A Framework for Open Source Projects

Master Thesis in Computer Science

submitted by

Gregor J. Rothfuss
rothfuss@abstrakt.ch
of Zurich, Switzerland
Register No. 97-711-915

Supervisor:
Prof. K. Bauknecht
Department of Information Technology
University of Zurich

Zurich, November 12, 2002

Abstract

The historical roots of Open Source are outlined. A comparison between Open Source projects and classical projects highlights strengths and weaknesses of both, and defines their attributes. Existing Open Source theories are evaluated, and the requirements for a framework for Open Source projects are determined. The framework introduces the notions of actors, roles, areas, processes and tools, and depicts their interrelationships in a matrix. Each aspect of the framework is then further developed to serve both as a conceptual foundation for Open Source and a help for organizing and managing Open Source projects.

Die Geschichte von Open Source wird aufgezeigt. Ein Vergleich zwischen Open Source und klassischen Projekten beleuchtet Stärken und Schwächen beider Ansätze, und definiert deren Atribute. Existierende Open Source Theorien werden evaluiert, und die Anforderungen für ein Rahmenwerk über Open Source ermittelt. Das Rahmenwerk führt die Begriffe Akteur, Rolle, Bereich, Prozess und Werkzeug ein und illustriert deren Zusammenspiel in einer Matrix. Die einzelnen Aspekte des Rahmenwerks werden vertieft und dienen als konzeptuelle Grundlage für Open Source sowie helfen bei der Organisation und dem Management von Open Source Projekten.

Acknowledgements

I would like to thank Prof. Dr. Kurt Bauknecht, head of the "Information Management" research group at the Department of Information Technology at the University of Zurich who made this master thesis possible.

I would like to thank for valuable discussions and suggestions Carl P. Corliss, Curtis Hays, Steven McGregor, Daniel Mettler, Andrew Vogel, Benjamin Voigt and Eric Wiseman. Special thanks to the participants in the PostNuke and OSCOM Open Source projects who inspired me to write this thesis in the first place, and to the University of California Berkeley for hosting me during the writing of this thesis.

A special thanks to Carl P. Corliss for careful spell checking and helping with the layout.

Table of Contents

Abstract 2
Acknowledgements 3
Table of Contents 4
List of Figures 7
List of Tables 8
List of Tables 8
1. Introduction 9
1.1 Why Open Source is relevant 9
1.2 Goals of this paper 10
2. The Open Source Phenomenon 11
2.1 Historical Overview 11
2.1.1 The early days 11
2.1.2 The rise of UNIX 11
2.1.3 The GNU Project 14
2.1.4 Linux 15
2.1.5 To the mainstream 16
2.2 Fundamentals 16
2.2.1 Intellectual Property Concepts 16
2.2.2 The significance of Contracts 19
2.2.3 Definitions of Open Source 19
2.2.4 Open Source Software Licenses 23
2.3 Theories about Open Source 26
2.4 The Open Source Community 27
2.4.1 Sociology 28
2.4.2 Software Engineering 29
2.4.3 Economy 33
2.5 Open Source Projects 35
2.5.1 Developers 36
2.5.2 Project Lifecycle 37
2.5.3 Example OSP 40
3. Open Source Projects versus Classical Projects 42
3.1 Underlying Trends 43
3.1.1 New Communication Technologies 43
3.1.2 More powerful Hardware 44
3.1.3 Higher-level Languages 44
3.1.4 Ubiquitous Standards 45
3.2 Defining Open Source Projects 46
3.2.1 Resources 47
3.2.2 Coordination 51
3.2.3 Structures 55
3.3 Defining Classical Projects 61
3.3.1 Resources 61
3.3.2 Coordination 63

3.3.3 Structures 66
3.4 Strengths of Open Source Projects 71
3.4.1 Release frequency 71
3.4.2 Customer Input 72
3.4.3 Scalability 72
3.5 Weaknesses of Open Source Projects 73
3.5.1 Communication 73
3.5.2 Redundant Efforts 73
3.5.3 Lack of Priorities 74
3.5.4 Lack of Conventions 74
3.5.5 Lack of focus 74
3.5.6 Dependency on key persons 74
3.5.7 Leadership 75
3.6 Strengths of Classical Projects 75
3.6.1 Predictability 75
3.6.2 Standards 76
3.6.3 Documentation 76
3.6.4 Accountability 76
3.7 Weaknesses of Classical Projects 76
3.7.1 Customer Input 76
3.7.2 Scalability 77
3.7.3 Bureaucracy 77
3.7.4 Skill levels 77
3.8 The Properties Matrix 78
4. Towards a unified Open Source Theory 80
4.1 Limitations of existing theories 80
4.1.1 The cathedral and the Bazaar 80
4.1.2 Open Source as academic Research 88
4.1.3 Economic Approaches 90
4.1.4 Virtual Decentralized Networks 94
4.1.5 Psychological Models 98
4.2 Challenges for Theory Formulation 99
4.2.1 Data Collection 99
4.2.2 Subject of Study 100
4.2.3 Theory Validation 101
4.3 A framework approach 101
4.3.1 Framework Goals 101
4.3.2 Framework Inputs 102
4.3.3 Framework Assumptions 103
4.3.4 Framework Methodology 105
5. A framework for Open Source Projects 106
5.1 Overview of the framework 106
5.1.1 Framework Components 106
5.1.2 The Framework Matrix 107
5.2 Actors and Roles 109
5.2.1 Actors 109

5.2.2 Roles 111
5.3 Areas 116
5.3.1 Marketing 116
5.3.2 Human Resources 117
5.3.3 Systems Management 117
5.3.4 Software Engineering 117
5.3.5 Project Management 118
5.4 Processes 118
5.4.1 Marketing 118
5.4.2 Human Resources 120
5.4.3 Systems Management 125
5.4.4 Software Engineering 128
5.4.5 Project Management 135
5.5 Tools 137
5.5.1 Marketing 137
5.5.2 Human Resources 139
5.5.3 Systems Management 142
5.5.4 Software Engineering 144
5.5.5 Project Management 147
6. Conclusion 148
6.1 Areas for further Research 149
Appendix A: Using the OSP framework 151
Project Success 151
Personal Success/Outcomes 152
Factors contributing to project success 152
Bibliography 153

List of Figures

Figure 1: A typical TCO Calculation (Source: UNISYS Corporation)............................. 34
Figure 2: Relations between roles in OSP (Source: [Evers00])... 59
Figure 3: Relations within an OSP 59
Figure 4: Social Network Analysis (Source: [Krebs02]) 60
Figure 5: Relations between roles in CSP (Source: [Evers00])... 70
Figure 6: Relation of CSP with their environment (Source: [Evers00]) 71
Figure 7: The Virtual Roof (Source: [Dafermos01]).. 96
Figure 8: Knowledge exchange in a virtual organization (Source: [Dafermos01]) 97
Figure 9: Visualization of Usenet postings 100
Figure 10: CSP versus OSP change process (Source: [Asklund01]) 114
Figure 11: Mozilla Roadmap 135
Figure 12: Affero online fundraising system 139
Figure 13: Radio Userland news aggregation 141
Figure 14: Sourceforge skill matrix... 142
Figure 15: Trillian Multi-Protocol Instant Messaging Client 144
Figure 16: Request Tracker.. 145
Figure 17: TortoiseCVS integration into the Windows explorer..................................... 146
Figure 18: JUnit framework... 147
Figure 19: Calendaring across platforms 148
Figure 20: Modularity of the Linux kernel 150

List of Tables

Table 1: Software Licenses compatibility 26
Table 2: Open Source projects by lifecycle stage 39
Table 3: OSP versus CSP properties 79
Table 4: The OSP framework matrix 108
Table 5: CMM Levels (Source: [Curtis95]) 121

1. Introduction

This paper aims to shed light on the hardly understood phenomenon of Open Source software development. Open Source has only gained widespread popularity as a term as recently as 1998, but is in fact far older. In the course of the appearance of Open Source in the public mind, many notions about its meaning have been formed, but few conclusions have been reached. Companies have been founded with great fanfare to exploit this new concept, and have since long gone bankrupt. Despite those business failures, Open Source is not a trend. It is here to stay. A casual Internet search reveals thousands of Open Source projects (OSP), with hundreds of thousands of participants. Academic knowledge of Open Source is in its very early stages, with few breakthrough papers to account for.

1.1 Why Open Source is relevant

Open Source has several properties that make it interesting for academic study, and relevant for the discipline of computer science.

Software Engineering
Open Source eschews many traditional software engineering concepts such as careful planning and prototyping in favor of a more organic approach. Some Open Source projects reach very high levels of quality while others linger along in the planning stages forever. Engineering feats like the Linux Kernel, one of the biggest software engineering projects ever seem impossible according to the notions of software engineering, and yet have been accomplished.

Economics
Participation in OSP defies common wisdom about economic principles. What makes highly skilled software engineers and programmers[1] participate in a project that they will not be paid for? Yet a COCOMO analysis of widely distributed Open Source programs derived a replacement value of $1 billion if they were recreated by conventional means.[2]

Social Sciences
OSP are a product of the Internet. Without the enormous advances in communication technologies and their widespread use around the world, such immense collaborative efforts would not be possible. The social sciences have discovered the Internet has a research medium, and OSP are interesting subjects of study because they produce tangible results from a process of discourse. OSP do present new opportunities to test theories of reputation, research trust relations between strangers and explore motivational behaviors.

---

[1] Empirical evidence suggests that OSP participants are among the most qualified software engineers in the world. One possible explanation is that the intense competition in OSP attracts the most talented in greater numbers than lesser skilled persons.
[2] http://www.dwheeler.com/sloc/

1.2 Goals of this paper

This paper presents a new attempt at a unified theory for Open Source. Building upon earlier theories, it brings together diverse areas of research (Software Engineering, Human Resources, Economics and Social Sciences) because existing theories fail to address Open Source as the interdisciplinary issue it is. The paper first puts Open Source into historical context in chapter 2 to try to understand the forces that led to its inception, and that will likely influence it in the future. The term Open Source is then examined in chapter 3 by looking at various aspects, and comparing Open Source projects to classical projects. The goal of this comparison is ultimately to arrive at a matrix that highlights the differences and similarities between the two approaches. A look at existing theories in chapter 4 reveals both their strengths and their deficiencies, and allows formulating the requirements for a new theory of OSP. This theory is designed to be extensible, and offers advice beyond explanation of the various aspects that are involved in OSP. Chapter 5 is devoted to the Open Source framework. The central organizing concept of the framework is the framework matrix, which ties together all the various aspects of Open Source under one roof. Each area of the framework is then discussed in detail. Finally, open areas of research are identified with the hope to include some of their results in a future version of the framework.

Besides being an academic treatise of Open Source, this paper also attempts to be readable by any interested individuals. Theoretical considerations are enriched with case studies, examples and advice. Readers with little time on their hands will find the comparison matrix at the end of chapter three helpful, and may want to explore the framework selectively. An extensive bibliography provides further resources.

It is assumed that the reader has a basic knowledge of Information Technology. Having encountered or worked in software engineering projects before helps greatly to relate to the case studies and terminology used, but is not a requirement.

2. The Open Source Phenomenon

Open Source goes back at least to the early 1960s. Yet as a term, the "Open Source Initiative" only coined it in 1998[3]. The history of Open Source is closely tied to the history of UNIX. Open Source needs legal prerequisites to make it work, like copyright and contract law. This in turn leads to a discussion of Open Source Licenses, a look at various theories of Open Source, and an examination of exemplary projects.

2.1 Historical Overview

When people talk about Open Source software, they normally refer to the operating system GNU/Linux and its applications. Linux has a long history that goes back to the creation of UNIX (1969) and further. This rich historical context is relevant to understand Open Source, because many of the philosophical issues involved were first encountered in the context of UNIX, and the tradition of UNIX permeates the Open Source community to this day.

2.1.1 The early days

Traditionally, hardware vendors like International Business Machines (IBM) delivered the source code for the operating systems and applications of their early computers with the shipment of the machine, because it was impractical to do otherwise. It was a strongly held belief that the value of a computer was contained in the hardware. Computer hardware was prohibitively expensive, and little thought was given to software. Operating systems and applications were necessary ingredients for the operation of a computer, but they were not seen as valuable in their own right. Furthermore, the users of computer technology were few and far between, and computer manufacturers actively encouraged users to share improvements to the software out of a belief that it would help save support costs. This belief changed only gradually with the advent of UNIX.

2.1.2 The rise of UNIX

[4]In 1965, the Massachusetts Institute of Technology (MIT), General Electric (GE) and Bell Telephone Labs (BTL) started the MULTICS (Multiplexed Information and Computing Service) project that had the objective to develop a new interactive, multi-user operating system. In 1969, BTL withdrew their resources from the MULTICS project, as success seemed to get out of reach.

"The problem was the increasing obviousness of the failure of MULTICS to deliver promptly any sort of usable system" [Ritchie79].

Even though the project was a failure, many lessons were learned from it, and subsequently applied to other projects. Among them:

- A file system with a tree structure

[3] http://www.opensource.org

[4] The following quotations, if not stated otherwise, are taken from the book "A Quarter Century of UNIX", by Peter H. Salus [Salus94]. Salus gives an excellent overview of UNIX history.

- A program to do command interpretation: the 'shell'
- The structure of files
- The semantics of "everything is a file"

Some of the BTL researchers which participated in the declined project did not want to give up the entire comfortable computing environment that was promised by MULTICS.

"They didn't want to lose the pleasant niche they occupied, because no similar ones were available" [Ritchie79].

Therefore, they began trying to find alternatives. They proposed the purchase of a new medium-scale computer, promising to write the operating system for it. The request was denied. They also developed the basic design for a file system and Ken Thompson[5] started to write some programs for a computer that was available at that time, a GE645. When it became clear that the machine would be removed from BTL within the following months, he stopped that work [Ritchie79]. Ken Thompson had developed a game called 'Space Travel' that was originally written for MULTICS[6] and then ported to GECOS[7], the operation system used on the GE645. 'Space Travel' did not run very well on the GE645 and was expensive to run. One game cost $75 US for CPU time. These facts prompted him to get it running on another machine. Dennis Ritchie[8] and Ken Thompson rewrote the game on the DEC PDP-7[9]. Thompson implemented the already designed file system soon after and continued to address all other requirements for a working operating system [Ritchie79]. Ken Thompson commented on how he started writing UNIX: *"I allocated a week each to the operating system, the shell, the editor, and the assembler... Yeah, essentially one person for a month."*[10] By promising to develop a text-processing tool for the system, they got a new PDP-11 computer in 1970 but had to share it with others.

