Free/Open Source Software Development: Results and Research Methods

Otso Kivekäs

Helsinki June 1, 2008

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

Tiedekunta/Osasto — Fakultet/Sektion — Faculty
Faculty of Science

Laitos — Institution — Department
Department of Computer Science

Tekijä — Författare — Author
Otso Kivekäs

Työn nimi — Arbetets titel — Title

Free/Open Source Software Development: Results and Research Methods

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's Thesis	June 1, 2008	78 pages

Tiivistelmä — Referat — Abstract

Free and open source software development is an alternative to traditional software engineering as an approach to the development of complex software systems. It is a way of developing software based on geographically distributed teams of volunteers without apparent central plan or traditional mechanisms of coordination.

The purpose of this thesis is to summarize the current knowledge about free and open source software development and explore the ways on which further understanding on it could be gained. The results of research on the field as well as the research methods are introduced and discussed.

Also adapting software process metrics to the context of free and open source software development is illustrated and the possibilities to utilize them as tools to validate other research are discussed.

ACM Computing Classification System (CCS):
D.2.1 [Software engineering] Requirements/Specifications
D.2.5 [Software engineering] Testing and Debugging
D.2.8 [Software engineering] Metrics
D.2.9 [Software engineering] Management
K.6.1 [Computing Milieux] Project and People Management
K.6.3 [Computing Milieux] Software Management

Avainsanat — Nyckelord — Keywords
Open source, Free software, Software engineering, Software process metrics

Säilytyspaikka — Förvaringsställe — Where deposited
Kumpula Science Library, serial number C-

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction **1**

2 Free and open source software **3**

2.1 Free software and open source software 3

2.2 The licenses 4

2.3 History of FOSS 5

2.4 FOSS as a social movement 6

2.5 The Cathedral and the Bazaar 7

3 Free/open source software development **8**

3.1 Characteristics of FOSSD 8

3.1.1 Team formation and reasons for contribution 10

3.1.2 Development tools as the process 12

3.1.3 Requirements analysis and management 13

3.1.4 Continuous design and defect management 15

3.1.5 Cooperation and coordination 17

3.1.6 Software evolution 21

3.1.7 Fork 22

3.1.8 FOSS ecosystem 25

3.2 Comparing FOSSD and traditional software engineering 27

3.2.1 Software engineering 28

3.2.2 Conceptual differences 29

3.2.3 Process differences 30

3.2.4 Statistical differences 32

3.2.5 A side note on agile methods 32

4 Research on free/open source development **34**

4.1 Descriptive research on software engineering 34

4.2 Research methods 35
4.2.1 Case studies and reflective analysis 36
4.2.2 Surveys 37
4.2.3 Virtual ethnography 38
4.2.4 Process simulations 39
4.2.5 Data mining on FOSS repositories 40
4.2.6 Extracting processes from data 42
4.2.7 Combining research methods 44
4.3 Process metrics for FOSSD 45
4.3.1 Usage of process metrics 47
4.3.2 Size-oriented metrics 48
4.3.3 Activity-oriented metrics 51
4.3.4 Project success oriented metrics 54
4.3.5 Other metrics 56
4.3.6 Constructing indicators 58

5 Future directions **61**
5.1 Future of FOSSD 61
5.2 Future directions in the research on FOSSD 63
5.3 Automated tool for FOSSD metrics 65

6 Conclusions **67**

References **69**

1 Introduction

In his speech at the Computing Community Consortium last year, Christos Papadimitriou said that the Internet is the first computational artifact that was never designed, and that it is changing computer science into natural science and social science [Pap07]. This requires new methods based on observation, experiments and falsifiable theories, as well as borrowing methods from various other disciplines.

Free and open source software development (FOSSD) is part of the same phenomenon as the Internet, and Papadimitriou's notes apply to it as well. It is a relatively new approach for the development of software, where a multitude of people work on the software without apparent central plan or traditional mechanisms of coordination. It takes place primarily in the Internet and is enabled by the Internet. Like the Internet, it is by its very nature an emergent process, where the results are not based on the plans of any individual party, but arise from the interaction during the process.

Free and open source software is developed by geographically distributed teams of volunteers with no central control that could define what or how the developers implement. However, the structure of the development teams is highly unequal with a small core of dedicated developers providing most of the code, and orders of magnitude larger spheres of supporting co-developers and active users who participate with much lower intensity. This model of development differs significantly from software engineering as it is described in text books.

The amount of free and open source software has increased exponentially along the expansion of the Internet, and currently there are at least hundreds of thousands of such software projects, though majority of them very small. Also the publications on the development of the software have increased rapidly since the turn of the millennium. Currently there is an annual conference and a few workshops on the topic and multiple article collections and special issues of various journals have been published.

The purpose of this thesis is to summarize the current knowledge about FOSSD and explore the ways on which further understanding on it could be gained. The main focus is on successful projects and a "pure" model of the development as a community project, without significant input from any company.

As this is a thesis on the field of software engineering, the focus is on analyzing the process of creating software, not the software itself or e.g. its economical implica-

tions. However, as the subject is relatively new for software engineering and differs significantly from the traditional subjects of the discipline, the commonly used research approaches and many methods of software engineering are not suitable. To aid the situation, some tools and results from other sciences have been borrowed, among them — perhaps surprisingly — disciplines such as sociology, anthropology and linguistics beside the more typical statistics, mathematics and economics.

Research methods used in studying FOSSD include case studies, surveys, virtual ethnography, process simulations, data mining, process extraction, automatic discourse analysis and social network analysis, among others. In particular, constructing software process metrics for FOSSD is a promising approach to validating research results obtained by other methods.

This thesis is organized as follows. Chapter 2 discusses free and open source software, and introduces relevant concepts related to it.

Chapter 3 summarizes and discusses, based on research results, the current understanding on how free and open source software is developed. Also comparisons between that empirical model and software engineering as text book discipline are made.

Chapter 4 introduces the research methods used in studying free and open source software and discusses the challenges the topic sets for software engineering in general. Also the usage of software process metrics as a tool for understanding free and open source software is thoroughly explained.

Chapter 5 discusses the future of FOSSD as well as the future of the research on FOSSD. Chapter 6 summarizes the results from previous chapters.

2 Free and open source software

Everyone knows the Linux operating system and the Mozilla Firefox browser. Many have also heard of the Apache web-server, and some know software such as Bind and Sendmail that keep the Internet running. But what are these software systems and the software projects that create them? Answering these questions is the purpose of this chapter.

Chapter 2.1 introduces the concepts *free software* and *open source software*. Chapter 2.2 explains the licenses used for free and open source software, Chapter 2.3 the history of them and Chapter 2.4 discusses them as social movements. Chapter 2.5 introduces the book that has set the research on the topic in motion.

2.1 Free software and open source software

The concepts *free software* and *open source software* are generally used inter-changeably, but sometimes seen as conflicting, depending on the point of view.

Free Software Foundation (FSF)[1] defines free software as software on which the user has the right to run, copy, distribute, study, change and improve it. More specifically, FSF defines the following criteria for free software [Frea]:

- The freedom to run the program, for any purpose.
- The freedom to study how the program works, and adapt it to your needs. Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor.
- The freedom to improve the program, and release your improvements to the public, so that the whole community benefits. Access to the source code is a precondition for this.

Open source software is defined by Open Source Initiative (OSI)[2] as a set of requirements for the distribution terms of the software. Essentially OSI requires the

[1] Free Software Foundation, http://www.fsf.org, is generally acknowledged as the authority that can define what free software means. Actually the whole concept of free software originates from the chairman of FSF, Richard Stallman.

[2] Open Source initiative, http:/www.opensource.org, was founded to steward the definition of term *open source*. It is the community-recognized body for reviewing and approving licenses as open source licenses.

following [Opeb]:

- The user has the right to redistribute the program.
- The user has access to the source code.
- The user can make and distribute derived works.
- The license does not discriminate groups of users or ways to use the software.
- There are no restrictions on distributing the program with other software or regarding technologies in use.

The terms *free software* and *open source software* have the same or almost exactly the same extension, i.e. the set of free software and the set of open source software are roughly speaking the same, which allows using either term when referring to both.

On the other hand, the intensions, or meanings, of the two terms do differ: free software emphasizes freedom and rights of the user-developer, whereas open source is generally understood to focus more on the practical matters of creating and distributing the software with no political implications. Thus it can be said that the difference or equality of the two terms depends on the context.

Some research papers referred to in this thesis use the term *free software*, while others use the term *open source software*. Also combinations such as *free and open source software (FOSS)* and *Free/Libre and open source software (FLOSS)* are used.

For the purpose of this thesis, the differences of free and open source software are not significant, and thus the term *free and open source software*, or *FOSS*, is used when referring to both of them throughout the thesis, except in literal quotes. Similarly the abbreviation *FOSSD* is used for free and open source software development or engineering.

2.2 The licenses

FOSS is, by definition, software distributed with free/open source licenses. For this reason it is necessary to give a short overview of the most important FOSS licenses.

GNU General public license (GPL) [Freb] is the most common of the FOSS licenses [LT05]. It is a 'copyleft' license, meaning that all parties modifying the pro-

gram or using it in new development and distributing their changes must distribute them with GPL license.

GNU Lesser general public license (LGPL) [Frec] is the other of the FSF licenses. It is a variant of GPL with weaker copyleft nature: only the changes made directly to the program need to be published with LGPL or GPL, but work that merely uses the LGPL licensed program does not need to.

BSD style licenses [Opea] are more flexible when it comes to distribution terms. The only requirement is to mention the original author in documentation and when the program is used. Creating modified versions and publishing them under a different license, also without publishing the source code, is allowed.

The license signals and institutionalizes the values and beliefs of the developer community [Sca05], and thus it plays a role in setting norms for the development and on what projects create alliances in the FOSS ecosystem. See Chapters 3.1.1 and 3.1.5 for the role of norms in development.

As this thesis is not focusing on licensing and distribution, but rather on software development, there is no need to go in details or in the economical implications of the licenses.

2.3 History of FOSS

The origins of FOSS as a concept and movement are often said to be in the hacker culture of the 60's and 70's in the US university computer science departments, particularly at Massachusetts Institute of Technology [Ray99a]. According to Stallman, back then it was seen natural that the source code of every program was always available to the user. All you had to do was to ask for it, as sharing the code was just normal neighbor help for the fellow hacker [Sta99].

In 1985 Richard Stallman founded Free Software Foundation to support his goal of writing a free Unix-like operating system runnable on modern computers. The system is GNU (recursive acronym for GNU is Not Unix). Since then FSF and Stallman as its chairman have been the central force in making free software known to the public.

In 1991 Linus Torvalds, a student in the Department of Computer Science at the University of Helsinki, announced in the newsgroup comp.os.minix:

> "I'm doing a (free) operating system (just a hobby, won't be big and

> professional like gnu) for 386(486) AT clones. [...] It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have."

This was the beginning of the Linux operating system that is today ported to dozens of processor architectures and has probably the largest list of supported hardware of all operating systems. The important aspect of Linux was not any of its technical qualities, but the model it was developed on, i.e. the model described in Chapter 3.1.

The development of Linux and the rapid expansion of the Internet that started in the early 90s created a huge expansion of FOSS. This started the trend of exponential growth seen in many characteristics of FOSSD as discussed in Chapter 3.1.6.

In 1998 the Open Source Initiative was founded by some major FOSS activists, such as Eric Raymond and Tim O'Reilly, to promote and guard *open source*, their new and more business friendly term for what was before that known solely as *free software*. When Netscape published their web browser as open source software in the same year, the role of FOSS as a publicly known alternative for "closed source" software was ensured.

2.4 FOSS as a social movement

FOSS is not just software, it is also the people who create it, the FOSS movement, or to be more precise, the free software movement and the open source movement. Since the introduction of the term *open source*, there has basically been two movements and the spokespersons of each have been trying to clearly separate themselves from each other.

Free software movement, grouped around FSF and Debian distribution emphasizes the value of freedom. The point is not what the software can do, but what the user can do with the software, his freedom to adjust it to his needs and distribute it further. Open source movement, on the other hand, pays attention to the practical benefits of the availability of the source code and tries to make producing FOSS appealing also to companies, which do not appreciate the freedom fighter rhetoric of FSF.

Stallman sees the rivalry between the free software movement and the open source movement as an antithesis of the faction fights of 60's radicals: instead of agreeing in goals and disagreeing in strategy, the FOSS movements disagree on the principles

and goals, but agree on strategy on most major issues [Sta02].

Due to this agreement in strategy, the distinction between free and open source software is not relevant for the most parts of this thesis. An exception to this are the situations where the values of the community can been seen as affecting the development [ES08]. More on this in Chapter 3.1.1.

2.5 The Cathedral and the Bazaar

It can be said that the research on FOSSD was started by one book, *The Cathedral and the Bazaar* by Eric S. Raymond [Ray99a]. In his book, originally published as a set of essays in the Internet, Raymond describes how FOSS is created and confronts that with traditional software engineering.

Raymond's book is far from scientific. He does not generally even try to find statistical evidence for his claims, and his concept of software engineering is that of a cultured practitioner without specific education. Not surprisingly, later research has shown many of his claims incorrect or too broadly interpreted [Sca06].

Raymond himself has said later that the facts in the essay are not novel or important, but the intention was to create metaphors and a narrative that would inspire the reader to think of the facts in a new way [Ray99b]. In this he clearly succeeded: besides the popularity in FOSS movement and software industry, the book also defined the framework against which the first scientific studies in FOSS were concluded. A vast majority of all scientific papers written on FOSSD between 1999 and 2002 refer to the book or the essay with the same name.

3 Free/open source software development

Walt Scacchi summarizes the key points of FOSSD and its relation to software engineering as follows [Sca06]:

> F/OSSD is mostly not about SE [software engineering], at least not as SE is portrayed in modern SE textbooks. Similarly, F/OSSD is not SE done poorly. Instead, F/OSSD is a different, somewhat orthogonal approach to the development of software systems where much of the development activity is openly visible, development artifacts are publicly available over the Web, and generally there is no formal project management regime, budget or schedule.

The first part of this chapter is dedicated to describing how FOSS is developed based on current scientific knowledge. The second part compares the findings to traditional software engineering.

3.1 Characteristics of FOSSD

Free and open source software development is a complex socio-technical phenomenon. In order to analyze it, it is necessary to focus at the same time to the software and processes affecting it, as well as the team working on the software: why are the developers working on it and how do they organize themselves. In the context of FOSSD, the team and the software are inseparable [RG03, YNYK05].

There is no single universal approach to FOSSD. Projects differ a lot from each other, and there are differences even in the workings and organizational approach of a single project over time [ØR03]. Classifications of different development styles have been made [YNYK05], but there is no general consensus on taxonomy of projects.

The focus here is on relatively large projects with multiple core developers and an active community of supporting developers and active users. Also projects where a single company has significant influence are outside the scope. All the results and proposals expressed in this chapter discuss such projects, unless specifically mentioned otherwise.

With these limitations, some general trends and typical features of FOSSD can be pointed out:

- FOSSD is based on geographically distributed teams that may never meet each other.
- There is no central authority comparable to project management: even though a hierarchical structure can be seen, the project leader cannot effectively assign tasks to people and he cannot force them to act on his will.
- The projects tend to form complex webs of inter-dependent projects, or FOSSD ecosystems.
- Many artifacts familiar from software engineering text books such as design, schedule, project plan and list of deliverables are missing.
- Many attributes of projects, most notably the number of developers are distributed on power law distribution.

Power law distribution is of form $N(k) \sim k^{-\gamma}$, where k is e.g. the number of developers in a project and γ is an exponent dependant on the phenomenon at hand, often between 2 and 3 [BA99]. Power law distributions are typical for large social networks and other self-organized networks. Reasons for this are not really known, but preferential attachment discussed in Chapter 3.1.6 has been proposed as an explanation.

As research on FOSSD is a relatively new field and differs significantly from traditional software engineering research, as discussed in Chapter 4, many of the results presented here are preliminary or just empirically grounded hypotheses and need still more research to be confirmed. This uncertainty is not expressed for each reference, but it applies throughout this chapter.

Each of the subchapters discusses FOSSD from a different perspective. They are not that much discussing different parts of the FOSSD, but rather the same phenomenon from different angles. Overlapping parts are shown in the text by referring to related subchapters. The subchapters are as follows:

- Chapter 3.1.1 discusses who the FOSS developers are and how FOSSD teams are formed
- Chapter 3.1.2 discusses the role of development tools in forming the processes
- Chapter 3.1.3 discusses requirements management and its absence

- Chapter 3.1.4 discusses software design and how that is interwoven into maintenance
- Chapter 3.1.5 discusses how cooperation and coordination of the development takes place
- Chapter 3.1.6 discusses free and open source software evolution
- Chapter 3.1.7 discusses the *forking* of projects, that is splitting them into two
- Chapter 3.1.8 discusses how the FOSSD projects relate to each other and constitute ecosystems.

3.1.1 Team formation and reasons for contribution

When studying software development, the questions regarding who and why create the software cannot be ignored. This is especially true for FOSSD, where the contributors generally contribute what and how they want, not necessarily what the project leaders ask them to contribute.

