloh 2002); (ii) participants are automatically notified of new activities, for example, through notification of changes to the CVS (Shaikh & Cornford 2003); and (iii) use of open source software automatically implies testing (Von Hippel & Von Krogh 2003).

The fourth observation is that sanctioning is the direct result of monitoring and participation. Programmers are sanctioned and stimulated to do things right in three different ways: (i) In the 'hall of blame'[31] almost everything programmers write (e.g. e-mails and source code) is and remains visible to others. This ensures that participants have an incentive to do things right. (ii) Participants can create a fork when they strongly disagree – or are annoyed – with the general direction of a community, with its atmosphere or with individual decisions. The fork results in a destruction of value, as the same resources are now divided between two competing projects. The fork thus sanctions the participants in the community. (iii) Flaming, spamming and shunning (Maggioni 2002, Osterloh 2002) are other sanctions used to penalize participants. "If [their code is] inscrutable, sloppy, or hard to understand, then others will ignore it or pummel them with questions. That is a strong incentive to do it right" (Wayner 2000, p. 118).

Individual behavioral rules

The reason why being ignored is considered a sanction can be understood with just two individual behavioral rules. These two rules do not support all of the observations related to monitoring and sanctioning, but they do go some way.

"*Participants want to increase the chance that others will accept and adopt their contributions*." This rule was introduced in chapter five. It argued that there are a number of reasons why participants would like others to adopt their contributions: to improve their reputation and reduce the effort they need to expend maintaining the source code.

"*Participants base their choices on tags, like the level of reputation of participants*." It was already argued that participants frequently base their choice on tags. One of these tags is the *reputation* of developers. Participants prefer to use and download software from communities or projects in which participants with a high level of reputation are involved. Similarly, participants listen more to participants who have a relatively better reputation.

Monitoring and sanctioning as emergent properties of individual behavioral rules

With every contribution, be it an e-mail or source code, the name of the contributor is connected. This system allows participants to judge the reputation of others in the communities. The reputation of participants is likely to rise if they make a positive contribution, for instance, if their name is attached to highly sophisticated and elegantly written source code or if they solved many problems of end users. In a similar line of reasoning, reputation will fall when participants write stupid e-mails or inelegant source code. The level of reputation is also affected when others write negatively about them.

The second rule states that participants base their choices on tags. Particularly relevant for this design principle is that a participant's reputation influences the choices made by other participants. Respondents interviewed in this research explained that they are more inclined to accept the opinion of a participant who has a good reputation. This gives participants an incentive to try and increase their reputation, since the higher their reputation the greater the chance that their solution will be accepted and adopted, which is the first rule.

The first rule also supports the observation that shunning, that is, ignoring, is viewed as a powerful sanctioning mechanism. It means that fewer people will download and adopt the ignored member's contribution.

Multiple layers of nested enterprise

The design principle multiple layers of nested enterprise draws attention to the diversity and interdependencies that exist in large and complex common pool resources. This is also true for open source communities. Particularly the larger open source software programs consist of millions of lines of source code and can be used in combination with hundreds of different applications. Interdependencies must be managed to ensure its proper operation.

The first observation concerning this principle is that open source communities have a highly developed division of labor (Koch & Schneider 2002). This division of labor is achieved in two ways, namely, through the modular design of the software, which results in the creation of many sub-communities, and through the division of activities.

Second, the division of labor and activities is emergent. Once again, there are no formal mechanisms by which tasks are identified and divided. No project leader can assign tasks to individuals. Instead, individuals choose for themselves what to work on. "[A]gents choose freely to focus on problems they think best fit their own interest and capabilities" (Bonaccorsi & Rossi 2003c, p. 1247).

Third, the division of labor results in high degrees of task specialization. For instance, Mockus et al. (2002) demonstrated that of the top-15 bug reporters in the Apache community only three are also core programmers.[32]

Fourth, the division of labor raises the level of efficiency in communities. The structure of open source communities is layered, resembling the structure of an onion (Crowston et al 2003b, Nakakoji et al 2002, Van Wendel de Joode et al 2003). The easier tasks, like reporting bugs or solving user problems, are carried out on the outside layers and the more difficult and challenging tasks are done closer to the center (Ye et al 2002). It is efficient for core developers to spend most of their time on the more difficult tasks and leave the easier ones to others. Indeed, they reportedly ignore simple tasks. "There are people who know practically everything. They won't react to trivial matters."[33] In contrast, less skilled programmers and even users perform the simpler tasks.[34]

Fifth, the outside layers of the community constitute a learning environment, and as participants learn they tend to move inward (Edwards 2001, Ye et al 2002). This learning environment is based on the presence of many fairly simple activities, which allows users to get acquainted with the practices and processes in the community.

Individual behavioral rules

The creation and the choice of activities are based on individual choices and actions. Individual choices are argued to result in task specialization. Some participants choose to adopt complex and challenging activities. Why do participants make such choices? What is the underlying logic? A possible explanation can be found in Hertel et al. (2003). They claim, based on extensive quantitative research, that three factors determine a participant's desire to work on an activity. Two of these factors are "the perceived importance of their own contributions for the subsystem" and "the perceived personal ability to accomplish the tasks" (p. 1175). These two factors can be translated into two behavioral rules that underlie individual actions:

"*Participants perform activities which they perceive they are able to complete successfully.*" Participant will be inclined to perform an activity, which they believe they are able to complete.

"*Participants perform activities of which they perceive their contribution to be important.*" One of the motives of participants to participate in open source community is to gain a reputation. This motive makes participants select and perform activities that are deemed important by others in the community.

Multiple layers of nested enterprise as an emergent property of individual behavioral rules

The more difficult the task is, the higher its visibility and importance (see also Ye et al 2002). Therefore, the two rules create a trade-off: the simpler the task the more certain participants will be that they will accomplish the task, but the lower the importance of the contribution. Participants therefore perform the most difficult task for which they perceive themselves able. This supports the observation that different participants create[35] and select different tasks to work on. It also supports the other observations. Highly skilled participants generally do not solve easy tasks, because performing more difficult tasks is considered more important. The rules also predict that participants will specialize in certain tasks. They know they have completed a similar activity before. Performing the same activity reduces their chance of failure and the time they need to perform the activity. Gradually participants increase their level of knowledge and improve their skills. When reassessing their personal skills and abilities, they might conclude that their ability to perform has changed and they might decide to work on more complex activities. Thus they move inward into the community.

External recognition

The final design principle is external recognition. External recognition focuses on the interdependencies between a community and its outside environment. Essentially, the principle claims that no community can exist and survive without external authorities acknowledging and respecting the processes and structures in the community.

Currently two debates surround open source software and the communities. Both debates are riddled with metaphors and stories. Furthermore, two extreme positions have emerged from them. The debates are relevant because they influence the perception that external authorities have of open source. The first debate is whether open source software is more secure or less secure than proprietarily developed software. The second debate is whether open source stifles or stimulates innovation in the software market. The root of both debates can be traced back to two underlying questions: (i) Is openness of the source code and the development process desirable? (ii) Are intellectual property rights on software desirable? Currently, the debates are undecided, making it as yet difficult for external authorities to act upon these issues.

In the meantime, participants in open source communities have created and adopted mechanisms to counter some of the critique that their opponents have on the way in which software is created and maintained in the communities. The first mechanism is the Developer's Certificate of Origin. This certificate was adopted by the Linux community in response to the SCO case and the potentially disastrous consequences of sloppy IPR management. The certificate explicates the path the source code has gone through before its inclusion in the Linux kernel.[36] One goal of the certificate is to address industry concerns over intellectual property rights.[37] Appendix D presents a copy of the certificate. The second mechanism is the creation of the Linux Standards Base (LSB) and the Free Software Group (FSG). To become LSB Certified, software must undergo a formal documented testing procedure (Claybrook 2004), which will likely increase the credibility of that software. The third mechanism is external representation through foundations and project leaders. Many organizations want to reduce their uncertainty and prefer to have one clear point of contact, one entity with which they can do business. Yet the participants in open source communities change constantly and are located in many different locations. Project leaders and foundations have proven able to take up the contact person's role.[38] The fourth mechanism is the use of disclaimers. The openness and perceived randomness of the software development process in the communities is due to the participation of both individual volunteers and organizations. One concern is that an organization will be sued if an employee has contributed sloppy source code that results in system crashes or security breaches.[39] The disclaimer is intended to alleviate some of these concerns.

An interesting characteristic of the four mechanisms is that they do not conflict with the observations made regarding the other design principles. The mechanisms do suggest a level of institutionalization, which is different from the observations related to the other principles. However they do not interfere with the ability of individuals to act based on individual choices. In other words, the mechanisms do not interfere with or limit the freedom of individuals to make their own informed choices.

The main findings

The role of institutions is minor compared to individual behavior

Institutions can be identified in open source communities. Consider, for instance, a voting system or project leadership. This research, however, arrived at the somewhat counterintuitive conclusion that many of the institutions in open source communities are 'light-handed.' Their influence on the actual behavior of individuals is surprisingly marginal. Some institutions do have an important role in the communities. The role of the Apache Software Foundation in the Apache community, for instance, is not at all marginal. The ASF performs the important function of safeguarding participants in the community from future legal claims. Yet, by and large, many of the institutions in open source communities perform only marginal roles in the communities' organization. One reason why is that, as reported by many respondents in this research, many participants ignore the institutions. They do what they think is best. They behave individualistically. Sometimes this means that they accept the choice of the project leader and the results of a voting system. In other situations, however, they might disagree and ignore these institutions. The previous pages identified and discussed a number of mechanisms that enable participants to ignore institutions and that allow them to act based on their own preferences.

Design principles addressed through emergent patterns of behavior

Now we are confronted with two questions. First, what is the role of institutions? This will be addressed in more detail later in this chapter. Second, how are open source communities organized if not through institutions? The functions described in the design principles are not solved through institutions. For instance, conflicts in open source communities are not resolved through formal conflict resolution mechanisms, like voting systems or project leadership. Indeed, the potential negative effects of conflicts are resolved differently. This research indicates that the function of conflict resolution emerges out of individual choice and behavior. This brings us to an important conceptual point. Apparently, the design principles can be met through other means than institutional arrangements, namely through mechanisms that emerge from individual choice and behavior. In other words, individual behavior aggregates into collective patterns of behavior and thus the functions described in them are solved.

Participants behave individualistically

But how does individual behavior aggregate into collective patterns of behavior? The simplest answer would be that every individual wants to collaborate and wants to act in the best interest of the community.

However, survey research proves otherwise. For one, research has demonstrated that many individuals join the communities primarily to receive personal benefits (e.g. Hars & Ou 2002, Hertel et al 2003). Furthermore, many choices and actions of participants could threaten the continuity of open source communities. Participants, for instance, create new and competing alternatives even when they know their alternative will only be used by a very limited number of people; they send e-mails that contain no relevant information and merely consume other participants' time; and they frequently write source code that is inelegant, does not do what it is supposed to do and which creates many instabilities when added to the existing software.

The fact that individuals ignore institutions and primarily act in response to their own preferences and choices causes redundancy (e.g. Egyedi & Van Wendel de Joode 2004). Typically, multiple solutions are created and maintained for every problem. According to respondents, this results in innovation, competition and eventually better software. However, it also implies that processes in the communities are not necessarily efficient. There is overlap between developers, who frequently perform activities and tasks that have already been done by others.

A small number of individual behavioral rules to understand how emergence is possible

Given that individuals have their own motives to participate and that they act based on their own choices, one might be tempted to assume that the organization of open source communities is random and chaotic. However, this is not the case. Intelligent collective patterns of behavior emerge from the acts of individuals. The question is 'How?'

Individual behavioral rules have been defined to demonstrate and understand how collective patterns of behavior emerge from the behavior of individuals. The individual behavioral rules describe the actual behavior of the individuals who participate in the communities (e.g. Waldrop 1992). The patterns that result from individual behavior are truly emergent because the collective behavior of the group is "qualitatively different from the behaviors of individuals composing the group" (Epstein & Axtell 1996). Although individuals participate for selfish reasons and make decisions that might appear to be dysfunctional, collectively their actions are intelligent and functional.

Table 11.1 summarizes the collective patterns of behavior. It also includes a list of individual behavioral rules. A brief description is also provided as to how intelligent collective patterns of behavior emerge from the individual behavior that is based on the rules. It is remarkable that the set of behavioral rules in the table, which is already limited, still has some overlap. This means that the total number of *unique* individual behavioral rules is very small indeed. For instance, four rules describe individuals' strive to minimize the time they spend on 'peripheral' activities; that is, activities perceived as less important. Examples of such activities are (i) searching for source code written by others and (ii) deciphering source code written by others.

Table 11.1 – The individual behavioral rules, observations and descriptions summarized

Design principle	**Observations**	**Individual behavioral rules**	**Description**
Chapter 4: Boundaries	- Licenses prevent the appropriation of source code. - A high level of divergence in licenses. - The licenses are dynamic and responsive. - A relatively small number of licenses predominate. - Foundations protect against appropriation and leave room for individual choice	- Participants adopt licenses that provide sufficient motivation for others to participate. - Participants adopt licenses that maximize the chance they will benefit from the participation of others. - Participants try to minimize the time they spend analyzing licenses.	- Participants weigh the trade-offs between the first two rules differently and thus create different licenses. - Participants prefer involvement in projects using a well-known license, as this reduces the time they spend analyzing licenses. Also, it limits the rise of divergence in licenses. - Participants will no longer select licenses that have proven to provide little guarantee, thus stimulating others to update licenses.
Chapter 5: Provision	- Tasks and activities are not centrally delegated or monitored. - Most open source software is modular and elegant. - Many mechanisms are present to structure software development. - Participants voluntarily use these mechanisms.	- Participants want to increase the chance others will accept and adopt their contributions. - Participants spend limited time searching for software and analyzing contributions. - Participants replace a contribution if another is easier to understand.	- Participants voluntarily adopt and create mechanisms to increase the visibility and/or understandability of source code, as this increases its chance of acceptance. - There is a constant drive to create elegant and modular software, as this (i) increases the chance of adoption and (ii) reduces the chance of replacement.
Chapter 6: Conflict resolution	- Communities have a high potential for conflicts. - Formal mechanisms do not resolve conflicts. - Modularity lowers the potential for conflicts to grow. - The negative consequences of conflicts are minimized through parallel development lines and the exit option. - The focus is on doing not talking.	- Participants want to demonstrate that they are right. - Participants want to minimize the time they spend in conflicts and discussions. - Participants remove or overwrite contributions if they deem them inappropriate.	- The first and third rule give rise to conflicts; participants can remove each other's contributions, but want to demonstrate that they are right. - The first and second rules are needed to understand why participants use parallel development or the exit option when too many conflicts arise or threaten to arise. - The second rule is also needed to understand the culture of doing.

Design principle	Observations	Individual behavioral rules	Description
Chapter 7: Collective choice	- Formal mechanisms have a limited role. - Selection is intelligent. - Many mechanisms influence individual choices. - Participants act in swarms; they mimic the behavior of others.	- Participants select software that meets their specific user needs. - Participants try to minimize the time they spend analyzing software and base their choice of software largely on tags.	- Tags, like reputation and level of activity, influence participants' choices and stimulate mimicking. - The tags are also indicators of quality, meaning that individuals cluster around the qualitatively higher software.
Chapter 8: Monitoring & sanctioning	- More formal sanctioning mechanisms are hardly used. - Participation results in monitoring. - Monitoring and participation results in sanctioning.	- Participants want to increase the chance that others will accept and adopt their contributions. - Participants base their choices on tags, like the reputation of participants.	- Participants will judge each contribution and adjust the perceived reputation of the contributor accordingly. - To increase the chance of adoption, they make their contributions as visible as possible. - Combined with the second rule, this could explain why a bad contribution becomes a punishment in itself.
Chapter 9: Layers of nested enterprise	- Communities display a high level of task division. - The division of tasks is emergent. - There is a high degree of specialization. - The division of labor raises efficiency. - Tasks on the outside layers of the community constitute a learning environment.	- Participants perform activities that they perceive they are able to complete successfully. - Participants perform activities of which they perceive their contribution to be important.	- The more difficult activities bring higher visibility and perceived importance. - The two rules create a trade-off, the result of which is different for each participant. - Participants select the most difficult tasks they perceive they can complete successfully. - Participants will not select activities they perceive to be easy, as these are less important. - They tend to repeat the same activities to increase the chance of successful completion

The power of ambiguity and trade-offs in the individual behavioral rules

While the rules themselves are rather simple, they still incorporate a significant degree of ambiguity. First, in real-world situations, interpretations may differ as to how a rule translates into behavior. Elegance of software, for example, may be judged differently by different participants. Second, the rules are not independent of each other. They create certain trade-offs, that, again, often give rise to different outcomes by different participants. At first sight the presence of ambiguity and trade-offs may seem to make rule-following less predictable and the rule-set as a whole less effective for organizing the community. On closer look, however, these turn out to be an effective and perhaps even necessary property.