"With several BTL staff members from outside the research group using the typesetting facilities of the PDP-11, the need to document the operating system grew. The result was the first UNIX Programmer's Manual by Thompson and Ritchie, which was dated November 3, 1971." [Salus94].

The manual[11] gave the first complete release of UNIX its name: "First Edition". It introduced most of the fundamental ideas. Many commands like 'mv', 'su' or 'find' had been invented at that time, and are still used today. Between 1971 and 1973, UNIX became a success inside certain parts of Bell Telephone Laboratories. The main improvements during that time were the invention of the C programming language, the port of the operating system to C and the invention of pipes. The result of these efforts

---

[5] Ken Thompson's biography is available at http://www.bell-labs.com/history/unix/thompsonbio.html
[6] http://www.multicians.org/
[7] http://www.tuxedo.org/~esr/jargon/html/entry/GCOS.html
[8] Dennis Ritchie maintains a biography at http://cm.bell-labs.com/cm/cs/who/dmr/bigbio1st.html
[9] http://www.montagar.com/dfwcug/VMS_HTML/timeline/1964-3.htm
[10] http://www.osdata.com/kind/unix.htm
[11] The manual is at http://cm.bell-labs.com/cm/cs/who/dmr/1stEdman.html

was the ’Third Edition’, released in February 1973. With the rewrite in C, UNIX was the first operating system that was portable to other hardware architectures. The “UNIX Philosophy” which Doug McIlroy[12], an early UNIX contributor, formulated as follows:

1. Write programs that do one thing and do it well
2. Write programs to work together
3. Write programs that handle text streams as the universal interface

led to a set of powerful tools, further increasing the appeal of UNIX. The third edition had been installed on 16 sites (all within AT&T/Western Electric) in 1973 and the development work had been going on for four years at that time. However, it was not known much outside AT&T. When Thompson submitted a paper at the “Symposium on Operating Systems Principles” in Yorktown Heights, NY, in October 1973 UNIX became popular in the field of computer science. Within six months of the paper’s delivery, the number of installations trebled. A revised version of the paper was published in the July 1974 issue of Communications of the ACM which “caused an explosion in demand for the fledgling operating system” [Salus94][Ritchie78]. AT&T had been convicted of antitrust violations in 1956, prohibiting it to start any other business than telephone or telegraph services. This had a very special effect on the development of UNIX. After people outside the company had started to get interested in the operating system AT&T had to avoid any conflict with the decree. Their policy was to license the software (allowed by the decree) but not to pursue software as a business. UNIX was provided “[a]s is, no support, payment in advance” Without support and bug fixes the growing community of UNIX users was forced to help themselves. They started to share ideas, information, programs, bug fixes, and hardware fixes. User groups were created wherever UNIX was introduced. Among them were universities in Australia, the United Kingdom, Germany and Japan. At ten years of age, UNIX was genuinely being used worldwide. The researchers at BTL had invented a great operating system, but the company’s management could not legally enter the business nor let the copyrights go. On the other hand, there was a growing community of users that wanted to use the system and the spirit that bonded UNIX users together in the 1970s and which continues today had its roots in an ’us-against-them’ attitude combined with a sense of humor. In the following years, users and developers of UNIX cooperated very closely:

“Something was created at BTL. It was distributed in source form. A user in the UK created something from it. Another user in California improved on both the original and the UK version. It was distributed to the community at cost [mainly for the distribution effort]. The improved version was incorporated into the next BTL release. There was no way the AT&T’s patent-and-licensing office could control this, and the system just got better and more widely used all the time.” [Salus94]

The “Seventh Edition” dated January 1979 was the first portable UNIX that ran on computers produced by DEC, IBM and Interdata. “Portability was born” [Salus94]. Unfortunately, the new license “prohibited the source code from being studied in courses” [Salus94] and many universities simply dropped the study of UNIX.

[12] A biography can be found at http://cm.bell-labs.com/cm/cs/who/doug/biography

Additionally, the development became more and more divergent. The Berkeley Software Distribution V3 (3BSD) emerged out of this situation.[13]

Over the years, many UNIX derivates followed. Some of them were based on original AT&T versions, others on the BSD line, but all of them required costly licenses from AT&T. In the beginning of the 1990s other systems like BSDI, 386/BSD and NetBSD[14] emerged that did not require these licenses. All this was accompanied by long-lasting legal battles and the users were confused and annoyed by the dozens of incompatible UNIX versions and an uncertain future. Today, most of these systems have disappeared from the market.

2.1.3 The GNU Project

The following quotations are taken from [DiBona99, The GNU Operating System and the Free Software Movement]. Starting his job at the MIT Artificial Intelligence Lab in 1971, Richard Stallman joined a "software sharing community that had existed for many years": Anytime you stumbled over an interesting program you asked the creator for the source code and read it, changed it or used parts of it to write a new program. They used the Digital PDP-10 series as their computer system at the time. Unfortunately, the model was discontinued in the early 1980s, all the created programs were unusable as they were written in assembler language and computers of that era had their own proprietary operating systems. "[Y]ou had do sign a nondisclosure agreement even to get an executable copy" A cooperative community was not possible anymore. According to Stallman, the rule behind proprietary software was "If you share with your neighbor, you are a pirate. If you want any changes, beg us to make them." The given situation forced Stallman to make "a stark moral choice" between the following three options:

1. He could "join the proprietary software world, signing nondisclosure agreements and promising not to help [his] fellow [programmer]". This would mean to spend his life "building walls to divide people".
2. Another option was to "leave the computer field" to avoid the misuse of his skills, but they would be wasted, too, and someone else would 'build the walls'.
3. Looking for a possibility to be a programmer and work on the establishment of a new cooperative community.

Stallman chose the last option and decided that the crucial component was an operating system, as you cannot use a computer without it. Fortunately, he had already been an operating system developer and therefore was the right person to do this job. Stallman describes the spirit behind this decision with the words of Rabbi Hillel:

"If I am not for myself, who will be for me?
If I am only for myself, what am I?
If not now, when?"

He chose to make his new system compatible with UNIX, which was owned and

[13] A more detailed account can be found at http://www.usenix.org/publications/login/1999-4/20years.html
[14] NetBSD http://www.netbsd.org

controlled by AT&T at the time. Therefore, he called his new project “GNU” which stands for the recursive acronym “GNU is Not UNIX” and coined the term “free software” in opposition to proprietary software. In order to speed up the project he decided to adapt existing components of free software wherever it was possible, e.g. TEX as text formatter and X as window system.

In January 1984, he gave up his job at MIT and focused entirely on the GNU project. To his delight, other people started to help him with his first project, the GNU Emacs[15] text editor, soon after. It became clear that simply producing software for the public domain would not serve the primary goal of the project to “give users freedom” as the programs could be slightly modified (e.g. by porting it to a specific machine) and turned into proprietary software. This led to the ’GNU General Public License’ (GPL). This license is based on a method that is called “Copyleft[16]”. It uses copyright law to keep software free.
In 1985, Stallman and other people engaged with the GNU project decided to found a tax-exempt charity called “Free Software Foundation” (FSF)[17] to handle the business area of the project like donations, selling copies of free software or offering other related services. Although the original intention was to complete the system first and then release it as a whole, people started using single finished components on the various compatible UNIX systems. This process had the advantage of improving the software and extending the user community, but “probably delayed completion of a minimal working system by several years”. Another problem was the projects choice to base Hurd, the ’heart’ of the system (the kernel), on the Mach[18] microkernel architecture because they had to wait for the Mach technology to be finished and their own part of implementation turned out to be much more difficult than expected. “[b]y 1990, the GNU system was almost complete; the only major missing component was the kernel.” The Hurd remains unfinished to this day.

2.1.4 Linux

Fortunately, Linus Torvalds, a Finnish university student, started to develop his own UNIX-compatible kernel “Linux” in 1991. Torvalds based the design on his kernel on a monolithic architecture, which was considered obsolete by Tannenbaum[19] and others. Torvalds provided Linux under copyleft and invited anyone to help him develop and improve the kernel. The developer community grew quickly and advances were made very fast. Thanks to his open development model and the growing role of the Internet, Linus Torvalds was able to work together with hundreds of developers. “Around 1992, combining Linux with the not-quite-complete GNU system resulted in a complete free operating system.” Heavily improved and extended versions of the Linux kernel and the GNU software tools have been released since then, millions of people have joined the GNU/Linux community and today it is the fastest growing operating system.

---

[15] GNU Emacs http://www.gnu.org/software/emacs/emacs.html
[16] The concept of Copyleft is explained in http://www.gnu.org/copyleft/copyleft.html
[17] FSF http://www.fsf.org
[18] The Mach project can be found at http://www-2.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html
[19] http://alge.anart.no/linux/history/linux_is_obsolete.txt

2.1.5 To the mainstream

The year 1998 started with a splash when Netscape Corporation announced to open source their flagship product Communicator in January. In the wake of this announcement, interest in Open Source soared, and mass media began to take notice. In May 1998, one of the first large-scale installations of Linux went live with the startup Google Inc. announcing their search engine. In June, Linux was offered certification from The Open Group to allow it to carry the UNIX name. In the same month, IBM announced its support for the Apache project, a web server software. Oracle and Informix pledged Linux support in early July, and Linus Torvalds appeared on the cover of Forbes magazine. In September 1998, Microsoft admitted Linux as a competitor for the first time. 1999 saw many Open Source companies going public, with spectacular stock rises after their IPO's. Open Source companies seemed to defy gravity for a while, and expectations towards open source skyrocketed. At various times Open Source was hyped as the solution to cure all ills in software engineering. It took the tech downturn of 2000 to reset expectations to a sensible level. Many of the former Open Source companies failed in the marketplace, and declared bankruptcy. Meanwhile, Open Source projects continue to be maintained, and new projects are being started every day. Reports of the death of Open Source have hence been premature.

2.2 Fundamentals

Open Source uses legal concepts to enable the special provisions it needs to encourage the unrestricted distribution of source code. It is therefore sensible to introduce these legal concepts. In addition, Open Source is a very broad term, and means different things to different people. The Open Source Definition elaborates on the central tenets of Open Source, and a look at various Open Source licenses highlights the crucial differences.

2.2.1 Intellectual Property Concepts

Intellectual property is defined as "intangible property that is the result of creativity, such as patents, copyrights, etc." [Oxford98, intellectual property] and grants individuals or groups certain control over valuable information. Due to the volatile nature of intellectual property, especially the ease with which it can be reproduced, it is granted special protection. The intent of these intellectual property laws is to protect the creator against unfair use of his material. The cost of copying a video tape is relatively small compared with several million dollars for the production of the actual movie. However, it still takes some resources. You need the equipment and the storage media for the transfer, which is normally significant compared with the regular price of a legal copy. Besides, the duplication process normally has some undesired side effects like a loss of quality. This is not the case with digital information. The duplicate normally cannot even be distinguished from the original, and the cost is often negligible. For this reason, intellectual property laws have an important effect on software value and thereby its development. There are no consistent international laws about intellectual property but most countries have treaties with each other in order to provide a minimum protection for their citizens. There are roughly four legal instruments to protect intellectual property.

2.2.1.1 Patents

"a government grant of the exclusive right to make, use, or sell an invention, usually for a limited period. Patents are granted for new and useful machines, manufactured products, and industrial processes and for significant improvements of existing ones. Patents are also granted for new chemical compounds, foods, and medicinal products, as well as to the processes for producing them. Patents can even be granted to new plant or animal forms developed through genetic engineering." [Britannica, patents] Governments grant patents as a payback for the disclosure of an invention. Therefore, patent law protects ideas, not specific expressions of ideas, as copyright law does. The usage of similar, independently achieved inventions is a violation of patent law. Patents are a powerful instrument to protect ideas. However, patents are threatening when they are misused, especially in the area of software. Let us assume that someone would be granted a patent on the basic arithmetical operations of addition and subtraction. Every calculation using these operations would be subject to the patent. Therefore, there are requirements and regulations for patents. In order to be granted a patent, the invention must be new, useful and non-obvious. An application must be filed, followed by a complex process to decide whether the patent should be granted. Once the patent is granted, the owner has the right to exclude others from making, using or selling his idea for a certain period in the country the patent was granted in. A patent remains valid between sixteen and twenty years in most countries. A patent has to be filed separately in each country although there are some treaties to simplify this process. [Britannica, patent]. While some software patents have been granted, their validity has been repeatedly challenged. [LPF91] argues that software patents are invalid, because software is based on mathematical ideas, which do not enjoy patent protection. Others have taken exception to the length of the patent period, which is considered too long for the software industry, as it hampers innovation. Opposing views hold that patents provide incentives to invent and allow the inventor to capitalize on the invention and recoup his investment.

2.2.1.2 Copyright

"the exclusive, legally secured right to publish, reproduce, and sell the matter and form of a literary, musical, dramatic, or artistic work." [Britannica, copyright] Copyright law was originally created for books. The purchase of a book buys you the actual physical medium (the paper) but not its contents. Instead, you obtain the legal right to use the contents with the purchase of the information media. Copyright laws that make it illegal to produce duplicates protect the text printed in the book. Copyright laws give the creator five exclusive rights over his work:

- Reproduction Right: The right to duplicate the work in fixed form.
- Modification Right: The right to modify the work to create something new. The result is called 'derivative work'
- Distribution Right: The right to distribute copies of the work to the public (e.g. sale or rental).
- Public Performance Right: The right to play, dance, act, or show the work at public place or to transmit it to the public.
- Public Display Right: The right to show a copy of the work at a public place or to transmit it to the public.

These rights have certain limitations.

- Idea: The idea that is expressed by the creative work is not protected. Copyright laws only cover a specific expression, not the ideas themselves. For instance, the recipes in a cookbook can be used without permission, but copying the text is prohibited.
- Facts: Analogous to ideas, the facts of copyrighted material are not protected either.
- Independent Creation: If an exact duplicate of the work is produced independently by someone else, it is not considered a copy and thereby does not violate copyright laws.
- Fair Use: The 'fair use' of creative work is not a violation of copyright laws even when it includes some duplication. Although the term is not precisely defined, it covers news reporting, research and criticism.

Several treaties guarantee international protection. The two most important ones are the Berne Convention (1886) and the Universal Copyright Convention (1952). Both agreements automatically grant foreign authors the same copyrights in the participating countries as local citizens. Today, most countries are members of at least one of these conventions [Britannica, copyright]. Copyright law has steadily increased the term of protection granted to rights holders, and efforts are underway to make the term unlimited. These laws are sponsored by the American record industry, and are challenged on constitutional grounds by [Lessig00] and others.