The motivations of individuals to contribute to a FOSSD project vary, of course. Motivations suggested in scientific or popular literature include need for a specific functionality, development as a hobby, learning, feeling of belonging to a community or movement, need for recognition, sense of achievement and career advancement [SC06]. In an on-line survey done in the United Kingdom Linux User Groups, the three most popular reasons for participating in the FOSS community were problem solving (86%), providing support (81%) and social exchange (70%) [SC06].

Typically people join FOSSD projects proactively. Quite typically new participants in a project have also prior social ties to some member of the project. Active recruiting of developers seems to be very rare [HMB+06]. The social ties of project members to other FOSS developers play a role also in the general development of the project as discussed in Chapter 3.1.8.

It is quite clear that the motivations of FOSS developers vary significantly and cannot be reduced to any simple explanation, whether that would be gaining status in gift economy, as Raymond proposes, or just wanting to help the neighbor, as Stallman puts it. However, it can be said that the motivations are not limited to interests related to the software itself, but the social interaction related to creating the software plays an important role too.

The motivations of the developers in a project are not only an individual matter, but an important part of the group dynamics of the FOSSD team. The common beliefs and values of the team constitute its norms and work practices [ES08, ES05], and affect even purely technical choices [ES05].

Elliot and Scacchi list the following beliefs[3] and values[4] in their empirical study on a FOSSD project called GNUe [ES05]:

- belief in free software: that it is important to use (only) free software when developing free software
- belief in freedom of choice: that every participant can select freely what he does or doesn't do
- value in community: developing a community is a goal in itself
- value in cooperative work: preference that other people should participate in one's work.

How these values and beliefs affect the functioning of a FOSSD team is discussed in Chapter 3.1.5.

The work of Elliot and Scacchi was based on the GNUe project, which defines itself as free software. It is reasonable to assume that the values and beliefs of a project that identifies itself as an open source software project would differ somewhat from what is described above, as the articulated ideologies of the two movements differ. As for today, there is no empirical study to confirm this hypothesis. Thus it is also unknown if related differences in norms and work practices could be found.

Companies and paid developers participating in FOSSD projects often do not differ significantly from "normal" project members [Ber06]. Statistically, companies are somewhat less active on average, and also projects initiated by a company are less likely to success. The lack of differences may be partly explained by the fact that many paid contributors have been participating in FOSSD projects already before their work assignment [BR04]. This means that the phenomena and findings explained here may have some applicability also to projects with significant company influence. Further investigation of this topic is outside the scope of this thesis.

[3]Beliefs: expressed cause and outcome relationships [ES05]

[4]Values: expressed preferences for certain behaviors or for certain outcomes [ES05]

3.1.2 Development tools as the process

The FOSSD methodology is largely based on using the typical FOSS development tools. The culture and methodology of FOSSD are conveyed to new developers via the tools: version management tools such as Concurrent Versions System (CVS) and Subversion teach to keep all the source artifacts available to everyone, issue management tools such as Bugzilla teach to accept feedback and proposals from anyone and mail list archives teach to keep all discussion available for everyone [Rob03]. A developer setting up a new project can easily define all major FOSSD processes just by setting up the tools (provided easily by FOSS portals) and by looking at some example on how they are used in other projects — this all without it crossing his mind once that he should think about processes.

On many development tool categories there is a multitude of good FOSS tools available and in widespread use, especially in version control, build systems, issue tracking, technical discussion and automated testing and documentation [Rob02]. On the other hand, there are important aspects of software engineering where FOSSD tools are missing, or at least relatively unused. These include requirements management, project management, metrics, estimation, scheduling and test suite design [Rob03].

The question why in some categories tools are widely used but in others almost nonexistent is certainly worth further studying. It looks as many of the tools in use are there to support experienced developers in their work, whereas tools that are not needed by them but focus e.g. on supporting management tend to be lacking. The tools that gain popularity tend to reduce the amount of non-preferred work a developer needs to do, such as documentation or testing [Rob03]. Gasser et al. propose that the lack of design tools is due to the fact that FOSS projects "have no need to fake a rational design process as is all too common in software engineering" [GSRP03].

The choice of tools is not only a practical matter, but has ideological background as well. It is important in many FOSSD projects that the tools used in development are also FOSS themselves [ES03]. Again, this should not be seen as a matter of individual preference, but as a part of building a community which is an important part of the social dynamics of FOSSD [Ell03]. Also, as discussed above, the set of common tools creates a tie to other FOSSD projects and makes it easier for outsiders to contribute to the project.

The usage of tools supporting the process, such as issue tracking or packaging tools,

has spread in FOSSD projects extremely fast, when compared to the much slower spreading of e.g. CASE tools [Rob02]. With respect to the findings so far, this should not be a surprise. Freely and easily available tools, as well as publicly available examples on how they are successfully used, makes it quite easy to take them into use.

Of the various tool categories, two are of special interest from the process perspective. The first is structured communication systems, or knowledge management environments, such as newsgroups, issue tracking systems and frequently asked questions (FAQ) documents [GSRP03]. The important aspect of these environments is that they store all the discussion in a way that keeps the observations and agreements available also in the future, thus enabling the development team to learn as a community — even when the participants change. Also these systems capture traces of design and analysis processes and thus serve as a valuable source of information when trying to analyze FOSSD, as discussed in Chapter 4.2.6.

The other interesting category is real time discussion channels, meaning instant messaging systems and especially IRC. Many FOSSD projects have their own IRC channels, and the debates in IRC result in reinforcement of the cultural beliefs, which then supports motivation to join a project and contributes to building the community [Ell03].

3.1.3 Requirements analysis and management

The most notable characteristic of requirements analysis and management in FOSSD is that it is not done, at least not in the text book sense. Traditionally the requirements process can be divided into requirements ecilition, analysis, specification and validation. Below each of these phases is discussed in the FOSSD context, based on a thorough empirical analysis of Walt Scacchi [Sca02b].

Scacchi notes that in his research on four FOSSD communities and multiple projects, no evidence of requirements ecilition was found. Instead requirements are typically just asserted, written on a discussion forum without reasoning or justifications. The requirements exist because some developer wants them. Often a requirement appears in discussion the first time when someone announces that he has implemented it [SJNE06].

Also systematic requirements analysis is missing. “Requirements for open systems are, in practice, analyzed via the reading of technical accounts as narratives to-

gether with making sense of how such readings are reconciled with one's prior knowledge" [Sca02b].

Not surprisingly, there are no requirement specifications either. The task of requirement specification seems to be taken by a network of discourses regarding the details of the requirements. Instead of writing a requirement systematically down, one just proposes it in free form and then participates in discussion about how the requirement should be interpreted.

The validation of requirements in FOSSD is not a goal of its own, but an implicit by-product of the process that constitutes the requirements for the software. If project participants do not smash down a requirement proposal, but develop it further in discussions, it is then judged worthy by experts and thus validated. Ultimately a requirement is valid if someone takes the time to get it implemented.

The requirements are managed as and by *software informalisms*. Software informalisms are web based descriptions in open form such as usage scenarios, code examples, HOWTO and FAQ documents, issue tracking system entries, etc. The name informalism is intended to contrast the practice to formal systematic notations, "formalisms".

The informalisms are situated discourse, answers to particular questions meaningful only in their original context. A key precondition for informalism based requirements management is that all the informalisms are freely available and persistent. They would not be meaningful if the whole original context were not easily reachable.

Even though there is no technical reason why informalisms used for requirements management could not be formalized, the developers do not seem to find any need for that. One possible explanation for this is that the developers are also active end-users of the system, and are thus a lot more familiar with the possible requirements than in traditional software engineering. The requirements are in a context familiar to them, and "just make sense".

It seems that requirements management is not a recognized first class task in FOSSD to the extent of programming or testing, something that would need to be done. Instead it is implied activity, a side product of community building. The requirements just emerge from the community. Also the requirement gathering is not an actual phase in a process, but rather ongoing activity that is done all the time regardless of the phase of the software release cycle [SJNE06].

Another face of FOSSD requirements management is that many projects reduce

the need for it by relying heavily on standards, especially regarding network communication [ET03]. The standards take the role of requirements documentation. This could also explain why FOSS appears to follow and support standards more rigorously than closed source software does [Rob03].

3.1.4 Continuous design and defect management

The defect management repositories of FOSSD projects are used for much more than just defect management. They are used for continuous, distributed and collective software specification and design [GSRP03].

The design and re-design are not bound to any particular phase of the development cycle or any specific time, but they are continuously going on whenever new issues that affect the design are introduced to the repository. It is a process where the user-developer community continuously builds its knowledge of what the target artifact, e.g. software module, is, what it does, what it should be and what it should do [GR03].

This construction of community knowledge is enabled by a knowledge management environment, typically an issue management system [GR03]. This kind of shared environment is a vital part of the process, as it does not just serve as a platform for design discussions, but also stores the knowledge of the community in its structure and gives a shape to the process as discussed in Chapter 3.1.2.

Traditionally the documents in an issue management system have been considered to have a conventional and straightforward relation to reality: an issue report describes an issue as it is, and any deviation from that relationship is a mistake and exception. However, empirical observations on how FOSSD projects use issue management frameworks show quite a different presentation model [GSRP03].

First, the documents record viewpoints, not reality, and these viewpoints are subject to interpretation. There are typically many comments on a problem report, and they may conflict each other arguing about the proper interpretation of the report. Also the same, or possibly the same, issue is often reported multiple times with somewhat different descriptions. The "sameness" of issues is also subject to interpretation and discussion. It can be said that the documents and discourses represent different aspects of a problem, not the problem itself.

The fidelity relationship between the documents and experiences is unclear, and it is not even possible to know the relevant real situation. The discussion on interpre-

tation of a problem report, about what the “real” issue is behind it, is a common phenomenon.

Also the scope of documentation is unknown, and the knowledge is always partial. There may always be more documentation somewhere, relevant to the matter at hand. Cases where two groups of people work on related or identical issues not knowing about each other are not rare.

The issue reports are typically linked to each other and can be analyzed as bug report networks (BRN) [San05]. Issues may be linked to each other by two forms of formal relationships, duplicate and dependency, as well as by informal “see also” type relationships. Also “meta bug reports” that do not try to represent any actual issue, but just group together a set of related issue,s are used as informal adaptation. For example in the Apache project 65% of the issues have a relationship to other issues [San05].

Not surprisingly, the defects in a FOSSD project tend not to follow formal models of defects — neither ones defined in software engineering literature nor ones defined by FOSSD projects themselves [Kop06]. Typically an issue starts as ’new’ and the only transition is directly to ’resolved’, which serves as the de facto end state of an issue.

The infrastructure that enables the continuous design process of FOSSD projects consists typically of the following:

- issue management system, e.g. Mantis or Bugzilla
- mailing list with a web archive, news group or web bulletin board providing a forum for non-realtime discussion with archives always available
- a chat room, typically an IRC channel, often with web archives.

The design processes in this infrastructure are not based on formal notations or explicit representations, but on informalisms, as discussed in Chapter 3.1.3.

The design style described above constitutes a fundamentally new type of design process based on a dynamic network of social processes [GSRP03]. This emergent knowledge process is enabled by a tool framework that enables accumulation of community knowledge [GR03] .

Issue management as seen in the issue management systems is actually information management, construction of the community knowledge [San05]. And as such, it is

also social ordering creating and codifying the social relationships in the project. For instance Barcellini et al. describe examples of design discussions where the social roles in the project can be seen to affect the process [BDBS05].

The time an issue remains open appears to be to a large extent dependent on the time it takes for the community to reach a consensus on what the issue is and what should be done about it [GR03]. Considering the process described above, this hypothesis sounds plausible. If there is no consensus, the design process on this particular issue is still going on and thus there can be no resolution.

3.1.5 Cooperation and coordination

The coordination in FOSSD projects is one of the most interesting issues in the whole phenomenon. How are projects with no common organization and no physical meetings managed? How is the apparent success in such a setting possible? Distributed organization is generally considered challenging due to discontinuities in physical and organizational settings [WMCC02], yet many FOSSD projects seem to successfully overcome these challenges, as the average productivity of programmers is almost stable with regard to project size [Koc05].

Chapter 3.1.1 discussed the FOSSD developers conceptualized as forming communities and the social dynamics that keep the communities together. Here the focus is more on the developers comprising a work team: how they act together to achieve a common goal. Needless to say, there is no one common model of coordination that would cover all FOSSD projects as even the same project at different times may be utilizing a different model [ØR03]. Some general observations and hypotheses can still be made.

As the nature of FOSSD projects makes cooperation and coordination challenging, the primary approach seems to be to avoid the need for cooperation. Instead the organizational model can be described as "cooperative individual work, where each developer works individually sharing his work products with others" [ØR03]. Also other studies confirm that minimizing the need for collaboration is a common practice [JS04].

Also the structure of the FOSSD programs seems to be designed to enable working with no need for heavy cooperation. FOSS tends to be highly modular, though in an unorthodox way, as developers do not respect module boundaries, but modify freely any components when doing changes to the source code. So the projects tend

to be modular in the level of source code, but the principle of information hiding is not followed [NR05]. This modularity of the source code may be a precondition for successful FOSSD, as it allows more individual work [MH02].

A FOSSD team appears to be highly centralized in nature. It is often described as an onion like structure with 4 different layers shown in Figure 3.1:

- core developers writing most of the code and generally responsible for software architecture
- co-developers contributing code infrequently or only on some part of the project
- active users providing feedback and bug reports as well as participating in discussions and helping each other in using the software
- passive users who just use the program.

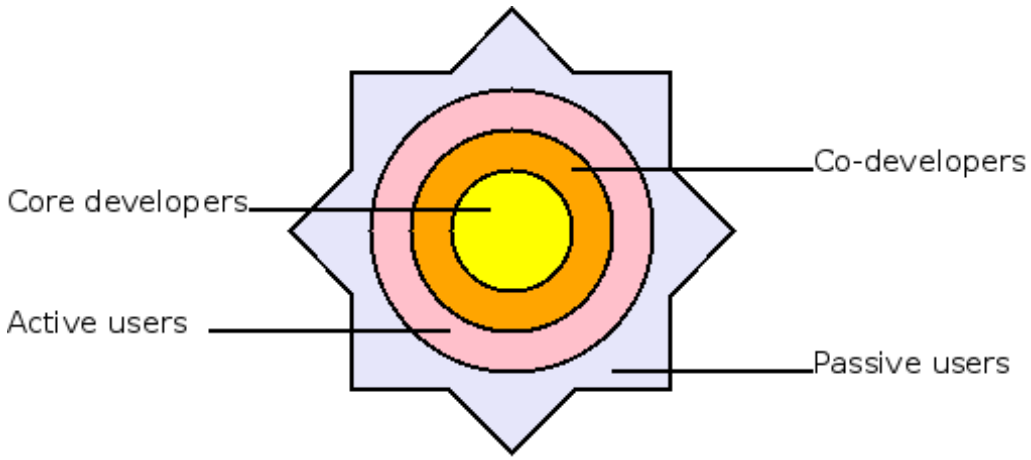


Figure 3.1: The onion model of FOSSD team [CH05b]

In the center of the core group is often one project leader, who decides what code is accepted to the project and who is generally acknowledged by project members, though in some projects decisions are also based on votes in the core group [CHAM05]. The core group is very small compared to the whole project size, often just 1-3 developers [CWLH06]. Each sphere around it is significantly, even orders of magnitude, larger than the previous one.

The passive users are not visible in the project tools or structures at all, as they just use the developed program. This makes the estimation of their number difficult, as

discussed in Chapter 4.3.2. As an interesting consequence of this, FOSSD projects do not seem to suffer from the "free rider problem", i.e. of user who just benefit from the work of others but do not give anything in return. As the passive users do not appear at all in the project, they do not consume any resources, and thus it does not matter that they do not give anything in return.

The validity of the onion model is not empirically proven, but it appears to be a general consensus model for conceptualizing FOSSD team structure both in academic community and among practitioners [CWLH06], though it has also been called into question [CH05b]. There is also some supporting evidence showing that the roles of participants as described in the model do effect the discussions and decisions in a project, namely that the opinions of the core developers carry more weight [HIC06, BDBS05].

Considering the onion model, identifying and analyzing the core group of a project is a relevant research topic. Despite the plausibility and popularity of the model, Crowston et al. found it surprisingly difficult to identify the core group reliably [CWLH06]. This is discussed more in Chapter 4.3.2.

Despite the difficulties in identifying the core, some results regarding its functionality have been found. The range of centralization seems to vary a lot between projects, with the larger projects somewhat less centralized [HIC06], which may partly explain the difficulties in automatic identification of the core group. Also changes in the members of the core seem to be relatively uncommon. On the periphery of the project they are much more common.

Typically new participants appear first on the sphere of active users, and climb only later to a position of co-developer or even core developer — if they ever do [SDD+06]. A vast majority of the participants appear only for one period or a few short periods of time [HIC06].

Mockus et al. came in their research to the conclusion that if the size of the team working actively on a project or subproject exceeds 10-15, the subproject needs to be split further to ensure efficiency, which would effectively establish an upper limit for the size of the core group of a project [MFH02]. On the other hand, no evidence of such splitting was found on a study on the module structure of the Apache web server [GBLR04].