Individual behavior that is mechanically determined by a few individual behavioral rules could lead to dysfunctional behavior. It could, for instance, result in fixation and lack of variety, because every individual would act in exactly the same way. Take the use of individuals' copying tags. If the rules were completely unambiguous, then every individual would copy the same tags and select the same alternative, resulting in fixation on one alternative. Or consider the rule that states that individuals select licenses that provide sufficient motive for others to participate. If there were one alternative that provided sufficient motive, then every individual would select this same license and no new alternatives would be created – thus reducing the trial and error in trading off protection of code versus attraction of new contributors.

The rules, however, predict a different pattern of behavior. The reason is the presence of ambiguity. The rules do not allow us to determine or predict the actual behavior of every single participant in the community. Neither do the individual behavioral rules predict that individuals faced with similar conditions will make the exact same decision. Instead, their behavior is ambiguous, for two reasons.

The first reason is that the meaning of the rules is ambiguous. What is, for instance, a 'simple activity'? Or, which license provides sufficient incentive for others to contribute? Participants have different answers to these questions and thus make different choices and act differently.

The second reason to claim that individual behavior is ambiguous is that the rules create trade-offs. The rules describe situations in which individuals must choose between alternative strategies. A trade-off can be identified for most rules. Consider, for instance, the design principle 'multiple layers of nested enterprise.' Two rules were defined: (i) Participants perform activities which they perceive they are able to complete successfully. (ii) Participants perform activities of which they perceive their contribution to be important. The two rules imply a trade-off, because participants must choose either to perform simple activities with low rewards but with a high chance of success, or they perform a more complex activity. If the activity is performed successfully they reap greater rewards; but the chance of success is lower.

The presence of ambiguity and trade-offs in the rules highlights the fact that the behavior of each individual cannot be predicted. The rules leave room for individual choice. They predict that participants have an incentive to diverge and look for new alternatives. The presence of ambiguity and trade-offs thus predicts a continuous rise of variation and they give rise to trajectories of trial and error.

Institutions as a special kind of Potemkin village: incentives for people to create institutions

The question that remains is 'Why did the communities create institutions?' Or perhaps more importantly, 'Why do so many respondents attach value to their presence?' This section proposes and discusses three possible explanations, which combined describe a strong incentive to create institutions, even though the actual influence of the institutions may be rather limited.

A first explanation is that few of the respondents understand their own way of working, simply because they are not worried about why and how things are done. Instead, they focus on actually doing. In fact, the observation that professionals do not understand their way of working is not new. Many researchers have pointed to the importance of tacit knowledge in the work of professionals and the difficulty of making this type of knowledge explicit (Nonaka & Takeuchi 1995, Schön 1983). There are some clues that this is the case for the professionals in open source communities. As an example, in one interview a respondent argued that conflicts are resolved through voting systems. Every time a conflict arises, the respondent said, a vote is held. However, when asked whether people actually act according to the outcome of a vote, the answer was no. Individuals were argued to be free to ignore the outcome and proceed with their activities. In other words, voting system in this community did not resolve the conflict. Thus, when asked to explain how conflicts are resolved the respondent referred to the most obvious mechanism, an institutional mechanism, the voting system. This could be one of the reasons why the voting system was created. Individuals might have created the system with the idea that it could help resolve conflicts. This strategy is understandable for individuals who overlook the emergent mechanisms through which conflicts are actually resolved.

A second explanation for the existence and proclaimed importance of formal institutions is the desire to gain external recognition. Chapter ten argued that many of the discussions about open source software and the communities are polarized. The critique of open source communities is that they are chaotic and ill-structured. They are also frequently portrayed as nerdish clubs. A strategy to counter some of this critique could be to copy and create formal collective institutions as are found in many software development companies. This could include mechanisms like voting systems, project teams and appointed project leaders. These mechanisms can then be used to counter some of the critique and explain and argue that the communities are far more structured and organized than portrayed by the critics. Whether these mechanisms actually influence the choices and actions of the participants is less important.

A third explanation, closely related but different from the first and second explanation, is that institutions are used as a means to understand and explain the apparent success of open source communities and the software. How can one understand the success of open source, if one realizes that the development process is driven by individual actions and choices? How is it possible that open source software is reliable in the absence of codified testing procedures or quality-control teams? How do things get done, if people on mailing lists only seem to discuss and argue that their solutions and ideas are better than everyone else's? These questions refer to a problem with a conclusion that is based on emergence, namely, it is difficult to understand. It is counterintuitive. It is much easier to assume that these collective patterns are purposefully created (e.g. Resnick 1994). Therefore, one solution is to look for institutions, like voting systems and elections of project leaders, for which there are existing organizational narratives connecting them to success. It is much easier to argue that a project was a success because of strong leadership than because of the aggregate outcome of individual activities that were pursued under different motives.

A summarized answer to the research question

We can now summarize the answer to the research question, which read as follows: 'How are open source communities organized and how do they sustain themselves?'

First, open source communities are self-organizing; the local interactions of individuals emerge into collective patterns of behavior. A limited number of individual behavioral rules combined with the mechanisms that were identified in each chapter are sufficient to understand how open source communities are organized and how they sustain themselves.

Second, the institutions in open source communities, like project leadership and voting systems, provide no answer to the research question. The institutions in fact appear to be constructed for reasons other than their effectiveness in regulating behavior.

Third, many individuals want to achieve their personal goals, which results in redundancy and waste. However, it also ensures the continuous drive for individuals to create variation, trial and error and innovation in the communities.

Open source communities are self-organizing: why this research is different from other explanations

The conclusion of this research is that open source communities are self-organizing. The interactions among individuals in the communities give rise to complex patterns of behavior on a collective level. The conclusion that open source communities are self-organizing is important, since it implies that there are limits to the malleability of open source communities. Processes and structures in the communities cannot be purposefully created or copied to other settings. To claim that open source communities are self-organizing does not mean, however, that the outcomes of this research are similar to other publications in which open source communities are claimed to be self-organizing.

First, this research has resulted in an organizational model of open source communities that is more than just a metaphor. One of the common points of critique of existing explanations based on self-organization is that they are merely metaphors. They fail to explain how local interactions result in a global order (Weber 2004). This research is different because it takes practice as its starting point. Adopting the lessons from community-managed common pool resources resulted in an empirical framework by which open source communities could be analyzed. Furthermore, this research strived to understand how self-organization comes about. Individual behavioral rules were defined to understand how individual behavior gives rise to patterns of collective behavior. All this moved the conclusion of this research beyond the level of metaphors.

Second, although the claim is that the communities are self-organizing, this research acknowledges the presence of institutions. Indeed, some of the institutions were argued to perform a crucial role in the way communities organize and sustain themselves. Partly, the institutions were created with the intent of protecting the communities from the outside world. In that sense, they appear to be necessary to allow self-organization within the communities themselves.

Third, the individual behavioral rules identified in this research are ambiguous. This is different from the rules identified in other self-organizing systems. These rules typically have an 'if… then…' structure, which allows the actions of individual agents to be modeled in a computer program (Resnick 1994). The rules put forward in this research, however, leave room for interpretation. Participants act differently even though they face similar conditions and even though they act based on the same basic rules.

Propositions

Based on the main findings in this research, a number of propositions can be put forward. On one hand, these propositions are intended as a summary of the main findings. On the other hand, they are intended as hypotheses that could lay the basis of future research.[40]

Proposition 1: The organizational structure of open source communities is intelligent since it can result in software that is comparable in quality to proprietary software.

Proposition 2: Intelligence at the organizational level of open source communities is by and large an emergent property and only marginally attributable to collective institutions.

Proposition 3: Intelligence at the organizational level emerges from individual behavior and can be understood with a limited number of basic individual behavioral rules.

Proposition 4: The individual behavioral rules contain incentives that reward convergence of individual behavior.

Proposition 5: The presence of ambiguity and trade-offs in the individual behavioral rules ensures constant variation, thereby creating many different trajectories of trial and error.

Proposition 6: The institutions in open source communities that do play a significant role do so because they (i) increase the external recognition of communities and (ii) are a means to protect the communities from outside pressures.

Implications and reflection

This section answers two questions: (i) what are the implications of this research and (ii) what are its limitations?

Implications

First, open source communities are innovative. One of the important implications of this research is related to the individual behavioral rules. Ambiguity and trade-offs in the behavioral rules suggest that participants in open source communities have a continuous drive to create variety and start different trajectories of trial and error. Combined with the relatively low barriers to create new development lines, it implies that open source communities are innovative. People who contribute to the communities are triggered to create new and different alternatives to existing solutions.

Second, innovation in open source communities requires redundancy. The fact that individuals have a drive to create alternatives to existing solutions implies that open source communities have high levels of redundancy. It results in the creation of unused alternatives and hence results in waste. Redundancy is, however, needed to ensure innovation in open source communities.

Third, although the level of innovation in open source communities is impressive, the underlying organizational model is surprisingly simple. We can understand the organization of open source communities and the way in which innovation in the communities is achieved. They emerge from the behavior of individuals. The communities are thus self-organizing. Such an explanation is surprisingly simple, simpler than we might have held possible. Especially when we realize that the organizational model is intelligent and has resulted in software which is often comparable in quality to proprietary software.

Fourth, although the model is simple, we cannot control open source communities. The fact that open source communities are self-organizing implies that their direction cannot be directed and/or controlled.[41] The development of software in the communities is ad hoc; it is a patchwork of contributions received from many different individuals.[42]

Fifth, although the model is simple, it is doubtful that we can purposefully design open source communities. We should seriously doubt whether we can actually design open source communities. Elements of the organizational model are interesting and can provide valuable lessons to other organizations. However, by and large the findings of this research imply that an open source community cannot be purposefully designed.

These implications might seem unsatisfying, but in fact they are a valuable lesson. They tell us that we need to carefully use and handle the organizational model and that we cannot simply copy the model to other industries. Furthermore, we cannot simply use the model to design and create new and successful communities from scratch.

Reflection

In every type of research choices have to be made. These choices affect the type of outcomes of the research and the nature of the conclusions. In order to put the findings of this study into perspective, we reflect on some of these choices and their consequences.

This research focuses on larger and more successful open source communities. The two communities that received most attention are Apache and Linux. One reason for selecting these communities was that they provide ample empirical material. Another reason was the choice of research framework. The design principles explain why and how communities are able to self-organize; they explain success. Obviously, this brings up a wide variety of methodological questions like what is success in the communities. Chapter three provided an answer to this question. Whether one agrees with that answer or not, the point is that the selected communities are not representative of the wide array of small and unknown open source communities that are, for instance, listed on the SourceForge website. Many of these have failed to attract users and a large percentage have even failed to create software (West & O'Mahony 2005). This means that the results of this explorative study cannot and should not be generalized to every open source community. Furthermore, research analyzing the reasons why certain projects have not been able to attract many participants could contribute to our understanding of the rise and fall of open source communities.

The data presented in this article is based on qualitative data: an extensive literature study, a sample of in-depth interviews (60 in total) and a number of discussions and statements from mailing lists. Other methods, like surveys and statistical analysis of forums like a mailing list and CVS, could have shed a different light on these finding. A constraint of these methods, however, is that researchers must first know what they need to analyze. They need to first have an understanding of the object of analysis and some sense of the causal linkages between the variables. However, when this research was started, there was little consensus about open source communities and the way they are organized. Even now there is little consensus. Therefore, exploration and sense-making were central elements of this study, which has led us to an open and qualitative approach. The inherent drawback of this approach is that it does not allow us to draw generalizable and more definite conclusions as to how open source communities are organized. It does, however, provide more support for future attempts in this direction.

The framework adopted in this research is based on the design principles as they were first identified by Ostrom (1990). The framework has provided a heuristics to analyze the communities and to identify a set of mechanisms that enable coordination and collaboration in the communities. Yet, as with any other framework, this framework too has the disadvantage that mechanisms and elements, which do not fit within the framework, might have been overlooked. A number of actions have been taken to try and minimize the chance that this would happen. Actions that were undertaken were for instance: (i) to conduct open-ended and semi-structured interviews in search of new mechanisms, (ii) to discuss the identified mechanisms with fellow researchers and participants in open source communities and (iii) to use four sources of data.

An interesting next step would be to test the individual behavioral rules defined in this research. However, one must remember that a test of the rules is far from straightforward. An environment would have to be created in which the actions of individuals can be isolated and simulated. Much thought and preparation would need to go into the creation of such an environment. One way to test the rules could be through the use of agent-based experiments (Dalle & David 2004, Resnick 1994). In this research the rules were primarily a means to validate the most important finding of the research, which is that seemingly random individual behavior can aggregate into complex and intelligent patterns of collective behavior. In other words, the rules were identified to support the findings; they are not a goal in and of themselves.

Directions for future research

The previous pages, starting with the propositions, have already listed and discussed a great number of directions for future research. In this section some additional directions are suggested.

The absence of collective institutions in open source

One interesting strand of future research would be to continue to analyze the role of self-organization in community-managed common pool resources. Ostrom claims that community-managed common pool resources are self-organizing and refers to the body of literature on self-organization (1999). However, she focuses on institutions to explain how self-organization in these communities is achieved. This research demonstrates that institutions are not always needed or present. In that sense, open source communities do provide a new and interesting field for investigating in more detail the emergence and role of institutions in self-organizing communities.

Lessons for corporations and governments

Organizations, both corporate and governmental, are increasingly interested in adopting lessons from open source communities as a 'new' way of creating software. This would make sense, as the communities do provide a new and different organizational form to motivate professionals and coordinate their joint efforts.

Currently, organizations adopt two strategies to adopt the lessons from open source communities. The first is to create a piece of software and to 'build' a community around it. Organizations hope that by licensing software with an open source license and attracting unpaid volunteers, they can improve and maintain their software without incurring high costs. In other words, they view open source communities as an inexpensive way to develop and maintain their software packages. The second strategy is to transfer lessons learned from open source communities into an organizational setting. An example is the Dutch consumer electronics giant Philips.[43] A large business unit of this company decided to change its software development tactics by adopting an 'inner source' development methodology.

Such attempts to transfer lessons from open source are exciting, but they do have risks. This research demonstrated that institutions in open source communities have little influence on the actual processes. Furthermore, the communities are self-organizing, which means that there are definite limits to their malleability. Combined, these factors create a challenge for anyone wanting to transfer the 'open source model' to other settings. Implementing a voting system or electing a project leader will not automatically result in a sustainable and effective software development community. Basically, the question is whether and, if so, how self-organization can be purposefully created. Research is needed to understand if and how open source communities can be purposefully created and/or the underlying organizational model can be copied to other settings.

Is open source software development better than proprietary software development?

The goal of this research was not to analyze whether open source communities are a better model for creating software, compared to proprietary software development models. Neither was the goal to understand whether open source software is better than proprietary software. However, the open source development model does differ from the proprietary software development model in a number of respects. The networked character of the community is probably the most fascinating of these. Also, many questions still surround open source software. For instance, is open source software secure and reliable?

Research on such questions is still very much dominated by the rhetoric of the proponents and opponents of open source. Future research should investigate such questions in more detail and should contribute to our understanding of open source software and its development model. The primary task of this strand of research would be to sensitize researchers and practitioners as to when an open source development model would better fit with a given set of criteria and when not.

Further research should also be conducted to understand aspects of the organization of open source communities in relation to the software they create. For instance, how do open source communities achieve innovation? How do open source communities create software that is reliable and sustainable? This latter question is at the heart of a study that began in January 2005, funded by the Netherlands Organisation for Scientific Research (NWO).[44]

How do open source communities differ from other types of organizations?