"We have entered a time when the code of our time can be written such that people who own intellectual property have the power — through law and through this code — to close off, to stop, to own an idea, and to make criminal, or at least extremely difficult, any use of that idea beyond the owners permission. We have entered a time when we can construct the world against nature."[20]

2.2.1.3 Trade Secrets

"information, including a formula, pattern, compilation, program, device, method, technique, or process that derives independent economic value from not being generally known and not being readily ascertainable and is subject to reasonable efforts to maintain secrecy." (Uniform Trade Secrets Act) Software source code is usually considered a trade secret and the binary code resulting from its compilation is protected by special copyright laws for computer programs like the Software Act of 1980 in the USA. Trade secrets are often invoked in the case of reverse engineering. Reverse engineering[21] determines the functions of hardware or software without access to the original source code or blueprints to allow for interoperability, and is often used to write drivers for hardware devices where no Open Source driver exists.[22]

[20] http://cyberlaw.stanford.edu/lessig/content/articles/works/lessigkeynote.pdf
[21] http://whatis.techtarget.com/definition/0,289893,sid9_gci507015,00.html
[22] http://www.troubleshooters.com/ucita/opensrc.htm outlines the dangers for Reverse Engineering with recent legislation in the United States.

2.2.1.4 Trademarks

"any visible sign or device used by a business enterprise to identify its goods and distinguish them from those made or carried by others. Trademarks may be words or groups of words, letters, numerals, devices, names, the shape or other presentation of products or their packages, color combinations with signs, combinations of colors, and combinations of any of the enumerated signs." [Britannica, trademark] While software cannot be protected by trademarks, packaging and slogans for commercial off the shelf software (COTS) are often trademarked. Trademarks have been used to fight Open Source alternatives to commercial software because the Open Source programs violated the trademarks of the commercial package.[23]

2.2.2 The significance of Contracts

Contracts as an enforceable promise are a very important instrument of the economic system. Two forms of contracts are especially relevant in the context of software.

2.2.2.1 License

Producers of creative work wanting to go beyond the provisions of copyright law provide special licenses for their work. In most cases, licenses place more restrictions on the use of the creative work than copyright law does. Licensing is a very important vehicle for regulating acceptable uses of software. It will be revisited later.

2.2.2.2 Non-Disclosure Agreement

The subject of non-disclosure agreements (NDA) are trade secrets. An NDA is the promise to keep the provided information secret. Non-disclosure agreements can be very complex, and are usually not acceptable for Open Source projects because the publication of source code would violate the terms of the non-disclosure agreement. The usual route being taken to accommodate both NDA and Open Source is to provide parts of Open Source software as binary modules only. Examples of this include driver software for advanced 3D functionality in the Linux kernel, and software to interface with software-based modems. Another reason for NDA is protection of trade secrets until a given date. Intel Corporation used this approach to protect its trade secrets for their new Itanium processor while a small group of programmers that had signed NDA's was preparing Linux support.

Most Open Source software is not written under explicit contracts, but rather started by individuals out of a need, or out of curiosity. The question of contracts, and especially licenses, only matters once a piece of software is distributed beyond its initial creator. Many OSP find it hard to settle on a license, because there are so many Open Source licenses to choose from, and they often only differ marginally. Choosing a license means to revisit ones notions about what Open Source means. What is Open Source?

2.2.3 Definitions of Open Source

To discuss Open Source, it is necessary to define the term Open Source in detail. Interestingly there is a broad variety of meanings attached to the term, and the discussion

[23] http://zdnet.com.com/2100-11-528438.html?legacy=zdnn

on definitions of Open Source is conducted ferociously. Two main camps can be identified within the Open Source community. Free Software, led by Richard Stallman, the creator of the GPL, and Open Source Software, led by the Open Source Initiative. While both movements share most of their practical goals, they differ philosophically. The Free Software movement believes it is unethical to use anything other than Free Software, while the Open Source movement is willing to accommodate proprietary software.

2.2.3.1 Free Software

The term Free Software was coined by Richard M. Stallman. Free Software requires the following four freedoms to be present [FSF01]:

1. The freedom to run the program, for any purpose.
2. The freedom to study how the program works, and adapt it to your needs.
3. The freedom to redistribute copies so you can help your neighbor.
4. The freedom to improve the program, and release your improvements to the public, so that the whole community benefits.

Stallman further clarifies the term:

Since "free" refers to freedom, not to price, there is no contradiction between selling copies and free software. In fact, the freedom to sell copies is crucial: collections of free software sold on CD-ROMs are important for the community, and selling them is an important way to raise funds for free software development. Therefore, a program which people are not free to include on these collections is not free software. Because of the ambiguity of "free", people have long looked for alternatives, but no one has found a suitable alternative. The English Language has more words and nuances than any other, but it lacks a simple, unambiguous, word that means "free," as in freedom--"unfettered," being the word that comes closest in meaning. Such alternatives as "liberated", "freedom" and "open" have either the wrong meaning or some other disadvantage.

2.2.3.2 Open Source Software

Open Source software as a term is both an informal term to refer to software where the source code is available, and a precise definition established by the Open Source Initiative [OSD02].

1. Free Redistribution
”The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.”

Rationale: By constraining the license to require free redistribution, the temptation to throw away many long-term gains in order to make a few short-term gains is eliminated. Without this clause, there would be a lot of pressure for cooperators to defect.

2. Source Code
"The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost – preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed."

Rationale: Obfuscating (disguising) source code defies the purpose of having access to the source code. Source code needs to be easily accessible and modifiable.

3. Derived Works
"The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software."

Rationale: The mere ability to read source code is not enough to support independent peer review and rapid evolutionary selection. For rapid evolution to happen, people need to be able to experiment with and redistribute modifications.

4. Integrity of The Author's Source Code
"The license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software."

Rationale: Encouraging improvement of the original code is advantageous, but users have a right to know who is responsible for the software they are using. Authors and maintainers have reciprocal right to know what they are being asked to support and protect their reputations. Accordingly, an open-source license must guarantee that source be readily available, but may require that it be distributed as pristine base sources plus patches. In this way, "unofficial" changes can be made available but readily distinguished from the base source.

5. No Discrimination Against Persons or Groups
"The license must not discriminate against any person or group of persons."

Rationale: In order to get the maximum benefit from the process, the maximum diversity of persons and groups should be equally eligible to contribute to Open Source. Therefore, the Open Source Definition (OSD) forbids any open-source license from locking anybody out of the process. Some countries, including the

United States, have export restrictions for certain types of software. An OSD-conformant license may not incorporate such restrictions itself.

6. No Discrimination Against Fields of Endeavor
"The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research."

Rationale: The major intention of this clause is to prohibit license traps that prevent open source from being used commercially.

7. Distribution of License
"The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties."

Rationale: This clause intends to forbid closing up software by indirect means such as requiring a non-disclosure agreement.

8. License Must Not Be Specific to a Product
"The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution."

Rationale: This clause prevents yet another class of license traps, which would prevent the reuse of source code outside the original product.

9. The License Must Not Restrict Other Software
"The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be Open Source software."

Rationale: Distributors of open-source software have the right to make their own choices about their own software. The GPL, arguably one of the most restrictive Open Source licenses, is conformant with this requirement. Software linked with GPL-licensed libraries only inherits the GPL if it forms a single work, not any software with which they are merely distributed. [FSF01a]

2.2.3.3 Licenses as a contentious issue

The philosophical differences between the Free Software and Open Source movements can ultimately be boiled down to different opinions about what constitutes a proper license for Open Source software[24]. Open Source implies a development methodology

[24] http://www.fsf.org/philosophy/free-software-for-freedom.html

that is shared by factions. Free Software implies a license designed to ensure the four freedoms noted above. The division between the two is often bitterly contested, both from inside and outside the communities. The holiest of holy wars are not fought over word processors, operating systems, or compilers. They are all about software licenses. In the end, it is up to the individual software author to pick a software license.

2.2.4 Open Source Software Licenses

In the course of the last few years, a multitude of Open Source licenses has been created. Most of the newer licenses are modified licenses to support a particular business model. Open Source licenses from Sun, IBM and Netscape Corporation belong to this category. It has become increasingly difficult to keep an overview on the various licenses, and to note their subtle differences. Both the GNU project and the Open Source Initiative have compiled extensive lists of licenses that qualify for the Open Source criteria. The list of licenses is steadily increasing[25]. Some of the better-known licenses include:

- GNU General Public License (GPL)
- GNU Library or 'Lesser' Public License (LGPL)
- BSD license
- MIT license
- Artistic license
- Mozilla Public License (MPL)
- Q Public License (QPL)
- IBM Public License
- MITRE Collaborative Virtual Workspace License (CVW License)
- Ricoh Source Code Public License
- Python license
- zlib/libpng license

At the time of this writing (June 2002) there were 43 different Open Source licenses acknowledged by the GNU project, and 32 acknowledged by the Open Source Initiative. A closer examination of selected licenses provides a good overview of the issues surrounding software licensing. Each license has their champions, as the examples indicate.

2.2.4.1 GNU Public License – GPL[26]

"The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users." [FSF91] The GPL is the most important open source license as most open source software is distributed under its terms. A major reason for its popularity is its viral effect, called Copyleft.

[25] news://news.gmane.org/gmane.comp.licenses.open-source.general conducts an ongoing debate about new Open Source licenses.
[26] http://opensource.org/licenses/gpl-license.php

Copyleft is a general method for making a program free software and requiring all modified and extended versions of the program to be free software as well. Copyleft says that anyone who redistributes the software, with or without changes, must pass along the freedom to further copy and change it. Empirical evidence suggests that Copyleft provides an incentive for programmers to add to free software. Important free programs such as the GNU C++ compiler exist only because of this. To Copyleft a program, first state that it is copyrighted; then add distribution terms, which are a legal instrument that gives everyone the rights to use, modify, and redistribute the program's code or any program derived from it, but only if the distribution terms are unchanged. Thus, the code and the freedoms become legally inseparable. Using the same distribution terms for many different programs makes it easy to copy code between various different programs. Since they all have the same distribution terms, there is no need to think about whether the terms are compatible. The Lesser GPL includes a provision that lets you alter the distribution terms to the ordinary GPL, so that you can copy code into another program covered by the GPL. Examples of software licensed under GPL include the Linux kernel and the K desktop environment (KDE).

2.2.4.2 GNU Lesser General Public License – LGPL[27]

This license differs from the GPL in one important aspect: it permits linking with non-free modules. It was originally designed for standard libraries to speed up the adoption of free software since such licensed libraries provide an opportunity for proprietary software to run in a free software system. It has since been argued by the Free Software Foundation that providing libraries under the terms of the LGPL weakens the cause of Free Software, and is discouraged by the FSF. [FSF99]

2.2.4.3 BSD License[28]

The BSD license and its relatives the X and Apache licenses are very different from the GPL and LGPL. These licenses let you do nearly anything with the software licensed under them. This is because the software that the X and BSD licenses originally covered was funded by monetary grants of the U.S. Government. Since the U.S. citizens had already paid for the software with their taxes, they were granted permission to make use of that software as they pleased. The most important permission, and one missing from the GPL, is that you can make BSD-licensed modifications private. In other words, you can get the source code for a BSD-licensed program, modify it, and then sell binary versions of the program without distributing the source code of your modifications, and without applying the X license to those modifications. This is still open source, however, as the Open Source Definition does not require that modifications always carry the original license. The BSD license and its variants, including the X license and the Apache license, are used in the FreeBSD operating system, the Apache web server, and the XFree86 window manager.

---

[27] http://opensource.org/licenses/lgpl-license.php
[28] http://opensource.org/licenses/bsd-license.php

2.2.4.4 Mozilla Public License[29]

The Mozilla Public Licenses (MPL) was developed by Netscape Corporation when it made its browser Netscape Navigator open source The MPL allows you to make modifications private. Many companies have adopted a variation of the MPL for their own programs. Among them are the Netscape Public License (NPL), the Interbase License, the Nokia Open Source License, and the IBM Public License.

2.2.4.5 Content Licenses

A recent development is the application of the Open Source licensing model to other forms of intellectual property. Adjustments have been made to deal with specific issues posed by other mediums. The creative commons[30] site lists the Design Science License, EFF Open Audio License, Free Art License, Free Music Public License, Open Content License, Open Music License, Open Publication License, Public Library of Science Open Access License, and the GNU Free Documentation License. Some book publishers have started to license books under the GPL, such as New Riders[31] and SAMS[32].

2.2.4.6 License Compatibility

Software systems consist of many components produced by different persons. As these modules are often distributed under different licenses, it is sometimes impossible to legally use them in combination. The problem is proliferating as many new companies start to participate in the Open Source community and create their own license instead of using an existing one. Since these conflicts only serve the opponents of Open Source software, such incompatibilities should be avoided as far as possible. Therefore, it is normally much better for all participating parties to use an existing license than to create yet another one. The GPL or LGPL provide the best protection against misuse and are compatible with most Open Source software already released as they are the most common licenses. Table 1 gives an overview of the various discussed licenses.