Many features of FOSSD projects coordination are probably results of the fact that they are voluntary projects, and usually ones without any formal organization. Thus

there is no way the project leader, or a supporting organization or some particular developer, could force any participant to do work some particular way — or at all. Instead the only options for the project leader are either to wait that an interested developer picks it up or to do it himself.

So, despite the highly centralized nature of the project group, there seems to be no hierarchy in assigning tasks [CWL+05]. Requests that someone should work on a particular task are formulated as proposals: "if you have time, you could look at X". There is also some evidence that when formal structures such as a management group have been defined in a project, they are not used in practice, but informal coordination is used instead [JS04].

Instead of commanding, the most common form of assigning a task is self-assignment, which serves also as a replacement for a coordination mechanism [CWL+05]. The tasks are not just self-assigned, but they typically also appear the first time when someone self-assigns them to himself. As discussed in Chapter 3.1.3, the legitimacy of a task is strongly linked to whether someone sees it relevant enough to get it implemented: "the code speaks louder than words".

The successful work of a FOSSD team is largely based on creating and utilizing shared mental models [CHAM05, CWL+05]. A shared mental model means that the team members have similar understanding on topics such as the project goals and limitations, the roles of different software modules, the target software architecture, the division of labor, etc. Crowston et al. propose that more highly developed and detailed shared mental models improve efficiency of cooperation by reducing the effort needed to collaborate. The hypothesis has not been empirically proven, but it is plausible.

Related to the mental models, the source code itself is an important institution of cooperation, i.e. questions regarding topics such as roles, planning and decision making are conceptualized by the participants as questions regarding the source code: who can change it, how should it be changed and when is it ready for release [Sca05]. This limited cooperation channel could be one reason for the claimed better quality of FOSS code: as the developers can make fewer assumptions about how their code will be used, they have to write it more robust and modularized [CAHM05].

The success of some FOSSD projects is also attributed to the large spheres of co-developers and active users who find and solve various issues in the software. Eric Raymond defines the phenomenon as Linus' law: "given enough eyeballs, all bugs are shallow" [Ray99a]. The work of finding issues or solutions to them does not require

much cooperation, and thus can be efficiently divided even to very large groups of people. Also finding issues doesn't require any special skills.

3.1.6 Software evolution

Studying the structure of a project team without taking the issue of time into consideration may lead to misleading results. As developers constantly join and leave the community and the structure may be totally different on different periods of time, it is important to also study how FOSSD projects change over time [HIC06].

Lehman et al. have been developing the theory of software evolution since 1980's focusing on the setting of typical commercial software development, trying to find and describe laws by which evolutionary pressure causes software to be maintained [Leh80, Leh96] . In this approach software is not seen as an object of actions but more as a natural process.

Considering the emergent and non-managed nature of FOSSD, this approach might actually be better suited for analyzing FOSSD than it is for traditional software engineering. Besides, when studying FOSSD the sample sizes may be thousands of projects and almost all project communication is available in textual form, which gives a more reliable basis for constructing general laws than the much more limited data on closed source development.

Applicability of the laws of software evolution in FOSSD has been studied, and it appears that at least the law of slowing growth [Tur96] is not applicable. Instead of slowing growth the Linux kernel seems to be growing in super-linear fashion [GT00]. Later research on over 4000 large projects showed that 61% of the projects were growing sub-linearly, but 39% — over 1500 projects — fitted better the exponential curve, making it clear that Linux was not an exception but an example of a more general rule [Koc05]. Also the larger projects are more likely to grow super-linearly, as are the projects where development effort is more unequally distributed. These findings support the validity of the onion model described in Chapter 3.1.5.

Naturally, such fast development is not the whole truth about FOSSD. In another similar large sample it was found that 97% of the projects had only less than one percent of change in a period of six months [CLM03]. As majority of the projects on major FOSS portals have only one developer, it appears that the distribution of projects regarding the amount of developers, activity or success is also extremely unequal, following power law distribution. This will be discussed in more detail in

Chapter 3.1.8.

The project evolution can be explained as application of peripheral attachment, which means that projects involving more developers are more likely to gain additional developers, i.e. "rich get richer" [WMZ06]. Also developers seem to move from project to project or find new projects in groups that were well connected beforehand. This can be explained by the fact that prior social ties play an important role in joining a FOSSD team, as discussed in Chapter 3.1.1.

Other interesting phenomena regarding FOSS evolution include the following:

- Module structure in a large project appears to stay relatively stable when studied as social ties of developers between modules, i.e. linking modules with common developers as edges [GBLR04] .
- The number of source lines of code per module seems to stabilize over time in large projects, i.e. the variation on average module size diminishes [CLM03]. The stabilized module size varies between projects, and no general rule for that has been found or hypothesized yet. Interestingly, also the average size of packages in Debian GNU/Linux distribution has been very stable while the total size has increased exponentially [GBRP+05], suggesting that there may be some general phenomenon in FOSSD causing this stability.
- Developers seem to have a tendency to specialize to a more limited set of modules over time [GBLR04].

The phenomena described above are based on studies on one project or a few projects, and thus should be seen more as empirical hypotheses for further study than reliable results.

In general it would appear that the projects develop more in incremental small steps than in major revolutions [Sca05]. The reason behind this could be that typically only a minority of the developers would support a revolutionary change, thus risking forking the project if such a change would be implemented, though this reasoning has not been verified empirically. For more discussion on forking, see Chapter 3.1.7.

3.1.7 Fork

One important aspect of FOSSD is forking, that is, starting an alternate development based on the same original code base. Historically the concept of forking comes from

commercial Unixes, but the forking in the FOSSD projects differs from the Unix forks, as discussed below.

In order to describe the phenomenon, it is useful to first introduce a few examples. Most of the details are based on the essay of Rick Moen[5] [Moe99]. Even though there are companies involved in many of the cases discussed below, all the project variants include an active developer community also outside the company, thus making them relevant examples for the purpose of this chapter.

NSCA and Apache [ØR03]: The Apache project started as a collection of patch files in the NSCA web server, development of which had basically stalled. A group of active web masters simply picked the project and continued development under a different name. The original NSCA got replaced by Apache basically everywhere. This is a good example of forking to replace an almost dead project.

GCC, PGCC and EGCS [Moe99]: GCC, a compiler produced by FSF, was lacking support for Pentium optimization in 1997, and FSF was not actively creating such functionality. A group of people at Intel produced a version of GCC, PGCC, with the optimizations for their processor, but lacking many other features. Some time after that another group of developers at a company called Cygnus developed EGCS mostly for their own needs. It not only supported Pentium, but was also technically superior to FSF GCC in other aspects. EGCS started to gain popularity over FSF GCC, and a few years later also FSF moved its support to EGCS thus making it the GCC mainline. PGCC is an example of a company forking a project for its own interest, whereas the EGCS case shows how a fork that supports the community needs better than the original project can occupy the whole "ecological niche" [JS05b] pushing the original out.

Glib, Linux libc [Moe99]: In the early phase of the Linux project, the developers needed a C library, and chose to fork the GNU C library (Glibc) which was lacking some features they needed. Years later they noticed that Glibc, which had also continued to be developed, included the necessary Linux support, as well as significant technical advances over the Linux libc. Thus Linux libc was abandoned and Linux was moved back to use Glibc. This is an example of a fork for technical reasons and of abandoning the fork for the benefit of the original once the technical issues are

[5]Even though Moen's essay is not formally peer-reviewed, it has been widely discussed in the Internet and corrected based on feedback from a number of people who have been involved in the incidents described in the essay. Thus it is reasonable to assume that the factual descriptions of forks presented in it are correct even though the analysis isn't scientifically verified.

solved.

GNU Emacs and XEmacs [Moe99]: Lucid, a company actively using Emacs and participating in its development got into difficulties in working with Richard Stallman, the head of GNU Emacs development. Mostly for this reason Lucid forked the Emacs project to develop what later became XEmacs. Today both GNU Emacs and XEmacs are active FOSSD projects with somewhat different focus but actively sharing code. This is an example of a fork due to personal disputes, and also a fork where both projects stay alive.

The reported reasons for forking a project include the following:

- personal disputes, possibly due to a "leader cult" [Bez99a]
- need or will of some developers to make a radical technical change [Sca05]
- stagnation of the original project [Moe99]
- interests of a company that differ from the project main line.

The interests of a company are a separate item, as they seem to be a special case. While also individual developers may have different interests than most of the project, they lack the resources to fork the project alone and thus have to adapt, unless there are some other reasons why the developer community might support their fork. A company, on the other hand, may well have enough resources to make a successful fork even with very limited support from the project community.

For a successful fork, the support of the community of developers and users is mandatory. A real fork is not just about duplicating the code base, but splitting the community, a task which does not appear to be easy. As seen in the examples above, a fork typically ends in either the original project or the forked project dying or merging into the other.

Counter examples of the rule of one survivor are besides the GNU EMACS–XEmacs dispute also the variants of BSD [Moe99] and maybe Debian and Ubuntu. In all these cases the projects have a different focus and thus can be seen as occupying slightly different ecological niches [NR05]. The discussion on ecological niches and competition between projects continues in Chapter 3.1.8.

The question about the main reasons that affect which fork(s) will survive, has not been addressed yet in detail. Technical quality and leadership skills of the leaders of

the forks are likely among the reasons, as well as the differences in focus as discussed above.

Even though forking is quite rare, the mere possibility affects the way projects function. If the project leaders fail to satisfy the developer community and the users, the risk of fork increases. Thus forking can be seen as a self-correction mechanism in FOSSD [ET03].

The possibility of fork does not only make the leaders pay attention to developers and users, but it also gives the developers a guarantee that their work is not draining to nothing even if the project leaders decide to abandon the project or change focus: they can always continue themselves. For this reason the right to fork projects is a fundamental part of FOSS and is included in both in the free software definition [Frea] and the open source definition [Opeb].

The copyleft licenses such as GPL [Freb] make an important difference regarding forking: as all of the forked code base must also be under GPL, the project mainline can always incorporate any novel technical solutions, thus making successful forking more difficult and less likely [NR05, Bez99b]. In cases where two forks remain alive, a cooperative sharing of new code seems to be common.

3.1.8 FOSS ecosystem

FOSSD projects typically do not exist alone, but instead they tend to form *ecosystems*, complex webs of projects related in various ways. A good example of this is the web infrastructure shown in Figure 3.2. The various projects in the web infrastructure depend on each other for support in communication protocols and software utilities as well as in providing requirements, often bi-directionally [JS05b]. There is no clear rule for what projects should be seen included in the same ecosystem, but the selection of projects is based on the point of view of the researcher. It is also possible to view almost all FOSSD projects as one large ecosystem.

An ecosystem is not static, but instead under constant change. Each project must continuously re-negotiate its position in the continuous define-implement-revise cycle [JS05b].

Typically the communication between projects does not appear to be direct or systematic. It can be seen in announcements of support in project change logs, reports and feature requests in issue trackers and discussions in related community forums [JS05b]. The communication process and dependency management do

not appear to be "owned" by anyone or located within any particular organization, though a particular interface is often asymmetrically managed and owned by one involved party [MH02].

The ways projects attach to each other vary. The following styles have been identified:

- common protocol standard, e.g. HTTP and related standards between Apache webserver and Mozilla browser [JS05b]
- one community integrates tools created by another community into its own project [JS05b], e.g. Tigris used the Bugzilla issue tracking system implemented by Mozilla as basis for its own Issuezilla tool
- projects share a common developer, a "linchpin-developer" [WMZ06].

Linchpin developers may play an important role in the forming of alliances and technical communication channels between projects. A study on Apache found that typically new subprojects are founded by a developer of an existing subproject, who then attracts his colleagues to the new project [WMZ06]. This phenomenon is discussed also in Chapter 3.1.6

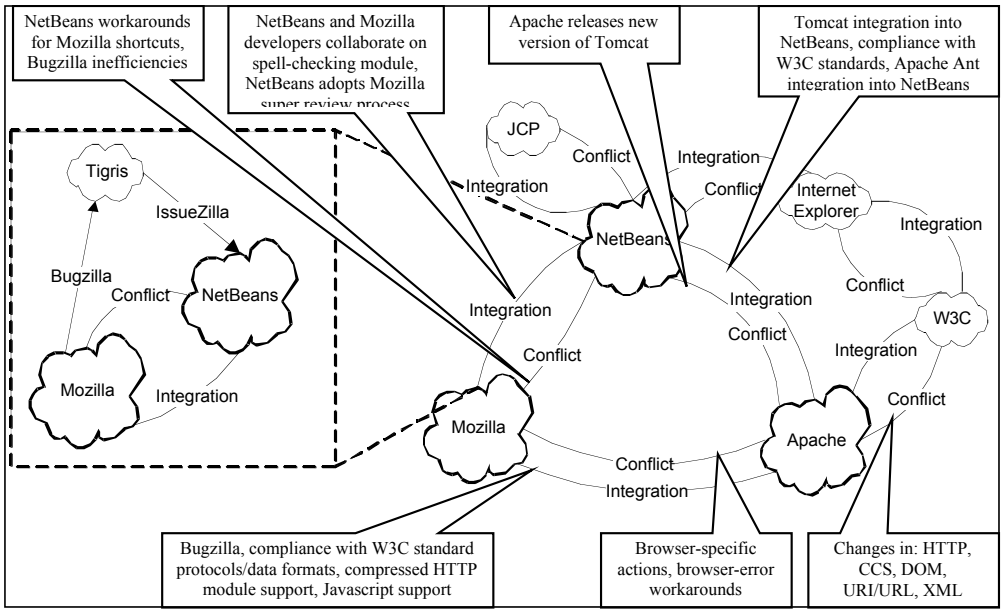


Figure 3.2: Example of a FOSS ecosystem: web information infrastructure [Sca07].

Many attributes of FOSSD projects appear to be spread on power law distribution [MFT02], i.e. a highly skewed distribution with a few high values and a multitude of lower values that appears linear on logarithmic scale. Such variables include developers per project and projects per developer [MFT02], as well as downloads per project and lines of code per project [Koc05]. As an example of this, 79% of the code in SourceForge.net is produced by the top decile of projects and 11% by the second decile [Koc05].

The FOSSD projects do not just happily coexist, but there is also fierce competition between projects that are in the same ecological niche and compete for "market share" in the user community [JS05b]. This competition manifests mostly in making the software better suitable for the users' needs. The intensity of competition seems also to vary significantly [JS04]. Some times two competing projects may collaborate to create common standards they both then implement, whereas in other cases they may see each other as rivals and fight for existence. The analogue to ecology could provide some hypothesis for studying what affects the intensity of the conflict, e.g. if the fundamental niches of the projects are the same or just partly overlapping. Competition of projects is discussed also in Chapter 3.1.7.

3.2 Comparing FOSSD and traditional software engineering

Comparing FOSSD to software engineering is quite problematic, as it is a comparison between reality and an idealized model. Software engineering is essentially a science about how software *should* be made, not how it is made, whereas FOSSD is an existing phenomenon without any clear ideal model.

Comparing FOSSD to the reality of more traditional software development would certainly be interesting, but would be challenged by the fact that analyzing the mentioned reality is not the primary interest of software engineering. Thus the issue is outside the scope of this thesis.

Constructing an ideal model of FOSSD and comparing that to software engineering would also be possible, as shown by e.g. Raymond [Ray99a]. However, that comparison wouldn't really be scientifically relevant, as the FOSSD model would then be just a creation of the author.

Thus the approach here is to compare the reality of FOSSD as described in Chapter 3.1 to the idealized model of software engineering as described in the text books [Pre00, Som07]. This comparison certainly cannot tell if one of them is "bet-

ter" than the other in some sense. Neither can it tell how much FOSSD really differs from the software development done in traditional company setting today. The best the comparison can do is to point out in which ways FOSSD does not follow the guidance of software engineering. The starting point for this comparison is the understanding that the objectives and practices of FOSSD and software engineering are different, and there is no reason to assume that they would be even closely related [Sca02a].

Interestingly, FOSSD may have more in common with academic research work than with software engineering. Both can be seen as emergent processes with self-correction based on peer review and in both one key motivator for contributors is reputations among peers. Also monetary rewards are not considered primary motivator in either of them [Bez99a].

Chapter 3.2.1 summarizes the key aspects of software engineering and discusses the aspects of it that are relevant to this comparison. Chapters 3.2.2 to 3.2.4 list and discuss the differences of FOSSD and software engineering, starting with more fundamental aspects going via processes to observed statistical differences. Chapter 3.2.5 discusses how the agile software development methods differ from FOSSD.

3.2.1 Software engineering

This chapter is based on the software engineering textbooks of Roger Pressman and Ian Sommerville [Pre00, Som07]. Text books are used as source, as the purpose of the chapter is to summarize the key aspects of software engineering as a text book discipline. The description given here is necessarily simplified and does not give a balanced picture of all aspects of software engineering.

Software engineering is a discipline of engineering, of applying technical and scientific knowledge to practical construction of software artifacts within some organizational and financial constraints. It is carefully planned activity, which aims for reliable and measurable development of software that satisfies the needs of the customer.