An interesting line of research would be to compare the open source communities to other types of organizations. Here the question is 'What organizations are comparable?' Should the communities be compared to other types of virtual teams (e.g. Rasters 2004)? Or should they be compared to other organizational forms in which software is developed (e.g. McKelvey 2001a)? The fact that in this research the communities are not compared with other types of organizations prevents us from claiming that open source communities are unique and that the mechanisms will not be found in other organizations. Mechanisms like modularity, elegance, a CVS and a coding style guide might also be found in other organizations. Yet many of the mechanisms and processes described are different from those typically found in commercial companies in which software is developed. The choice to study open source communities in isolation limits our ability to reflect on whether there are other organizations with a structure similar to that of open source communities and/or whether a structure based on open source communities is relevant for other sectors, organizations or professions.

Notes on chapter eleven

[1] Many researchers claim that the pressure of free riding is not a serious problem. Yet it is a potential problem, which is more than just hypothetical. Open source software will only be created if the communities manage to continuously attract new contributors and retain existing ones.
[2] The claim is not that software developed in open source communities is qualitatively better than proprietarily developed software. The quality of proprietary software or open source software differs for each individual software program. Furthermore, the relationship between the quality of the software and the way in which it was developed is all but understood. The only claim made here is that in certain segments of the software market open source software has been able to gain a share of the market and has apparently reached a satisfactory level of quality, which in itself could be seen as a surprise.
[3] One example is the *New York Stock Exchange*: http://www.it-director.com/article.php?articleid=2125 (November 2003) and in May 2003 the city council of Munich decided to switch 14,000 desktops away from Microsoft to open source software (http://www.wired.com/news/infostructure/0,1377,62236,00.html, March 2004).
[4] Apache is estimated to host more than 65 percent of all active websites (from http://www.netcraft.com/ (July 2004).
[5] From an article on the Internet: http://www.infoworld.com/article/03/07/01/HNreasoning_1.html (March 2004).
[6] From the Internet: http://www.reasoning.com/news/pr/02_11_03.html (July 8, 2003).
[7] One exception is an article by O'Mahony SC. 2003. Guarding the Commons: How Community Managed Software Projects Protect Their Work. *Research Policy* 32: 1179-98 She explicitly addresses how open source communities protect themselves against external pressures. She then focuses on the boundaries constructed in open source communities.
[8] Next to the licenses communities have other boundaries to protect them against external pressures. These include, for instance, trademarks and foundations. Ibid.. This section, however, is limited to open source licenses and foundations, as they are believed to be the most important boundaries in open source communities.
[9] Measured September 10, 2003.
[10] From an interview with the vice-president of the FSF.
[11] According to the statistics on the Freshmeat website, 65 percent of the projects listed use the GPL. From the website software.freshmeat.net/stats/ (May 21, 2003).

[12] For example, the Apache Software Foundation spends thousands of dollars each year in legal representation of the foundation and the individual contributors.
[13] Also based on an interview with two members of the ASF, who argue that the BSD license is a very reasonable one. They reason that a company that invests time and effort in the improvement of a software program should be free to make a return on that investment and therefore the license should have fewer restrictions than imposed by the GPL.
[14] The maintainer of the translation of KDE into Dutch describes the lack of such an institution: "Coordination right? In the beginning we did that: 'he is doing that part and he is doing the other.' But that did not work. Why? … We spent more time keeping track of who did what than translating."
[15] Elegant source code is that which is structured and reductive. It is a term used to indicate that the software works and, at the same time, is a notion of beauty. "So from the standpoint of a small group of engineers, you're striving for something that's structured and lovely in its structuredness." Cited from an interview with Ellen Ullman by Scott Rosenberg in 1997. The interview was published on the Internet: http://archive.salon.com/21st/feature/1997/10/09interview.html (August 2002).
[16] Modularity is based on the idea of divide and conquer, see: Dafermos GN. 2001. Management and virtual decentralised Networks: The Linux Project. *First Monday. Peer reviewed journal on the Internet* 6: downloaded from the Internet: http://www.firstmonday.dk/issues/issue6_11/dafermos/ (December 2004) Modular software is divided into smaller pieces, building blocks, which together create a working software program.
[17] From an interview with the editor in chief of the Linux journal.
[18] A maintainer in the Linux kernel community states, "You can only work when the software is beautiful and elegant, to be able to implement changes easily."
[19] Agreement is required on what the modules are and how the interfaces should be defined. This could result in much conflict and disagreement. However, for this too the answer appears to be in 'doing,' as evidenced in many communities analyzed in this research. For instance, in the Apache community one of the developers sat down and decided to rewrite the HTTP server and made the design modular. Others accepted and adopted the changes. The principle of doing is explained more elaborately elsewhere in this book.
[20] See www.apache.org/dev/styleguide.html (May, 2003) and www.linuxjournal.com/article.php?sid=5780 (May, 2003).
[21] The website can be found at http://www.debian.org/devel/wnpp/work_needing (April 8, 2003).
[22] From an interview with one of the two maintainers of the Lilypond software.
[23] From an interview with a member of the KDE community.
[24] From an interview with the editor in chief of the Linux journal.
[25] Both quotes are from an interview with the project leader of the PostgreSQL community.
[26] From the Internet: http://freshmeat.net/ (July 2004).
[27] Consider, for instance, this statement by a respondent: "You judge the maintainability of the code, the level of activity of the community and whether or not the code is understandable."
[28] From an interview with a package maintainer in the Debian community.
[29] From an interview with a member of the steering committee of PostgreSQL.
[30] From an interview with a member of the ASF Board of Directors.
[31] The term 'hall of blame' is borrowed from two members of the Apache Software Foundation who used it to refer to the fact that everything they write remains visible on the Internet, creating an incentive to do things right.
[32] Another example is given by a system maintainer in CNet who is also a member of the ASF: "We are not experts in coding, but we bring in something else… we contribute in giving cases or certain situations where Apache can be improved, based on real business situations."
[33] From an interview with a Linux kernel developer.
[34] One of the project leaders of Lilypond states, "Users also have many questions. Other users frequently solve them."
[35] A respondent from the BlueFish community explained how he created his own new task, which is to translate BlueFish into Dutch. "I tried some programs and BlueFish was the first at which I succeeded. I translated a little part of the software and then joined the BlueFish mailing list. There I asked whether someone already translated BlueFish into Dutch. No one had."
[36] From an article on the Internet: http://news.com.com/Linux+contributors+face+new+rules/2100-7344_3-5218724.html (October 2004).
[37] From the Internet: http://www.nwc.com/showitem.jhtml?docid=1511buzz1 (October 2004).
[38] This observation is based on quotes from a number of respondents. Consider this statement from a former project leader of the Debian community: "With Corel I talked about distributions, with IBM

about support." Or consider the statement about the creation of the Apache Software Foundation, in which the respondent described how IBM wanted to have "a contact point. Someone they could talk to."

[39] This point was made at a number of workshops with, among others, representatives of Dutch government agencies. One concern that has prevented them from participating in the communities is the threat of liability.

[40] Thanks to Tom van Engers who suggested the addition of propositions and to Maura Soekijad who provided a good example of how propositions can be included in the concluding chapter of a thesis.

[41] There are, however, ways to try to influence the direction of a project. One could for instance pay one or more developers in a community to create a certain feature.

[42] This finding was confirmed in a session at the Holland Open Software Conference. The president of the Apache Software Foundation and the project leader of Blender both agreed that the development of open source is ad hoc.

[43] Based on a talk by a Philips representative, given at the MMBase conference in Delft, June 9, 2004.

[44] The research will be performed by the Delft University of Technology. The project number is 458-03-003.

SAMENVATTING (SUMMARY IN DUTCH)

DE ORGANISATIE VAN OPEN SOURCE COMMUNITIES

Dit onderzoek richt zich op open source communities, welke ook wel bekend staan onder namen als 'free software communities' en 'libre software communities.' Open source communities zijn virtuele organisaties waarin software ontwikkeld wordt. Bekende voorbeelden zijn de Linux en de Apache community. Een kenmerk van open source communities is dat de broncode van de software open is en vrij gedeeld wordt. De broncode is noodzakelijk om software te kunnen begrijpen en aan te passen. De openheid van broncode stelt duizenden vrijwilligers en betaalde werkkrachten uit alle delen van de wereld in staat om software te ontwikkelen en te onderhouden. Het eigendom van de software is meestal niet in handen van één persoon of bedrijf, maar is verdeeld over de bedrijven en individuen die aan de ontwikkeling van de software hebben bijgedragen.

Er zijn een aantal redenen waarom onderzoek naar open source communities waardevol is. Ten eerste is er verrassend weinig bekend over de wijze waarop ontwikkelaars in de communities georganiseerd zijn en hoe de continuïteit van de communities gewaarborgd wordt. Veel onderzoek lijkt uit te wijzen dat mechanismen - zoals centrale sturing (Kuwabara, 2000; Mockus et al., 2003) contractuele relaties (Franck et al., 2003) en organisatiegrenzen (Fielding, 1999) - welke voor coördinatie noodzakelijk worden geacht, in de communities afwezig zijn.

Ten tweede neemt de afhankelijkheid van open source software toe. Cijfers wijzen uit dat zevenenzestig procent van alle actieve Internet websites ondersteund wordt door het open source programma Apache en tachtig procent van al het e-mail verkeer wordt mogelijk gemaakt door het open source product Sendmail. Open source software wordt niet alleen gebruikt door kleine organisaties en hobbyisten. Een toenemend aantal grote professionele organisaties ondersteunt bedrijfskritische toepassingen met open source software. Een voorbeeld is *Amazon* die haar belangrijkste verkoopkanaal, de Amazon.com website, baseert op open source software. Andere voorbeelden zijn de *New York Stock Exhange* en *IBM*. Ook overheden en gemeentes wagen de overstap naar open source software. Voorbeelden zijn de gemeente Munchen en de centrale overheid in Brazilië.

Ten derde lijken overheden het gebruik van open source software te stimuleren. Een goed voorbeeld hiervan is de beleidsnotitie: "Software open U!" (2002). In de notitie bepleiten de auteurs het gebruik van open standaarden en open source software in de Nederlandse publieke sector. Mede naar aanleiding van de notitie is het programmabureau OSOSS (Open

Standaarden en Open Source Software) opgericht. Eén van de doelstellingen van het programmabureau is het informeren van publieke organisaties over open source software.

Samenvattend kan gesteld worden dat we relatief weinig begrijpen van de wijze waarop de communities georganiseerd zijn en hoe de continuïteit van open source communities en software gewaarborgd worden. Een dergelijk begrip lijkt echter noodzakelijk, immers a) steeds meer organisaties zijn afhankelijk van open source software en b) overheden lijken het gebruik van open source software te stimuleren. Dit brengt ons bij de vraag die centraal staat in dit onderzoek, namelijk:

Hoe zijn open source communities georganiseerd en hoe wordt de continuïteit gewaarborgd?

Het huidige onderzoek naar de organisatie van open source communities kan worden ingedeeld in twee groepen. De eerste groep maakt gebruik van metaforen om de communities te typeren. Een deel van het onderzoek in deze categorie beargumenteert dat de communities zelforganiserend zijn. Zelforganiserende systemen zijn systemen waarin centrale sturing afwezig is en waarin coördinatie het gevolg is van de interactiepatronen van lokale agenten. Zij stellen dus dat open source communities niet volgens een vooropgezet plan ontworpen zijn en dat centrale sturing afwezig is (Axelrod et al., 1999; Bekkers 2000). Een verklaring van de communities moet gezocht worden op het niveau van de individuele ontwikkelaar.

De tweede groep richt zich niet zozeer op de gehele organisatie van de communities, maar probeert kleinere puzzelstukjes die de werking van de communities kunnen verklaren, te begrijpen. Dit type onderzoek wijst uit dat steeds meer processen op een collectief niveau ingericht en aangestuurd worden. Voorbeelden van mechanismen die coördinatie mogelijk maken, zijn open source licenties, project leiderschap en geautomatiseerde mailing lijsten. Een verklaring op het niveau van het individu wordt door onderzoekers in deze groep vaak gezien als te generalistisch en simplistisch.

De verklaringen uit beide groepen lijken niet met elkaar te rijmen. Immers, de één beweert dat een verklaring gezocht moet worden in het gedrag van individuele ontwikkelaars. De ander claimt dat processen op een collectief niveau bepaald worden.

Hoofdstuk 2 & 3: Theoretisch raamwerk en onderzoeksaanpak

Het theoretische raamwerk in dit onderzoek is gebaseerd op de uitkomsten van onderzoek naar collectieve goederen die in communities beheerd worden. Elinor Ostrom en anderen hebben uitvoerig empirisch onderzoek gedaan naar specifieke collectieve goederen, namelijk "common pool resources". Een common pool resource onderscheidt zich van andere collectieve goederen doordat er sprake is van rivaliserend gebruik. Bij overmatige consumptie wordt de continuïteit van het goed bedreigd. Voorbeelden van common pool resources zijn drinkwatervoorzieningen en visserijgronden.

Ostrom en anderen tonen aan, dat onder sommige condities individuen in staat zijn zichzelf te organiseren in communities en er gezamenlijk in te slagen de continuïteit van het goed te waarborgen. Ostrom (1990) definieert ontwerpprincipes die kunnen verklaren waarom in sommige communities zelforganisatie mogelijk is en waarom in andere niet. In tabel S.1 zijn de principes in vereenvoudigde en aangepaste vorm weergegeven. Dit onderzoek is gebaseerd op de ontwerpprincipes van Ostrom. De principes doen recht aan het vermogen van mensen om zichzelf te organiseren zonder sturing van buitenaf.

Tabel S.1 – de ontwerpprincipes

1.	Eenduidig gedefinieerde grenzen
2.	Regels voor productie
3.	Toegang tot conflicthanteringmethodes
4.	Mechanismen voor collectieve besluitvorming
5.	Toezicht en sanctionering
6.	Meerdere niveaus van organiseren
7.	Erkenning door externen

In dit onderzoek zijn de ontwerpprincipes systematisch afgelopen. Voor elk principe is dezelfde vraag gesteld, namelijk: hoe wordt in de communities invulling gegeven aan het principe? Om deze vraag te beantwoorden zijn 60 diepte interviews gehouden. Naast interviews is secundaire literatuur geraadpleegd en is de communicatie tussen ontwikkelaars op internet mailing lijsten geobserveerd. De communities die in dit onderzoek de meeste aandacht hebben gekregen zijn Linux, Apache, Debian, PostgreSQL en Python. De communities genieten grote naamsbekendheid en ze weten relatief veel ontwikkelaars te mobiliseren.

Op de komende pagina's worden per principe de belangrijkste observaties besproken. Vooruitblikkend op de bevindingen kan gesteld worden dat één bevinding dominant is, namelijk: individuen laten zich in hun handelen voornamelijk leiden door hun eigen keuzes. Veel respondent verklaren dat zij de formele instituties, zoals stemsystemen, negeren. In plaats daarvan maken ze hun eigen keuzes. Op de komende pagina's wordt het keuzegedrag en de bijbehorende structurerende mechanismen in meer detail besproken.

Hoofdstuk 4: Eenduidig gedefinieerde grenzen

Open source communities kennen geen echte organisatiegrenzen, dat heet grenzen die bepalen wie wel of niet mag toetreden tot de community. Wel zijn er grenzen die bepalen wat er met de software gedaan mag worden. Deze regels zijn er vooral op gericht om de communities te beschermen tegen partijen die zich (delen van) open source software programma's willen toe-eigenen en vervolgens het gebruik door anderen onmogelijk willen maken. Deze vorm van toe-eigening van open source software wordt mede mogelijk gemaakt door intellectuele eigendomsrechten, als softwarepatenten (Boyle, 2003; Bollier, 2001).