	Can be mixed with non-free software	**Modifications can be taken private and not returned to you**	**Can be re-licensed by anyone**	**Contains special privileges for the original copyright holder over your modifications**
GPL				
LGPL	X			
BSD	X	X		
NPL	X	X		X
MPL	X	X		
Public Domain	X	X	X	

29 http://opensource.org/licenses/mozilla1.1.php
30 http://www.creativecommons.org/concepts/
31 http://www.newriders.com/
32 http://www.samspublishing.com/

Table 1: Software Licenses compatibility

Underlying the notions expressed in the various Open Source licenses are fundamental assumptions about the nature of Open Source both as a movement and as a software engineering phenomenon. Software licenses are not the only legal issue facing Open Source. Software patents pose a host of issues.[33] Numerous approaches have been made to explain Open Source with a consistent theory. These theories are important tools for understanding Open Source.[34]

2.3 Theories about Open Source

No consistent, generally agreed on model for Open Source software exists yet. However, there are many theoretical approaches that try to explain the phenomenon of Open Source. Eric Raymond describes the Open Source community and its method of writing software in his book "The Cathedral & the Bazaar" [Raymond99b]. The title is an allegory: proprietary software production as the carefully planned building of a cathedral, Open Source software production as the chaotic interactions of the participants in an oriental bazaar. Strong, centralized management versus loosely related developers organized in several thousand seemingly independent projects. The following quotes from Raymond's book are helpful to understand the process of Open Source software development:

1. "Quality was maintained not by rigid standards or autocracy but by the naively simple strategy of releasing every week and getting feedback from hundreds of users within days, creating a sort of rapid Darwinian selection on the mutations introduced by developers." [Raymond99b, page 24]
2. "Linus Torvalds' style of development [is:] release early and often, delegate everything you can, be open to the point of promiscuity" [Raymond99b, page 30]
3. "Users are wonderful things to have, and not just because they demonstrate that you're serving a need, that you've done something right. Properly cultivated, they can become co-developers." [Raymond99b, page 36]
4. "It is not only debugging that is parallelizable; development and (to a perhaps surprising extent) exploration of design space is, too. When your development mode is rapidly iterative, development and enhancement may become special cases of debugging–fixing 'bugs of omission' in the original capabilities or concept of the software." [Raymond99b, page 51]
5. "I don't think it's a coincidence that the gestation period of Linux coincided with the birth of the World Wide Web, and that Linux left its infancy during the same period in 1993-1994 that saw the takeoff of the [Internet service provider] industry and the explosion of mainstream interest in the Internet. Linus Torvalds was the first person who learned how to play by the new rules that pervasive Internet made possible." [Raymond99b, page 63]

---

[33] http://petition.eurolinux.org/reference/economy.html

[34] Eric Raymond's seminal paper "The Cathedral and the Bazaar" has been mentioned as a decisive factor by many companies for their embrace of Open Source ideas.

6. “The [Open Source] world behaves in many respects like a free market or an ecology, a collection of selfish agents attempting to maximize utility which in the process produces a self-correcting spontaneous order more elaborate and efficient than any amount of central planning could have achieved.” [Raymond99b, page 64]
7. “[I]n a world of cheap PCs and fast Internet links, we find pretty consistently that the only really limiting resource is skilled attention. Open Source projects [...] die only when the developers themselves lose interest. That being the case, it’s doubly important that Open Source [developers] organize themselves for maximum productivity by self-selection–and the social milieu selects ruthlessly for competence.” [Raymond99b, page 71]
8. “A happy programmer is one who is neither underutilized nor weighed down with ill-formulated goals and stressful process friction. Enjoyment predicts efficiency.” [Raymond99b, page 75]
9. “It may well turn out that one of the most important effects of Open Source’s success will be to teach us that play is the most economically efficient mode of creative work.” [Raymond99b, page 75]

Raymond draws heavily on sociology and psychology to explain Open Source. Nikolai Bezroukov considers Raymond’s bazaar model as “a too simplistic view of the Open Source software development process”. Instead, he “tries to explore links between open source software development and academic research as a better paradigm [...]” and thinks it “should be better viewed as a special case of academic research.” [Bezroukov99b] Considering the Open Source phenomenon as academic research leads to the wide field of philosophy of science. Additionally, the combination of the “rapid Darwinian selection” mentioned in quote one and Bezroukov’s parallel to scientific research might lead to advances in the theory of software development, scientific research or creative work in general. Quote two epitomizes the belief that software should be released in short intervals, tested by as many users as possible. Quote four asserts that Brook’s law may be wrong, and five illustrates the importance of cheap and ubiquitous communication. Arguments six and seven produce economic explanations for Open Source, and eight and nine focus on psychological motivations. While “The Cathedral and the Bazaar” feels a bit like a hodgepodge of wildly different areas of science, wittily mixed together, it is by far the most popular and influential theory of Open Source. Other theories have been proposed, and will be revisited in chapter four. So far, we have lumped together the legalistic definition of Open Source with the meaning of Open Source as a community. It is not sufficient to look at software licenses to understand Open Source. The Open Source community is a vital part of Open Source, and has unique characteristics.

2.4 The Open Source Community

The Open Source community is influenced by mix of sociological, software engineering and economic forces. Open Source is a way of life, a way to develop software, and a new economic model. Each aspect of Open Source influences the others, and they are hard to separate. Some factors stand out though.

2.4.1 Sociology

Software does have an increasing impact on society. With the advent of the Internet, many things that used to take place in the real world are now 'moving' into cyberspace. People are trading, communicating, studying and being entertained on the Internet. Software not merely provides the capabilities to connect to the Internet; it controls it completely. This level of importance to completely control the experience for a software user is unprecedented. Recent developments like "Browser Wars[35]", "Spyware[36]" and "Censorware[37]" illustrate the leverage software has on the lives of individuals. The use of software thus raises new questions, especially the questions of observation and independence.

2.4.1.1 Observation

Software controls many valuable goods. It is essential that it operate correctly. Software, or rather its actions, needs to be observed to make sure it performs the desired operation without undesired side effects. Using software means placing trust in the intentions of the entity that wrote the software. It is usually infeasible to audit software for its actions. Most users of software do not have the necessary skills to do so, nor does software complexity lend itself to effortless observation. Some of these considerations include:

1. if the software does exactly what it should do
2. if the software only does what it should do
3. if all appropriate precautions are taken to control potentially dangerous processes
4. if data and resources are sufficiently protected against undesired actions of 'hostile' attackers, or data loss

Source code availability enables these observations, and is therefore an important consideration. [Stoltz99] argues that many government organizations require source code access to be able to verify the security of software. Closely related to the issue of observation is independence.

2.4.1.2 Independence

Independence means keeping full control over application data and information systems. As long as there are many different compatible software systems to choose from, application data can be migrated to other systems as required. Unfortunately, compatibility of software systems has turned out to be very weak. The replacement of installed software systems and successful data migration is difficult. The increasing complexity of software will make this process more difficult still. Choosing a software system is a far-reaching decision. Dependence on a software company and its future products means a loss of sovereignty. This dependence poses large business risks in the event of technical difficulties with software or the bankruptcy of the software vendor.

[35] The intense competition between Netscape Corporation and Microsoft Corporation for the control of the web browser market.

[36] Spyware is software installed by advertising companies that reports statistical data. This is often done without the knowledge or consent of the user.

[37] Censorware is software which is designed and optimized to prevent *another person* from sending or receiving information. Censorware has been known to infringe on civil liberties.

Open Source software and companies do not naturally provide better technology. However, sovereignty is preserved as you can leave your service provider at any time and keep using the same software. Availability of the source code allows for outsourcing of maintenance in the event that the original vendor ceases to maintain a product. Iceland and other nations with too few inhabitants for a profitable software market have already experienced such problems when they were looking for native language support, even though they were willing to pay for the translation effort[38]. Many countries and companies do not like to depend that much on private companies and therefore look for alternatives. Open Source software might turn out to be the best choice.[39]

2.4.2 Software Engineering

There are several thousand Open Source projects (OSP)[40], with widely varying levels of sophistication. Some projects use sophisticated strategies of software engineering while others just start working without any planning at all. Although there is no general method of producing software in OSP, their special open distribution policy raises several interesting questions. The special considerations for software engineering are far reaching, and will be revisited later in more detail. For now, an overview of how Open Source influences various software engineering aspects will provide an introduction.

2.4.2.1 Security

It has been claimed that Open Source software is inherently insecure due to the openness of the source code.[41] The question if Open Source software is more secure than proprietary software has never been settled with proof. It will most likely never be settled because both the notion of security is vague, and it is hard to gather relevant data. A few issues are worth consideration though.

1. Security by obscurity is a flawed concept, and has been repeatedly dismissed by the scientific community[42].
2. Trusting the creating party without appropriate observation of the result requires a large amount of trust in each single person and party that has access to the (secret) source code. Only one defector would destroy the protection provided by secrecy. Besides, the loss of protection might not become known for a long time.
3. The scientific process has long relied on peer review for quality control. [Bezroukov99b] establishes that the Open Source process can be seen as a special type of academic research.

2.4.2.2 Reliability

[Schmidt01] argues that "Open Source projects make it easier to address... quality assurance, end-user confidence and good will, and the coherency of system-wide

---

[38] http://www.menntamalaraduneyti.is/mrn/mrn-engAfrit/mrn-eng.nsf/888deacc045556e80025664300576c77/7f8f562e9fa1a40a002566ff0065cee0?OpenDocument

[39] A new development, TCPA (Trusted Computing Platform Alliance) might seriously undermine these choices. http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html

[40] Sourceforge contains thousands of projects alone: http://sourceforge.net

[41] See http://www.linuxworld.com/linuxworld/lw-1998-11/lw-11-ramparts.html&e=42

[42] http://www.counterpane.com/crypto-gram-0205.html#1

software and usability properties .. compared with traditional closed-source approaches to building software." A study by the University of Wisconsin about the "Reliability of UNIX Utilities and Services" remarks: "[T]he reliability of the freely distributed GNU and Linux software was surprisingly good, and noticeably better than the commercially produced software.[43]" Open Source software is not more reliable in general, but evidence suggests it can be, and often is.

2.4.2.3 Reusability

The evolutionary process that is made possible by permissive licenses helps to shape reusable software components. It is reasonable to consider the distribution of the source code and the granted right to modify the software as a stimulating factor for the reusability of software components. [Price99]: *"Other Open Source software provide frameworks for interprocess communication and networking (Linux, Bind and Sendmail) which allow programs such as Ghostview to be coupled with Web browsers and servers (Mosaic, Lynx and Apache) to enable the rapid initial growth of the World Wide Web. Programs such as TeX and Gimp aid creation of Web documentation and images. Since most of these packages were never intended as Web components, the argument could be advanced that the Web constitutes the world's most successful example of software reuse in the large."*

2.4.2.4 Compatibility and Standards

Why should developers use provided standards? The members of OSP tend to choose the most effective option they can find without much effort in advance. Standards are very handy for this strategy as they contain a lot of theoretical work and developers can concentrate on the actual task instead of spending most of their time thinking about theoretical frameworks and a fundament to provide consistency and compatibility. [Spangler01]: *"Reconciliation of different interpretations of the standard, clarifications and extensions can be realized by discussion systems and some generally accepted conciliators of Open Source projects. As long as companies have a commercial interest in selling solutions based on the protocol, it makes economic sense for them to jointly support its common implementation."* Examples for standards work in the Open Source world abound..

> **Free Standards Group**[44]
> "A non-profit corporation organized to accelerate the use and acceptance of open source technologies through the application, development and promotion of standards."
>
> **Linux Standard Base**[45]
> "The goal of the Linux Standard Base is to develop and promote a set of standards that will increase compatibility among Linux distributions and enable software applications to run on any compliant Linux system. In addition, the LSB will help

---

[43] Barton P. Miller et al.: 'Fuzz Revisited: A Reexamination of the Reliability of UNIX Utilities and Services', February 2000, ftp://grilled.cs.wisc.edu/technical_papers/fuzz-revisited.ps
[44] http://www.freestandards.org/
[45] http://www.linuxbase.org

coordinate efforts to recruit software vendors to port and write products for Linux." Current members are Caldera Inc., Corel Corporation, the Debian Project, delix Computer GmbH, Enhanced Software Technologies Inc., IBM, LinuxCare, Linux for PowerPC, Mandrake Soft, Metro Link Inc., Turbolinux Inc., Red Hat Software, Software in the Public Interest Inc., SuSE GmbH, VA Linux, WGS Inc., and SGI.

Linux Internationalization Initiative[46]
"Li18nux is a voluntary working group, consisting of Linux and Open Source related contributors who are working on Globalization, a combination of Internationalization and Localization. The organization was formed in August 1999. The ultimate goal of the organization is to achieve software/application portability and interoperability in the International context for Linux and other Open Source projects. Its activities are focused on the internationalization of a core set of APIs and components of Linux distributions to achieve a common Linux environment. This will allow an internationalized Linux application to be executed regardless if different flavors of distributions are used. The results of the working group will be open to everyone, and be proposed for adoption to the Free Standards Group"

X Desktop Group[47]
"The X Desktop Group is a free software project to work on interoperability and shared technology among desktop environments for the XWindow System. The most famous X desktops are GNOME and KDE."

Filesystem Hierarchy Standard[48]
"FHS defines a common arrangement of the many files and directories in UNIX-like systems (the filesystem hierarchy) that many different developers and groups have agreed to use. [...] The implementors of Linux distributions and other UNIX-like operating systems, application developers, and Open Source writers use the FHS specification. In addition, many system administrators and users have found it to be a useful resource. FHS [...] is currently implemented by most major Linux distributions, including Debian, RedHat, Caldera, SuSE, and more."

Austin Common Standards Revision Group[49]
"The Austin Common Standards Revision Group (CSRG) is a joint technical working group established to consider the matter of a common revision of ISO/IEC 9945-1, ISO/IEC 9945-2, IEEE Std 1003.1, IEEE Std 1003.2 and the appropriate parts of the Single UNIX Specification. The approach to specification development is 'write once, adopt everywhere', with the deliverables being a set of specifications that will carry both the IEEE POSIX designation and The Open Group's Technical Standard designation, and if adopted an ISO/IEC designation.

[46] http://www.li18nux.net
[47] http://www.freedesktop.org
[48] http://www.pathname.com/fhs/
[49] http://www.opengroup.org/austin/

The new set of specifications will form the core of the Single UNIX Specification Version 3, with delivery in [the second quarter] 2001"

Debian Policy Manual[50]
"This manual describes the policy requirements for the Debian GNU/Linux distribution. This includes the structure and contents of the Debian archive, several design issues of the operating system, as well as technical requirements that each package must satisfy to be included in the distribution."

[50] http://www.debian.org/doc/debian-policy/

2.4.3 Economy

The economic motivations of OSP have been the subject of many studies. [Bessen02], [Ghosh98], [Goldhaber97], [Hippel02], [Iannicci02], [Il-Hom02], [Lancashire01], [Lerner01], [Weber00], [Wegberg00]. Many of these studies involve new notions of economic activity, and their conclusions are tentative at best. That said, economic consequences due to the use of Open Source can be observed.

2.4.3.1 Total Cost of Ownership

Total Cost of Ownership (TCO[51]) is a type of calculation designed to assess both direct and indirect costs and benefits related to the purchase of any IT component. Calculation of TCO includes the factors as outlined by figure 1.

System Preparation
Several additional components are usually required in order to get new software running, e.g. hardware devices, infrastructure or other software.

Operation Efficiency
All phases in the life cycle of a software component require additional resources like time of human actors or hardware devices. The efficiency by which these resources perform operations strongly influence total costs. Degradations in efficiency could be caused by non-intuitive or too complex user interfaces, incompatibility of data formats or software defects.