Software engineering is often set in contradiction with "programming", plain creation of software. This distinction is explained by a metaphor of construction, so that building software should be done like building houses: based on rigid following of detailed plans, division of labor and well defined processes. The details of the processes vary, as developing new process models is a major part of software engineering.

The model described above has a few implications that need to be explicated for the comparison of FOSSD and software engineering. First of all, software engineering requires a central authority that can dictate what and how the participants of the process should do. Implicitly software engineering is expected to happen in a company setting with a customer giving requirements and the management defining tasking and working methods. Exceptions to this setting, e.g. distributed development, do exist of course, but they are exceptions.

Secondly, the key concept of software engineering is the process. More than a science about software, software engineering is the science of utilizing and optimizing software processes. The more abstract research in the field focuses on development of new conceptual models for developing software, whereas empirical software engineering is about analyzing the execution of those models.

Thirdly, the source of requirements for software, "the customer", is outside the scope of software engineering. Requirements are elicited, analyzed and validated. They may be questioned or rejected, but requirements are never invented. The need for software that fulfills some particular given requirements is the starting point of software engineering, and what happens before that point is outside its scope.

Finally, it is important to remember that software engineering — in the sense the term is used here at least — is more about how software should be made than about how it is made [6]. It can even be argued that the process models, such as the waterfall introduced in software engineering courses, were never actually intended to be used as they are described, but to serve as a reference for software development [AB05].

3.2.2 Conceptual differences

This subchapter lists the fundamental differences between FOSSD and software engineering, the incompatibilities of the two concepts. These differences likely are the reasons behind many of the differences discussed in Chapters 3.2.3 and 3.2.4. Bear in mind that FOSSD refers here to the specific way of developing FOSS as described in Chapter 3.1, not to development of any FOSS, as it can in principle be developed with any kind of model and process.

In FOSSD there is no single authority who could dictate what or how the team develops, as described in Chapter 3.1.5. Instead FOSSD is an emergent process, where the tasking is based on self-assignment or suggestions. The project leaders may

[6] For a longer explanation on this, please see Chapter 4.1.

have significant influence on other developers, but the position is not comparable to a project manager in software engineering.

This lack of the central authority implicitly required by software engineering makes basically all the process models and a large part of the cumulated experience of software engineering unsuitable for FOSSD. Thus FOSSD is not just another process model, but entirely another approach to the development of software.

As described in Chapter 2, the source code for FOSS needs always to be available. While this can be true also for any software engineering project, it rarely is. This availability of source code is likely the cause of many process-oriented and statistical differences between FOSSD and software engineering described below.

FOSSD is striving towards informality, whereas software engineering aims for formalizing processes, tools and methodologies where ever possible. Sometimes formal methods and processes are used in FOSSD as well, but there is no push towards them, and there is evidence that they are not a requirement for success, unlike generally considered true for software engineering.

FOSSD is practically always distributed development. While there is a lot of literature on distributed software engineering as well, it is an exception to the general rule that software engineering projects are executed by a co-located teams.

In FOSSD the team and the software are inseparable, and it is not reasonable to study one without studying also the other, as discussed in Chapter 4. In software engineering, it is the tradition to treat the team as resources who execute orders given to them, some better, some worse, but ultimately just doing what they are told by the management.

The requirements for FOSSD development are not coming from an external source, as is assumed in software engineering. Instead they emerge in the community, as described in Chapter 3.1.3.

3.2.3 Process differences

This subchapter describes the major differences between the perceived FOSSD processes and the processes postulated in software engineering text books [Pre00, Som07].

- There is no tasking, instead the most common form of assignment is self-assignment [CWL+05].

- Plans, schedules and lists of deliverables are generally missing [ØR03]. Particularly the lack of schedules can be seen as a key point of FOSSD: "the software is ready when it is".
- Instead of a systematic design phase present in most software engineering processes, FOSSD is continuously re-designed based on informalisms instead of formal designs [ØR03, GR03]. See Chapter 3.1.4 for more information.
- Requirements analysis is generally not done [Sca02b]. Instead the requirements just emerge from the community, and are justified by the effort someone is willing to put to implementing them, as discussed in Chapter 3.1.3. Despite lacking analysis, the requirements in FOSSD may actually be clearer to the developers than in traditional software engineering, as the users/developers are the original source of the requirements and thus certainly understand them [PC04].
- There is no clear change from the development phase to the maintenance phase, and the maintenance activities such as bug management are closely linked to the design and continuous re-design of the software as discussed in Chapter 3.1.4.
- Software engineering processes are iterative, stepping from phase to another and then possibly iterating, whereas a FOSSD process is more evolutionary, moving to different directions based on external and internal pressures [PC04]. Also there are no clear boundaries for different phases, but the process can be seen as being in all phases at the same time [Sca02b].
- There is no separate testing or quality assurance personnel in FOSSD, but testing is done by the users of the software.
- FOSSD projects generally do not produce much documentation. Especially process documentations tend to be lacking or, when present, unreliable.
- FOSSD projects may fork, as described in Chapter 3.1.7, a possibility that affects the processes even when it does not realize.

Some of these differences may partially be result of the imbalance in the comparison of idealized process to observed reality. Others may be artifacts in the perception caused by the fact that FOSSD projects "have no need to fake a rational design process as is all too common in software engineering" [GSRP03]. However, it appears quite clear that there are significant differences between the software engineering and FOSSD processes.

3.2.4 Statistical differences

This subchapter lists some interesting ways on which the FOSSD development on average appears to differ from typical software engineering in a company setting. As discussed previously, the comparison is not entirely balanced, which may cause undeserved emphasis in some details. By themselves these observations do not show any major difference, but they may be symptoms of some more fundamental differences, like the ones discussed in Chapters 3.2.2 and 3.2.3.

FOSSD projects tend to release software very rapidly compared to software engineering models. When in software engineering a 60-90 day release cycle can be considered "very short" [Pre00], the Linux Kernel has been released multiple times on the same day. Naturally the role and scope of changes for a release are quite different when they are done in such high speed. The releases have quite a different role in the process: they serve as a way to deliver the software also to bug fixers and potential developers, not just to the end users.

Many FOSSD projects appear to grow in super-linear fashion, a trend that has been considered impossible in software engineering literature [Koc05]. There are also notions that FOSSD would thus deny Brooks' law that adding manpower to a late project will delay it more [Ray99a, BJ78]. Considering that there is no one to add man-power to a FOSSD project, it might be more adequate to state that the Brooks' law is simply not applicable to FOSSD, and thus can be seen neither valid nor invalid. This difference is discussed more in Chapter 3.1.6.

The adaptation of new tools appears to be really fast in FOSSD, whereas in software engineering the tool usage spreads a lot slower [Rob02].

Also there are some signs that the turnover rate of the core developers is smaller in FOSSD than typical employee turnover rate in traditional software engineering, though the numbers are not really comparable [RGB06]. So despite the fact that the developers are entirely volunteers, they are in a way more reliable than paid employees.

3.2.5 A side note on agile methods

The agile development methods that have been introduced in software engineering in the last decades share some common attributes with FOSSD, such as rapid releasing and loose and changing requirements. This has lead some researchers to speculate

whether FOSSD is an agile method by comparing FOSSD, "plan-driven development" and agile development, reaching conclusions that FOSSD can be placed between agile and traditional process [WA03].

While it is true that the agile processes are closer to FOSSD in many ways than the traditional processes are, the comparison is not really meaningful, due to the fundamental differences between FOSSD and software engineering in general, as discussed in Chapter 3.2.2. As a clarifying example, consider a typical decision situation where a company manager is deciding if a new product or customer project should be developed based on agile or more traditional processes. Selecting FOSSD is simply not an option, as it is not something a manager can just decide to put his team to use.

Even though FOSSD is not an agile method, it may be possible to improve agile methods by analyzing the FOSSD practices and including some of the process features or tools in agile methods, a course of research with some preliminary results already [AB05, ASS05a, ET03]. Features related e.g. to distributed development or continuous design may be borrowable, but some key aspects such as requirements management, task assignment or testing by users are likely not, as they are based directly on the nature of FOSSD as an emergent voluntary process with no actual customer.

4 Research on free/open source development

The amount of research on FOSSD has expanded rapidly in the recent years. Much of this research differs from typical software engineering research due to the differences in nature of the subjects, as discussed in Chapter 3.2.

As showed in Chapter 3.1, the team and the software are inseparable in the context of FOSSD and the research targeting one of them needs to take also the other into account. Some of the research methods and topics discussed in this chapter appear to be more related to the software than the development process, but due to this combined nature of FOSSD, covering also them is necessary.

The availability of huge quantities of process related data — practically the whole development process to the detail — in FOSSD gives new unique possibilities not previously available for software engineering. This is discussed further in Chapter 4.2.5.

Chapter 4.1 discusses the descriptive approach on research and its suitability to studying FOSSD. Chapter 4.2 introduces the methods used in such research. Chapter 4.3 discusses the use of software process metrics in FOSSD as a part of research and for validating research results.

4.1 Descriptive research on software engineering

As discussed in Chapter 3.2, software engineering, as all engineering disciplines, is a *prescriptive science*. The goal of prescriptive sciences is to define how things should be done. Observations and experiments are done of course, but their target is to see if the prescribed approach works and how it could be improved.

In particular the study of software development methods in software engineering is prescriptive, trying to define the best possible ways to develop software instead of observing how engineers appear to be doing it. This is understandable, considering that software engineering was founded to solve the "software crisis" by creating better ways to develop software.

The opposite of prescriptive science is *descriptive science*. Examples of descriptive science are, e.g., physics and sociology. In general all natural sciences and most social sciences are descriptive. Descriptive sciences aim at understanding phenomena with no particular intention to change them, though the results of the research can of course be used to change things.

In this dichotomy, meaningful research on FOSSD is necessarily descriptive. As discussed in Chapters 3.2.2 and 3.2.5, in FOSSD there is no central authority who could define how the software is to be developed in the project, and thus research that aims at giving advice to this imaginary figure cannot achieve much.

This change in the basic approach is certainly a challenge for scientists in the field of software engineering who wish to study FOSSD. Also, studying FOSSD is about studying how a group of people appears to function as a group and as individuals when they are trying to achieve a goal, and that is exactly what social sciences study. So another challenge is to understand and utilize methods and tools that come from disciplines with very different background and culture than engineering, such as anthropology, linguistics and economy.

Besides challenges, FOSSD opens also new possibilities for software engineering by forcing it to look for new approaches. The attitude of empirical observing analysis as well as many of the techniques used in FOSSD research could be usable also in research focusing on traditional closed source software development.

Gasser et al. define the following requirements for empirical studies of software artifacts, processes, communities and knowledge [GRS04]. It is worth noting that they did not limit the list to FOSSD. Such studies should:

1. reflect reality from actual experience rather than assumed, artificially constructed phenomena
2. give adequate coverage of naturally-occurring phenomena
3. examine representative levels of variance in key dimensions and phenomena
4. demonstrate adequate statistical significance
5. provide results that are comparable across projects
6. provide results that can be repeated, tested, evaluated, and extended by others.

4.2 Research methods

This chapter aims at providing an overview on the various methods that are used on studying FOSSD. The taxonomy used here is partially based on the one used by Walt Scacchi [Sca07]. The purpose is not to list all possible methods that can be utilized in FOSSD research, but to introduce those methods that have been used

as primary methods of at least a few scientific studies with promising results. Thus supporting methods such as social network analysis or discourse analysis do not have their own subchapters.

A large part of the research on FOSSD is done based on some research framework, a collection of methods with implicit and explicit assumptions about the subject as well as about the methods themselves. In this, FOSSD research differs somewhat from typical engineering or science discipline, where the basic assumptions are so universally accepted that discussing them is not necessary.

Research frameworks that have been used in FOSSD research with varying success include socio-technical interaction networks [Sca05], competence rallying [CS02], collaborative innovation networks [WMZ06], grounded theory [GSRP03] and information spaces [SDD+06], among others. The challenges related to frameworks and ways to overcome them are discussed in Chapters 4.2.7 and 5.2.

The taxonomy of research methods is the following:

- case studies are discussed in Chapter 4.2.1
- surveys and polls are discussed in Chapter 4.2.2
- virtual ethnography and other ethnographically informed methods are discussed in Chapter 4.2.3
- simulating software processes is discussed in Chapter 4.2.4
- data mining of FOSSD repositories is discussed in Chapter 4.2.5
- extracting processes from that data is discussed in Chapter 4.2.6
- combining the methods is discussed in Chapter 4.2.7.

4.2.1 Case studies and reflective analysis

A large part of FOSSD research is case studies in one or two projects. Such studies are often based on a "common sense" approach with no particular framework for analysis, or they try to follow practices of a typical case study in closed source development without considering if the approach is suitable.

As a consequence of this, the results found regarding software development depend largely on what kind of expectations and questions the researcher had in mind. Since

it has to be said that often the focus appears to be more in analyzing the software itself rather than the development processes, the limited descriptions of the process are understandable.

The descriptions of FOSSD written by practitioners, such as Raymond [Ray99a] and Pavlicer [Pav00], tend to have similar problems. They include valuable insight into the FOSSD processes, but the treatise tends to be unsystematic and uncritical thus providing quite little scientifically useful results. Also the models described in the advocate literature do not always survive in scientific analysis [Sca02b].

At best, a case study is a good analysis of a particular example, a starting point for further research providing hypothesis and research questions to be confirmed. As an example, the study of Thomas Østrelie is an excellent analysis of one particular question, the relationship of perceived hierarchy and control via one example [Øst04]. Another such example is the study of Jensen and Scacchi [JS05b]. Naturally, all such studies do always require more research with different approaches before the results can be considered reliable and universal.

4.2.2 Surveys

Survey methods are used especially for analyzing why people participate in FOSSD projects or communities around them, as well as for other statistical information on the participants [SC06, BR04, GGKR]. For this kind of data, surveys are a good method, as they base the results on significantly larger samples than a case study on a single project and thus provide more reliable results.

However, there are significant limitations on what can be studied with a survey. Surveys are best suited for studying frequencies and variations of a variable or correlations of variables. For complicated interdependent and evolving systems such as FOSSD project, the applicability of a survey research method is quite limited.

Surveys are also based on the respondents' knowledge and interpretation of the questions and thus can only address questions the respondents can be assumed to have an informed opinion about. In practice, this limits their applicability to data about the respondents themselves and to their motivations and opinions, but excludes questions on how the project operates. For instance, a survey on how software architecture is designed in FOSSD projects would likely give no meaningful results.

The highly unequal nature of FOSSD sets additional challenges to survey research.

Who should be included in the population on which the survey is done, considering that there are orders of magnitude more participants with low involvement, but the highly involved participants produce orders of magnitude more results such as source code and newsgroup messages? The answer to this question depends on the specific research question at hand, as well as on prior understanding on how FOSSD projects generally work.

Therefore, efficient survey research actually requires a highly developed theoretical model of the phenomenon at hand in order to produce more than just the most obvious results. Also it is quite a work-intensive method, which may be the main limiting factor on the amount of FOSSD surveys done.

4.2.3 Virtual ethnography

Ethnography is a holistic research method based on combining quantitative and qualitative aspects in "hands on" field work: face-to-face interaction between the researcher and the informants, i.e. the subjects of the research. It is based on the assumption that the aspects of the studied social phenomenon or system cannot be fully understood separately but the system needs to be analyzed as a whole. Ethnography is a commonly used research method in social sciences, especially in anthropology.

Virtual ethnography is adaptation of ethnography to the context of Internet [Hin00]. Instead of face-to-face interaction, the researcher participates in chat sessions, does email interviews and analyzes the online texts that are part of the phenomenon. The basic approach of participating in and observing the life of the informants in its normal setting is the same as in classic ethnography, only the setting is changed to be the Internet.

In the context of FOSSD, virtual ethnography has been used, for example, in studying the organizational culture of free software developers and how that culture affects their work [ES05], how shared mental models appear and affect the efficiency of FOSSD teams [CS06], as well as how the ethics of free software are constructed on developers' practices [CH05a]. Each of these topics would have been very difficult to study solely with quantitative methods.

As a qualitative method, virtual ethnography has the typical shortcomings and challenges shared by all qualitative research methods. It is relatively work intensive and requires that the researcher has a good understanding of the method and how

to use it. It is easy for the researcher to find the results he wants to find and it is difficult to reproduce the results independently. Also the research is typically limited to only one group, and it is not obvious to what extent the results can be generalized.

Virtual ethnography is best suited for "mapping the terrain", that is, creating a hypothesis on the dynamics of a social phenomenon such as FOSSD and thus creating a frame on which further studies can be based on. On a discipline based on science and engineering such as software engineering, an ethnographic analysis alone cannot be considered to provide validated results, but statistical or empirical research is required in addition.

4.2.4 Process simulations

Software process simulations have been used in traditional software engineering to test new process models before taking them into use, especially to anticipate possible bottlenecks. While simulating FOSSD differs from simulating traditional software engineering in both its goals and its methods, the basic idea remains the same:

1. create or obtain a detailed model of the process
2. run simulations based on the model
3. compare simulation results to reality
4. adapt the model parameters
5. iterate.

In FOSSD simulation the objective is not to plan process changes. Instead, it is typically used for understanding how the process works or for anticipating future success of a particular project [WHC05].