De belangrijkste grenzen zijn de *open source licenties*. De licenties zijn gebaseerd op auteursrecht. De meeste licenties stellen dat de broncode van de software open is en open dient te blijven. Aan de ene kant vormen de licenties de belangrijkste manier waarmee open source software beschermd wordt tegen buitenstaanders die zich de software willen toe-eigenen. Aan de andere kant dienen de licenties open te zijn zodat ze anderen een incentive geven om aan de ontwikkeling van de software bij te dragen. Ontwikkelaars maken hun eigen keuze in deze trade-off tussen openheid en geslotenheid. Gecombineerd met het feit dat iedereen in staat is een nieuwe licentie te maken, verklaart dit de grote variëteit aan licenties. Volgens het Open Source Initiative (OSI), een stichting die het concept "open source" promoot, zijn er bijvoorbeeld 48 verschillende open source licenties. De licenties zijn dynamisch; ze worden aangepast aan nieuwe bedreigingen en uitdagingen die ontstaan door bijvoorbeeld technische veranderingen. Ondanks de grote variëteit aan licenties is convergentie ook waarneembaar. Zo is de General Public License (GPL) veruit de populairste licentie en wordt zij in ongeveer 75% van de projecten gebruikt.

Een tweede grens wordt gevormd door *stichtingen*. In sommige communities zijn stichtingen opgericht. De voornaamste doelen van de stichtingen zijn a) het voorkomen en bestraffen van overtredingen op de licenties en b) de bescherming van ontwikkelaars in de communities tegen mogelijke rechtszaken (O'Mahony, 2003). De stichtingen lijken vooral gericht op het bewaken en het juridisch inbedden van open source licenties. Daarnaast valt op dat de stichtingen, gemeten in aantallen mensen en hoeveelheid geld, verrassend klein zijn. Dit ondersteunt de bevinding dat de stichtingen bovenal juridische constructen zijn die een sociale omgeving in stand houden waarbinnen ontwikkelaars vrijelijk broncode met elkaar kunnen delen. De stichtingen hebben weinig invloed op de processen in communities.

Een derde mechanisme wordt gevormd door *Internet mailing lijsten*. Op de lijsten informeren ontwikkelaars elkaar over de nieuwste ontwikkelingen met betrekking tot de licenties. De mailing lijsten worden ook gebruikt om overtredingen op de licenties onder de aandacht te brengen. Ze versterken daarmee de doorwerking van de open source licenties.

Hoofdstuk 5: Regels voor productie

Veel respondenten in dit onderzoek gaven aan, dat zij zelf besluiten waar ze aan willen werken. Dit betekent vaak dat ze doen waar ze op dat moment zin in hebben. De vraag is hoe een dergelijk proces kan resulteren in software die werkt. Op enige wijze is afstemming tussen de ontwikkelaars nodig. Het blijkt, dat in open source communities een groot aantal mechanismen aanwezig zijn die de ontwikkelaars in staat stellen met minimale inspanning en zonder veel onderlinge afstemming een nuttige bijdrage te leveren.

Twee mechanismen verminderen de noodzaak tot het maken van onderlinge afspraken. De mechanismen liggen beide besloten in de broncode van open source software. Zo streven ontwikkelaars *elegantie* na. Elegantie refereert aan schoonheid en eenvoud. Elegante broncode maakt het mogelijk dat ontwikkelaars in één oogopslag kunnen begrijpen wat die broncode probeert te bereiken. Elegantie minimaliseert de energie die nodig is om broncode, geschreven door anderen, te begrijpen. Een ander mechanisme is *modulariteit*. Modulaire software bestaat uit kleinere, relatief onafhankelijke delen. Ontwikkelaars kunnen hierdoor relatief onafhankelijk van elkaar veranderingen aanbrengen in modules.

Ook zijn er mechanismen die enige vorm van afstemming tussen ontwikkelaars mogelijk maken. Deze mechanismen zijn opgesomd in tabel S.2.

Hoofdstuk 6: Toegang tot conflicthanteringmethoden

Open source communities kennen een grote verscheidenheid aan ontwikkelaars, die verschillende en vaak conflicterende belangen hebben. Dit draagt bij aan een groot potentieel voor conflicten, die de continuïteit van de communities in gevaar kunnen brengen. De vraag is hoe conflicten in open source communities opgelost worden.

Tabel S.2 – Mechanismen die afstemming tussen ontwikkelaars mogelijk maken.

Mechanisme	**Omschrijving**
Concurrent Versions System	Database systeem waarin ontwikkelaars gelijktijdig en gedistribueerd aan software kunnen werken.
Mailing lijsten	Een gedistribueerd communicatiemiddel
Handboeken	Beschrijving hoe broncode geschreven moet worden.
"Bug-tracking" systemen	Fouten in de software kunnen in dit systeem asynchroon en door iedereen worden gemeld en opgelost.
"To-do" lijsten	Lijsten met activiteiten die nog opgepakt moeten worden.
Weeshuis	Parkeerplaats voor projecten die geen leider hebben.
Extra tekst in broncode	Extra tekst bij broncode als uitleg
Namen bij veranderingen	Bij elke toevoeging van broncode wordt de naam van auteur toegevoegd die daarmee aanspreekpunt wordt
Kleine en incrementele "patches"	Broncode wordt in kleine brokken aangeleverd, zodat het makkelijker is om veranderingen te begrijpen

Conflicten in de communities lijken niet opgelost te worden. Conflicten vormen echter geen bedreiging voor de continuïteit van processen in de communities. De reden is dat ontwikkelaars conflicterende meningen, handelingen en oplossingen kunnen negeren. De aanwezigheid van een aantal mechanismen maakt het mogelijk dat een proces niet geblokkeerd wordt door een conflict en zorgt ervoor dat ontwikkelaars hun activiteiten kunnen continueren. Deze bevinding is het makkelijkst uit te leggen aan de hand van twee scenario's.

In het eerste scenario is er één oplossing voor een bepaald softwarematig of andersoortig probleem en er ontstaat een conflict of die oplossing de juiste is. In deze situatie zullen de voorstanders van een bestaande oplossing geneigd zijn de tegenstanders te negeren; tegenstanders moeten eerst maar bewijzen dat hun alternatief beter is. Opponenten van een oplossing worden aangespoord om hun woorden in daden om te zetten. Dit kunnen zij doen door een *parallelle ontwikkelingslijn* op te starten, waarin zij hun ideeën in een alternatief kunnen omzetten. Het staat iedereen vrij om een dergelijke lijn op te starten. Deze mogelijkheid zorgt ervoor dat conflicterende partijen hun activiteiten kunnen continueren, waardoor het conflict geen effect heeft op de continuïteit van de communities. Een negatief neveneffect is dat het kan leiden tot redundantie: er worden meerdere oplossingen gemaakt voor 1 probleem.

In het tweede scenario zijn twee of meer alternatieven beschikbaar. Ook nu kan het conflict genegeerd worden. Iedereen kan zijn eigen keuze maken tussen de alternatieven.

Hoofdstuk 7: Mechanismen voor collectieve besluitvorming

De voorgaande pagina's beschreven mechanismen die bijna allemaal bijdragen aan een toename van divergentie in software. Ontwikkelaars kunnen als ze het ergens niet mee eens zijn hun eigen alternatief ontwikkelen. Dit geldt voor software maar ook voor bijvoorbeeld de licenties. Om fragmentatie en incompatibiliteit tegen te gaan zal op enig moment de divergentie ingedamd moeten worden (Egyedi et al., 2004). De vraag is hoe?

Wederom ligt het antwoord op de bovenstaande vraag bij de keuzes en het gedrag van individuele ontwikkelaars. Zij besluiten zelf welke software ze willen gebruiken, in welke community ze een bijdrage leveren, welke open source licentie ze adopteren etc. Rem op de divergentie vormt de beperkte tijd van ontwikkelaars en gebruikers. Het is onmogelijk om elke keer de software en licenties te onderzoeken en te analyseren. Dit zou te veel tijd kosten. Om die reden laten veel ontwikkelaars zich in hun keuze leiden door een aantal "tags." "Tags" zijn in essentie niet meer dan signalen die door anderen gevolgd worden (Axelrod, et al., 1999). Voorbeelden van signalen zijn de kleding, het handschrift en de haardracht van een persoon. Deze signalen vertellen ons iets over een bepaalde persoon en beïnvloeden onze perceptie en keuzegedrag. De "tags" in open source communities vormen een indicatie voor de kwaliteit van de software en verminderen de tijd die nodig is om een keuze te maken. Ze creëren positieve feedback, wat op collectief niveau leidt tot dominante keuzes. De "tags" zijn een rem op divergentie.

Het eerste mechanisme is *elegantie van de broncode*. Elegante broncode is vaak kwalitatief beter dan onelegante broncode. Ontwikkelaars prefereren daarom broncode die elegant is. Veel respondenten vinden het eenvoudig om te beoordelen welke broncode elegant is. Het tweede mechanisme is *reputatie van ontwikkelaars*. Ontwikkelaars zijn geneigd alternatieven te kiezen die (mede) ontwikkeld is door ontwikkelaars met een hoge reputatie. De reputatie van een ontwikkelaar is een indicatie voor de kennis en inzet van die ontwikkelaar. Participatie van dergelijke ontwikkelaars verhoogt de kans dat de software kwalitatief goed is. Het derde mechanisme, *mate van activiteit* van een project of community, wordt op veel websites automatisch door een teller bijgehouden. Hoe hoger de activiteit hoe groter de kans dat de software kwalitatief goed is. Het vierde en vijfde mechanisme zijn *distributies* en *websites met een softwarebibliotheek*. Distributies als Red Hat en SuSE en websites als Freshmeat, maken een keuze tussen verschillende programma's en projecten. Ze beïnvloeden daarmee de populariteit van projecten en dus de verwachte kwaliteit.

Hoofdstuk 8: Toezicht en sanctionering

Open source communities kennen veel kleine vergrijpen die regelmatig voorkomen. Voorbeelden zijn: een ontwikkelaar die broncode schrijft maar zich niet houdt aan de standaardstijl van de community of een ontwikkelaar die geen gebruik maakt van het databasesysteem dat door de community gebruikt wordt om software in op te slaan.

Een eerste bevinding is dat de meeste vergrijpen geen directe bedreiging zijn voor de continuïteit van een community. De reden hiervoor is dat communities een grote redundantie kennen aan mechanismen. Redundantie van de mechanismen maakt het systeem minder afhankelijk van één mechanisme en maakt het systeem als geheel minder kwetsbaar voor individuen die een overtreding begaan en een mechanisme negeren.

In het licht van de bevinding dat open source communities vooral kleine vergrijpen kennen die geen direct gevaar vormen voor de continuïteit, zijn de meeste bevindingen ten aanzien van toezicht en sanctionering begrijpelijk. Ten eerste wordt nauwelijks gebruik gemaakt van formele mechanismen om overtreders te sanctioneren. Zo is er de mogelijkheid om iemand het recht af te nemen om software direct in een database toe te voegen. Echter, dergelijke mechanismen worden volgens de respondenten niet of nauwelijks gebruikt.

Ten tweede betekent participatie in de communities automatisch ook het houden van toezicht. Er zijn drie redenen: a) de communities zijn zeer transparant waardoor de activiteiten van anderen eenvoudig te observeren zijn, b) ontwikkelaars worden automatisch op de hoogte gehouden van nieuwe ontwikkelingen en dus ook overtredingen en c) gebruik van de software houdt automatisch in dat de software wordt getest.

Ten derde zijn er veel mechanismen om overtredingen te 'bestraffen.' De consequenties van deze sancties zijn echter relatief mild en hebben betrekkelijk weinig consequenties. Voorbeelden van sanctiemechanismen zijn het schrijven van e-mails met vaak beledigende taal waarin uitgelegd wordt wat verkeerd is gedaan of het negeren van personen die een 'overtreding' hebben begaan.

Hoofdstuk 9: Meerdere niveaus van organiseren

De grote open source communities, zoals Linux en Apache, worden geconfronteerd met een enorme complexiteit. De software bestaat vaak uit miljoenen regels aan broncode, er zijn meerdere versies van dezelfde software in omloop en in sommige communities zijn duizenden ontwikkelaars en gebruikers actief. Een strategie om met deze complexiteit om te gaan is om de software modulair te maken en de activiteiten te verdelen in kleinere stukken. Op enig moment wordt afstemming tussen de taken en modules bereikt. Dit wordt vooral mogelijk gemaakt door de aanwezigheid van gerespecteerde ontwikkelaars en softwaredistributies. Zij maken verbindingen tussen modules en activiteiten mogelijk.

In open source communities wordt werk verdeeld. Opvallend is dat die werkverdeling op emergente wijze plaatsvindt. Respondenten geven aan dat ze zelf bepalen wat ze doen. Ze doen vooral die activiteiten die ze leuk vinden en die ze op basis van hun kunde en ervaring tot een succesvol einde denken te kunnen brengen. In sommige gevallen kan dit betekenen dat ze nieuwe activiteiten en rollen creëren die nog niet bestonden. De verdeling van werk leidt tot taakspecialisatie: ontwikkelaars kiezen er vaak voor om gedurende langere tijd dezelfde werkzaamheden te verrichten. Door specialisatie kunnen ontwikkelaars in relatief korte tijd een nuttige bijdrage leveren.

De verdeling van werk leidt ook tot mogelijkheden voor leren. Ontwikkelaars met veel kennis en ervaring besteden hun tijd vaak aan complexe taken. Reden is dat de waardering voor het succesvol afronden van relatief complexe taken relatief hoog is. De eenvoudigere taken laten ze daarom regelmatig liggen. Deze taken kunnen nu uitgevoerd worden door ontwikkelaars met minder kennis van zaken en door nieuwkomers. Het aanbod van minder complexe activiteiten creëert voor hen een leeromgeving.

Hoofdstuk 10: Erkenning van externen

De erkenning van externen voor de processen in open source communities is belangrijk om de continuïteit van de communities te waarborgen. Momenteel is er veel discussie over de processen en de structuur van open source communities. Twee goede voorbeelden van discussies zijn discussies rondom a) de veiligheid van open source software en b) de mate van innovatie. De discussies zijn vaak ongenuanceerd en ze worden omgeven door retoriek en metaforen. Zo wordt door tegenstanders van open source software gesteld dat de software onveilig is, omdat openheid van de software ertoe leidt dat iedereen met opzet ongewenste broncode kan toevoegen. Op hun beurt geven voorstanders aan, dat de openheid juist aan iedereen de mogelijkheid geeft om de software te controleren en problemen en fouten in de broncode te verwijderen.

Interessant is dat er in de communities mechanismen zijn gecreëerd die een deel van de kritiek van externe partijen wegnemen en daarmee wordt getracht om de erkenning van externe partijen te vergroten. Eén van die mechanismen is het Developers Certificate of Origin. Dit document is geschreven door Linus Torvalds, de oprichter van één van de meest bekende open source programma's, te weten Linux. Het document dient bij iedere bijdrage van een ontwikkelaar ondertekend te worden. De bedoeling is dat dit iets wegneemt van de angst van bedrijven die vinden dat het te onduidelijk is door wie de software geschreven is en waar ze vandaan komt. Andere voorbeelden zijn: het opzetten van de Linux Standard Base (LSB), de aanstelling van projectleiders en de oprichting van stichtingen en zelfs hele marketingafdelingen. Deze mechanismen hebben gemeen, dat ze ogenschijnlijk een eenduidiger structuur aanbrengen in de organisatie van open source communities. Daarnaast delen ze nog een ander kenmerk; ze hebben namelijk nauwelijks invloed op de keuzevrijheid en handelingsvrijheid van ontwikkelaars.

Conclusie

Een eerste conclusie is dat de principes uit Ostrom een vruchtbaar kader vormen voor een analyse van open source communities. Toepassing van de principes heeft geleid tot inzicht in de organisatorische processen in de communities.

Een tweede conclusie is dat de communities instituties kennen die relevant zijn, maar die een beperkte rol hebben in de organisatie van de communities. Sommige instituties, met name de stichtingen, vervullen een essentiële rol. Echter veel respondenten geven aan dat ze het overgrote deel van de instituties, zoals stemsystemen en projectleiderschap, negeren. Potentiële verklaringen voor het bestaan van instituties zijn: a) de instituties zijn een alternatieve verklaring voor het functioneren van de communities, b) de instituties vormen een manier om externe erkenning te krijgen, en gerelateerd aan de vorige redenen is c) de instituties zijn een manier om het succes van de communities te verklaren.