Failures
Software failures can be very costly. Consider the case of software that controls machinery in a factory, or air traffic control software.[52]

Training
Most software is complex enough to require training, either through printed documentation, classes or hands-on coaching.

Service
Service includes support, helpdesk and other complimentary processes.

Updates
Software updates correct errors or enable new functionality. Most software vendors charge a fee for updates.

Purchase
The purchase grants permission to use a copy of the software. It usually does not grant ownership.

[51] For notes on the origin of the term, consult
http://search390.techtarget.com/sDefinition/0,,sid10_gci342316,00.html
[52] http://catless.ncl.ac.uk/Risks contains many accounts of software failures.

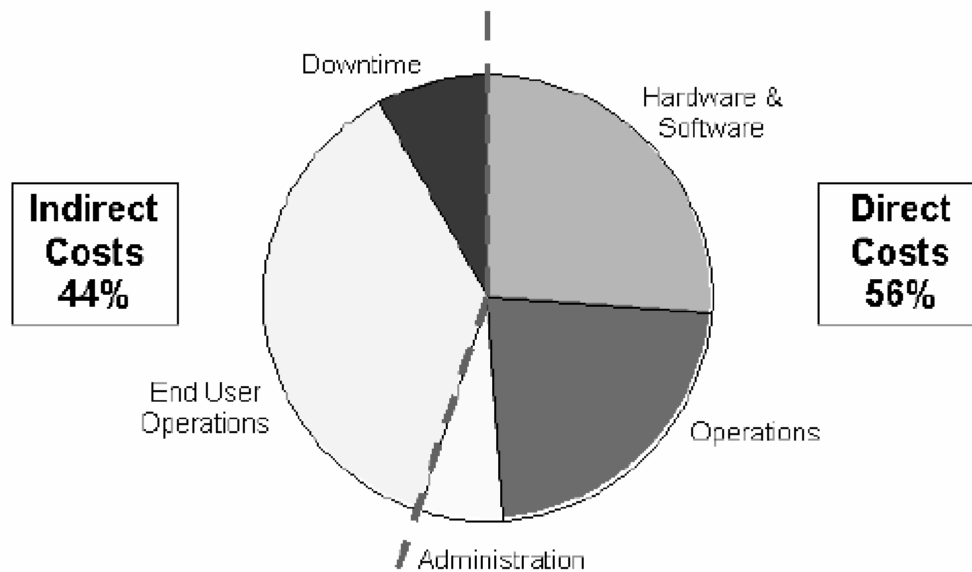

Figure 1: A typical TCO Calculation (Source: UNISYS Corporation)

2.4.3.2 Giving Away Software For Free

One of the economic incentives for giving software away is the "Loss-Leader" model. [Raymond00] In this model, Open Source software is used to create or maintain a market position for proprietary software that generates a direct revenue stream. In the most common variant, Open Source client software enables sales of server software, or subscription/advertising revenue associated with a portal site.[53]

2.4.3.3 Making Money with Open Source Software

Many companies have tried to make money with Open Source, despite claims about the "gift culture as a result of material abundance" [Raymond98b] that try to negate the economic interests of Open Source participants. Open Source success stories, [Miller02] being a good example, lead the way. Successful business models for Open Source are based on one or several of the following components:

Software Distribution
Distributors simply sell copies of Open Source software. This business is based on the idea that the regular user of Open Source software is willing to pay a small amount for convenient access to the software. These businesses sell the packaging, not the software, and are protected by trademark and copyright laws from copycats.[54]

Service

[53] Netscape Communications, Inc. was pursuing this strategy when it open-sourced the Mozilla browser in early 1998.
[54] Examples include RedHat (http://www.redhat.com) or SuSE (http://www.suse.com)

There are many different services in the software field as described in the section about TCO above. Examples are support, training or simply paid bug fixing.[55] International Business Machines (IBM) reportedly recouped its investment of $1 billion into Linux within one year with service contacts.[56]

Hardware
As hardware devices cannot be used without appropriate software, vendors usually spend a remarkable amount of financial resources on the production of driver software. It is the usual procedure to make this software available for free, but without source code. However, more and more companies also start to participate in Open Source projects to assure the compatibility and support for their products.[57]

Information
Books, magazines and news services provide information about Open Source software for a reasonable price, e.g. nicely printed manuals.[58]

Another way to look at Open Source is to look at the output of the community, Open Source software. Software is built in projects, and it makes sense to take a closer look at how Open Source Projects (OSP) operate.

2.5 Open Source Projects

The Open Source community is very hard to investigate as an abstract social phenomenon. It is difficult to determine who is a part of it and who is not.[59] Fortunately, OSP can be observed and analyzed due to their presence on the Internet and their publicly available communication. Clearly, there are as many ways to run an OSP as there are projects, but some common threads emerge nevertheless. What is an OSP?

Definition
Any group of people (or sole individuals) developing software and providing their results to the public under an Open Source license constitute an Open Source project (OSP).[60]

The major productive assets of OSP are developers. Developer is a wide term, and need not be confined to programmers, but can also include documentation writers, graphic artists and others. To understand Open Source means to understand developers. Who are those developers?

---

[55] Sendmail Inc (http://www.sendmail.com) and CodeWeavers (http//www.codeweavers.com)
[56] http://www-1.ibm.com/services/e-business/linux_8.html
[57] Adaptec Inc. (http://www.adaptec.com) and Nvidia Inc. (http://www.nvidia.com)
[58] O'Reilly and Associates (http://www.oreilly.com) employs several world class Open Source developers, like Larry Wall (Perl).
[59] [Edwards00] provides a nice overview of the community.
[60] The Open Source Definition [OSD02] is used here to define Open Source licenses.

2.5.1 Developers

Open Source developers come from a wide variety of backgrounds [Ghosh00]. Some of the major groups are educational institutions, companies, governments and individuals.

2.5.1.1 Educational Institutions

Universities and other institutions produce a lot of software for educational and research purposes. Although some parts of it become proprietary software, many developments are released under legal terms that conform to the Open Source definition.[61] Universities produce a steady supply of talented programmers that have not yet entered the work force, and are thus able to devote significant amounts of time to Open Source. Some of the best-known OSP started in academia, such as Linux, BSD and Apache.

2.5.1.2 Research Institutions

Research is often closely associated with educational or public institutions by their work, personnel or financing. Additionally, many projects are based on intensive collaboration between many different organizations. Therefore releasing the research results under a permissive license is often a natural choice because it allows any involved party to use them. Besides, such a license is sometimes also a condition for financial sponsorship.[62] Research and Open Source do have an affinity for each other, as [Bezroukov99a] points out.

2.5.1.3 Software Distributors

The producers of Open Source software distributions normally participate in various capacities in several OSP. Their motivation is normally increases in their user base, as users often demand specific features that are not yet available, e.g. sound support or a word processor.[63] Another scenario is consulting engagements where software distributors develop targeted functionality for a client and integrate the work back into their main product line.

2.5.1.4 Commercial Companies

Aside from distributors, any other business based on Open Source software might participate in some capacity in OSP. Today, this includes most of the large information technology companies, e.g. IBM, Intel or Hewlett Packard. Those companies are often motivated by a desire to commoditize infrastructure, such as the operating system. In the 1980s many of those companies had produced their own variant of UNIX, which resulted in a fragmented market for UNIX, and allowed Windows NT to gain market share. Over the years, Linux has started to displace HP-UX, Ultrix, AIX, Irix and SCO Unix, and today all UNIX versions do support Linux binaries.[64]

---

[61] Not all universities have a sensible open source policy though. http://www.fsf.org/philosophy/university.html discusses the issues in more detail.

[62] IBM AlphaWorks releases many research projects as Open Source. http://www.alphaworks.ibm.com/

[63] RedHat Inc. founded its Advanced Development Labs for this purpose. http://www.labs.redhat.com/

[64] http://www.atai.org/softwarewar.png provides a humorous account.

2.5.1.5 Corporate Users

Considering the enormous financial resources that many companies or governmental administrations spend on their software systems (usually several million dollars for licenses alone), sponsoring OSP is often much cheaper than paying the license fees for proprietary software. [Dinkelacker01] defines Corporate Source as "the application of Open Source concepts, perspectives and methodologies within the corporate environment – i.e. "open" to all developers behind the firewall." Another important element for corporations is the business risk of running critical infrastructure on software without source code access. Interestingly, this requirement is increasingly being heard and answered by proprietary software companies, too.[65]

2.5.1.6 Private Users

Anyone using Open Source software is interested in improving it as it gives him a direct benefit. For this reason, many users participate one way or another in OSP. Since many of them are also working in the IT business [BCG02], their participation is often essential for a project. Most OSP are primarily run by volunteers who invest an average of 14 hours a week of their time on it. [BCG02] Open Source was started by talented individuals, and is overwhelmingly run by persons with programming skills to this day. Integrating non-programmers has always been a challenge, and will become ever more important, as the user base for Open Source grows.[66]

2.5.1.7 Governments

Many governments have started to encourage Open Source for critical infrastructure. A good example is the sponsoring of GNU Privacy Guard (GPG), an Open Source implementation of the PGP protocol, by the German Ministry of Economics[67]. The case for Open Source in government is one of national security, sovereignty, and a willingness to foster local software engineering talent. In particular, countries of the third world have realized that Open Source makes sense. For instance, the author is involved in a project to create a software industry in Bahrain, a country of the Persian Gulf. Faced with the prospect of oil reserves that will run out, Bahrain is looking into ways to start new industries. It has settled on software as a promising avenue, lacking natural resources or industry. To jumpstart the industry, Bahrain has decided to embrace Open Source. Their first foray was to run their national elections (the first ever) with PostNuke.[68]

Another defining aspect of OSP is life cycles. The range of projects reaches from the planning stage to mature projects that have been stable for decades, but are still being maintained. What defines OSP life cycles?

2.5.2 Project Lifecycle

OSP are organic. They do not follow strict pattern for releases, and oscillate between different cycles. For the purposes of illustration, a typical life cycle would be:

---

[65] http://www.microsoft.com/licensing/sharedsource/default.asp
[66] http://www.creativecommons.org is an interesting approach to foster much broader contribution.
[67] http://www.gnupg.org
[68] http://www.bahraintoday.net

1. Someone has an unmet need and tries to devise solutions for it. [Raymond99b] calls this "scratching an itch"
2. That person asks some friends and colleagues what they know about the issue. Some of them may have similar problems, but probably no solution either.
3. All interested persons start to exchange their knowledge on the topic and thereby create a vague picture about the central issue of the group.
4. Interested people who are willing to spend some resources on finding a solution for the issue create an informal project.
5. The project members work on the issue until they achieve some satisfactory result.
6. They make their work publicly available at a place where many people are able to access it. They may announce their project at places like mailing lists, newsgroups or online news services.[69]
7. Other persons recognize some of their own concerns in the project and are interested in a convenient solution, too. Therefore, they review the projects result (e.g. by using it). As they look at the issue from a different perspective, they suggest improvements and even might join the project.
8. The project grows and a lot of feedback helps to get a better understanding of the issue, and possible strategies to solve it.
9. New information and resources are integrated into the research process. The solution grows, and addresses the issue in ever better ways.
10. The research cycle is closed and returns to stage five.
11. The project's community is established and will react to future changes the same way it emerged originally.

A common classification of the various stages of an OSP is Planning, Pre-Alpha, Alpha, Beta, Stable, Mature.

Planning
No code has been written, the scope of the project is still in flux. The project is but an idea. As soon as tangible results in the form of source code appear, the project enters the next stage.

Pre-Alpha
Very preliminary source code has been released. The code is not expected to compile, or even run. Outside observers may have a hard time to figure out the meaning of the source code. As soon as a coherent intent is visible in the code that indicates the eventual direction, the project enters the next stage.

Alpha
The released code works at least some of the time, and begins to take shape. Preliminary development notes may show up. Active work to expand the feature

[69] The Linux project started with such a message.
http://groups.google.com/groups?selm=1991Oct5.054106.4647%40klaava.Helsinki.FI

set of the application continues. As the amount of new features slows down, the project enters the next stage.

Beta
The code is feature-complete, but retains faults. These are gradually weeded out, leading to software that is ever more reliable. If the number of faults is deemed low enough, the project releases a stable version, and enters the next stage.

Stable
The software is useful and reliable enough for daily use. Changes are applied very carefully, and the intent of changes is to increase stability, not new functionality. If no significant changes happen over a long time, and only minor issues remain, the project enters the next stage.

Mature
There is little or no new development occurring, as the software fulfills its purpose very reliably. Changes are applied with extreme caution, if at all. A project may remain in this final stage for many years before it slowly fades into the background because it has become obsolete, or replaced by better software. The source code for mature projects remains available indefinitely, however, and may serve educational purposes.[70]

The distribution of project stages among thousands of OSP is interesting. As of July 3, 2002 the two biggest sites for OSP displayed this distribution of project stages:

Stage / Project	Sourceforge.net	Freshmeat.net
Planning	9006 (33%)	75 (1%)
Pre-Alpha	6003 (22%)	380 (3%)
Alpha	5329 (20%)	1510 (14%)
Beta	6318 (23%)	3385 (31%)
Stable	4813 (18%)	4952 (47%)
Mature	530 (2%)	781 (7%)

Table 2: Open Source projects by lifecycle stage

Most OSP on Sourceforge.net are in the planning stage. This can be explained by the ease of setting up a project. A new project can be set up in minutes, and very often, little thought is given into the repercussions of starting an OSP. Success rates do seem very low. Assuming the Stable stage means success, only 20% of Sourceforge.net project are successful. On the other hand, most OSP are leery to label themselves Stable or Mature due to their focus on "getting it right." Some projects take this view to a silly extreme by

[70] The original UNIX source is still used to study operating system design around the world, after 30 years.

remaining at version .99 for years, as if the magical 1.0 could only be reached asymptotically, or not at all.

Every OSP is different. What does a typical OSP look like? No one knows, as there is no such thing as a typical OSP. There are some well-known OSP that must be doing something right, considering they are in use all over the world, by millions of people. What do these projects look like?

2.5.3 Example OSP

The following examples of large OSP show the breadth and depth of efforts in the Open Source community. The project descriptions originate from the projects themselves.

2.5.3.1 XFree86[71]

"XFree86 is a freely redistributable implementation of the X Window System that runs on UNIX(R) and UNIX-like operating systems (and OS/2). The XFree86 Project has traditionally focused on Intel x86-based platforms (which is where the '86' in our name comes from), but our current release also supports other platforms. One of our current goals is to increase the range of platforms that XFree86 runs on."