The process models used for software engineering simulation are not suitable for FOSSD due to the major differences in the development approaches. For example, a typical software engineering simulation model uses the amount of developers and tasks as input variables, whereas in FOSSD the amount of developers is unknown and the tasks to be done are not defined centrally but emerge from the community [ASS+05b]. Instead of using models from software engineering simulations, the FOSSD models are constructed by the researchers doing the simulation either based on some prior studies or just on their own understanding on how FOSSD works.

The validity of the model in use is the key question for the reliability of FOSSD simulations and as the understanding on the phenomenon is still partial, there are no obvious solutions. The situation is not helped by the fact that it is very difficult to validate the model used for simulation. If the results a simulation provides do not match with reality, parameters can always be adapted to make the model fit better, as all the models contain various ad hoc parameters. Also the high variance of many key variables of FOSSD, discussed in Chapter 3.1.8, reduces the reliability of the estimates given by models [ASS+05b].

It appears that in order to create FOSSD simulations that are more than curiosity or testing of an approach, the models used in the simulations would need to be more strongly based on research results about how FOSSD works. This in turn would require more converged and confirmed understanding on the whole phenomenon than what is available today.

4.2.5 Data mining on FOSS repositories

Case studies, surveys, simulations and ethnographical research have been used for a long time in software engineering and social sciences, so there is lot of experience and tradition in utilizing them. In studying FOSSD there are also unique new methods available, provided by open and efficient access to the internal workings of projects. Due to the nature of FOSSD, a large majority — sometimes virtually all — of communication and actions between participants are available to researchers.

While similar data on classic software engineering projects has in some cases been available to a researcher, it is no match due to the sheer quantity of the FOSSD data available today. Classical software engineering is also typically done by co-located teams and it relies significantly on verbal communication that leaves no trace, whereas in FOSSD typically a large percentage of informal communication is also available in textual form. Besides, the fact that the same data is available to all researchers is a significant benefit for scientific research on the topic.

Due to the accessibility of the source code and project communications, various automatic and semi-automatic analysis techniques can be utilized, once the data is gathered. Typically this takes the form of either web spidering or acquisition of copies of databases.

The sources of data to be analyzed vary according to the objectives of the research. Typical sources for data mining are the following:

- The most popular source are project portals such as SourceForge [7] and Savannah [8]. They provide various statistics over the projects and typically also access to a source repository and an issue tracking system.

- The source repository can be used to analyze all changes done to the code. In some projects the commit access is limited to a few core developers and code changes from other participants are committed by them, which reduces the usability of the data, as the original contributor information is not available.

- Issue tracking systems show traces not only over defect management, but also feature requests and design activity, as discussed in Chapter 3.1.4 [RG03].

- Mailing lists and discussion forums provide access to design discussions as well and give opportunity for analyzing social structures of the project [BDBS05]. Most projects keep online archives of their mailing lists.

- News letters and release repositories can be used for tracing when releases have been made, and what is included in them.

- Other available data include project documentation such as FAQ and Changelog documents, online logs of IRC and chat discussions, developers' blogs and social networking sites such as Ohloh [9].

The availability of the data does not mean it can be easily utilized. In traditional empirical software engineering acquiring data has always been a challenge, but the data itself is of good quality, consistent, rich and optimized for research use. In FOSSD research the situation is quite the opposite: there is plenty of data available, but it is not intended for research purposes and requires a lot of clearing and standardization before it can be automatically analyzed [Con06].

The unstructured nature of the data does not only cause additional work for the researchers, but also introduces the risk of misinterpretation. If the researchers are not familiar with the practices of FOSSD, they may not realize the true meaning of some artifacts, e.g. that issue repositories actually contain design discussion [HC04]. Also there is a risk of fitting the theory to the data that happens to be available, for example using the number of downloads from a project portal to measure project popularity or success, as the number of downloads is easy to measure. However,

[7] http://www.sourceforge.net
[8] http://savannah.gnu.org
[9] http://www.ohloh.net

popular FOSSD applications are typically not downloaded from project portals but from distributions [HC04].

An additional challenge to statistical analysis is that universal sets such as the whole population of FOSSD projects or the population of developers of one project are not known, and that the distributions of most interesting variables such as developers per project or lines of code per developer are highly skewed [HCC05]. See Chapter 3.1.8 for more discussion on the distributions of such variables.

One promising approach for solving some of the issues related to utilizing the FOSSD data is to create open data repositories for cleared and standardized research data. The most well-known repository is FLOSSmole[10], which provides a large standardized collection of data on various project gathered from FOSSD portals as well as scripts for gathering and analyzing the data, under a FOSS license of course [HCC06].

Collective data repositories can be seen as a step towards common and open research infrastructure on empirical FOSSD research — a collaborative approach that would naturally fit the subject. However, the benefit they currently provide for new research appears to be limited, as they lack almost all data on communication inside and across projects, the very data that can be expected to be most fruitful for process discovery. As a result of this, most of the studies utilizing FLOSSmole are actually carried out by the same group that created the tool.

The limited usability of the data repositories can be seen as result of the fact that gathering data is not just preparation work that has to be done in order to do science, but an integral part of research guided by the theoretical framework of the research. To simplify a little, a researcher gathers the data he expects to support or reject his hypothesis and does not look for aspects he does not expect to be relevant. In situations where there is no common theoretical framework telling what aspects of FOSSD are relevant, the data of one researcher may be of little value to another one.

4.2.6 Extracting processes from data

Extracting development processes from observed data is one of the Holy Grails of empirical software engineering — an approach with significant difficulties and limited results so far [JS05a]. While FOSSD research cannot claim to have avoided the challenges, the public availability of large amounts of project data provides a

[10] http://ossmole.sourceforge.net

promising starting point for process extraction.

Creating a process state machine or similar formalization for FOSSD from scratch, just based on raw data, is still not a realistic goal. First, there is no basis to assume that a state machine, even a probabilistic one, is a suitable formalism for FOSSD development, and second, the states of such machine would inevitably be affected by how the researcher assumes the development to work, as they are based on the taxonomy of concepts used in the analysis [JS05a].

Instead of trying to construct a model of FOSSD just based on data, more fruitful results can be expected if the researchers use prior knowledge on how software development in general or FOSSD in particular. Also, aiming at constructing partial models of some parts or aspects of FOSSD may lead to better results than trying to gasp the whole phenomenon to a single model [JS05a].

For limited processes where the states are obvious, such as issue states in an issue repository, a probabilistic finite state automata can be a useful formalism [RG03]. Even then it should be considered whether the states actually properly present the relevant aspects of the phenomenon. For example, the issues in FOSS issue repositories tend to stay in state "new" until they are solved [Kop06]. Thus the division to the issues that are recognized but not fixed and to the ones that are not understood yet is not recognized by an automata based on the formal issue states.

Another partial step is constructing a social network of the developers based on how they refer to each other in mailing lists or an issue tracking system [CWLH06]. A social network is a weighted network where the nodes are project members and the edges represent their connections to each other. Social network analysis can also lead to mis-interpretations, as e.g. a lengthy discussion about one particular issue with a core developer could make an issue reporter look like a core developer as well [CWLH06]. Besides a weighted network, this kind of analysis may provide also hints about roles different developers play in processes based on e.g. who initiates or ends discussion threads [BDBS05].

As many of the interesting aspects of the process are based on free text messages among members instead of formal state changes on any system, text analysis or computer based discourse analysis can be applied in the analysis. The reliability of such analysis is of course always subject to doubt. Gasser and Ripoche report that they have reached 80% accuracy in recognizing aspects such as uncertainty and conflicts in issue repository data [RG03].

In general, automated applications based on methods of social sciences, such as discourse analysis or rich picture, seem to be suitable tools for FOSSD process extraction. In utilizing them it is good to recall that each tool is based on a particular theoretical framework with its own drawbacks and limitations. Thus a researcher needs at least cursory understanding of the methodological discussions and disputes in social sciences when interpreting the results.

Ensuring the accuracy of an extracted process model is a very challenging problem with no obvious solution. Validating a model of some limited part instead of a full FOSSD project is likely to be easier. Some partial solutions are also discussed in Chapter 4.3.1.

4.2.7 Combining research methods

A natural approach to overcome the shortcomings of the methods presented above is to combine them. Examples of such work are those of Sack et al. [SDD+06], Scacchi et al. [SJNE06] and Gasser et al. [GSRP03].

Each of these research groups is — each on its own way — creating understanding on a FOSSD project by ethnographic methods and discourse analysis and then utilizing this understanding in data mining, process extraction and formal process modeling. This multi-method approach can be seen as a way to combine the result-yielding but worksome ethnographic methods with automatic or semi-automatic data-mining providing access to larger data sets and thus potentially more general and reliable results.

The approach is relatively new, and each of these research approaches has been conducted on a single project. No research that would have sought to combine these methods with automated comparison between projects has been made, and neither have the results of these particular studies been verified on other projects. Thus it is unknown if these approaches can be scaled up to be applied to a larger set of projects.

Combining research methods is no silver bullet to FOSSD research. Besides the scaling issue, these methods still suffer of some limitations of the ethnographic and process extraction methods as discussed in Chapters 4.2.3 and 4.2.6. In particular, the research approach is always based on a framework that may imprint some of its own limitations to the research. E.g. Sack et al. are dividing projects to three "information spaces" [SDD+06], while Scacchi et al. conceptualize them as "socio-

technical interaction networks" [Sca05].

While both of these approaches seem to provide promising results, there is no obvious way to validate whether one of them is in some sense better or worse with respect to analyzing FOSSD than the other. Some partial answers to this challenge are discussed in Chapter 5.2.

4.3 Process metrics for FOSSD

Software process metrics are a quantitative, systematic way to gain insight into the functioning of a software process. The main purpose of process metrics is to provide software engineering teams better tools than just subjective evaluation for improving their work. This is basically achieved via systematic measurement of some identified metrics and by comparing those to the past average in the same organization [Pre00].

In the context of metrics, some vocabulary needs to be defined:

- *Attribute* is the quantity or aspect a metric is trying to measure. The attributes may be clear objective quantities, but also vague concepts such as success or activity.
- *Measure* is an observation of a particular attribute against a standard, for example the lines of code of a module with a specified counting mechanism. Similarly, *measurement* is the act of obtaining a measure [Rag95].
- *Metric* is a comparison of two or more measures. It combines measurements, sets them in time and context and gives them interpretation, e.g. found errors per thousand lines of code [Rag95].
- *Indicator* is a metric or combination of metrics that provides insight into a phenomenon, in this case to the software process [Rag95]. Creating indicators is always based on a theoretical framework and assumptions on the studied phenomenon, e.g. what is a normal amount of bugs per thousand lines of code, which percentage of the bugs have likely been found with the used methods, etc.
- *Screening variable* is a variable used to pre-select only a part of available data for statistical analysis, e.g. select from SourceForge dataset the FOSSD project with seven or more developers. Screening the data with a variable causes always a risk of changing also dependant variables in the data.

- *Proxy metric*, or just a proxy, is a metric that is used to estimate something else than what it actually describes, usually because it is infeasible to measure the target attribute directly. For example the amount of downloads from the web page of a project can be used as a proxy for the popularity of the project. Proxy metrics are always based on the assumption that the distribution of the measured attribute actually represents the distribution of the target attribute reliably.

Utilizing metrics in FOSSD research differs from their usage in software engineering. In software engineering, metrics are typically used tightly integrated to the development process, by the development team or the managers and with the aim of improving the process. On the FOSSD side, on the other hand, the metrics are used by an external actor, the researcher, and with the aim of understanding or classifying the projects. Also, in FOSSD there is no generally accepted "known good process" a particular project could be compared to.

Due to the differences in the setting, many typical process metrics used in software engineering are not suitable for FOSSD. The same is true for metrics for project success. In a typical software engineering project, the usage of the software is relatively open for measurement, whereas the access to the internals of the development might be more restricted. In FOSSD the situation is vice versa: the development work is available to a very large extent, but it is hard to get information on how much and how the software is used [CHA06]. Some of the FOSSD metrics discussed here could be usable also for studying software development in typical company setting, but that topic is outside the scope of this thesis.

As discussed in the beginning of Chapter 4, the FOSSD team and process are interwoven to the software they develop, and studying one without considering the other gives an incomplete and possibly twisted picture of the phenomenon. Thus some of the metrics discussed here are closer to traditional software metrics than process metrics.

When utilizing metrics, especially in research, it is important to pay attention to their validity. The relationship between the metric and the target attribute it is assumed to represent is never trivial. Thus a good metric should have both a theoretical explanation on why the measured variables would indicate the intended target attributes, as well as empirical tests to prove that the metric indeed works. Besides this, a good metric is also relatively easy to measure and objective, i.e. independent from the researchers' subjective estimations.

The discussion on individual metrics in this chapter is brief, but the theoretical basis for non-obvious metrics are discussed in the context of the model of FOSSD introduced in Chapter 3. The validity of the metrics is not tested here, but references to validation done by others are given when available.

Chapter 4.3.1 discusses the utilization and purpose of FOSSD metrics. Chapters 4.3.2 to 4.3.5 introduce various metrics, both ones introduced in literature as well as new ones. Chapter 4.3.6 discusses the construction of indicators by combining metrics. Table 4.1 summarizes all the metrics introduced in this chapter.

4.3.1 Usage of process metrics

As discussed above, the possible usage of process metrics in FOSSD differs somewhat from the typical software engineering setting. This subchapter lists possible use scenarios for metrics and motivations for gathering them.

The free and open source communities and especially the community leaders could use metrics to improve the way the projects function, in a way that is analogous to how metrics are used in software engineering. This usage, while listed in the motivation chapters in many studies, has at least so far been mostly theoretical. The FOSSD communities and leaders do not seem to see a need for improving their projects with metrics any more than they see the need for process descriptions or formal documentation.

Software companies that utilize free and open source software would certainly have use for FOSSD metrics. The most obvious usage is to predict the likelihood that a particular project is going to be properly supported in the future. Currently such predictions are done mostly based on subjective evaluation, but scientifically verified indicators for the continuation of support would help companies in the prediction. Such indicators would typically be based on success metrics introduced in Chapter 4.3.4.

Another usage for FOSSD metrics in the industry is analyzing if a company working actively in a project is disturbing the dynamics of the project. This could be done by comparing various metrics of the project to the “known good values” collected from projects where there is no active company participation. Creating indicators for this kind of usage would require significant advancements and consolidation in FOSSD research, as currently there is no good baseline to compare the projects to.

An EU/NSF workshop on priorities for FOSSD research identified the need for

learning "from open source modes of organization and production that could perhaps be applied to other areas". But to be able to learn from projects that are working well, a definition for "working well" is first needed [CHA06]. This is the third way industry could benefit from FOSSD metrics.

For the scientific community the value of FOSSD metrics might be even higher. As discussed in Chapter 4.2, many of the research methods utilized in FOSSD research would benefit from additional validation of the results. In this metrics, due to their quantitative nature, may come to help.

If a study based on e.g. virtual ethnography or simulations provides interesting results, a limited validation could be done by constructing statistical hypotheses based on the findings and then utilizing metrics on testing them. For example, if a theory implies that projects with a more centralized communication are more successful, a statistical test for this claim could be constructed using metrics for the level of project centralization and success.

4.3.2 Size-oriented metrics

Project size is one of the most obvious things to measure, but while it might sound trivial, it is actually quite a complex thing already in itself. Does it mean the size of the software, or perhaps the size of the team? If the former, should it be only actual source code, or also documentation and examples; if the latter, who should actually be counted, just the core developers or everyone who has submitted bugs to the software?

An additional challenge is how to take time into consideration. Source code can be measured at a particular moment, but when measuring the amount of people involved, some time scale need to be taken into use, e.g. all users who submitted issues during the last month could be counted as current project members. Also social networks show quite a misleading picture if the time is not taken into account.

Since the FOSSD projects consist of both the people and the software, metrics for both are given here. The metrics presented below are targeted to measure the static size of a project, i.e. the size at a given moment.

Source lines of code (SLOC) is maybe the oldest of all software metrics. The normal version of SLOC is simply the amount of lines in any programming language. If the purpose is to estimate the total effort put to the project, also documentation and other non-source products of the team should be counted in. The accuracy of the

SLOC count as a metric to compare projects is in any case limited, due to different verbosity of programming languages and different programming styles.

Another estimate for project size is the amount of software modules. If the result that SLOC/module stabilizes in FOSSD projects [CLM03] can be confirmed, this metric should in normal cases correlate with SLOC after the initial phase. Also combined metrics for average SLOC/module and developers/module with the respective distributions can be constructed, though it is not obvious what such metrics would indicate.

The amount of developers is more complex to measure than the software size. The first question is, who should be counted. Considering the onion model introduced in Chapter 3.1.5, the possible groups to measure are core developers, co-developers, active users and all users, each group containing the previous ones. As discussed, the amount of all users is generally hard if not impossible to measure, and is thus excluded here. For the other groups some possible metrics are introduced below.

In measuring the core group size one obstacle is that the actual meaning of the core group is not clear or obvious. A typical definition is "the developers who provide most code or input", but this leaves significant room for interpretation. Metrics constructed for measuring the core group include:

1. The list of developers the project announces in its web site [CWL+05]. This metric seems to identify significantly larger core groups than other metrics described here [CWLH06].