Een derde conclusie, welke overeenkomt met de bevindingen uit ander onderzoek naar open source, is dat open source communities zelforganiserend zijn. Ontwikkelaars in de communities geven aan, dat ze zelf kiezen wat ze willen doen. De bevindingen uit dit onderzoek laten zien dat het individuele keuzegedrag van ontwikkelaars, welke nauwelijks beïnvloed wordt door instituties, leidt tot verassend intelligente patronen op een collectief niveau. De aanwezigheid van zelforganisatie is belangrijk omdat het betekent dat de communities niet of nauwelijks maakbaar en kopieerbaar zijn.

De vierde conclusie is dat slechts een zeer beperkt aantal individuele gedragsregels nodig is om zelforganisatie in de communities te kunnen begrijpen. Een overzicht van de gedragsregels en de wijze waarop ze de bevindingen ondersteunen, kan gevonden worden in tabel 11.1 op pagina 211 van dit boek. Een aantal voorbeelden van de regels zijn: a) ontwikkelaars willen de tijd minimaliseren die ze nodig hebben om de broncode te begrijpen die door anderen is geschreven, b) ontwikkelaars willen aantonen dat zij gelijk hebben, c) ontwikkelaars willen de kans vergroten dat anderen hun bijdragen accepteren, en d) ontwikkelaars ondernemen activiteiten waarvan ze denken dat ze deze succesvol kunnen afronden. Deze regels sluiten goed aan bij de meeste bevindingen in dit onderzoek. Verder hebben de regels gemeen, dat ze zijn gebaseerd op de notie dat ontwikkelaars hun eigen nut proberen te maximaliseren. Dit betekent aan de ene kant dat de regels stellen dat ontwikkelaars hun tijd en energie zo efficiënt mogelijk willen inzetten. Ze zullen bijvoorbeeld zo min mogelijk tijd willen steken in het zoeken naar en het analyseren van de broncode die door anderen is ontwikkeld. Aan de andere kant betekent het ook dat ze de opbrengsten van hun inzet willen maximaliseren. Zo proberen ze de oplossingen die ze zelf hebben ontwikkeld door anderen geaccepteerd te krijgen.

We zouden geneigd zijn te denken dat het keuzegedrag van individuen dat bepaald wordt door een beperkt aantal regels leidt tot fixatie en stagnatie. Immers, ontwikkelaars laten zich leiden door dezelfde regels en maken daarom dezelfde keuze. Echter, het gedrag op basis van de gedragsregels is ambigu. Ten eerste zijn de regels zelf ambigu, ze laten ruimte voor interpretatie. Ten tweede creëren ze een trade-off. Zo kunnen ontwikkelaars proberen hun toekomstige opbrengsten te maximaliseren, maar ze kunnen ook hun verwachte kosten minimaliseren. In veel gevallen sluit de ene strategie de andere uit. Iedere ontwikkelaar zal een andere keuze maken in een dergelijke trade-off. Aan de ene kant maken de ambiguïteit van de regels en de trade-offs tussen de regels het onmogelijk om het gedrag van individuele ontwikkelaars te voorspellen. Aan de andere kant betekenen ze ook, dat er een continue stroom van variatie, divergentie en trial-and-error is. Het stelt de communities in staat te innoveren.

Tot slot. De uitkomst dat open source communities zelforganiserend zijn, lijkt misschien een magere en teleurstellende uitkomst, maar dat is het niet. Het leert ons een waardevolle les, namelijk dat we het organisatiemodel van open source communities voorzichtig moeten gebruiken en dat we het niet zonder meer in andere sectoren kunnen inzetten. We kunnen het model ook niet zomaar gebruiken om nieuwe en succesvolle communities te ontwerpen.

Ruben van Wendel de Joode, juli 2005.

APPENDIXES

Appendix A: List of respondents[1]

Respondents in the United States

Name	Company / Community	Profession / role in the community
Behlendorf, Brian	CollabNet Apache Software Foundation	Founder and CEO Member of the Board of Directors
Berkus, Josh	OpenOffice.org PostgreSQL	Volunteer head of marketing Volunteer head of marketing
Chahal, Raj	IXsystems	Sr. Sales Engineer
Coleman, Biella	University of Chicago Debian	PhD student Researcher
Davidson, James Duncan	Apache Software Foundation	Member
Dawson, Bruce	Greater New Hampshire Linux User's Group	Coordinator
Feldman, Jerry	Boston Linux/Unix User Group	Associate Director
Gulik, Dirk-Willem van	Covalent Technologies Apache Software Foundation	Vice-President, Research Member of the Board of Directors
Hall, John Maddog	Linux International Linux	Executive Director User and advocate
Hathaway, Shane	Zope Corporation (Python)	Software developer
Holsman, Ian	CNET Networks Apache Software Foundation	Director, Performance Measurement & Analysis Member
Hunter, Jason	Apache Software Foundation	Member
Kipping, David	Trolltech	COO, Executive Vice-President
Kuhn, Bradley M.	Free Software Foundation	Vice-President
Lakhani, Karim	MIT Apache	PhD student Researcher

Name	Company / Community	Profession / role in the community
Lane, Tom	Red Hat PostgreSQL	Employed fulltime to work on PostgreSQL Member of the steering committee
Lessig, Lawrence	Stanford Law School Other laws of Cyberspace	Professor of law Author
Marti, Don	Linux Journal	Editor in chief
Momjian, Bruce	PostgreSQL	Member of the steering committee
Olander, Matt	IXsystems	Mgr. of Technical Services
Parikh, Samir	IXsystems	Sales Engineer
Perens, Bruce	Debian Open Source Definition Hewlett-Packard	Former project leader Primary author Former Senior Strategist, Linux and Open Source
Rigo, Armin	Psycho (spin-off from Python)	Author
Rossum, Guido van	Python Zope Corporation	Author and 'benevolent dictator' Director of PythonLabs
Schoolcraft, Bill	Linuxcare	Technical Support Engineer
Smith, Michael	FreeBSD	Former member of the Core Team
Stallman, Richard M.	Free Software Foundation GNU project GPL license	President Creator Author
Thangavelu, Kapil	Zope (Python)	Developer
Turner, David	Free Software Foundation	Free Software Licensing Guru
Zoest, Sander van	Yahoo Apache	Software developer Contributor

Respondents in the Netherlands/Germany/Belgium

Name	**Company / Community**	**Profession / role in the community**
Akkerman, Wichert	Debian	Former project leader
Brouwer, Andries	University of Technology, Eindhoven Linux	Professor of discrete mathematics Maintainer
Baal, Joost van	Debian Logreport	Package maintainer 'Boss'
Bekker, Stijn	Debian	Package maintainer
Bezold, Wolfgang	IBM	Manager
Dassen, Ray	Debian	Package maintainer
Germaschewski, Kai	Linux	Maintainer
Heisterberg, Wolfgang	IBM	Manager
Hemel, Armijn	Linux	Contributor
Immens, Arnold	Frog	Employee
Joseph, Brian	Connectux	Marketing manager
Kooij, Joost	Debian	Package maintainer
Lankamp, Edwin	Linux	Active user
Leeuwen, David A.	Debian Linux	Package maintainer Contributor
Lehmann, Holger	IBM	Manager
McDonnell, Robert D	Sun Microsystems	Solaris trainer at Sun educational services
Molenaar, Bram	Vim	Author and maintainer
Mos, Seth	Linux Coltex BV	Contributor System maintainer
Nieuwenhuizen, Jan	LilyPond	Project leader
Offerman, Aad	Linux News	Former editor in chief
Rasters, Gaby	University of Nijmegen Debian	PhD student Researcher
Roest, Gerben	Linvision	Director
Schwidefsky, Martin	IBM Linux	Programmer Maintainer
Sessink, Olivier	BlueFish	Author and maintainer
Sliekers, Olav	BlueFish	Translator

Name	Company / Community	Profession / role in the community
Sliepen, Guus	Debian	Package maintainer
	Linux	Programmer
Vandenabeele, Peter	Mind.be	CEO
Vreeken, Jeroen	Linux	Maintainer
Vries, Rinse de	KDE	Head of Dutch translation
Winter, Brenno de	Philips	Freelancer
	Polder Linux User Group	Coordinator

Note on Appendix A

[1] The company/community names and the profession/role in the community are based on the profession and the interest of each respondent at *the time of the interview*. This table is in some ways outdated, since respondents will have changed jobs, participate in a different community or moved to a different company.

Appendix B: Primary data collection activities per community

	Interviews	Secondary sources of literature[1]	Direct observation	Archival records
Apache	7	(Fielding 1999, Franke & von Hippel 2003, Lakhani & Von Hippel 2003, Mockus et al 2002)	Presentations at the Holland Open Software Conference (Amsterdam).	www.apache.org
Debian	9	(Bezroukov 1999, Wayner 2000)		www.debian.org
Linux	9	(Benkler 2002a, Iannacci 2003, Kuwabara 2000, Lee & Cole 2003, Moon & Sproull 2002, Torvalds & Diamond 2001, Tuomi 2000, Tuomi 2002)	Presentations at FOSDEM, user meetings	Kernel traffic[2] and other Linux websites
PostgreSQL	3		PostgreSQL track OSCON (San Diego)	www.postgresql.org
Python	4		Python track OSCON (San Diego)	www.python.org

Note on appendix B

[1] The listed literature is an overview of some of the most important sources of literature that were used to analyze each of the communities. These sources included an extensive analysis of a particular community. More books and articles were used, but they provided a less extensive analysis of the community.
[2] On the Internet: http://www.kerneltraffic.org/ (March, 2004).

Appendix C: The collection of open source licenses

The table lists the licenses that were available on the website of Open Source Initiative (OSI) in September 2003.

Name of the license	Author of the License	Comments
Academic Free License v. 1.2	Lawrence E. Rosen	Rosen is a lawyer and wrote a number of licenses. This is probably one of the few that are not directly connected to a particular community.
Apache Software License v. 1.1	ASF	The license is adapted from the BSD license and includes a number of specific Apache clauses.
Apple Source Public Source License v. 1.2	Apple	The license is used for a number of open source projects initiated by Apple.
Artistic License	Larry Wall	Wall is the project leader of the Perl community. It was originally only intended for use in Perl.[1]
Attribution Assurance Licenses	Edwin A. Suominen	Adapted from the BSD license. It is written by an individual (not a lawyer)
BSD license	University of California, Berkely	The current version is a modified version. The original version contained an advertising clause.
Common Public License v. 1.0	IBM	This is the next version of the IBM Public License. At least one project that was initiated by IBM uses the CPL, namely Eclips.[2]
Eiffel Forum License v. 2.0	Non-profit International Consortium for Eiffel	Version 1 was not compatible with the GPL; this one is. Eiffel contains an object-oriented method, language, libraries and environment.
Entessa Public License v. 1.0	Entessa	The license is specifically written for an open source project called OpenSEAL.
GNU General Public License	Free Software Foundation	This is the most commonly used open source license. According to Freshmeat 65 percent of the projects on the Freshmeat index, are licensed under the GPL.
GNU Lesser General Public License	Free Software Foundation	This license is essentially the same as the Library GPL, which was first written in 1991. The name 'lesser' refers to the fact that this license, according to its authors, does less to protect the interests of the developers.

Name of the license	Author of the License	Comments
Lucent Public License	Lucent Technologies	The intent was to stay close to the BSD license, but allegedly the BSD license does not address some issues that should be addressed. Therefore, the author states that the license is based on the IBM Public License, but leaves out the viral aspect.[3]
IBM Public License	IBM	This license is not compatible with the GPL, because it includes the requirement that a royalty-free grant of patent license be included upon redistribution of the original work.[4]
Intel Open Source License	Intel	This license is based on the BSD license, but includes one additional condition that regulates export of the software. Its intended use is for the Open Source Computer Vision Library.
Historical Permission Notice and Disclaimer	Bruce Dodson	According to the author, much software is licensed under a permission notice and disclaimer. This license allows this software to become OSI approved. It is not intended for newly developed software.[5]
Jabber Open Source License	Jabber Open Source community	This license is primarily created for the Jabber server. The license is incompatible with the GPL because it provides a list of licenses that may be used to re-license the software. The GPL is not among them.
MIT License	MIT	The MIT license is also known as the X11 license and resembles the BSD license.
MITRE Collaborative Virtual Workspace License	MITRE	The MITRE license is actually nothing more than an agreement to hand over the copyright of improvements to MITRE. It allows developers to license the software under the GPL or MPL.
Motosoto License v0.9.1	Motosoto (Dutch company, went bankrupt)	The Motosoto license was specifically created to cover the 'community portal server' and related software.
Mozilla Public License 1.1	Netscape and Mozilla	The MPL was written for the Mozilla browser. The license has been adopted and refined by a great number of companies.
Naumen Public License	Naumen	The Naumen Public license was created by the Russian company Naumen, and resembles the Zope public license.

Name of the license	Author of the License	Comments
Nethack General Public License	M. Stephenson and Nethack	This license is based on the GPL and is used to cover NetHack, which is a computer game.
Nokia Open Source License	Nokia	The license is a variant of the MPL.
OCLC Research Public License 2.0	OCLC Research	Developed to cover software developed by OCLC.
Open Group Test Suite License	Open Group	This license specifically addresses the process of testing. The Open Group is a consortium of software developers that develops software for 'boundaryless information flow.'
Open Software License v. 2.0	Lawrence E. Rosen	According to the FSF website, this license is not compatible with the GPL. Rosen claims the license to be copyleft, but this is doubted by the FSF.
PHP license 3.0	PHP Group	The license is intended to cover PHP software (a scripting language) and is incompatible with the GPL.
Python license 1.6	CNRI	This license is incompatible with the GPL because it is governed by the laws of the state Virginia.
Python Software Foundation License	Python Software Foundation and CNRI	This license is based on the Python License and is also called Python License 2.1. It is compatible with the GPL.
Qt Public License	Trolltech	This license was intended for sharing the source code. The OSI judged the license to be compatible with the open source definition. Under pressure of a number of open source communities, Trolltech decided to no longer use the QPL.
RealNetworks Public Source License V1.0	RealNetworks	RealNetworks is a company that makes a number of applications. It claims that the license is compatible with the GPL, but the software may not include code covered by the GPL.
Reciprocal Public License v. 1.1	Technical Pursuit	According to the company, the license is similar to the GPL. However, the creators considered the GPL too broad. Furthermore, this license is supposed to include private use as well.

Name of the license	Author of the License	Comments
Ricoh Source Code Public License	Ricoh Company	According to the authors, the license is similar to the NPL, but without some controversial clauses. Therefore it might resemble the MPL more than the NPL.
Sleepycat License	Sleepycat	The license is supposedly similar to the GPL and compatible with the GPL. The only difference is that when the software is used in a non-free application, a license must be purchased.
Sun Industry Standards Source License	Sun Microsystems	The license is a derivative of the LGPL, but is not compatible with the GPL.
Sun Public License	Sun Microsystems	According to the FSF this license is basically the same as the MPL and is incompatible with the GPL.
Sybase Open Watcom Public License 1.0	Sybase	This license is used to make the previously proprietary Sybase Watcom compilers open source.
University of Illinois/NCSA Open Source License	University of Illinois	The license allows developers from the university to create open source software with the permission of the university.
Vovida Software License v. 1.0	Vovida Networks	Vovida is an initiative to collect open source software for datacom and telecom environments.
W3C License	World Wide Web Consortium	This license is GPL compatible and resembles the Simple Public License.
WxWindows Library License	Julian Smart, Robert Roebling et al	This license is GPL compatible, resembles the LGPL and has a notice that allows people to distribute modifications and keep the source code private.
X.Net License	X.Net, a California based company	This license is extremely unrestrictive.
Zope Public License 2.0	Zope Corporation	This license is GPL compatible. The first version was not, but the community changed the license to make it compatible.
zlib/libpng license	Zlib community	This license is compatible with the GPL and resembles the BSD license.