2.5.3.2 KDE[72]

"KDE is a powerful graphical desktop environment for UNIX workstations. It combines ease of use, contemporary functionality and outstanding graphical design with the technological superiority of the UNIX operating system. KDE is an Internet project and truly open in every sense. Development takes place on the Internet [...]. No single group, company or organization controls the KDE sources. [...] All KDE sources are [...] subject to the well-known GNU licenses. [...] KDE has developed a high quality development framework for UNIX, which allows for rapid and efficient application development. Applications developed with this framework include KOffice, a full-featured Office Suite, KDevelop, a C/C++ IDE (Integrated Development Environment), and many others."

2.5.3.3 The Gimp[73]

"The GIMP [...] is a freely distributed piece of software suitable for such tasks as photo retouching, image composition and image authoring.[The GIMP home page] contains information about downloading, installing, using, and enhancing GIMP [and] serves as a distribution point for the latest releases, patches, plugins, and scripts. We also try to provide as much information about the GIMP community and related projects as possible."

---

[71] XFree86 Project, Inc http://www.xfree86.org
[72] K Desktop Environment http://www.kde.org
[73] GNU Image Manipulation Program http://www.gimp.org

2.5.3.4 Apache[74]

"The Apache Project is a collaborative software development effort aimed at creating a robust, commercial-grade, featureful and freely-available source code implementation of an HTTP (Web) server. The project is jointly managed by a group of volunteers located around the world, using the Internet and the Web to communicate, plan, and develop the server and its related documentation. These volunteers are known as the Apache Group. In addition, hundreds of users have contributed ideas, code, and documentation to the project."
"In February of 1995, the most popular server software on the Web was the public domain HTTP daemon developed by [NCSA]." Development of that software had stalled and a small group of webmasters gathered together for the purpose of coordinating their private changes. They put together a mailing list and shared information space for the core developers. "By the end of February, eight core contributors formed the foundation of the original Apache Group". "[W]e added all of the published bug fixes and worthwhile enhancements we could find, tested the result on our own servers, and made the first official public release (0.6.2) of the Apache server in April 1995." After a new design for the server architecture, extensive beta testing, many ports to several platforms, new documentation, and many additional features, Apache 1.0 was released on December 1, 1995. "Less than a year after the group was formed, the Apache server passed NCSA's [software] as the [number one] server on the Internet. The survey by Netcraft shows that Apache is today more widely used than all other web servers combined.[75]"

2.5.3.5 Linux[76]

One of the most famous OSP is the Linux kernel. Linus Torvalds started Linux in 1991 and has been leading it since then. The source code has a size over 100 Megabytes, and its growth rate is increasing. [Wheeler01] did a source code analysis of a popular Linux distribution (RedHat)[77] that concluded. "In particular, it would cost over $1 billion to develop this GNU/Linux distribution by conventional proprietary means in the U.S. (in year 2000 U.S. dollars). Also, Red Hat Linux 7.1 includes over 30 million physical source lines of code (SLOC), compared to well over 17 million SLOC in version 6.2. Using the COCOMO cost model, this system is estimated to have required about 8,000 person-years of development time (as compared to 4,500 person-years to develop version 6.2). Thus, Red Hat Linux 7.1 represents over a 60% increase in size, effort, and traditional development costs over Red Hat Linux 6.2". Linux is a clone of the operating system UNIX. It aims for POSIX compliance, having all the features of a modern fully-fledged UNIX, including true multitasking, virtual memory, shared libraries, on-demand loading, shared copy-on-write executables, memory management, and TCP/IP networking. Linux was first developed for x86-based PCs (386 or higher). It has been ported to Alpha AXP, Sun SPARC, Motorola 68000, MIPS, PowerPC, ARM, PA-RISC, IA-64, SuperH and dozens of embedded processors." The size of the Linux project is unprecedented in the history of software development. At times, thousands of

[74] Apache HTTP Server http://www.apache.org
[75] http://www.netcraft.co.uk/survey/
[76] Linux Kernel http://kernel.org
[77] RedHat is one of the largest Linux distributions. It can be found at http://www.redhat.com

programmers have volunteered their time and effort in the daily development of numerous components and functions that comprise the operating system. According to one estimate, the project has involved over 40000 people worldwide.[78]

2.5.3.6 Mozilla[79]

On January 23rd, 1998 Netscape Communications announced that they would release a version of their product 'Netscape Communicator' as free software and the source code was released to the public on March 31st. They named the new project 'Mozilla'. "Mozilla is an Open Source web browser, designed for standards compliance, performance and portability. [Netscape Communications] coordinate the development and testing of the browser by providing discussion forums, software engineering tools, releases and bug tracking."

2.5.3.7 PostNuke[80]

The PostNuke project was founded in May 2001 to develop an excellent content management system (CMS) with a focus on community features. The project grew very quickly to several hundred active contributors. In its first year, it was downloaded over 500'000 times, and tens of thousands of sites operate on PostNuke. "PostNuke is a free multi-lingual CMS written in PHP[81] and licensed under the GNU General Public License. PostNuke software dynamically manages website content submitted through browsers. PostNuke allows administrators to work dynamically within a structured environment to rapidly deliver diverse content including articles, links, news, job boards, frequently asked questions, resume listings, dynamic headlines, weather, file download areas, and much more. PostNuke reduces web site development costs by introducing sophisticated administration tools and services which separate form, function, content, and design."

A look at Open Source is incomplete and misleading without comparing it to classical methods of software engineering. Open Source is by far no panacea, and its advantages are often offset by disadvantages. What are the areas where Open Source may open new venues for software development, and where could it learn from classical ways? How does Open Source stack up to classical approaches, and when is it appropriate to use Open Source methodologies? When are you better advised to stick to traditional methods?

3. Open Source Projects versus Classical Projects

What is an OSP? What is a classical project? The lines between them are blurry. Both OSP and classical projects are heavily influenced by technological advances, new forms of collaboration, and changing work ethics. It is helpful to name these trends, as they will likely be influential in the future, and may lead to entirely new forms of software projects. A definition of both OSP and classical projects needs to take their contributing resources, their means of coordination and their structures into account. Equipped with

[78] [Raymond99b]
[79] Mozilla.org http://www.mozilla.org
[80] PostNuke http://www.postnuke.com
[81] PHP http://www.php.net

such a definition, one may then attempt to compare these two approaches, and identify their strengths and weaknesses. It is ultimately helpful to contrast the major attributes of OSP versus classical projects in a matrix for a convenient overview.

Why do OSP exist? Are OSP a logical result of advances in technology, or did they happen due to social forces at work? If trends influenced the emergence of OSP, what were those trends?

3.1 Underlying Trends

Technological trends have influenced computer science since its beginnings, and there is no reason to believe these trends will not play influential roles in the future. Technological innovation happens on many fronts at once, and on many levels. Out of many forces that continue to shape the field of computer science, new communication technologies, advances in computer hardware, the move to increasingly higher-level languages and the emergence of ubiquitous standards seem to be particularly fruitful for the field of software engineering.

3.1.1 New Communication Technologies

In 1945, Vannevar Bush wrote a very influential paper: "As We May Think"[82] Bush argued for the creation of a memex: "A memex is a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory." The invention of the Internet, and later on applications running on top of it, laid the groundwork for realizing the "memex". The first innovation with widespread appeal to come out of the Internet was email.[83] Email enabled researchers to quickly and inexpensively exchange scientific arguments. What began with work-related subjects soon reached out into personal matters as well, with scientists establishing discussion groups for thousands of topics. Thusly, newsgroups were born.[84] In 1991, the invention of the worldwide web by a researcher at CERN, Tim Berners-Lee enabled the publication of research materials with pictures and, most importantly, hyperlinks. Hyperlinks are elements in an electronic document that link to another place in the same document or to an entirely different document. Hyperlinks are essential for Hypertext[85]. Hypertext, a term coined by Ted Nelson, is the implementation of the ideas first articulated by Bush. The emergence of hypertext, albeit in a much more primitive form than envisioned by Nelson[86], enabled new forms of online collaboration. For the first time in history, it became feasible to access a large part of human knowledge at low cost, and independent of location. Originally conceived by Berners-Lee to facilitate scientific exchange, the web soon outgrew its initial focus on academia, and spread to become a mass medium. In its wake, it invigorated economic growth in countries that used web technology to

[82] http://www.theatlantic.com/unbound/flashbks/computer/bushf.htm
[83] A history of email is given at http://www.vicomsoft.com/knowledge/reference/email.history.html
[84] More information about newsgroups can be found at http://www.learnthenet.com/english/html/26nwsgrp.htm
[85] Hypertext is a term created to describe non-linear writing in which you follow associative paths through a world of textual documents. The most common use of hypertext are the links on World Wide Web pages.
[86] http://ted.hyperland.com/buyin.txt

increase productivity. So great were the expectations that a new economy was declared and, fueled by an infusion of trillions of dollars, thousands of new companies were formed to capitalize on the web. The initial excitement about the web turned out to be premature, but neither is the web obsolete. The original vision of Berners-Lee has since been expanded to encompass the notion of the Semantic Web.[87]

"The Web was designed as an information space, with the goal that it should be useful not only for human-human communication, but also that machines would be able to participate and help. One of the major obstacles to this has been the fact that most information on the Web is designed for human consumption, and even if it was derived from a database with well defined meanings (in at least some terms) for its columns, that the structure of the data is not evident to a robot browsing the web. Leaving aside the artificial intelligence problem of training machines to behave like people, the Semantic Web approach instead develops languages for expressing information in a machine processable form."

3.1.2 More powerful Hardware

Concurrent with the rise of communication technologies, computer hardware made very large leaps. Computing power rose roughly in accordance with Moore's Law[88], i.e. doubled every twelve months for the past two decades. This led to a democratization of computing, as more and more computing resources became available for individuals. Hobbyists formed clubs to tinker with computers. This in turn led to many innovations from outside the traditional academic communities. Computer science was arguably one of the few disciplines in recent history where amateurs were able to contribute to a significant extent. Advances in hardware technology spurred growth in new fields of application. While early usages of computers were limited to computation, the focus shifted to communication as the most important application for computers in the last decade. Computing power has become cheap and plentiful, so plentiful that it is increasingly embedded in everyday devices. It is no longer feasible to understand a computer system in all its details. Growing complexity is met with ever-increasing levels of abstractions. What used to be specified explicitly is generalized, and ultimately abstracted away. New hardware at affordable prices simplifies costly processes. Producing movies in a basement becomes feasible, and the scope of computer technology increases constantly. More and more individuals are subjected to computers every waking minute, making them more familiar, and allowing easier entry into the field of computer science that used to be dominated by mathematically minded persons. Advances in hardware create new demand for software that utilizes the new capabilities.

3.1.3 Higher-level Languages

As computing power grew, programmer time was substituted by CPU time. Programmer time had become scarce. It was no longer reasonable to optimize applications for maximum performance at the expense of additional efforts by programmers. The ever-increasing complexity led to the introduction of higher and higher levels of abstraction. Each new generation of languages sacrificed some execution efficiency for gains in

[87] http://www.w3.org/DesignIssues/Semantic.html

[88] Gordon Moore predicted in 1965 that computing power would double every 12 months. His prediction was astonishingly accurate. http://www.intel.com/update/archive/issue2/feature.htm

expressive power. As computing power continued to rise, compiled languages made room for interpreted languages. Interpreted languages were far slower, but allowed for easy changes in the source code of a program and immediate execution of the changes. The concept of virtual machines abstracted the hardware itself to insulate dependencies and make software run on a wide variety of hardware platforms. Programming languages also became easier to master, introducing more and more individuals to programming. Scripting languages[89] were developed to harness the facilities of operating systems, and making them available to casual programmers. A notable example of scripting languages is Visual Basic[90]. Conceived by Microsoft, it allowed millions of non-programmers to write simple applications, while making computer science professionals squeal at the horrible code and sloppiness it produced. Scripting languages became very popular despite all their weaknesses, and most OSP are based on scripting languages such as Perl[91], Python[92], PHP[93], and Bash[94].

3.1.4 Ubiquitous Standards

Standards were the laughingstock of the information technology industry for the longest time. Andrew Tanenbaum95 once remarked. "The nice thing about standards is that there are so many of them to choose from." The growth of a worldwide communication infrastructure, especially the Internet, placed a new emphasis on standards though. Standards were no longer optional, but critically required to enable full participation in the economies of scale the Internet offered. [Chuang98]: "With the digitization and networking of information, many herald the arrival of the era where the marginal cost of information dissemination is virtually zero. The ease with which data may be duplicated and transported over the global Internet leads pundits to proclaim that 'bandwidth is free' and 'distance is dead'" The Internet established a new approach to standardization. Instead of designing standards in a lengthy process that often took years (the traditional method used by the ISO[96] and others), Internet standards would often evolve to a useable state in mere months. Chiefly responsible for this remarkable success are the IETF[97] and W3C[98] committees. The process of RFC (Request for Comment) favors incremental approaches. "In outline, the process of creating an Internet Standard is straightforward: a specification undergoes a period of development and several iterations of review by the Internet community and perhaps revision based upon experience, is adopted as a Standard by the appropriate body, and is published." Strong and ubiquitous standards play into the hands of OSP, as [Vallopillil98] notes. The widespread use of standards has allowed for the rapid spread of new applications across millions of appliances, and created enormous markets overnight. To target a standard means to reach millions, or hundreds of millions of potential users, increasing the potential rewards for successful software manifold.

---

[89] http://compilers.iecc.com/comparch/article/95-03-064 has an interesting discussion on the term.
[90] http://www.iessoft.com/scripts/vbhistry.asp gives a historical account.
[91] Perl http://www.perl.org
[92] Python http://www.python.org
[93] PHP http://www.php.net
[94] Bash http://www.gnu.org/software/bash/bash.html
[95] Andrew Tanenbaum http://www.cs.vu.nl/~ast/
[96] International Standards Organization http://www.iso.ch
[97] Internet Engineering Task Force http://www.ietf.org
[98] Worldwide Web Consortium http://www.w3.org

These technological trends were reinforced by societal changes that happened in parallel. As access to information became universal and cheap, barriers to entry evaporated in many knowledge-based industries, intensifying competition. Consumers were able to make informed decisions, and a single Internet search could reveal potentially sensitive information about any entity. Openness was no longer an option, it became increasingly the norm. The public appreciation for being open grew, and Open Source software entered many new areas. At the same time, open access to information threatened traditional cartels founded on scarcity of information. Empires built on exclusive distribution of information (the recording industry, for instance) began to crumble, and the incumbents fought technological progress by introducing new legislation designed to stop innovation. This legislation increasingly threatens Open Source.[99]

Open Source manifests itself through the actions of OSP. What are characteristics for OSP? Which elements need to be present for OSP to prosper? How would a definition of OSP look like?