2. The most active 20% of the developers [RGB06]. This is not a very reliable metric considering that the amount of contributions per developer appears to be distributed based on power law [MFT02], so a fixed percentage is likely to estimate unreasonably large core groups in big projects. Also, this metric depends on a metric for co-developer team size which is another source of errors, and would make a comparison of the two attributes meaningless.

3. The participants who contribute one third of the total input (postings in issue tracking) [CWLH06]. This can be seen as an improved version of the previous method, but can be assumed to be more reliable.

4. Social network analysis (SNA), see Chapter 4.2.6, on reply-to structures in an issue tracking system [CWLH06]. This analysis appears to include some false positives in situations where a peripheral member is involved in a lengthy

discussion with some core member. Improving the method to be more reliable could be feasible e.g. by creating the SNA based on mailing list data where peripheral members are less likely to participate actively.

Except the announced developer list, each of these metrics would need to be calculated over some specified time interval, e.g. 6 months, as the core group may have changed in the project history. If the purpose is to study changes in the core group, another option is to divide the project history to a fixed amount of phases regardless of the resulting phase length [RGB06]. It is not obvious whether this kind of approach would cause systematic deviation in the metric.

Measures 2 and 3 are based on the "activity" of the developers, which is not a straightforward concept. The activity visible in either code repository, mailing list or issue tracker can be used as a proxy for it. Code repository is the most intuitive approach, but many projects use a system where most people deliver their changes as patches to a release manager who then commits them, thus distorting any metric that relies on the code repository. Considering how the requirements analysis and the design of the software is done on mailing lists and issue trackers, also input to these can be used as a proxy for activity. Selection of the attribute to measure is also a question of what particular activity is of interest, though the correlations between the three activity measures can be assumed to be quite high, so they are also possible proxies for each other.

The most natural metric for co-developer team size would be the amount of people who commit to the source repository. However, due to the release manager practice mentioned above, this cannot be measured directly in many projects. Assuming that there is a relatively high correlation between the amount of code commits and active participants on mailing list and/or issue tracker, a proxy metric for co-developer team size can be created based on those. The validity of the metric can be verified using projects where the code commit credits are known to be correct. Also an easy, though most likely unreliable metric would be to use the number of the mail list subscribers.

The community size, or the amount of active users, which are roughly speaking the same thing, can be measured using the amount of individual identities available in the issue tracking system. As the definition of active user is that he submits error reports, this metric is in principle simple.

However, there are also problems with this metric, as a significant amount of the issues are filled by anonymous users [CAHM04]. Possible strategies towards the

anonymous issue reporters are to count them as one user, to count each message as a separate user, to count the anonymous reporters in the same thread as the same user or just to ignore the anonymous users [HC04]. The first and the second strategy give the lower and the upper limit for the amount of active users retrospectively. If an assumption is made that regular reporters register almost always, the upper limit can be used as a metric for community size as it is, but this assumption would certainly require verification.

Time can be taken into account also in the community size metrics to avoid counting all the active users a project ever had. Though if the project grows exponentially, as discussed in Chapter 3.1.6, the error caused by past users could be relatively small.

The amount of correlations between the metrics for the code size, the core group size, the co-developer team size and the community size are an interesting question which has not been studied much. A viable hypothesis would be to assume relatively high correlations, while the cases where the correlations do not hold may provide the most interesting insight to FOSSD.

4.3.3 Activity-oriented metrics

Besides size, the other obvious attribute to measure is project activity or vitality. While there is likely a large correlation between size and activity of a project, they are still clearly different attributes. While size focuses on the project's present state and perhaps looks to the past, activity is more likely to tell also about the future of the project: the increase in the size tomorrow is based on the activity today.

The easiest metric for activity is the release cycle. While far from reliable, it gives a rough view on the activity in the project.

The utilization of typical FOSSD tools such as a mailing list, and an issue repository is a clear indication of activity in itself. Also more abstract tools such as documented coding style could be added to the list. To improve the metric, the projects that are active according to other metrics should be analyzed to establish a weighted list of tools that correlate with activity. For more discussion on FOSSD tools and their relevance to the process, see Chapter 3.1.2.

The number of co-developers or most of the other size-oriented metrics can be used as proxy for activity, due to the probably high correlation. On the other hand, size metrics would fail to catch the activity measure precisely in the cases where it would be most interesting: on projects that are changing, i.e. small but very active

projects or large but inactive ones. Thus size-oriented metrics alone are in most cases bad proxies for activity.

New lines of code per time unit is another obvious activity metric. It was used in the studies showing that many FOSSD projects grow exponentially [Koc05]. Depending on the goal, it may make more sense to analyze the percentual change instead of just the number of new lines.

The frequency of new developers joining the community and the increase in the activity of old peripheral members is clearly an indicator of the project's activity and its capability to sustain its functioning. Metrics for this activity are maybe the likeliest to be useful for FOSSD communities. The frequency of joining new members can be estimated by metrics such as new reporters in the issue system, new members subscribing the mailing list or new members posting to the mailing list. Each of these metrics estimates a slightly different attribute. Considering the onion model and how developers climb towards the inner circles, as discussed in Chapter 3.1.5, constructing a series of metrics each following a smaller but more contributing part of the team than the previous one would provide more sophisticated insight into the activity of a project. More complicated metrics, such as the number developers who post over five mails a month, could be used as a part of such a model.

An activity metric derived from the previous one would be the derivate of the joining function. As the amount of new developers joining the community is derivate of the size, this function would be the second derivate of the size, giving perhaps the earliest warnings if a community is starting to loose its popularity.

Besides the joining of new developers, also keeping the old ones is a sign of healthy and active project. Developer half-time has been used as a metric when studying Debian [RGBM05]. Half-time is measured by following how long it takes until half of the developers who were active at the starting point have ceased to be active. Due to its nature as a Linux distribution, Debian has a large amount of volunteers in specific named positions. Typical FOSSD projects are lacking this kind of obvious membership criteria, and the developer half-time would need to be a combination to some of the team size metrics. Using half-time as metric includes the implicit assumption that the studied attribute decreases exponentially, which would need to be verified before relying on such a metric. As an additional note, it is not obvious that higher developer half-time is always better, so the connection of developer half-time to e.g. project success measures should be studied.

Besides developer half-time, also SLOC half-time could reveal interesting details of

the project's dynamics. Considering that the projects are in constant state of redesign and also the requirements change, it is reasonable to assume that even when the project grows exponentially, some part of the code gets deleted for every new release. SLOC half time tells how long it takes until half of the lines of code of some particular release have been removed. Source repository tools such as CVS and SVN provide a trace of every line of code that has been removed, so this metric is possible to obtain.

It is possible that the amount of deletions practically never reaches half of the lines that existed at some point. Also the amount of deletions may well not be exponentially diminishing, but follow some different function. Thus a survival function for a line of code would probably be a better metric than half-time.

A survival function is a function over a random variable of the form

$$S(t) = P(T > t) = 1 - P(t)$$

where S is the survival function, P is the probability distribution of the random variable T and t is the given time. In other words, survival function tells the likelihood that the system, e.g. line of code, represented by T survives until time t.

The issue repository tells also a lot about project activity. The time an issue stays open is an obvious metric for activity [CHA06]. Due to the fact that some issues are still open at the time of analysis, the metric cannot be just average, but a survival function is better suited for the purpose. In a study over a few projects, the function turned out to be exponentially diminishing [CHA06]. If this is generally true, the metric could also be formulated as issue half-time.

As a large amount of the issues are judged duplicates, as much as 33% in one study [San05], the metric could be improved by excluding the duplicates. Alternatively, only the resolving time of duplicates could be studied. Considering the nature of the issue repository as a space for design discussions, as discussed in Chapter 3.1.4, the recognition time of duplicates could tell about the efficiency of the project in reaching conclusion about the issues, whereas the resolving time of non-duplicates could be more related to the efficiency in solving the found issues. Both of these derived metrics would require empirical studies to confirm if they indeed can be used as proxies for the attributes described above.

Also more trivial metrics based on the issue repository, such as the amount of new issues per time unit, the amount of reports in general and the amount of open reports could have some usability as metrics for activity. There appears to be a positive

correlation between the amount of issues and faster fixing of issues [CAHM04].

The issues are also connected to each other, forming bug report networks (BRN), as discussed in Chapter 3.1.4. In a study, 65% of the issues in the Apache project were found to be connected [San05], including the duplicates discussed above. The percentage of issues belonging to a BRN as well as the distribution of the BRN sizes tell about the amount of information management done in the issue management system, and thus could be used as a proxy for one aspect of project activity.

Finally, one relatively easy and popular but suspectable measure for activity is simply to use the SourceForge activity rank [Sou, CAHM04]. The activity rank is constructed based on multiple attributes in the SourceForge databases, including downloads, web page hits, source commits and releases and issues submitted during the last seven days. Due to the complex formula with no theoretical justification and the fact that the formula has been changed a few times in the past, the usability of the activity rank as a metric for activity is questionable. However, it provides a useful comparison for any activity metrics.

4.3.4 Project success oriented metrics

Project success is perhaps the most interesting attribute of FOSSD projects. When a minority of the projects is extremely successful, but most fail, it is important to ask what makes the difference. In order to analyze the causes of success, the mandatory precondition is to recognize the successful projects.

The obvious problem is the definition of "success" in the context of FOSSD. DeLone and McLean have defined the de facto standard model of information system success as constituting of six inter-related aspects: system quality, information quality, system use, user satisfaction, individual impact and organizational impact [DM03]. While the applicability of their framework to FOSSD is not obvious, it have been successfully used [CHA06]. The particular questions in measuring FOSSD success is whether success means wide spread usage of the software, or perhaps smooth running and steady growth of the development project. In the lines of the DeLone and McLean model, the best approach is probably to measure both.

A typical success attribute is popularity, for which various metrics have been introduced. Even though popularity is a clearer concept than success, and arguably easier to measure, it is not without problems. Comparing popularity of two projects with totally different amounts of potential users can give a misleading picture, as

even a mildly popular instant messaging client has more users than a deep space astronomy tool used by almost all practitioners. As the size of the potential user space is generally infeasible to measure, also metrics for popularity that do not depend on it have been developed, and in particular relative growth can be used as an alternative in most popularity metrics.

The easiest metric for popularity is the amount of downloads from a project portal. While easy to obtain, it may introduce significant errors, as most users do not download the software directly, but install it from Linux distributions [CAH03]. Downloading is more common in projects where new beta versions come out regularly and are used by large amounts of active users. Thus amount of downloads from a project portal might actually measure more the activity of the project than its popularity.

As most users obtain FOSS from distributions, the inclusion of a project in the Linux distributions is a sign of its popularity. Naturally this applies only for software that is intended to be run on Linux. The biggest distribution, Debian, contains also a tool called *Debian popularity contest (Popcon)*[11] that gathers installation and usage statistics on end users' computers and constructs a centralized report based on them. Currently there are approximately 80 000 computers reporting to the system. As a metric for popularity, Popcon obviously works just for packages that are included in Debian, which currently contains 18733 packages, likely including all even modestly successful FOSSD projects that run on Linux. The people running Popcon in their machine are a self-selected minority, and likely a quite biased sample of the full population of FOSS users. Thus Popcon statistics are not a reliable metric for popularity, but perhaps still the best available metric for the usage of a piece of FOSS.

A more novel way to measure project popularity is to count the appearance of the project in web search engines. As the FOSSD projects exist largely in the Internet, and also large part of the user population appears online, basing success metrics on search engines is a justified approach. It can also be compared to the way the success is understood for academic publications, i.e. by references. Apparently the amount of results by a search with the project name is prone to errors, but counting links to the project home page is quite reliable [Wei05].

As discussed above, success does not mean only popularity, but there are also other aspects. When considering successful running of a FOSSD project, the activity-

[11] http://popcon.debian.org/

oriented metrics discussed in Chapter 4.3.3 are natural proxies for this aspect of success. In particular, the expansion of the active user and co-developer spheres would be useful metrics.

Considering that many developers participate in a FOSSD project voluntarily, in order to fulfill some goals of their own, their satisfaction with the software and the project can be seen as a form of success — after all, they participate in the project just for themselves [CHA06]. The obvious measure for estimating this is conducting a survey, which would be infeasible to use on a regular basis, but could be used for verifying reliability of some proxies. Suitable candidates for proxies for developer satisfaction would be the project activity metrics, since if people participate actively in a project, they are most likely satisfied with it.

4.3.5 Other metrics

Besides size, activity and success, also many other attributes of a FOSSD team can be estimated with metrics. The selected attributes depend of course on the goal of the research. Some attributes that either have appeared in the literature or can be derived from research results are introduced here.

The age of the project is often an important dependant variable in statistical analysis. While the exact age of the project — if understood as the time since the project initiator started to work on it — is often infeasible to estimate, there are multiple suitable proxies. A typical proxy in research based on SourceForge is to use the date the project was set up in SourceForge. Other possible options include the first entry in revision control, the first mailing list mail and generally the first item on any data set available. All of these share the same weakness, namely that the project may not have used that particular tool for its whole lifetime.

Another, slightly different approach is to base the age on the date of the first release, usually numbered 0.01. As FOSSD projects tend to release the first release as early as possible and there is a community norm to label the first very incomplete release as 0.01, this metric could be more reliable than the other proxies for age. Also, considering the nature of FOSS as software with the source code available, it can be argued that the project starts as FOSSD project the moment the first release is made. In this sense, this metric actually estimates the target attribute and is not a proxy.

The reputation of the core developers in FOSS social networking site Advogato[12] can be used as a proxy for how experienced or well connected the project leaders are [CHA06]. Advogato has a "trust" system where every developer can mark his trust towards other developers. The resulting network shares the power law dynamics with to many other attributes of FOSSD.

The process maturity of a FOSSD project can also be measured with various metrics, though this requires some adaptation on the concept of process maturity [Mic05]. Possible metrics can be based on the utilization and usage patterns of typical FOSSD tools such as version control, mailing list and issue repository, as well as on whether release testing is performed. As the concept of process maturity is not well established in FOSSD, it is worth paying attention to whether the intention is to analyze maturity as a dependant variable and success as a depending variable, or vice versa. When studying success based on maturity, the metrics for maturity would first need to be empirically verified with tests that are not based on metrics for success, so as to avoid circular reasoning.

The degree of the social network of the project constructed from, e.g., a mailing list or an issue tracker can be used as a metric for the "tightness" of the project team. Alternatively, the average or median degree of the nodes could be used, as well as some other metrics utilized in social network analysis. Similarly connections to other projects can be estimated from a network based on e.g. developer-project relationships in SourceForge or Advogato [WMZ06, WHC05].

The modularity of the software, while being mostly a software metric, is a valid attribute in FOSSD analysis, as the code structure of the project correlates with its social structure [NR05]. Possible metrics for modularity include average SLOC/module, as discussed in Chapter 4.3.2, and the utilization of modules in general, i.e. whether there are 1, 2 or more sourced directories [CLM03]. Also analyzing who modifies what parts of the software could give valuable information, as there is some evidence that in successful FOSSD projects the developers do not respect module boundaries when doing changes [NR05].

Also metrics for individual issues in the issue repository can be made, estimating e.g. the level of uncertainty related to a particular issue in order to anticipate the survival function of the issue [San05]. The metrics used in this can be based on automatic text analysis, which is also an interesting potential tool for metrics construction in general.

[12]http://www.advogato.org

4.3.6 Constructing indicators

In order to efficiently use the metrics described above, they would need to — besides validating and refining — be combined to indicators. As basically all metrics have various sources of error, and there are often multiple possible metrics for more or less the same attribute, the best approach would be to combine them. For instance various different size metrics can be combined to one more reliable indicator for the project size.

In order to create such indicators the metrics would need to be normalized, which would require significant further research on the metrics to collect necessary comparison data for normalization. Also considering that many of the attributes appear to be exponentially distributed, logarithmic transformation would be needed for many metrics.

Many of the sources of error appear to render the value entirely useless instead of just modest error. For instance a five years old project may appear to be founded a few months ago. Thus a good approach for error correction is to use multiple metrics for same attribute and use the median of the normalized values instead of the average. This does not entirely solve the issue, as some erroneous values may also distort the normalization.

Table 4.1: Summary of the metrics discussed in chapter 4.3.