Notes on appendix C

[1] From http://www.catb.org/~esr/writings/taoup/html/ch19s05.html and http://www.perl.com/pub/a/language/misc/Artistic.html (both pages August 2003).
[2] From http://www-106.ibm.com/developerworks/library/os-cplfaq.html (August 2003).
[3] From the Internet: http://article.gmane.org/gmane.comp.licenses.open-source.general/1306 (September 2003).
[4] From the Internet: http://www.gnu.org/philosophy/license-list.html (September 2003).
[5] From the Internet: http://www.geocities.com/brucedodson.rm/hist_pnd.htm (September 2003).

Appendix D: The Developer's Certificate of Origin

The Developer's Certificate of Origin version 1.0 was posted on the Linux kernel mailing list by Linus Torvalds on Saturday, May 22nd. The entire e-mail with some comments can be found on the Internet at http://kerneltrap.org/node/view/3180 (last visited October 2004). This is what the Certificate looks like:

Developer's Certificate of Origin 1.0

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

(b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.

Appendix E: Bibliographical references

2002. *Free/Libre and Open Source Software: Final Report*, International Institute of Infonomics and Berlecon Research GmbH, downloaded from the Internet: http://www.infonomics.nl/FLOSS/report/ (December 2004), Maastricht and Berlin

Agrawal A. 2002. Commons Resources and Instituational Sustainability. In *The Drama of the Commons*, ed. E Ostrom, T Dietz, N Dolšak, PC Stern, S Stonich, EU Weber, pp. 41-85. Washington, D.C.: National Academy Press

Aldrich H. 1999. *Organizations Evolving*. Trowbridge, Wiltsgire: Cromwell Press Ltd

Aldrich HE, Herker D. 1977. Boundary Spanning Roles and Organizational Structure. *Academy of Management Review* 2(2): 217-30

Amason AC. 1996. Distinguishing the effects of functional and dysfunctional conflict on strategic decision making: resolving a paradox for top management teams. *Academy of Management Journal* 39(1): 123-48

Anderies JM, Janssen MA, Ostrom E. 2003. *Design Principles for Robustness of Institutions in Social-Ecological Systems*. Presented at Joining the Northern Commons: Lessons for the World, Lessons from the World, Anchorage

Andrew AM. 1989. *Self-Organizing Systems*: Gordon and Breach Science

Arrow KJ. 1951. *Social Choice and individual values*. New York and London: John Wiley & Sons and Chapman & Hill

Arrow KJ. 1962. Economic welfare and the allocation of resources from invention. In *The Rate and Direction of Inventive Activity: Economic and Social Factors*, ed. RR Nelson. Princeton: Princeton University Press

Arrow KJ. 1974. *The limits of organization*. New York: WW Norton & Company

Arthur WB. 1990. Positive Feedbacks in the Economy. *Scientific American* 262 : 92-9

Axelrod R, Cohen MD. 1999. *Harnessing Complexity: Organizational Implications of a Scientific Frontier*. New York: Free Press

Bauer A, Pizka M. 2003. *The Contribution of Free Software to Software Evolution*. Presented at Sixth International Workshop on Principles of Software Evolution (IWPSE'03), Helsinki, Finland

Becking J, Course S, van Enk G, Hangyi HT, Lahaye JJM, et al. 2005. MMBase: An open source content management system. *IBM Systems Journal* 44(2)

Bekkers VJJM. 2000. *Voorbij de virtuele organisatie? Over de bestuurskundige betekenis van virtuele variëteit, contingentie en parallel organiseren (Past the Virtual Organization. The meaning for public administration of virtual variety, contingency and parallel organizing)*. 's Gravenhage: Elsevier bedrijfsinformatie

Bendor JB. 1985. *Parallel Systems: Redundancy in government*. Berkeley: University of California Press

Benkler Y. 2002a. Coase's Penguin, or, Linux and the Nature of the Firm. *Yale Law Journal* 112(3): 369-446

Benkler Y. 2002b. Intellectual Property and the Organization of Information Production. *International Review of Law and Economics* 22(1): 81-107

Bennett E. 2000. *Institutions, economics and conflicts: fisheries management under pressure*. Presented at "Constituting the Commons: Crafting Sustainable Commons in the New Millennium," the Eighth Biennial Conference of the International Association for the Study of Common Property, Bloomington, Indiana, USA

Berge E. 2003. *Joining the Northern Commons: Lessons for the world, Lessons from the world*. Presented at IASCP Northern Polar Conference, Anchorage

Bergmann Lichtenstein BM. 2000. Emergence as a process of self-organizing; New assumptions and insights from the study of non-linear dynamic systems. *Journal of Organizational Change Management* 13(6): 526-44

Bergquist M, Ljungberg J. 2001. The power of gifts: organizing social relationships in open source communities. *Information systems Journal* 11: 305-20

Berkes F. 2000. *Cross-Scale Institutional Linkages: Perspectives from the Bottom Up*. Presented at "Constituting the Commons: Crafting Sustainable Commons in the New Millennium," the Eighth Biennial Conference of the International Association for the Study of Common Property, Bloomington, Indiana, USA

Berkes F. 2003. *Can Cross-Scale Linkages Increase the Resilience of Social-Ecological Systems?* Presented at RSCD International Conference, Politics of the Commons, Chiang Mai, Thailand

Bernbom G. 2000. *Analyzing the Internet as a Common Pool Resource: The problem of Network Congestion*. Presented at "Constituting the Commons: Crafting Sustainable Commons in the New Millennium," the Eighth Biennial Conference of the International Association for the Study of Common Property, Bloomington, Indiana, USA

Bezroukov N. 1999. Open Source Software Development as a Special Type of Academic Research (Critique of Vulgar Raymondism). *First Monday. Peer reviewed journal on the Internet* 4: downloaded from the Internet: http://www.firstmonday.dk/issues/issue4_10/bezroukov/index.html (December 2004)

Blumer H. 1954. What is wrong with social theory. *American Sociological Review* 19: 3-10

Bollier D. 2001a. The cornucopia of the commons. In *YES! Magazine*

Bollier D. 2001b. *Public Assets, Private Profits; Reclaiming the American Commons in an age of Market Enclosure*. Washington: New America Foundation

Bollier D. 2002. *Silent Theft, the private plunder of our common wealth*. New York: Routledge

Bonabeau E, Dorigo M, Theraulaz G. 1999. *Swarm Intelligence: from natural to artificial systems*. New York: Oxford University Press

Bonaccorsi A, Rossi C. 2003a. *Comparing motivations of individual programmers and firms to take part in the Open Source movement. From community to business*. Unpublished paper. Laboratory of Economics and Management, Sant'Anna School of Advanced Studies, Pisa, Italy. Downloaded from the Internet: http://opensource.mit.edu/papers/bnaccorsirossimotivationlong.pdf (December 2004)

Bonaccorsi A, Rossi C. 2003b. *Licensing schemes in the production and distribution of Open Source software. An empirical investigation*. Unpublished paper, Institute for Informatics and Telematics (IIT), Sant'Anna School of Advanced Studies, Pisa, Italy. downloaded from the Internet: http://opensource.mit.edu/papers/bnaccorsirossilicense.pdf (December 2004). 32 pp.

Bonaccorsi A, Rossi C. 2003c. Why Open Source Software can succeed. *Research Policy* 32(7): 1243-58

Boyle J. 2003. The second enclosure movement and the construction of the public domain. *Law and contemporary problems* 66: 33-74

Bromley DW. 1992. Commons,Property, and Common-Property Regimes. In *Making the Commons Work: Theory, Practice, and Policy*, ed. DW Bromley, pp. 3-15. San Francisco: ICS Press

Brown LD. 1984. Managing conflict among groups. In *Organizational Psychology, a book of readings*, ed. DA Kolb, IM Rubin, JM McIntyre. Englewood Cliffs: Prentice-Hall

De Bruijn H, Ten Heuvelhof E. 2000. *Networks and Decision Making*. Utrecht: LEMMA Publishers

De Bruijn JA. 2002. *Managing Performance in the Public Sector*. London and New York: Routledge

Bruns B. 2000. *Nanotechnology and the Commons: Implications of Open Source Abundance in Millennial Quasi-Commons*. Presented at "Constituting the Commons: Crafting Sustainable Commons in the New Millennium," the Eighth Biennial Conference of the International Association for the Study of Common Property, Bloomington, Indiana, USA

Buck SJ. 1998. *The Global Commons, An Introduction*. London: Earthscan Publications

Butler B, Sproull L, Kiesler S, Kraut R. forthcoming. Community effort in online groups: Who does the work and why? In *Leadership at a distance*, ed. S Weisband, L Atwater: Erlbaum

Carlsson L. 2003. *Managing Commons across Levels of Organization*. Presented at Workshop on Commons: Old and New, Centre for Advanced Study, Oslo

Chisholm D. 1989. *Coordination without Hierarchy: Informal structures in multiorganizational systems*: University of California Press

Claybrook B. 2004. Linux Standard Base: Enough Support? Why LSB compliance is important. In *LinuxWorld*, pp. 26-8

Coase RH. 1937. The nature of the firm. *Economica* 4: 386-405

Costanza R, Low B, Ostrom E, Wilson J. 2001. Ecosystems and human systems: a framework for exploring the linkages. In *Institutions, Ecosystems, and Sustainability*, ed. R Costanza, B Low, E Ostrom, J Wilson, pp. 3-20. Boca Raton: CRC Press LLC

Costigan JT. 1999. Introduction: Forests, Trees, and Internet Research. In *Doing Internet Research; Critical Issues and Methods for Examining the Net*, ed. S Jones, pp. xvii-xxiv: SAGE Publications, Inc.

Cowan R, Harison E. 2001. Protecting the digital endeavor: prospects for intellectual property rights in the information society. *AWT-achtergrondstudie nr. 22*

Crawford SES, Ostrom E. 1995. A grammar of institutions. *American Political Science Review* 89(3): 582-600

Crowston K, Annabi H, Howison J. 2003a. *Defining open source software project success*. Presented at Twenty-Fourth International Conference on Information Systems, Seattle, Washington, USA

Crowston K, Scozzi B, Buonocore S. 2003b. *An explorative study of open source software development structure*. Presented at Eleventh European Conference on Information Systems, Naples, Italy

Cunningham BM, Alexander PJ, Adilov N. 2004. Peer-to-peer file sharing communities. *Information Economics and Policy* 16(2): 197-213

Dafermos GN. 2001. Management and virtual decentralised Networks: The Linux Project. *First Monday. Peer reviewed journal on the Internet* 6: downloaded from the Internet: http://www.firstmonday.dk/issues/issue6_11/dafermos/ (December 2004)

Dalcher D. 2003. Beyond Normal Failures: Dynamic management of software projects. *Technology Analysis & Strategic Management* 15(4): 421-39

Dalle J-M, David PA. 2004. *SimCode: Agent-based Simulation Modelling of Open-Source Software Development*, Unpublished research paper, downloaded from the Internet: http://opensource.mit.edu/papers/dalledavid2.pdf (November 2004)

Dalle J-M, Jullien N. 2003. 'Libre' software: turning fads into institutions? *Research policy* 32(7): 1-11

Davies J. 2001. *Traditional CPRs, New Institutions: Native Title Management Committees and the State-Wide Native Title Congress in South Australia*. Presented at "Tradition and Globalisation: Critical Issues for the Accommodation of CPRs in the Pacific Region," the Inaugural

Pacific Regional Meeting of the International Association for the Study of Common Property, Brisbane, Australia

Demil B, Lecocq X. 2003. Neither market nor hierarchy or network: the emerging bazaar governance. Unpublished research paper, downloaded from the Internet: http://opensource.mit.edu/papers/demillecocq.pdf (December 2004)

Denzin NK, Lincoln YS. 1994. Introduction: Entering the Field of Qualitative Research. In *Handbook of Qualitative Research*, ed. NK Denzin, YS Lincoln, pp. 1-17. Thousands Oaks: Sage Publications

Deutsch M. 1973. *The Resolution of Conflict: constructive and destructive processes*. New Haven and London: Yale University Press

Dietz T, Dolšak N, Ostrom E, Stern PC. 2002. The Drama of the Commons. In *The Drama of the Commons*, ed. E Ostrom, T Dietz, N Dolšak, PC Stern, S Stonich, EU Weber, pp. 3-35. Washington, DC: National Academy Press

Dipboye RL, Smith CS, Howel WC. 1994. *Understanding Industrial and Organizational Psychology*. Fort Worth: Hartcourt Brace College Publishers

Edmonds B. 2002. Three challenges for the survival of memetics. *Journal of Memetics - Evolutionary Models of Information Transmission* 6: downloaded from the Internet: http://jom-emit.cfpm.org/2002/vol6/ (November 4)

Edwards K. 2001. *Epistemic communities, situated learning and open source software development*. Presented at 'Epistemic Cultures and the Practice of Interdisciplinarity' Workshop at NTNU, Trondheim

Van Eeten MJG. 1999. *Dialogues of the Deaf: Defining New Agendas for Environmental Deadlocks*. Delft: Eburon

Van Eeten MJG, Dicke WM. 2004. Het Groene Hart: Afbeeldingen van een maatschappelijk vraagstuk. *Stedebouw en Ruimtelijke Ordening* 85(4): 40-2

Egyedi TM, Van Wendel de Joode R. 2003. *Standards and coordination in open source software*. Presented at 3rd IEEE Conference on Standardization and Innovation in Information Technology, Delft, The Netherlands

Egyedi TM, Van Wendel de Joode R. 2004. Standardization and other coordination mechanisms in open source software. *Journal of IT Standards & Standardization Research* 2(2): 1-17

Elliot MS, Scacchi W. 2002. *Communicating and Mitigating Conflict in Open Source Software Development Projects*, Unpublished research paper, Institute for Software Research, UC Irvine, downloaded from the Internet: www.ics.uci.edu/~melliott/commossd.pdf (December 2004), Irvine

Epstein JM, Axtell R. 1996. *Growing Artificial Societies: social science from the bottom up*. Washingtom: The Brookings Institution

Falk A, Fehr E, Fischbacher U. 2002. Appropriating the Commons: A Theoretical Explanation. In *The Drama of the Commons*, ed. E Ostrom, T Dietz, N Dolšak, PC Stern, S Stonich, EU Weber, pp. 157-91. Washington, D.C.: National Academy Press

Farrell A, Morgan MG. 2000. *Multi-Lateral Emission Trading: Lessons From Inter-State NOx Pollution*. Presented at "Constituting the Commons: Crafting Sustainable Commons in the New Millennium," the Eighth Biennial Conference of the International Association for the Study of Common Property, Bloomington, Indiana, USA

Farrell J, Saloner G. 1988. Coordination through committees and markets. *Rand Journal of Economics* 29(2): 235-52

Ferscha A, Scheiner C. 1999. *Collective Choice in Virtual Teams*. Presented at IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Palo Alto, California

Fielding RT. 1999. Shared Leadership in the apache Project. *Communications of the Association for Computing* 42(4): 42/3

Fitzgerald B, Kenny T. 2004. Developing an information systems infrastructure with open source software. *IEEE Software* January/February 2004: 50-5

Fontana A, Frey JH. 1994. Interviewing: the art of science. In *Handbook of Qualitative Research*, ed. NK Denzin, YS Lincoln. Thousand Oaks: Sage Publications

Franck E, Jungwirth C. 2003. Reconciling Rent-Seekers and Donators - The Governance Structure of Open Source. *Journal of Management and Governance* 7(4): 401-21

Franke N, von Hippel E. 2003. Satisfying heterogeneous user needs via innovation toolkits: the case of apache security software. *Research Policy* 3(7)2: 1199-215

Frey BS. 1997. On the relationship between intrinsic and extrinsic work motivation. *International journal of industrial organization* 15(4): 427-39

Garzarelli G. 2003. Open Source Software and the Economics of Organization. In *Austrian Perspectives on the New Economy*, ed. J Birner, P Garrouste. London: Routledge

Gefu JO, Kolawole A. 2002. *Conflict in Common Property Resource Use: Experiences from an Irrigation Project*. Presented at "The Commons in an Age of Globalisation", the Ninth Biennial Conference of the International Association for the study of Common Property, Victoria Falls, Zimbabwe

German DM. 2002. *The evolution of the GNOME Project*. Presented at Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering, 24th International Conference on Software Engineering, Orlando, USA

Ghate R. 2000. *Joint forest management: constituting new commons*. Presented at "Constituting the Commons: Crafting Sustainable Commons in the New Millennium," the Eighth Biennial Conference of the International Association for the Study of Common Property, Indiana, Bloomington, USA