3.2 Defining Open Source Projects

Open Source has many characteristics, and remains poorly understood. It makes sense to attempt a definition by looking at major characteristics such as resources, coordination and structures. These arguably appear in all OSP, and are broad enough to apply. First, why is Open Source called Open Source?

Open Source as a way of life

To understand OSP, one should ponder the significance of the term “Open Source” first. For many participants, Open Source is a philosophy[100]. Open Source participants adopt a rationale for their actions different from that of their peers in the proprietary software world. Open Source participants as well as researchers into the phenomenon often bring up the concepts of community, gift, shared ideals and so forth[101]. The motives that drive Open Source participants clearly determine the development of the phenomenon. Unfortunately, monetary data is not available for Open Source communities except at the anecdotal level, such as the salary levels for “star programmers”[102]. The main reason for this lack of data is the fact that monetary transactions are largely non-existent within the Open Source production process.

Open Source as a way of work

There are people who make a living not being paid for software they write. This raises many questions. What sort of a living do such people make; who pays them and why; what benefits accrue to employers of such people who pay for what may

[99] The website of the Electronic Frontier Foundation (http://www.eff.org/) contains about a dozen such laws.

[100] http://www.fsf.org/philosophy/philosophy.html

[101] Refer [Hannemyr99],[Kishida01],[Weber00]

[102] Several well-known open source personalities report economic benefits stemming from their status. Examples include Linus Torvalds, who went to work for chip design company Transmeta, or Alan Cox, who works from home for RedHat Corporation.

be freely available. Another set of questions is related to the non-monetary economy that results from the production of goods without payment – if they are not receiving cash for their software, what, if anything, do they receive instead? Moreover, how much? What do they give for access to other free software? The value of money as a measuring tool is immeasurable. Lacking this, alternatives must be found in order to identify power structures; ownership and effective control of systems; vulnerabilities and dependencies in the "economy" surrounding Open Source systems. To illustrate: Microsoft's position in the economy is easy enough to analyze, since its property and influence is quantified in monetary terms. The position of Linus Torvalds or the Apache team is hard to quantify even within the Open Source community (let alone in the economy at large), even if a system of definable reputation is accepted as a way of doing so[103]. The further development of measurement and modeling methods is, therefore, crucial to the understanding and better functioning of Open Source, and its integration into the monetary economic system.

Open Source as a way of software development
Finally, Open Source is a method of developing software. It is often quite different from the formal development methodologies of proprietary software companies. The element of collaborative authorship is much discussed[104]; less so is the element of competition between programmers.[105] Open Source faces several challenges as a software development technique: intellectual property rights[106]; software quality and reliability[107]; version control and responsiveness to environmental changes[108]; credit and liability management. Open Source as a development technique has been studied in far better detail than other approaches based on economic or sociologic models. This is only to be expected, as most people whose expertise lies closest to Open Source software development (as participants themselves) do not have a significant expertise in economics, but can share their development experiences.

OSP need resources to function. What constitutes OSP? Which resources are critical? Given these resources, how are they coordinated? Which structures emerge in OSP?

3.2.1 Resources

Although OSP differ from classical projects in their resource needs, they still do require them, despite overblown accounts of "virtual organizations". OSP do not exist in the void, but rather use the infrastructure of the Internet (both hardware and software), require people to staff the project and funding to keep the services and infrastructure running.

[103] [Goldhaber97] and [Kaisla01] attempt such a measure of reputation.
[104] See [Feller00], [Jones00], [Mockus02],
[105] [Hippel02], [Lerner01]
[106] [FSSF02b], [Kelsey99], [Rasch01]
[107] [Halloran02],[Kuan02], [Schach02],[Schmidt01]
[108] [Asklund01],[Cubranic99], [Hippel02]

3.2.1.1 Software

Very few OSP start from scratch by developing all the required software themselves. Reuse is the norm. [Price99]:" Open Source tools represent both reuse in the small and reuse in the large. Examples of reuse in the small include Emacs and Perl (which provide the custom languages for creation of reusable libraries and channels for their distribution) and flex (which provides generative reuse)." This introduces dependency problems for these projects, but allows them to leverage existing work. Three types of software reuse can be distinguished:

Tools
Software developers use tools to produce software. Tools used in the creation of software are not required at runtime. Examples include compilers, editors, linkers, configuration management packages and issue tracking software.

Components
Components are necessary for the operation of a software package but are developed separately. Examples include the C libraries, windowing toolkits, header files and binary data for hardware drivers.

Integrated Code
This category comprises source code that is directly integrated, usually by copying. This kind of reuse is affected by the license that the originating software is released under, and is unique to OSP as intellectual property laws usually forbid the inclusion of copyrighted material.

Most OSP rely on freely available tools for their work, as licensing fees would be an impediment for large groups of developers. Individuals may use commercial tools for their own use though. Some commercial tools have a liberal licensing so that they can be used for Open Source work.[109] An example of a restricted tool that is nevertheless used in the Open Source world is Java. The Java license does not conform to the Open Source Definition, but allows free distribution. OSP do prefer to use Open Source software both for their toolset, their components and especially for integrated code. While this rule does not hold universally, it is still a useful characterization. [Ghosh02] conducts a fascinating empirical study that looks at authorship information, clusters of authorship and code dependency between packages: "Dependencies can be identified through automatic scanning. This provides a high level of detail regarding dependencies (i.e. at a function call level) well beyond the present purposes of analysis. Such detailed analysis would be computationally exceptionally hard to perform for 30,000 software packages!" An earlier survey by [Ghosh00] found 12706 developers authoring OSP, a size of 1.04 Gigabytes for the source code of the projects (roughly 25 million lines) and 3149 identifiable OSP.

[109] The BitKeeper configuration management system is now being used for Linux kernel development, although the decision remains controversial. http://www.bitkeeper.com/

3.2.1.2 Hardware

There are no special hardware characteristics for OSP beyond what is required for remote collaboration over the Internet. Thus, only hardware that has Internet connectivity is of interest for OSP. In essence, there are hardware resources at the edge of the network (individual resources) and at the center (shared resources). Central, shared resources require funding by some entity. Many OSP rely on individuals or corporations to provide them with hardware resources. A notable example is Sourceforge.net, which as of July 5th, 2002 hosts 42835 projects with 448'419 developers.[110] The wide availability of cheap and powerful hardware has allowed individuals to provide servers for OSP that would have been prohibitively expensive a few years ago.

3.2.1.3 People

OSP participation is voluntary in almost all cases. Very few persons are paid to develop Open Source software, and traditional chains of command do not apply. This has serious implications for OSP. No one can be forced to work on tasks that do not interest him. Correspondingly, tiring work such as fixing bugs or writing documentation is often neglected in favor of writing new code (which is considered more rewarding by most developers). Project leaders will occasionally ask for specific tasks to be completed, and rely on the motivation of individuals to take on these tasks. It is considered more important that tasks are being done at all than them being done perfectly or within a given time. However, failures and irresponsible actions always have consequences for participants as they lose social status. Scheduling, or indeed any form of project management, is hard to do in OSP, and is faced with many challenges. [Welch00] mentions that "laziness, ignorance, fear and denial prevent using good management, because people can get by for awhile using bad management and learning good communication skills to avoid accountability for reliance on bad management." Co-dependant tasks often stall because a precursor task is not yet completed. This leads to duplication of effort and frustration within a project team. Cynical observers could claim that nothing gets ever done in an OSP unless someone is bothered enough to fix it. Sometimes this pain is actively used to steer the course of a project. Linus Torvalds, the project leader for the Linux kernel, repeatedly stalled large parts of kernel development to increase the pain level just enough for someone to step up and assume responsibility.[111]. A crucial element of OSP is motivation. Lacking financial incentives, OSP participants are driven by motivational factors such as peer esteem, a sense of wonder, a desire to gain new skills, and to have fun. OSP should work to foster these motivations by recognizing contributions, creating a learning environment, and making it easy for new participants to get accustomed with the culture of an OSP. Epistemic communities [Edwards00] are a good model for the social and psychological interactions between OSP participants, and will be revisited in chapter five.

3.2.1.4 Funding

Most OSP do not have financial resources at their disposal. Instead, they get by with donations of resources from their participants. Money is usually not an important

[110] http://sf.net
[111] http://www.wideopen.com/reprint/573.html contains an account of Torvalds leadership.

resource. Those OSP that do have income, be it through donations or the sale of related products like T-shirts do have trouble disbursing the funds. The distributed nature of OSP makes it very hard to establish a fair division of funds among participants, and this in turn means that only the most active contributors to an OSP benefit from funding. Potential uses for funding are exhibitions, congresses, project meetings in real life or hardware resources. Various funding models for OSP have been proposed. [Bessen02] argues that Open Source is a more efficient way to provide customized software, and attracts investments from companies. [Roberts00] and [Rasch01] describe markets where companies can pay for the development of functionality, and [Schmidt02] argues for government subsidies for OSP. All proposals have in common that they make the case for efficiency gains through Open Source. It is argued that OSP allow harnessing economies of scale to a larger degree than proprietary software because their openness makes it easier for standards to spread. Public funding for Open Source is a very recent concept, and remains poorly understood. The European Commission commissioned a study in 2002 [EC02] that concluded that funding Open Source made sense given that a few conditions were met.

- A reasonable number of persons "sharing the same problem"
- Initial but flexible repartition of "ownership / leadership" between diverse persons, from diverse organizations;
- Documentation everywhere;
- A roadmap (navigate into the code as in a web site);
- A common trunk easily understandable + functional modules / no monolithic code;
- Software organized in many relatively small pieces of code (in order to facilitate individual ownership);
- Clearly identify (and declare) what parts are "mature/stable" and what parts are "to improve" (according to the "release often" principle);
- Launch permanent discussion forums on requirements, objectives, and priorities for further development.

As it turns out, these conditions are prerequisites for successful OSP anyway, and thus place no additional hurdles for public subsidies.

3.2.1.5 Service and Infrastructure

Members of OSP are usually spread all over the world. They depend on a communication infrastructure, in most cases, the Internet. Collaboration requires central infrastructure. Fortunately, many participants do have access to such infrastructure in order to subsequently donate that infrastructure to their OSP. Examples of central infrastructure include email accounts, home pages, storage capacity or entire computer systems. OSP rely on such donated services to a large degree, and are liable to the loss of that donation if the responsible person decides to withdraw his donation. One glaring liability is Sourceforge.net, which hosts almost 40'000 projects. The impact of a termination of service by Sourceforge.net is probably the single biggest liability for OSP. OSP are lean in the sense that they piggyback on existing infrastructure. Very often, OSP participants use Internet access for their OSP that they had anyway. OSP infrastructure and required

services are thus very much falling through the cracks, and it is exceptionally hard to account for these hidden costs. As long as OSP are able to leverage devices at the edge of the Internet and avoid excessive centralized resources, they do not create significant costs. The biggest cost for centralized infrastructure is very often the bandwidth cost for the downloads of popular projects. Future peer-to-peer file sharing networks may be able to mitigate these costs.[112]
Assuming the necessary resources for an OSP are in place, they need to be coordinated somehow. The distributed nature of most OSP presents special challenges for coordination.

3.2.2 Coordination

Poor coordination is one of the main factors inhibiting the growth of a successful OSP. Only very few individuals in any given project are knowledgeable enough about the finer points of a project to assist with the coordination of work. These persons are usually overwhelmed with work. Empirical evidence suggests that OSP participants value the coordination efforts of the project leadership (assuming for a moment that there is a leadership[113]) as it allows them to apply their work more effectively. [Malone93] developed a framework for studying coordination that will be used in this section. Coordination tries to resolve dependencies, which can be categorized into shared resources, producer-consumer relationships, simultaneity constraints and task dependencies. [Malone93]

3.2.2.1 Shared Resources

"Whenever multiple activities share some limited resource (e.g., money, storage space, or an actor's time), a resource allocation process is needed to manage the interdependencies among these activities." [Malone93] The resources of OSP can be roughly categorized into human actors, support systems and information.

Human Actors
Unlike paid employees, no one can force OSP participants to perform a task. There is no authority with these powers. All project members decide for themselves how they spend their time and their resources. Nevertheless, OSP need coordination. All coordination efforts rely on persuasion and are therefore complex social interactions. These social interactions can take on many forms, and it is hard to generalize them. Coordination happens by social conventions that are particular to a project and need to be learned as they are usually not documented.[114]. Social interactions are crucial for OSP, and require closer attention. They will be revisited later.

Support Systems
It may be necessary to allocate the limited resources of support systems (e.g. storage space on a OSP web server). Fortunately, most OSP mainly utilize the resources at the edge of the network, i.e. personal computing equipment of OSP

[112] http://www.openp2p.com/
[113] [Raymond99b] essentially argues that most management is a waste of time.
[114] [Edwards00] provides a good account of these hidden conventions and rules.

participants. To the extent that OSP use centralized resources, their usage must be monitored and excessive utilization prevented. This can be achieved by replicating critical infrastructure.[115]

Information
Ensuring timely access to information is crucial for project success. While information is an unlimited resource, provisions need to be taken to emphasize important information. Often, there is too much information that demands attention, and participants are overwhelmed[116]. Another area that requires coordination efforts is sensitive information, such as information about security vulnerabilities.[117]

3.2.2.2 Producer/Consumer Relationships

"[...] a situation where one activity produces something that is used by another activity." [Malone93] Producers may be part of a different project, or other team members within the same project. All OSP are both consumers (of third party functionality, or services) and produce services and goods for other projects to consume. These interactions are very complex, and resemble the food chain observed in nature. [Ghosh02] studies these relationships by a survey of a large body of source code (over 1 Gigabyte), and determining producers and consumers.