Metric	**Measured attribute or proxy for**	**Category**	**Automatable**	**Introduced in**	**Related research**
Source lines of code (SLOC)	Code size	Size	Yes	Trad.	
Number of modules	Code size	Size	Partial	Trad.	
Announced developer list	Core group size	Size	Yes	[CWL+05]	[CWLH06]
Most active 20% of developers	Core group size	Size	Yes	[RGB06]	
Contributors that produce 33% of the output	Core group size	Size	Yes	[CWLH06]	
SNA analysis of reply-to structures	Core group size	Size	Yes	[CWLH06]	
Number of developers committing source code	Co-developer sphere size	Size	Yes	Here	[MFH02]
Number of active participants on mailing list or issue tracker	Co-developer sphere size, proxy for project activity	Size	Yes	Here	[MFH02]
Number of mailing list subscribers	Co-developer sphere size, proxy for project activity	Size	Yes	Here	
Number of issue reporters	Active user sphere size	Size	Yes	[CAHM04]	[HC04]
Release frequency	Project activity	Activity	Partial	Here	
Usage of typical FOSSD tools	Proxy for process maturity	Activity	No	[Mic05]	
Derivative of SLOC	Project growth, proxy for project activity	Activity	Yes	[Koc05]	
New bug reporters	Project growth, proxy for project activity	Activity	Yes	Here	
Derivative of team size (based on some size metric)	Project growth, capability to attract new developers, proxy for project activity	Activity	Yes	Here	
Second derivative of team size	Growth pattern, early warning	Activity	Yes	Here	
Developer half-time or survival function	Keeping members	Activity	Yes	[RGBM05]	
SLOC half-time or survival function	Speed of changed	Activity	Yes	Here	
Issue survival function (time to solution)	issue fixing speed, proxy for project activity	Activity	Yes	[CHA06]	[San05]
Time it takes to recognize duplicate issues	Time to consensus about the issue	Activity	Yes	Here	[San05]
Survival function for non-duplicate issues	Solving speed for recognized issues	Activity	Yes	Here	[San05]

Table 4.1: Continuation

Metric	**Measured attribute or proxy for**	**Category**	**Automatable**	**Introduced in**	**Related research**
Frequency of new issues	Issue finding activity, proxy for project activity	Activity	Yes	Trad.	
Percentage of connected issues	Information management	Activity	Yes	Here	[San05]
SourceForge activity rank	Proxy for project activity	Activity	Yes	[CAHM04]	
Number of mails, issues or source commits of a developer	Activity of a developer	Activity	Yes	Here	
Number of downloads per unit of time	Popularity or activity	Success	Yes	[CAH03]	[CAHM04]
Inclusion in Linux distributions	Popularity	Success	Partial	[CHA06]	
Debian Popularity contest	Popularity	Success	Yes	[CAH03]	
Web search engine hits for project name	Popularity	Success	Yes	[Wei05]	
Link count for project web page in web search engines	Popularity	Success	Yes	[Wei05]	
Any metric for project activity, in particular team growth	Successful running of project, proxy for developer satisfaction	Success	Yes	[CAHM04]	[CHA06]
Developer satisfaction survey	Developer satisfaction	Success	No	[CHA06]	
Start date in SourceForge	Age	Other	Yes	Here	
First issue, mail in archive or source commit	Age	Other	Yes	Here	
The first release	Age	Other	Partial	Here	
Reputation in Advocado.org	How well connected core developers are	Other	Yes	[CHA06]	
Degree of a SNA of the developers	Team “tightness”	Other	Yes	[WMZ06]	
Degree of node in a project-developer graph	Connectedness to other projects	Other	Yes	[WMZ06]	[WHC05]
Average SLOC per module, also distribution	Modularity	Other	Yes	Here	
Automatic text analysis on issue data	Uncertainty on an issue	Other	Yes	[San05]	
Source directories, 1, 2 or many	Proxy for modularity	Other	Yes	[CLM03]	
Developers per module, also distribution	Unclear	Other	Partial	Here	

5 Future directions

This chapter discusses the future directions of FOSSD and research on FOSSD. Chapter 5.1 discusses the future of free and open source software as well as software development. Chapter 5.2 describes some likely trends in the research on FOSSD and points out some particular questions that need further study. Chapter 5.3 introduces a new tool and an approach for FOSSD research that looks particularly promising.

5.1 Future of FOSSD

The future of free and open source software looks generally bright. Unlike a proprietary software product, a FOSS product cannot be bought out and it cannot disappear in bankrupt. As long as there is at least some level of interest in using a piece of FOSS, it will remain available. Due to this persistence, it is reasonable to assume, that the successful FOSS products that exist today will remain available, and new ones will appear.

When a successful FOSS product appears on some software category, e.g. Firefox web browser or Apache web server, it sets a minimum standard for all proprietary products on that category. As the amount of software is growing exponentially, and the amount of software developers cannot compete with that growth, more and more software becomes part of infrastructure that is not actively developed, but just assumed to be there. In the market of infrastructure software, changes are slow and minimal, so it is virtually impossible for proprietary software to compete against FOSS. Thus an increasing part of the basic software infrastructure will become FOSS in the future. This trend can already be seen in the Internet infrastructure.

However, the bright future of FOSS does not necessarily mean similar success for FOSSD. As discussed in Chapter 3.1, FOSSD is just one of the many methods to develop FOSS. An increasing partition of FOSS is created by companies and they do it for many reasons, not only to get external voluntary developers for free, but also e.g. to create a "minefield" on a market, by making a superior FOSS solution available and thus preventing competitors from creating successful business. They may not want to create FOSSD projects but use more traditional software engineering approaches instead. So the future of FOSSD depends at least partially partially on the interest the companies have on it.

Whether companies see encouraging FOSSD as beneficial does not depend only on their goals, but even more on their understanding of the phenomenon. As the results of FOSSD are always hard to predict, the less it is understood, the bigger is the temptation to use traditional software engineering approach instead. For this reason, the future of FOSSD may well depend on the success of the research on FOSSD. Be that as it may, there will certainly be FOSSD projects beside the growing number of FOSS products that are developed based on more conventional software engineering. The question is about the relative sizes of the groups, not their existence.

According to Clay Shirky, the entire Wikipedia project has taken about hundred million hours of work. It may sound like a large number, but it is only 1/2000 of the time people spend watching television per year in the United States [Shi08]. The Debian Linux distribution contains majority of all even moderately successful FOSSD projects. The Debian version 3.0, released 2002, would have taken 50 million work hours if it was implemented on conventional software engineering methods [GBRP+05]. If we assume that for each hour of development, nine hours have been spent on configuring the software, debugging it and other related activities, this makes approximately 500 million hours of work, 1/400 of the time people spend watching television yearly in the United States.

Based on this calculation, if one percent of US population would spend 10% of the time they currently watch television on FOSSD instead, it would take them two years to rewrite most of existing FOSS. So, if the amount of people who can program or the amount of free time they have are not changing drastically, FOSSD is here to stay.

Many FOSSD projects have been expanding exponentially for a long time. It sounds plausible to assume that such growth cannot continue forever and that there will inevitably be a transition to more stable phase. However, considering that also e.g. economic systems are based on models that assume eternal exponential growth, the possibility of continuing exponential growth cannot be deemed unrealistic a priori, but some evidence of such transition would be needed. So far no study have been published that would show slowdown in the growth of major FOSSD projects. Until some contradicting evidence is found, it is reasonable to assume that the exponential growth will continue.

As discussed in Chapter 3.1.2, the FOSSD process is largely shaped by the FOSSD tools. Thus future changes in the process are likely to be introduced by new kind

of tools. It is difficult to foresee what such tools would be, but one possibility is discussed in Chapter 5.3.

5.2 Future directions in the research on FOSSD

The directions the research on FOSSD will take in the future, depend on two things: the goals of the research and the success of the research that was using different methods and approaches. Most of the research so far has been basic research aiming at constructing general understanding of the phenomenon. Considering the increasing interest of companies in FOSSD discussed above, and their need for better estimation of particular projects, it is reasonable to assume that the amount of applied research focusing on some particular projects will increase.

The success of different methods and approaches is in general hard to predict. However, some trends can already be seen, and they are likely to continue in the near future.

A lot of research on FOSSD is based on some research framework, as discussed in Chapter 4. As the frameworks allow the researcher to focus on more detailed questions and connect the results to other research using the same framework, it is reasonable to assume that their usage still increases. Also, connecting results more to other research, i.e. supporting or rejecting older results or using their results as basis for research problems, will increase when the field ages, as there is more research to connect to. With some optimism, a trend towards more coherent and connected understanding of the whole phenomenon can also be seen.

One particular category of studies that will likely increase due to more connected research are those aiming at validating results of other studies. Considering that a large part of the research on FOSSD is utilizing somewhat unreliable methods such as virtual ethnography, case studies and simulations, as discussed in Chapters 4.2.1 to 4.2.4, there is a clear need for validating results via other means. Validating the results with metrics is discussed more in Chapter 4.3.1.

Another trend connecting the research are the FOSSD data and tool repositories discussed in Chapter 4.2.5. The usability of the data in the repositories has been so far limited due to missing some key attributes, but such obstacles will likely be solved. However, perhaps more important aspect of the FOSSD repositories is the practice of publishing also the created scripts and tools with a FOSSD license. If this practice becomes more common in the FOSSD research, it can lead to more

spontaneous collaboration and in general faster progress in the research — much the same way it works in FOSSD. It is good to notice, that there is no reason why this kind of "open source science" approach should be limited only to research on FOSSD or even software engineering, but it can be applied also to e.g. biology [Mau03].

Also the increasing connections to social sciences, in the form of borrowing research methods, tools and research frameworks, is a trend that will most likely continue. In particular, software tools that are based on social science research, such as automatic discourse analysis and social network construction tools, are likely to be an integral part of FOSSD research in the future.

The close connection to social sciences unfortunately brings along their methodological challenges, most notably the multitude of different approaches and theoretical models that each postulate their own concepts and cannot be easily combined or compared to each other. However, compared to most social sciences, the research on FOSSD has some advantages in fighting this division, especially the unique access to virtually all communication data in the projects. This access enables data mining approaches to gain statistical proofs for detailed questions that would have been infeasible to study in most other fields. Thus it is in principle possible to make quantitative comparisons between frameworks based on how well the hypotheses created based on them reflect the reality. In practice this is of course far from a trivial task.

Besides the general trends, there are also some specific research questions that deserve a more detailed investigation in the future.

- The team dynamics, especially the movement of people from role to role is lacking quantitive analysis. The general principles are known, as discussed in Chapter 3.1.5, but e.g. the relative sizes of the spheres and the amount of movement between them have been studied only cursorily.

- The forking of the projects, the reasons leading to it and the reasons why some forks survive and others do not, are not well covered in the scientific literature.

- Further study on software and project evolution in FOSSD is likely to provide more detailed and reliable picture of the project life span as well as models to be used on e.g. simulations and metrics.

- The reasons for the power law dynamics of many attributes of FOSSD needs further study. For example, the stability of the package sizes in the Debian

project and module sizes in the Apache project discussed in Chapter 3.1.6 gives a hint about some general principle that is still not known.

- The utilization of software process metrics in the context of FOSSD has significant potential, as discussed in Chapters 4.3.1 and 5.3.

Using the concepts of Thomas Kuhn, the research on FOSSD is still in *pre-scientific* phase, without a commonly agreed framework that would unite it, unlike software engineering, which is in the phase of *normal science*, where researchers share common assumptions on how their subject behaves in general and do not have to bother themselves with questions about the very nature of their field [Kuh70]. Development towards normal science would be beneficial also for the research on FOSSD, as according to Kuhn, most of the advancements in science happen in the phase of normal science. The trends described above raise some hope regarding such future.

5.3 Automated tool for FOSSD metrics

As discussed in Chapter 4.3.1, the FOSSD process metrics have significant potential benefits for FOSSD projects, for industry and for scientific research. This chapter discusses one practical approach to realize some of those benefits, namely automated tool for obtaining FOSSD project metrics.

Technically such tool would be a collection of scripts for obtaining data from primary sources such as version control repositories, mail list archives as well as issue trackers and for calculating metrics and indicators based on the data. The metrics would include some combination of size and activity metrics at least, and they would need to be throughoutly tested and validated. Also the system would include a central repository for collecting the results for comparison.

As a fully automated tool cannot use common sense when a project appears hundred years old or done by just one developer, the demands for the reliability of the indicators are higher than in manual usage. For this reason, full automation may be infeasible to start with, but manual verification of data is needed at least in the beginning.

The fact that there is several different variants of source repositories, issue trackers and mailing list archive software, causes some additional challenges, as all major systems should be supported. Avoiding this issue by using existing repositories of pre-processed data is not a realistic option, as they lack many major details, and

this would also practically prevent creating a FOSSD project around the tool, as discussed below.

Considering the fast adaptations to new tools in FOSSD discussed in Chapter 3.1.2, an automatic metrics system that provides real value to the project leaders could be taken into use relatively fast. This may not be as easy as it sounds: the policy and process questions in FOSSD projects tend to be formalized as questions about the source code, as discussed in Chapter 3.1.5 and the process metrics have nothing to do with the source code, which may make them appear irrelevant to FOSS developers. However, mere curiosity and a possibility to compare own project against others might be a reason enough for using the tool.

Creating the tool as a FOSSD project is a requirement for spreading it among FOSS developers. The first reason for this is that they tend to be more reserved regarding tools that are not FOSS, as discussed in Chapter 3.1.1. The second and more important reason is the very nature of FOSSD as an emergent and self-correcting process: the FOSS developers themselves could adapt the tool to their own data formats and report the issues with fix proposals back to the project.

The benefits such tool and its widespread usage would provide for the research on FOSSD are clear. As discussed in Chapter 4.3.1, metrics are a valuable tool for verifying research results and comparing results obtained by incompatible means to each other. If an automatic metric tool was available for all researchers, using it for verifying results could become a common practice on a large part of FOSSD research. This, in turn, could increase the connections of the research results to each other and provide a more coherent picture of the whole phenomenon.

6 Conclusions

Free and open source software development is not software engineering as it is described in text books and cannot be measured by the traditional tools of software engineering. Instead FOSSD is an alternative approach to the development of complex software systems with its own strengths and limitations that differ from those of traditional software engineering.

The understanding of FOSSD is still incomplete, but the basic structure and functionality of the FOSSD projects and the accompanying development teams and communities are relatively well known. The development process is not divided to different phases, but instead development happens on all phases at the same time: requirements analysis, design, implementation and testing are all going on simultaneously.

The project community is typically divided in an onion like structure, where *core developers* are responsible for most of the code and design and *co-developers* contribute infrequently on some parts of the project. *Active users* take new versions in use and test them and also report issues and requirement proposals back to the projects, whereas *passive users* just use the software without participating in the community, which makes identifying or counting them generally infeasible.

The research on FOSSD has gained some momentum since it started in the late 1990's. Currently there is an annual conference on the topic and several collections and special issues on the topic have been published. Many of the studies have been done in isolation, with quite limited connections to other research, but the trend is towards more connected research, which will hopefully lead to a more coherent picture of the whole phenomenon.

As a field of study, research on FOSSD is *descriptive science*, aiming at understanding and describing the phenomenon unlike traditional software engineering, which is *prescriptive* by its nature, aiming at improving the things it studies. Due to the descriptiveness and the fact that the main focus is essentially in studying the behavior of people, FOSSD has a lot in common with social sciences. Many methods and theoretical frameworks utilized in FOSSD research have indeed been borrowed from social sciences. It is reasonable to assume that this connection will grow even stronger in the future.

Using the concepts of Thomas Kuhn, the research on FOSSD is still in *pre-scientific* phase, without a commonly agreed framework that would unite it, whereas soft-

ware engineering is in the phase of *normal science*, where researchers share common assumptions on how their subject behaves in general and do not have to bother themselves with questions about the very nature of their field [Kuh70]. Development towards normal science would be beneficial also for the research on FOSSD, as according to Kuhn, almost all the advancements in science happen in the phase of normal science.

One tool in creating a more coherent picture on FOSSD could be the utilization of software process metrics. Metrics would in many cases allow validation or disregarding of results that are originally obtained in a method that cannot validate its own results, e.g. via ethnography or simulations. Common metrics utilized regularly as a part of research would also make studies based on entirely different approaches comparable in their results, thus providing at least a limited way to connect them.

In the near future the field would benefit from more research that aims at validating the already found results and connecting them to each other. The metrics and approaches discussed in this thesis can be of use in that work.

References

AB05 Adams, P. and Boldyreff, C., Towards supporting agile practice within the libre software paradigm. *Proceedings of the First International Conference on Open Source Systems*. IFIP, 2005, pages 303–304.

ASS05a Angioni, M., Sanna, R. and Soro, A., Defining a distributed agile methodology for an open source scenario. *Proceedings of the First International Conference on Open Source Systems*. IFIP, 2005, pages 209–214.

ASS+05b Antoniades, I., Samoladas, I., Stamelos, I., Angelis, L. and Bleris, G., Dynamical simulation models of the open source development process. In *Free/Open Source Software Development*, Koch, S., editor, Idea Group Publishing, London, 2005, pages 174–202.

BA99 Barabási, A. and Albert, R., Emergence of scaling in random networks. *Science*, 286,5439(1999), pages 509–512.

BDBS05 Barcellini, F., Détienne, F., Burkhardt, J. and Sack, W., Thematic coherence and quotation practices in OSS design-oriented online discussions. *Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*. ACM Press, 2005, pages 177–186.

Ber06 Berdou, E., Insiders and outsiders: paid contributors and the dynamics of cooperation in community led F/OS projects. *Proceedings of the Second International Conference on Open Source Systems*. IFIP, 2006, pages 201–208.

Bez99a Bezroukov, N., Open source software as a special type of academic research. *First Monday*, 4,10(1999).

Bez99b Bezroukov, N., A second look at the cathedral and the bazaar. *First Monday*, 4,12(1999).