Ghate R. 2002. *Global Gains at Local Costs: Imposing Protected Areas: A Case Study From India*. Presented at "The Commons in an Age of Globalisation," the Ninth Conference of the International Association for the Study of Common Property, Victoria Falls, Zimbabwe

González-Barahona JM, Robles G. 2003. Free Software Engineering: A Field to Explore. *UPGRADE: the European Journal for the Informatics Professional* IV(4): 2-7

Goodin RE. 1996. Institutions and their Design. In *The theory of institutional Design*, ed. RE Goodin, pp. 1-53. Cambridge: Cambridge University Press

Hann I-H, Roberts J, Slaughter S, Fielding RT. 2002. *Why Do Developers Contribute to Open Source Projects? First Evidence of Ecomic Incentives*. Presented at Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering, 24th International Conference on Software Engineering, Orlando, USA

Hardin G. 1968. The Tragedy of the Commons. *Science* 162: 1243-8

Harison E. 2002. *Intellectual Property Rights in Software and Information Technologies, A Preliminary Study for ITER Project 014-38-413*, Merit University of Maastricht, Maastricht

Hars A, Ou S. 2002. Working for Free? Motivations for Participating in Open-Source Projects. *International Journal of Electronic Commerce* 6(3): 25-39

Heller MA. 1998. The Tragedy of the Anticommons: Property in the Transition from Marx to Markets. *Harvard Law Review* 111: 621-88

Hemetsberger A, Reinhardt C. 2004. *Sharing and Creating Knowledge in Open-Source Communities: The case of KDE*. Presented at Fifth European Conference on Organizational Knowledge, Learning, and Capabilities, Innsbruck, Austria

Hendriks F. 1999. *Public Policy and Political Institutions; The Role of Culture in Traffic Policy*. Cheltenham, UK: Edward Elgar Publishing

Hertel G, Niedner S, Herrmann S. 2003. Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel. *Research Policy* 32(7): 1159-77

Hess C, Ostrom E. 2001. *Artifacts, Facilities, and Content: Information as a Common Pool Resource.* Presented at Conference on the Public Domain, Duke Law School, Durham, North-Carolina, USA

Himanen P. 2001. *The Hacker Ethic and the spirit of the Information Age.* New York: Random House

Von Hippel E. 2001. Innovation by User Communities: Learning from Open-Source Software. *Sloan Management Review* 42(4): 82-6

Von Hippel E, Von Krogh G. 2003. Open Source Software and "Private-Collective" Innovation Model: Issues for Organization Science. *Organization Science* 14(2): 209-23

Hirschman AO. 1970. *Exit, Voice, and Loyalty: responses to decline in firms, organizations, and states.* Cambridge, Massachusetts and London: Harvard University Press

Hodgson GM. 2003. John R. Commons and the Foundations of Institutional Economics. *Journal of Economic Issues* 37(3): 547-76

Holland JH. 1995. *Hidden Order: How Adaptation Builds Complexity.* Cambridge: Perseus Books

Holman J. 2000. *Electronic Information as a Commons: The Issue of Access.* Presented at "Constituting the Commons: Crafting Sustainable Commons in the New Millennium," the Eighth Biennial Conference of the International Association for the Study of Common Property, Bloomington, Indiana, USA

Hunter D. 2003. Cyberspace as Place and the Tragedy of the Digital Anticommons. *California Law Review* 91: 439-519

Iannacci F. 2002. The Economics of Open-Source Networks. *Communications & Strategies* 48(4)

Iannacci F. 2003. The Linux Managing Model. *First Monday. Peer reviewed journal on the Internet* 8, downloaded from the Internet: http://www.firstmonday.org/issues/issue8_12/iannacci/ (December 2004)

Janesick VJ. 1994. The Dance of Qualitative Research Design: Metaphor, Methodolatry, and Meaning. In *Handbook of Qualitative Research*, ed. NK Denzin, YS Lincoln. Thousands Oaks: Sage Publications

Jehn KA. 1995. A miltimethod examination of the benefits and detriments of intragroup conflict. *Administrative Science Quarterly* 40(June): 256-82

Jehn KA, Mannix EA. 2001. The dynamic nature of conflict: a longitudinal study of intragroup conflict and group performance. *Academy of Management Journal* 44(2): 238-51

De Jong M. 1999. *Institutional Transplantation: How to adopt good transport infrastructure decision-making ideas from other countries?* Delft: Eburon

De Jong M, Van der Voort H. 2004. Evolutionary Theory in the Administrative Sciences: Introduction. *Knowledge, Technology and Policy* 16(4): 19-29

Kahin B. 2002. *An Agenda for Policy Research on Open Source.* Presented at NSF Workshop on Open-Source Software, Arlington, Virginia

Karlsson S. 2000. *Heterogeneity and Harmonization in Layered Institutional Approaches to Global Commons Issues: The Case of Pesticide Use in the South.* Presented at "Constituting the Commons: Crafting Sustainable Commons in the New Millennium," the Eighth Biennial Conference of the International Association for the Study of Common Property, Bloomington, Indiana, USA

Kaufman H. 1981. *The administrative behavior of federal bureau chiefs.* Washington: The Brookings Institution

Kelly K. 1994. *Out of Control; The new biology of machines, social systems and the economic world.* Cambridge: Perseus Books

Klijn E-H. 1996. *Regels en sturing in netwerken: de invloed van netwerkregels op de herstructurering van naoorlogse wijken*. Delft: Eburon

Koch S, Schneider G. 2002. Effort, co-operation and co-ordination in an open source software project: GNOME. *Information systems Journal* 12(1): 27-42

Kogut B, Metiu A. 2001. Open-Source software development and distributed innovation. *Oxford Review of Economic Policy* 17(2): 248-64

Kollock P. 1996. *Design Principles for Online Communities*. Presented at Harvard Conference on the Internet and Society, Harvard, USA

Kollock P. 1999. The economies of online cooperation: Gifts and public goods in cyberspace. In *Communities in Cyberspace*, ed. MA Smith, P Kollock, pp. 220-39. London: Routledge

Kollock P, Smith MA. 1996. Managing the Virtual Commons: Cooperation and Conflict in Computer Communities. In *Computer-Mediated Communications: Linguistic, Social, and Cross-Cultural Perspectives*, ed. S Herring, pp. 109-28. Amsterdam: John Benjamins

Kollock P, Smith MA. 1999. Communities in Cyberspace. In *Communities in Cyberspace*, ed. MA Smith, P Kollock, pp. 3-25. London: Routledge

Von Krogh G, Haelfliger S, Spaeth S. 2003a. Collective Action and Communal Resources in Open Source Software Development: The Case of Freenet. Research paper, University of St. Gallen, Switzerland, downloaded from the Internet: opensource.mit.edu/papers/vonkroghhaefligerspaeth.pdf (November 2004)

Von Krogh G, Von Hippel E. 2003. Editorial: Special issue on open source software development. *Research Policy* 32(7): 1149-57

Von Krogh G, Spaeth S, Lakhani KR. 2003b. Community, joining, and specialization in open source software innovation: a case study. *Research Policy* 32(7): 1217-41

Kuit M. 2002. *Strategic behavior and regulatory styles in the Netherlands energy industry*. Delft: Eburon

Kuwabara K. 2000. Linux: A Bazaar at the Edge of Chaos. *First Monday. Peer reviewed journal on the Internet* 5: downloaded from the Internet: www.firstmonday.dk/issues/issue5_3/kuwabara/ (December 2004)

Laan M. 2003. Amsterdam is duur Microsoft zat (Amsterdam is fed up with expensive Microsoft). In *Het Parool*, pp. 9. Amsterdam

Lakhani K, Von Hippel E. 2003. How Open Source Software Works: "Free" User-to-User Assistance. *Research Policy* 32(7): 922-43

Lakhani K, Wolf RG. 2003. Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects. Boston: MIT Sloan Working Paper no.4425-03, downloaded from the Internet: freesoftware.mit.edu/papers/lakhaniwolf.pdf (November 2004)

Laland KN, Brown GR. 2002. *Sense and Nonsense in Evolutionary Theory: evolutionary perspectives on human behaviour*. Oxford: Oxford University Press

Landau M. 1969. Redundancy, Rationality, and the Problem of Duplication and Overlap. *Public Administration Review* 29(4): 346-58

Langlois RN. 2002. Modularity in technology and organization. *Journal of Economic Behavior and Organization* 49(2): 19-37

Lanzara GF, Morner M. 2003. *The Knowledge Ecology of Open-Source Software Projects*. Presented at 19th EGOS Colloquium, Copenhagen

Lazega E. 2000. Rule enforcement among peers: a lateral control regime. *Organization Studies* 21(1): 193-214

Lee GK, Cole RE. 2003. The Linux kernel development as a model of knowledge creation. *Organization Science* 14(6): 633-49

Lerner J, Tirole J. 2002a. *The Scope of Open Source Licensing*, NBER Working Papers 9363, downloaded from the Internet: http://ideas.repec.org/p/nbr/nberwo/9363.html (December 2004)

Lerner J, Tirole J. 2002b. Some simple economics of open source. *Journal of Industrial Economics* 50(2): 197-234

Levacic R. 1991. Markets and government: an overview. In *Markets, hierarchies & networks; the coordination of social life*, ed. G Thompson, J Frances, R Levacic, J Mitchell, pp. 35 - 65. London: The Open University

Likert R, Likert JG. 1976. *New ways of managing conflict*: McGraw-Hill

Lin Y. 2004. *Epistemologically Multiple Actor-Centred System: or, EMACS at work!* Presented at 3rd Oekonux Conference, Vienna, Austria

Ljungberg J. 2000. Open source movements as a model for organising. *European Journal of Information Systems* 9(4): 208-16

Lovelace K, Shapiro DL, Weingart LR. 2001. Maximizing cross-functional new product teams' innovativeness and constant adherence: a conflict communications perspective. *Academy of Management Journal* 44(4): 479-93

Madey G, Freeh V, Tynan R. 2002. *Understanding OSS as a Self-Organizing Process.* Presented at Meeting Challenges and Surving Success: 2nd Workshop on Open Source Software Engineering, Orlando

Maggioni MA. 2002. Open source software communities and industrial districts: a useful comparison? Unpublished research paper, downloaded from the Internet: opensource.mit.edu/papers/maggioni.pdf (November 2004)

Mannix EA, Griffith T, Neale MA. 2002. The Phenomenology of Conflict in Distributed Work Teams. In *Distributed Work*, ed. PJ Hinds, S Kiesler, pp. 213-34. Cambridge and London: The MIT Press

March JG, Olson JP. 1989. *Rediscovering Institutions*. New York: The Free Press

Markus ML, Manville B, Agres CE. 2000. What Makes a Virtual Organization Work? *Sloan Management Review* 42(1): 13-26

Mauss M. 1990. *The gift: the form and reason for exchange in archaic societies*. London: W. W. Norton

McCormick C. 2003. *"The Big Project That Never Ends": Role and Task Negotiation Within an Emerging Occupational Community*. recent summary of findings, as part of PhD research thesis. University of Albany, downloaded from the Internet: opensource.mit.edu (December 2002).

McGowan D. 2001. Legal Implications of Open-Source Software. *University of Illinois Review* 241(1): 241-304

McKelvey M. 2001a. The economic dynamics of software: three competing business models exemplified through Microsoft, Netscape and Linux. *Economics of Innovation and New Technology* 10: 199-236

McKelvey M. 2001b. *Internet Entrepreneurship: Linux and the dynamics of open source software*. working paper thesis. Centre for Research on Innovation and Competition, The University of Manchester, Manchester, downloaded from the Internet: http://les1.man.ac.uk/cric/Pdfs/DP44.pdf (December 2004)

Mockus A, Fielding RT, Herbsleb JD. 2002. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology* 11(3): 309-46

Montoya-Weiss MM, Massey AP, Song M. 2001. Getting IT together: temporal coordination and conflict managements in global virtual teams. *Academy of Management Journal* 44(6): 1251-62

Moon JY, Sproull L. 2000. Essence of Distributed Work: The Case of the Linux Kernel. *First Monday. Peer reviewed journal on the Internet*, downloaded from the Internet: http://www.firstmonday.org/issues/issue5_11/moon/ (December 2004)

Moon JY, Sproull L. 2002. Essence of distributed work: the case of the Linux kernel. In *Distributed Work*, ed. PJ Hinds, S Kiesler, pp. 381 - 404. Cambridge and London: The MIT Press

Morgan G. 1986. *Images of Organization*. Thousand Oaks, California: Sage Publications

Muchapondwa E. 2002. *Sustainable Commercialised Use of Wildlife as a Strategy for Rural Poverty Reduction: The Case of 'Campfire' in Zimbabwe*. Presented at "The Commons in an Age of Globalisation," the Ninth Conference of the International Association for the Study of Common Property, Victoria Falls, Zimbabwe

Mudiwa M. 2002. *Global Commons: The Case of Indigenous Knowledge, Intellectual Property Rights and Biodiversity*. Presented at "The Commons in an Age of Globalisation," the Ninth Conference of the International Association for the Study of Common Property, Victoria Falls, Zimbabwe

Nakakoji K, Yamamoto Y, Nishinaka Y, Kishida K, Ye Y. 2002. *Evolution Patterns of Open-Source Software Systems and Communities*. Presented at International Workshop on Principles of Software Evolution (IWPSE 2002), Orlando, Florida

Narduzzo A, Rossi A. 2003. *Modularity in Action: GNU/Linux and Free/Open Source Software Development Model Unleashed*. working paper, Quaderno DISA n. 78 thesis. downloaded from the Internet: opensource.mit.edu/papers/narduzzorossi.pdf (December 2004)

Naughton J. 1999. *A brief history of the Future; The origins of the internet*. London: Weidenfeld & Nicolson

Nelson RR, Sampat BN. 2001. Making sense of institutions as a factor shaping economic performance. *Journal of Economic Behavior and Organization* 44(1): 31-54

Nonaka I, Takeuchi H. 1995. *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*. Oxforf: Oxford University Press

Noonan DS. 1998. Internet Decentralization, Feedback, and Self-Organization. In *Managing the Commons*, ed. JA Baden, DS Noonan, pp. 188-94. Bloomington and Indianapolis: Indiana University Press

Nooteboom B. 2002. *Trust: forms, foundation, failures and figures*. Cheltenham: Edward Elgar Publishing Limited

Nordhaus WD. 1969. *Invention, Growth and Welfare: A Theoretical Treatment of Technological Change*. Cambridge: MIT Press

North DC. 1991. *Institutions, institutional change and economic performance*. Cambridge: Cambridge University Press

Oakerson RJ. 1992. Analyzing the Commons: A framework. In *Making the Commons Work: Theory, Practice, and Policy*, ed. DW Bromley, pp. 41-59. San Francisco: ICS Press

Oksanen M. 1998. *Authorship, Communities and Intellectual Property Rights*. Presented at "Crossing Boundaries", the seventh biannual conference of the International Association for the Study of Common Property, Vancouver, British Columbia, Canada

Olson M. 1965. *The logic of collective action; public goods and the theory of groups*. Cambridge: Harvard University Press

O'Mahony SC. 2003. Guarding the Commons: How Community Managed Software Projects Protect Their Work. *Research Policy* 32(7): 1179-98

O'Mahony SC, Ferraro F. 2004. *Hacking Alone? The Effects of Online and Offline Participation on Open Source Community Leadership*. research paper, downloaded from: http://opensource.mit.edu/papers/omahonyferraro2.pdf (December 2004) thesis

Oommen TK. 1994. *Reconciling Equality and Pluralism: an agenda for the "developed" societies*. discussion Paper, No. 8 thesis. Collegium Budapest, Institute for Advanced Study, downloaded from: http://www.colbud.hu/main/PubArchive/DP/DP08-Oommen.pdf (December 2004), Budapest

O'Reilly T. 1999. Lessons from Open-Source software development. *Communications of the Association for Computing* 42(4): 33-7

Orr SW. 2001. The economics of shame in work groups: how mutual monitoring can decrease cooperation in teams. *Kyklos* 54(1): 49-66