BJ78 Brooks Jr, F., *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Publishing Company, Boston, MA, USA, 1978.

BR04 Bonaccorsi, A. and Rossi, C., Contributing to OS projects: a comparison between individual and firms. *Proceedings of the 4th Workshop on Open Source Software Engineering*. ACM, 2004, pages 18–22.

CAH03 Crowston, K., Annabi, H. and Howison, J., Defining open source software project success. *Proceedings of the International Conference on Information Systems (ICIS 2003)*. Association for Information Systems, 2003, pages 327–340.

CAHM04 Crowston, K., Annabi, H., Howison, J. and Masango, C., Towards a portfolio of FLOSS project success measures. *Proceedings of the 4th Workshop on Open Source Software Engineering*. ACM, 2004, pages 29–33.

CAHM05 Crowston, K., Annabi, H., Howison, J. and Masango, C., Effective work practices for FLOSS development: A model and propositions. *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05)*, Los Alamitos, CA, USA, 2005, IEEE Computer Society, pages 197–206.

CH05a Coleman, E. and Hill, B., The social production of ethics in Debian and free software communities: Anthropological lessons for vocational ethics. In *Free/Open Source Software Development*, Koch, S., editor, Idea Group Publishing, London, 2005, pages 273–295.

CH05b Crowston, K. and Howison, J., The social structure of free and open source software development. *First Monday*, 10,2(2005).

CHA06 Crowston, K., Howison, J. and Annabi, H., Information systems success in free and open source software development: Theory and measures. *Software Process—Improvement and Practice*, 11,2(2006), pages 123–148.

CHAM05 Crowston, K., Heckman, R., Annabi, H. and Masango, C., A structurational perspective on leadership in free/libre open source software teams. *Proceedings of the First International Conference on Open Source Systems*. IFIP, 2005, pages 9–15.

CLM03 Calicuppi, A., Lago, P. and Morisio, M., Evidences in the evolution of OS projects trough changelog analyses. *Proceedings of the 3rd Workshop on Open Source Software Engineering*. ACM, 2003.

Con06 Conklin, M. S., Beyond low-hanging fruit: seeking the next generation in FLOSS data mining. *Proceedings of the Second International Conference on Open Source Systems*. IFIP, 2006, pages 47–56.

CS02 Crowston, K. and Scozzi, B., Open source software projects as virtual organisations: competencyrallying for software development. *Software, IEE Proceedings*, 149,1(2002), pages 3–17.

CS06 Crowston, K. and Scozzi, B., The role of mental models in FLOSS development work practices. *Proceedings of the Second International Conference on Open Source Systems*. IFIP, 2006, pages 91–97.

CWL+05 Crowston, K., Wei, K., Li, Q., Eseryel, U. and Howison, J., Coordination of free/libre open source software development. *International Conference on Information Systems*. ACM, 2005, pages 11–23.

CWLH06 Crowston, K., Wei, K., Li, Q. and Howison, J., Core and periphery in free/libre and open source software team communications. *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS 06)*. IEEE Computer Society, 2006, pages 118–124.

DM03 Delone, W. and McLean, E., The DeLone and McLean model of information systems success: A ten-year update. *Journal of Management Information Systems*, 19,4(2003), pages 9–30.

Ell03 Elliott, M., The virtual organizational culture of a free software development community. *Proceedings of the 3rd Workshop on Open Source Software Engineering*. ACM, 2003, pages 45–50.

ES03 Elliott, M. and Scacchi, W., Free software developers as an occupational community: resolving conflicts and fostering collaboration. *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*. ACM, 2003, pages 21–30.

ES05 Elliott, M. and Scacchi, W., Free software development: Cooperation and conflict in a virtual organizational culture. In *Free/Open Source Software Development*, Koch, S., editor, Idea Group Publishing, London, 2005, pages 152–172.

ES08 Elliott, M. and Scacchi, W., Mobilization of software developers: The free software movement. *Information, Technology and People*, 21(2008), pages 4–33.

ET03 Erenkrantz, J. and Taylor, R., Supporting distributed and decentralized projects: Drawing lessons from the open source community. *1st Workshop on Open Source in an Industrial Context*. Technische Universität München, 2003, pages 21–30.

Frea Free Software Foundation, The free software definition, `http://www.fsf.org/licensing/essays/free-sw.html` (6.9.2007).

Freb Free Software Foundation, GNU general public license, `http://www.gnu.org/licenses/gpl.html` (6.9.2007).

Frec Free Software Foundation, GNU lesser general public license, `http://www.gnu.org/licenses/lgpl.html` (6.9.2007).

GBLR04 Gonzalez-Barahona, J., Lopez, L. and Robles, G., Community structure of modules in the Apache project. *Proceedings of the 4th Workshop on Open Source Software Engineering*. ACM, 2004, pages 43–47.

GBRP+05 Gonzalez-Barahona, J., Robles, G., Perez, M., Rodero-Merino, L., Gonzalez, J., Matellan-Olivera, V. and Castro-Barbero, E., Analyzing the anatomy of GNU/Linux distributions: methodology and case studies (Red Hat and Debian). In *Free/Open Source Software Development*, Koch, S., editor, Idea Group Publishing, London, 2005, pages 27–58.

GGKR Ghosh, R., Glott, R., Krieger, B. and Robles, G., Free/libre and open source software: Survey and study, part iv: Survey of developers, `http://flossproject.org/report/FLOSS_Final4.pdf(16.03.2008)`.

GR03 Gasser, L. and Ripoche, G., Distributed collective practices and free/open-source software problem management: perspectives and methods. *Proceedings of the Conference on Coopération, Innovations et Technologies (CITE 03)*, 2003, pages 349–365.

GRS04 Gasser, L., Ripoche, G. and Sandusky, R., Research infrastructure for empirical science of F/OSS. *Proceedings of the First International Workshop on Mining Software Repositories*. IET, 2004, pages 12–16.

GSRP03 Gasser, L., Scacchi, W., Ripoche, G. and Penne, B., Understanding continuous design in F/OSS projects. *16th International Conference on Software Engineering & its Applications (ICSSEA-03)*, 2003.

GT00 Godfrey, M. and Tu, Q., Evolution in open source software: A case study. *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 2000, pages 131–142.

HC04 Howison, J. and Crowston, K., The perils and pitfalls of mining sourceforge. *Proceedings of Workshop on Mining Software Repositories*. University of Waterloo, 2004, pages 7–11.

HCC05 Howison, J., Conklin, M. and Crowston, K., OSSmole: A collaborative repository for FLOSS research data and analyses. *Proceedings of the First International Conference on Open Source Systems*. IFIP, 2005.

HCC06 Howison, J., Conklin, M. and Crowston, K., FLOSSmole: a collaborative repository for FLOSS research data and analyses. *International Journal of Information Technology and Web Engineering*, 1,3(2006), pages 17–26.

HIC06 Howison, J., Inoue, K. and Crowston, K., Social dynamics of free and open source team communications. *Proceedings of the Second International Conference on Open Source Systems*. IFIP, 2006, pages 319–330.

Hin00 Hine, C., *Virtual Ethnography*. Sage Publications, London, 2000.

HMB+06 Hahn, J., Moon, J., Bay, C., Kowloon, H. and Zhang, C., Impact of social ties on open source project team formation. *Proceedings of the Second International Conference on Open Source Systems*. IFIP, 2006, pages 307–317.

JS04 Jensen, C. and Scacchi, W., Collaboration, leadership, control, and conflict negotiation in the netbeans.org open source community. *Proceedings of the 4th Workshop on Open Source Software Engineering*. ACM, 2004.

JS05a Jensen, C. and Scacchi, W., Experience in discovering, modeling, and reenacting open source software development processes. *Unifying the Software Process Spectrum: International Software Process Workshop*, 2005, pages 449–462.

JS05b Jensen, C. and Scacchi, W., Process modeling across the web information infrastructure. *Software Process–Improvement and Practice*, 10,3(2005), pages 255–272.

Koc05 Koch, S., Evolution of open source software systems—a large scale investigation. *Proceedings of the First International Conference on Open Source Systems*. IFIP, 2005, pages 148–153.

Kop06 Koponen, T., Life cycle of defects in open source software projects. *Proceedings of the Second International Conference on Open Source Systems*. IFIP, 2006, pages 274–286.

Kuh70 Kuhn, T., *The Structure of Scientific Revolutions*. University of Chicago Press, 1970.

Leh80 Lehman, M., Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68,9(1980), pages 1060–1076.

Leh96 Lehman, M., Laws of software evolution revisited. *Proceedings of European Workshop on Software Process Technology*. Springer, 1996, pages 108–124.

LT05 Lerner, J. and Tirole, J., The Scope of Open Source Licensing. *Journal of Law, Economics, and Organization*, 21,1(2005), pages 20–56.

Mau03 Maurer, S., New institutions for doing science: from databases to open source biology. *Paper presented at European Policy for Intellectual Property Conference, Maastricht*, 2003.

MFH02 Mockus, A., Fielding, R. T. and Herbsleb, J. D., Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11,3(2002), pages 309–346.

MFT02 Madey, G., Freeh, V. and Tynan, R., Understanding OSS as a self-organizing process. *Proceedings of the 2nd Workshop on Open Source Software Engineering*. ACM, 2002.

MH02 Mockus, A. and Herbsleb, J., Why not improve coordination in distributed software development by stealing good ideas from open source? *Proceedings of the 2nd Workshop on Open Source Software Engineering*. ACM, 2002.

Mic05 Michlmayr, M., Software process maturity and the success of free software projects. *Software Engineering: Evolution and Emerging Technologies*, 130(2005), pages 3–14.

Moe99 Moen, R., Fear of forking essay, http://linuxmafia.com/faq/Licensing_and_Law/forking.html (17.5.2008), 1999.

NR05 Narduzzo, A. and Rossi, A., The role of modularity in free/open source software development. In *Free/Open Source Software Development*, Koch, S., editor, Idea Group Publishing, London, 2005, pages 84–102.

Opea Open Source Initiative, The BSD license, `http://www.opensource.org/licenses/bsd-license.php` (6.9.2007).

Opeb Open Source Initiative, The open source definition, `http://www.opensource.org/docs/osd` (6.9.2007).

ØR03 Østerlie, T. and Rolland, K., Unveiling distributed organizing in open source development: Practices of using, aligning, and wedging. *Proceedings of the Workshop on Open Source Software Movements and Communities*, 2003, pages 310–316.

Øst04 Østerlie, T., In the network: Distributed control in Gentoo Linux. *Proceedings of the 4th Workshop on Open Source Software Engineering*. ACM, 2004, pages 76–81.

Pap07 Papadimitriou, C., The algorithmic lens: How the computational perspective is transforming the sciences, `http://www.cra.org/ccc/fcrc/` (30.3.2008), 2007.

Pav00 Pavlicek, R., *Embracing Insanity: Open Source Software Development*. Sams Publishing, Indianapolis, IN, USA, 2000.

PC04 Potdar, V. and Chang, E., Open source and closed source software development methodologies. *Proceedings of the 4th Workshop on Open Source Software Engineering*. ACM, 2004, pages 105–109.

Pre00 Pressman, R., *Software Engineering: A Practitioner's Epproach: European Adaptation*. McGraw-Hill, 2000.

Rag95 Ragland, B., Measure, metric, or indicator: What's the difference?, http://www.stsc.hill.af.mil/crosstalk/1995/03/Measure.asp (27.3.2008, 1995.

Ray99a Raymond, E. S., *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly and Associates, Sebastopol, California, 1999. URL `http://www.catb.org/~esr/writings/cathedral-bazaar(31.5.2008)`.

Ray99b Raymond, E., The revenge of the hackers. In *Open Sources: Voices from the Open Source Revolution*, DiBona, C., Ockman, S. and Stone, M., editors, O'Reilly, Cambridge, Massachusetts, 1999, pages 207–219, URL `http://www.oreilly.com/catalog/opensources`.

RG03 Ripoche, G. and Gasser, L., Scalable automatic extraction of process models for understanding F/OSS bug repair. *Proceedings of the 16th International Conference on Software & Systems Engineering and their Applications (ICSSEA-03)*, 2003.

RGB06 Robles, G. and Gonzalez-Barahona, J., Contributor turnover in libre software projects. *Proceedings of the Second International Conference on Open Source Systems*. IFIP, 2006, pages 274–286.

RGBM05 Robles, G., Gonzalez-Barahona, J. and Michlmayr, M., Evolution of volunteer participation in libre software projects: Evidence from Debian. *Proceedings of the First International Conference on Open Source Systems*. IFIP, 2005, pages 100–107.

Rob02 Robbins, J., Adapting OSS methods by adapting OSS tools. *Proceedings of the 2nd Workshop on Open Source Software Engineering*. ACM, 2002.

Rob03 Robbins, J., Adapting open source software engineering (OSSE) practices by adapting OSSE tools. In *Making Sense of the Bazaar: Perspectives on open source and free software*, Feller, J., Fitzgerald, B., Hissam, S. and Lakhani, . K., editors, MIT press, Cambridge, Massachusetts, 2003.

San05 Sandusky, R., Software problem management as information management in a F/OSS development community. *Proceedings of the First International Conference on Open Source Systems*. IFIP, 2005, pages 11–15.

SC06 Schofield, A. and Cooper, G. S., Participation in free and open source communities: an empirical study of the community member's percep-

tions. *Proceedings of the Second International Conference on Open Source Systems*. IFIP, 2006, pages 221–230.

Sca02a Scacchi, W., Is open source software development faster, better, and cheaper than software engineering. *Proceedings of the 2nd Workshop on Open Source Software Engineering*. ACM, 2002.

Sca02b Scacchi, W., Understanding the requirements for developing open source software systems. *IEE Proceedings - Software Engineering*, 149,1(2002), pages 24–39.

Sca05 Scacchi, W., Socio-technical interaction networks in free/open source software development processes. In *Software Process Modeling*, Acuña, S. and Juristo, N., editors, Springer, 2005, pages 1–27.

Sca06 Scacchi, W., Understanding free/open source software developement processes. *Software Process–Improvement and Practice*, 11,2(2006), pages 95–105.

Sca07 Scacchi, W., Free/open source software development: Recent research results and methods. *Advances in Computers: Architectural Advances*, 69(2007), pages 243–290.

SDD+06 Sack, W., Détienne, F., Ducheneaut, N., Burkhardt, J., Mahendran, D. and Barcellini, F., A methodological framework for socio-cognitive analyses of collaborative design of open source software. *Computer Supported Cooperative Work (CSCW)*, 15,2(2006), pages 229–250.

Shi08 Shirky, C., Gin, television, and social surplus, `http://www.shirky.com/herecomeseverybody/2008/04/looking-for-the-mouse.html` (25.5.2008), 2008.

SJNE06 Scacchi, W., Jensen, C., Noll, J. and Elliott, M., Multi-modal modeling, analysis and validation of open source software requirements processes. *International Journal of Information Technology and Web Engineering*, 1,3(2006), pages 49–63.

Som07 Sommerville, I., *Software Engineering, 8th edition*. Addison Wesley, 2007.

Sou SourceForge inc, What statistical information does SourceForge.net collect?, http://alexandria.wiki.sourceforge.net/Statistics (29.3.2008).

Sta99 Stallman, R. M., The GNU operating system and the free software movement. In *Open Sources: Voices from the Open Source Revolution*, DiBona, C., Ockman, S. and Stone, M., editors, O'Reilly, Cambridge, Massachusetts, 1999, pages 53–70, URL `http://www.oreilly.com/catalog/opensources`.

Sta02 Stallman, R. M., Why "free software" is better than "open source". In *Free Software, Free Society: Selected Essays of Richard M. Stallman*, Gay, J., editor, Gnu Press, 2002, URL `http://www.gnu.org/philosophy/free-software-for-freedom.html(31.5.2008)`.

Tur96 Turski, W., Reference model for smooth growth of software systems. *IEEE transactions on software engineering*, 22,8(1996), pages 599–600.

WA03 Warsta, J. and Abrahamsson, P., Is open source software development essentially an agile method. *Proceedings of the 3rd Workshop on Open Source Software Engineering*. ACM, 2003, pages 143–147.

Wei05 Weiss, D., Measuring success of open source projects using web search engines. *Proceedings of the First International Conference on Open Source Systems*. IFIP, 2005, pages 93–99.

WHC05 Wagstrom, P., Herbsleb, J. and Carley, K., A social network approach to free/open source software simulation. *Proceedings of the First International Conference on Open Source Systems*. IFIP, 2005, pages 16–23.

WMCC02 Watson-Manheim, M., Chudoba, K. and Crowston, K., Discontinuities and continuities: a new way to understand virtual work. *Information, Technology and People*, 15,3(2002), pages 191–209.

WMZ06 Weiss, M., Moroiu, G. and Zhao, P., Evolution of open source communities. *Proceedings of the Second International Conference on Open Source Systems*. IFIP, 2006, pages 21–32.

YNYK05 Ye, Y., Nakakoji, K., Yamamoto, Y. and Kishida, K., The co-evolution of system and community in open source software development. In *Free/Open Source Software Development*, Koch, S., editor, Idea Group Publishing, London, 2005, pages 59–82.