Osterloh M. 2002. *Open Source Software Production The Magic Cauldron?* Presented at LINK Conference, Copenhagen

Osterloh M, Rota S, Kuster B. 2003a. *Open Source Software Production: Climbing on the Shoulders of Giants*. working paper thesis. University of Zurich, Zurich, downloaded from the Internet: http://opensource.mit.edu/papers/osterlohrotakuster.pdf (December 2004)

Osterloh M, Rota S, Von Wartburg M. 2003b. *Open Source - new rules in software development*. working paper thesis, downloaded from the Internet: www.iou.unizh.ch/orga/downloads/OpenSourceAoM.pdf (January 2004)

Ostmann A. 1998. External control may destroy the commons. *Rationality and Society* 10(1): 103-22

Ostrom E. 1986. An agenda for the study of institutions. *Public Choice* 48(1): 3-25

Ostrom E. 1990. *Governing the Commons; The Evolution of Institutions for Collective Action*. Cambridge: Cambridge University Press

Ostrom E. 1992. The Rudiments of a Theory of the Origins, Survival, and Performance of Common-Property Institutions. In *Making the Commons Work: Theory, Practice, and Policy*, ed. DW Bromley, pp. 293-318. San Francisco: ICS Press

Ostrom E. 1993. Design principles in long-enduring irrigation institutions. *Water Resources Research* 29(7): 1907-12

Ostrom E. 1998. A behavioral approach to the rational choice theory of collective action. *The American Political Science Review* 92(1): 1-22

Ostrom E. 1999. Coping With Tragedies of the Commons. *Annual Review Political Science* 2: 493-535

Ostrom E. 2000. Collective Action and the Evolution of Social Norms. *Journal of Economic Perspectives* 14(3): 137-58

Ostrom E. 2003. How types of goods and property rights jointly affect collective action. *Journal of Theoretical Politics* 15(3): 239-70

Ostrom E, Burger J, Field CB, Norgaard RB, Policansky D. 1999. Revisiting the Commons: Local Lessons, Global Challenges. *Science* 284: 278-82

Ostrom E, Dietz T, Dolšak N, Stern PC, Stonich S, Weber EU, eds. 2002. *The Drama of the Commons*. Washington: National Academy Press

Ostrom E, Gardner R, Walker J. 1994. *Rules, Games, and Common-Pool Resources*. Ann Arbor: The University of Michigan Press

Ostrom V, Ostrom E. 2004. The Quest for Meaning in Public Choice. *American Journal of Economics and Sociology* 63(1): 105-47

Perens B. 1999. The Open Source Definition. In *OpenSources: Voices from the Open Source Revolution*, ed. C DiBona, S Ockman, M Stone, pp. 171-89. Sebastopol: O'Reilly & Associates

Pondy LR. 1967. Organizational Conflict: Concepts and Models. *Administrative Science Quarterly* 12: 296-320

Poteete AR, Ostrom E. 2004. Heterogeneity, Group Size and Collective Action: The Role of Institutions in Forest Management. *Development and Change* 45(3): 435-61

Powell WW. 1990. Neither market nor hierarchy: network forms of organization. *Research in Organizational Behaviour* 12: 295-336

Pruitt DG. 1998. Social Conflict. In *The Handbook of Social Psychology*, ed. DT Gilbert, ST Fiske, G Lindzey, pp. 470-503. New York and Oxford: Oxford University Press

Raiffa H. 1992. *The art and science of negotiation*. Cambridge: The Belknap Press of Harvard University Press

Rasters G. 2004. *Communication and Collaboration in Virtual Teams: Did we get the message?*: Print Partners Ipskamp

Raymond ES. 1999a. A brief History of Hackerdom. In *OpenSources: Voices from the Open Source Revolution*, ed. C DiBona, S Ockman, M Stone, pp. 19-29. Sebastopol: O'Reilly & Associates

Raymond ES. 1999b. *The Cathedral and the Bazaar: Musings on Linux and Open Source from an Accidental Revolutionary*. Sebastapol: O'Reilly

Raymond ES. 2000. Homesteading the Noosphere. *First Monday. Peer reviewed journal on the Internet* 4: downloaded from the Internet: http://www.firstmonday.dk/issues/issue3_10/raymond/ (December 2004)

Resnick M. 1994. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. Cambridge: The MIT Press

Rose CM. 2002. Common Property, Regulatory Property, and Environmental Protection: Comparing Community-Based Management to Tradable Environmental Allowances. In *The Drama of the Commons*, ed. E Ostrom, T Dietz, N Dolšak, PC Stern, S Stonich, EU Weber, pp. 233-57. Washington, D.C.: National Academy Press

Rosenthal U. 1988. *Bureaupolitiek en Bureaupolitisme; om het behoud van een competitief overheidsbestel (inaugurele rede Leiden)*. Alphen aan de Rijn: Samsom H.D. Tjeenk Willink

Runge CF. 1992. Common Property and Collective Action in Economic Development. In *Making the Commons Work; Theory, Practice, and Policy*, ed. DW Bromley, pp. 17-41. San Francisco: ICS Press

Russ GS, Galang MC, Ferris GR. 1998. Power and influence of the human resources function through boundary spanning and information management. *Human Resource Management Review* 8(2): 125-48

Ruttan LM. 1998. Closing the Commons: Cooperation for Gain or Restraint. *Human Ecology* 26(1): 43-66

Sarker A, Itoh T. 2001. Design principles in long-enduring institutions of Japanese irrigation common-pool resources. *Agricultural Water Management* 48(2): 89-102

Scacchi W. 2004. Free/Open Source Software Development Practices in the Computer Game Community. *IEEE Software* 21(1): 56-66

Schön DA. 1983. *The Reflective Practitioner: How Professionals Think in Action*. New York: Basic Books

Schwartz T. 1986. *The Logic of collective choice*. New Yok: Columbia University Press

Scott JC. 1999. *Seeing like a state; How Certain Schemes to Improve the Human Condition Have Failed*. New Haven: Yale University Press

Sekher M. 2001. Organized participatory management: insights from community forestry practices in India. *Forest Policy and Economics* 3(3/4): 137-54

Selz D. 1999. *Value Webs, Emerging forms of fluid and flexible organizations*. dissertation thesis. University of St. Gallen, St. Gallen, downloaded from the Internet: http://www.businessmedia.org/modules/pub/download.php?id=businessmedia-11&user=&pass= (December 2004)

Sen AK. 1970. *Collective choice and social welfare*. San Fransisco: Holden-Day

Shah S. 2003. *Community-Based Innovation & Product Development: Findings From Open Source Software and Consumer Sporting Goods*. dissertation thesis. Massachusetts Institute of Technology, Cambridge, MA

Shaikh M, Cornford T. 2003. *Version control software for knowledge sharing, innovation and learning in OS*. Presented at Open Source Software Movements and Communities workshop, ICCT, Amsterdam

Shapiro C, Varian HR. 1998. *Information Rules: A Strategic Guide to the Network Economy*. Boston: Harvard Business School Press

Sharma S, Sugumaran V, Rajagopalan B. 2002. A framework for creating hybrid-open source software communities. *Information systems Journal* 12(1): 7-25

Simon HA. 1996. *The Sciences of the Artificial*. Cambridge: MIT Press

Sinha PC. 2000. *Towards Crafting Sustainable Commons, Third World, and the Antarctica Model*. Presented at "Constituting the Commons: Crafting Sustainable Commons in the New Millennium," the Eighth Biennial Conference of the International Association for the Study of Common Property, Bloomington, Indiana, USA

Smith AD. 1999. Problems of conflict management in virtual communities. In *Communities in Cyberspace*, ed. MA Smith, P Kollock, pp. 134-63. London: Routledge

Spencer L, Ritchie J, O'Connor W. 2003. Analysis: Practices, Principles and Processes. In *Qualitative research practice: A guide for social science students and researchers*, ed. J Ritchie, J Lewis, pp. 199-218. Londen: Sage Publications

Sproule-Jones M. 1998. *Restoring the Great Lakes: Institutional Analysis and Design*. Presented at "Crossing Boundaries", the seventh annual conference of the International Association for the Study of Common Property, Vancouver, British Columbia, Canada

Stake RE. 1994. Case studies. In *Handbook of Qualitative Research*, ed. NK Denzin, YS Lincoln, pp. 236-47. Thousand Oaks: Sage Publications

Stallman R. 2002. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. Boston: GNU Press

Steins NA, Röling NG, Edwards VM. 2000. *Re-'designing' the principles: An interactive perspective to CPR theory*. Presented at "Constituting the Commons: Crafting Sustainable Commons in the New Millennium," the Eighth Biennial Conference of the International Association for the Study of Common Property, Bloomington, Indiana, USA

Stern PC, Dietz T, Dolšak N, Ostrom E, Stonich S. 2002. Knowledge and questions after 15 years of research. In *The Drama of the Commons*, ed. E Ostrom, T Dietz, N Dolšak, PC Stern, S Stonich, EU Weber, pp. 445-89. Washington: National Academy Press

Tang SY. 1991. Institutional Arrangements and the Management of Common-Pool Resources. *Public Administration Review* 51(1): 42-51

Tang SY. 1992. *Institutions and Collective Action: Self-Governance in Irrigation*. San Francisco: Institute for Contemporary Studies

Ten Heuvelhof E, De Jong M, Kuit M, Stout H, eds. 2003. *Infrastratego: Strategisch gedrag in infrastructuurgebonden sectoren (Infrastratego: Strategic Behavior in infrastructures*. Utrecht: Lemma

Theraulaz G, Gautrais J, Camazine S, Deneubourg JL. 2003. The formation of spatial patterns in social insects: from simple behaviours to complex structures. *Philosophical Transactions of the Royal Society of London A* 361: 1263-82

Thomas K. 1976. Conflict and conflict management. In *Handbook of Industrial and Organizational Psychology*, ed. MD Dunnette, pp. 889-935. Chicago: Rand McNally College Publishing Company

Thomson JT, Schoonmaker Freudenberger K. 1997. *Crafting institutional arrangements for community forestry*. Rome: Food and agriculture organization of the United Nations (Community Forestry Field Manual, No. 7)

Torvalds L. 1999. The Linux Edge. *Communications of the Association for Computing* 42(4): 38, 9

Torvalds L, Diamond D. 2001. *Gewoon voor de fun (Just for Fun)*. Uithoorn: Karakter Uitgevers

Travers M. 2001. *Qualitative Research Trough Case Studies*. London: Thousands Oaks

Tucker C. 1998. *Evaluating a Common Property Institution: Design Principles and Forest Management in a Honduran Community*. Presented at "Crossing Boundaries", the seventh biannual conference of the International Association for the Study of Common Property, Vancouver, British Columbia, Canada

Tuomi I. 2000. *Learning from Linux: Internet, innovation and the new economy*. Berkely University, Berkeley, downloaded from the Internet: http://emlab.berkeley.edu/users/bhhall/tuomi00.pdf (February 2001)

Tuomi I. 2001. Internet, Innovation, and Open Source: Actors in the Network. *First Monday. Peer reviewed journal on the Internet* 6: downloaded from the Internet: http://www.firstmonday.org/issues/issue6_1/tuomi/ (December 2004)

Tuomi I. 2002. Evolution of the Linux credits file: methodological challenges and reference data for open source research. Joint Research Centre, Institute for Prospective Technological Studies, available on the Internet: http://www.jrc.es/~tuomiil/articles/EvolutionOfTheLinuxCreditsFile.pdf (January 2004)

Turkle S. 1995. *Life on the screen: Identity in the age of the Internet*. New York: Touchstone

Vemuri VK, Bertone V. 2004. Will the Open Source Movement Survive a Litigious Society? *Electronic Markets* 14(2): 114-23

Vendrik K, Van Tilburg R. 2002. *Software open u! Plan van aanpak ter stimulering van open source software (Software open yourself! Plan of approach to stimulate open source software)*, Nota van de Groenlinks Tweedekamerfractie

Waldrop MM. 1992. *Complexity: the emerging science at the edge of chaos and order*. London: Penguin Group

Walker JM, Garzarelli G, Herr A, Ostrom E. 2000. Collective choice in the commons: experimental results on proposed allocation rules and votes. *The Economic Journal* 110(January): 212-34

Wayner P. 2000. *FREE FOR ALL: How Linux and the Free Software Movement Undercut the High-Tech Titans*. New York: HarperBusiness

Weber S. 2004. *The Success of Open Source*. Cambridge: Harvard University Press

Van Wendel de Joode R. 2004a. *Analyzing software development and maintenance in open source communities with the CPR framework*. Presented at The Tenth Biennial Conference of the International Assiociation for the Study of Common Property (IASCP), Oaxaca, Mexico

Van Wendel de Joode R. 2004b. Conflicts in open source communities. *Electronic Markets* 14(2): 104-13

Van Wendel de Joode R. 2004c. *Continuity of the commons in open source communities*. Presented at SSCCII-2004 Conference, symposium of Santa Caterina on Challanges in the Internet and Interdisciplinary Research (IEEE cosponsored conference), Amalfi, Italy

Van Wendel de Joode R, De Bruijn JA, Van Eeten MJG. 2003. *Protecting the Virtual Commons; Self-organizing open source communities and innovative intellectual property regimes*. The Hague: T.M.C. Asser Press

Van Wendel de Joode R, Kemp J. 2002. The Strategy Finding Task within Collaborative Networks, based on an exemplary Case of the Linux Community. In *Collaborative*

Business Ecosystems and Virtual Enterprises, ed. LM Camarinho-Matos, pp. 517-27: Kluwer Academic Publishers

West J, O'Mahony SC. 2005. *Contrasting Community Building in Sponsored and Community Founded Open Source Projects*. Presented at 38th Annual Hawaii International Conference on System Sciences, Waikoloa, Hawaii

Wichmann T. 2002. *FLOSS Final Report Part 2. Firms' open source activities: motivations and policy implications*, Berlecon Research GmbH, Berlin

Witt U. 1997. Self-organization and economics - what is new? *Structural Change and Economic Dynamics* 8(4): 489-507

Ye Y, Kishida K, Nakakoji K, Yamamoto Y. 2002. *Creating and Maintaining Sustainable Open Source Software Communities*. Presented at International Symposium on Future Software Technology, Wuhan, China

Yin RK. 1989. *Case study research: design and methods*. Newbury Park: Sage Publications

Zeitlyn D. 2003. Gift economies in the development of open source software: Anthropological reflections. *Research Policy* 32(7): 1287-91

Curriculum Vitae

Ruben van Wendel de Joode was born in Winterswijk in the Netherlands, 24 June 1977. In 1995 he graduated from the "Rijkscholengemeenschap" in Epe. That same year he started his study of economics and business at the Erasmus University of Rotterdam, where he specialized in business policy and business management. In September 2000 he passed the preliminary exam of the study technology, policy and management (TPM) at Delft University of Technology.

He completed his master's thesis at the Erasmus University in October 1999. The title of the thesis was *The Contribution of Assessment Models in Product Development*. The thesis was part of an internship at Robert Bosch GmbH and the Fraunhofer Institute in Stuttgart, Germany. The goal was to analyze how assessment models can be adopted in product development and to judge whether they can contribute to improve the quality of these processes.

In September 2000 he became a member of the department of organization and management at the Faculty of Technology, Policy and Management of Delft University of Technology and he became a member of the research school the Netherlands Institute of Government. From September 2000 until February 2004, he was also a member of a Delft Interfaculty Research Program called Betade. Over time, the topic of his research settled on open source communities and, more specifically, their organizational structure. Fascination led him to choose this topic. How could volunteers who are geographically distributed and who have no contractual agreements with each other possibly create sophisticated software, which is of high quality and adopted by an increasing number of organizations?

In 2001 Van Wendel de Joode was part of a team that was awarded a grant from the research program Information Technology and Law (IT*e*R), which is part of the Netherlands Organisation for Scientific Research (NWO). The grant was to conduct research on open source communities, and allowed him to travel to the United States where he interviewed many individuals who are in some way related to open source communities.

Currently he is assistant professor in the Department of Organization and Management at the Delft University of Technology, where he continues research on open source communities. A large part of this research is funded through a new grant from NWO. The current focus is the interplay between businesses and open source communities, in particular, issues of continuity and reliability.