Relations to other projects
Many OSP utilize many components produced by other projects. Usually such reuse is based on stable releases, and involvement with the producing project is kept to a minimum. Reuse may introduce additional dependencies and can lead to delays when the producer has not yet released an important release. The consuming project has little influence over the producer, and cannot usually demand target dates for releases. Nevertheless, many developers participate in several projects over time and much information and experience is exchanged. These relations have been called the Open Source community. Most producer-consumer relationships are transient though. A typical OSP may rely on dozens of other projects, and profound relations are just not feasible.[118]

Relations inside the project
As a project grows sufficiently large, dependencies develop within the project. Bottlenecks arise because tasks have interrelationships. Due to the voluntary nature of most OSP work, these bottlenecks cannot be resolved with orders, but need persuasion. Peer pressure is a strong motivator to resolve such bottlenecks. OSP take architectural precautions to reduce these dependencies. Code is kept as modular as possible, and project contributors often work fairly independently, without a large amount of communication with other project members.

[115] http://kernel.org/ set up a sophisticated international mirroring system.
[116] The term "data smog" has been coined to describe the phenomenon.
http://www.valt.helsinki.fi/comm/argo/anet00/data.htm
[117] CERT has a vulnerability disclosure policy: http://www.kb.cert.org/vuls/html/disclosure
[118] [Cubranic99],[Ghosh02] and [Mockus02] analyze project interactions.

Additionally, the users of a project are consumers, and often do not contribute anything back to the project. When the balance between consumers and producers gets out of hand, ugly discussions break out in OSP. Key contributors often feel that they produce far more than they consume. This leads to burnout and frustration, and may endanger the future of OSP if they are unable to reconcile these differences in their project. The author found that the project he was involved in, PostNuke, had many demanding consumers, and few producers. The consumers felt entitled to the fruits of the producers' labor without contributing themselves, and were very vocal about their wishes. After a reassessment of the situation, he felt that to comply too much with the wishes of random users meant risking his own motivation, and subsequently worked only on areas of the project he had a personal interest in. It had been established that most of the tens of thousands of users of the project only cared for the result, and that by producing results without worry for the cries of vocal users, he could actually deliver more value to the project, and maintain his motivation.

3.2.2.3 Simultaneity Constraints

"Another common kind of dependency between activities is that they need to occur at the same time (or cannot occur at the same time)." [Malone93] Simultaneity constraints happen daily in OSP. Two prominent examples are real-time communication, and document modification.

Real-Time Communication
Meetings, conferences, phone calls, discussions on Internet relay chat channels[119] and other kinds of real-time communication must take place at the same time for all participants. This can be very challenging for projects that span 20 time zones. It is also hard for participants in these online meetings to maintain a high concentration because they are subjected to their individual environments during the meeting.

Document Modification
Most OSP use configuration management software to avoid conflicts arising from simultaneous edits of their source code. Configuration management can be very simple, like a locking system for files that are being edited. They can also be very complex, like the distributed configuration management system used for the Linux kernel.[120]

3.2.2.4 Task Dependencies

"[...] a group of activities [that] are all 'subtasks' for achieving some overall goal." [Malone93] Several persons working on the same task must coordinate their activities to make sure that the results of their activities integrate. Three different methods to handle this dependency can be identified. Not all are equally suitable for OSP.

---

[119] IRC provides a way of communicating in real time with people from all over the world. http://www.irchelp.org/

[120] [Asklund01] discusses configuration management for OSP

Top-Down Goal Decomposition
"[...] an individual or group decides to pursue a goal, and then decomposes this goal into activities (or subgoals) which together will achieve the original goal." [Malone93] This approach uses significant resources to produce a plan of future actions; subtasks are assigned to team members. This approach is often used in classical projects, but poses problems for OSP.

1. There is no real management with enforcement power.
2. Plans become outdated quickly (persons leaving a project, new technologies, etc.).
3. Many problem domains have no clear-cut solutions
4. Team members have little knowledge about the skills of their coworkers.

Bottom-Up Goal Identification
"[...] several actors realize that the things they are already doing (with small additions) could work together to achieve a new goal." [Malone93] Considering the large number of OSP in the world and the philosophy of free knowledge exchange they are based on, it is only natural that new projects emerge from already existing ones and new goals are achieved by combining ideas and approaches from different projects. This is one of the fundamental ideas of the Open Source philosophy. Sometimes, developers discover that they are working on the same problem as another project is, and merge their projects.

Concurrent Task Processing
Another possibility to coordinate goals and their subtasks is to let people work concurrently. This method is similar to the bottom-up method. The advantage of concurrent task processing is that participants have a lot of freedom, keep the control over their resources and still improve their productivity by collaboration. A common procedure is the following: Everyone starts working, produces some results, the group compares achieved (partial) solutions, discusses them and starts a new concurrent working cycle with the parts they agreed on as the most promising ideas. Configuration management software supports this approach with the concept of branches that allow parallel changes to the same files with the prospect of a merge at a later time.

Most OSP start with ad hoc coordination. As they grow, this becomes increasingly unwieldy, and poses a barrier to entry for new participants who might not be familiar with the informal ways of coordination and information exchange in a particular project. Interestingly, larger OSP are often closer to the cathedral model than the bazaar model. Freewheeling chaos is a very nourishing ground for an initial exchange of ideas, but shows deficiencies once a project enters the Alpha or Beta stage. Thus, OSP do have structures.

3.2.3 Structures

Analyzing the structure of OSP provides many insights into their mode of operation. Structure may be the area where OSP diverge most from classical projects, as research by [Dafermos01], [Kishida01], [Kuwabara00] and [Madey02] indicates. Structure appears on multiple levels. OSP conduct activities, its members assume roles, activities are related to each other, and can be subsumed into larger processes.

3.2.3.1 Activities

While activities in OSP are varied, they broadly fall into the categories of communication, coordination, documentation, decision-making and software production.

Communication

Project information is distributed among its participants. In order to work together and to achieve results, participants need to exchange a lot of information between each other. This creates a strong need for efficient communication facilities. Efficient and effective communication needs to answer these questions:

1. Which knowledge should be transferred?
2. Who should provide the knowledge?
3. Who should receive it?
4. How should knowledge be transferred?

Knowledge should be transferred by communication. Communication means to share the relevant knowledge of the project between all participants to support their work. Questions one to three are very hard to answer generically. Every OSP handles these questions somewhat differently. All OSP do share the results of their work, inviting peer review. The source code of their project often constitutes the most reliable knowledge store. In addition to results, knowledge about the process to achieve results is often exchanged. This knowledge could be preliminary ideas, requests for comments, or observations. Requested knowledge should be transferred (first question) from everyone who thinks to have something to contribute to the specified topic (second question) to the participant who announced the information request (third question). Active requests for information indicate which areas of a project draw interest and help to filter the amount of knowledge available. Documentation efforts on the other hand are more driven by a goal to be complete rather than to answer specific questions. Writing documentation tries to answer anticipated questions, and does not pay off immediately. Often, documentation is compiled from answers to frequently asked questions (FAQ).[121]

Decision-Making

Setting up goals or priorities for the future, choosing between contradicting ideas, changing a project's policies are all part of decision making. One person, a group or all members of a project might be involved in the decision making process.

[121] http://www.faqs.org/faqs/ contains thousands of FAQ documents.

Decisions can be taken formally following rules the members agreed on before or informally by a simple conversation. Decisions might be considered final or provisional. Collecting good data to make informed decisions is a time-consuming task. Decision-making draws heavily on other activities. For instance, the collection of required information is part of the communication activity. Many decisions are made subconsciously because the number of decisions each individual makes every day is enormous. These inherent decisions and hidden assumptions can lead to communication problems within a project. On the other hand, it is impractical to decide minor issues in the group. Balancing individual and collective decisions is therefore critically important. Several projects have evolved sophisticated decision-making processes. Due to the chatty nature of online communication, it is sometimes hard to reach conclusions. The Apache project has developed a voting mechanism[122] that boils down decisions to a choice of +1 and -1.

Coordination

All activities of an OSP need some kind of coordination as they all depend on their context including other activities. Many of these dependencies are hidden. Coordination and communication are very closely related activities, and communication is the major means available to OSP for coordination. Since OSP are usually staffed by volunteers, they normally do not have a strong management authority that is capable of a centralized coordination process. Most dependencies are resolved by social conventions and interactions. OSP participants are usually aware of the coordination bottleneck and take action to prevent dependencies wherever possible. This may mean that only high-level objectives are coordinated, and individuals set sub-goals on their own. [Cubranic99]: "Developing 'social contracts' among participants in computer-mediated communication is often more effective than looking for a technological solution"

Documentation

The goal of documentation is to provide any interested party with a detailed description of the development process and the product. Documentation is a sort of asynchronous communication. The primary goal of a project is to produce software. Documentation activities decrease productivity at first, but become valuable in the long term. Documentation of the development process should not be confused with documentation of the product. Process documentation provides insight into the development process and rationales for decisions. Process documentation is especially valuable for OSP as is provides common ground for new team members and establishes enough contexts to make communication more effective. Some benefits of process documentation include:

- Manuals: It is much easier to write manuals for a product if you have access to some of the assumptions that went into the software. In addition, many OSP have a rather technical audience, which is interested in extending the software and adapting it to their needs. For these users,

[122] http://httpd.apache.org/dev/voting.html

process documentation is more valuable than documentation about the product itself.

- Bugs: It is helpful to have a detailed change history to track down bugs. Additional documentation can be used to investigate less obvious bugs.

- Compatibility: Good process documentation helps to enforce consistency. Established processes make it easier for contributors to develop additions to a project, and make it easier to integrate it later.

Process documentation is usually longer-lived than the underlying source code because implementations may change, but fundamental ideas stay the same. Documentation is a scarce resource in any software project, so any little piece of it helps, even simple log messages. Successful OSP have established policies that make the best use of process documentation. The Linux kernel project now has facilities to collect author commentary into release notes. This has made the release notes much more informative.

Software Production
Software production is the most important activity and all others only serve to optimize it. Software production can be described as a Meta activity, as it encompasses other activities. Some projects have their software production activity defined precisely; others only have vague goals. Some people like to collaborate on software production without splitting the activity into smaller sub-tasks, which requires more communication, others want to work more independently and have clear boundaries between their responsibilities. Depending on perspective, whole OSP may be activities within still larger OSP. Software production happens in groups that vary from one to several hundred or thousands of participants. Some OSP take an engineering approach to software production, others advance their software more due to brownian motion than any discernible design decisions. That said, software production is at the very center of OSP activity, and warrants a closer look later.

As part of performing activities, OSP participants assume various roles. These roles are helpful to understand why activities occur, and reveal the decision-making structures in a project.

3.2.3.2 Roles

The activities of OSP are normally not planned much in advance, and there is no central authority with the power to assign responsibilities to participants. Therefore it is rather difficult to identify abstract roles in specific projects and even harder to find suitable roles for OSP in general. Most projects have very fluid boundaries between roles. Developer, Manager, Maintainer, Administrator and Commenter are roles that appear in most projects in some form.

Developer
A developer is responsible for implementing the projects goals. He participates mostly in the production and documentation activities.

Manager
A manager directs a project. Decision-making and coordination are his major activities. A manager leads by virtue of competency, not fiat, and is therefore very often a developer.

Maintainer
A maintainer keeps track of issues with released components of a project and is responsible for their resolution. His major activities are coordination and communication. Being a maintainer entails having profound knowledge about a project, its components and its architecture, and is therefore mostly a job for seasoned developers.

Administrator
An administrator is responsible for a smooth operation of the project by maintaining project resources like centralized servers, communication facilities, and configuration management systems. Administrators may not have a glorious role, but they are essential for keeping a project running. Often, administrative roles are filled by less technical people for whom these tasks are a very good venue to contribute in a project.

Commenter
Anyone who provides some kind of feedback and does not implement it himself in the production activity is a commenter. Developers usually working on a different part of the same project, other participants, and maintainers of other projects or regular users are possible commenters.

Participants are somewhere in between these abstract roles and perform actions that belong to several different roles. In turn, most roles are occupied by several participants. Developers and maintainers provide the required software, managers manage a project and commenters review all activities and give feedback. The administrator is the invisible agent who supports all these activities. Figure 2 illustrates these roles.

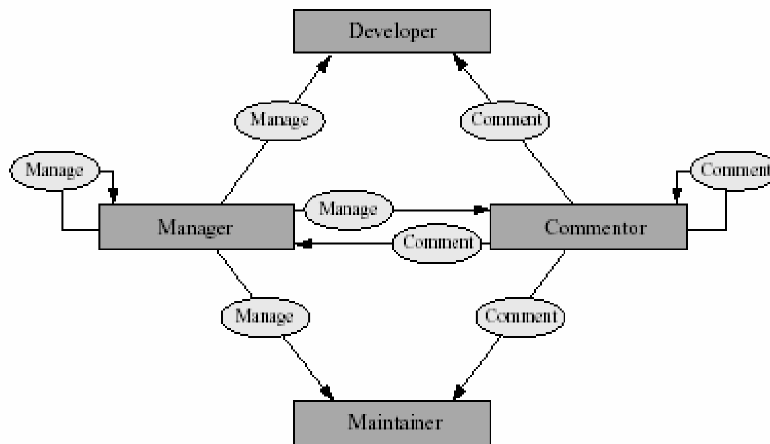

Figure 2: Relations between roles in OSP (Source: [Evers00])

Relations are important elements of the social fabric that constitutes OSP. Probably the most elusive structural element, relations are the conduits for decisions and information flow. By studying the relations in a project, it is possible to learn much about the driving forces behind a project.

3.2.3.3 Relations

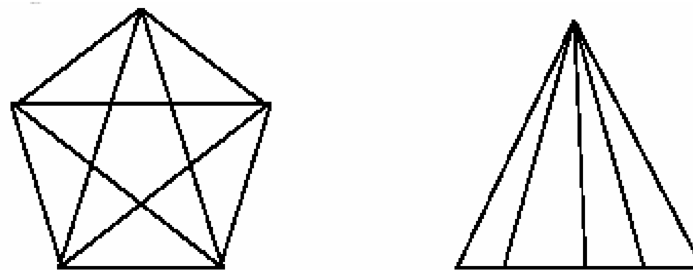

Figure 3: Relations within an OSP

OSP have very complex relations between their various cknstituents, and with their environment. [Kaisla01] mentions that "OSP conventions are based on fairness, nondiscrimination and equal treatment of all parties" Looking at OSP, one discovers that rules, social conventions and behaviors guide OSP to a very large degree. Relations are typically somewhere between full connectivity (Figure 3, left), and hierarchical connectivity, (Figure 3, right). These two relation models lie at opposite ends; one could call them cathedral and bazaar, respectively. [Krebs02] researches relations between individuals in his social network analysis. *"[SNA] is the mapping and measuring of relationships and flows between people, groups, organizations, computers or other information/knowledge processing entities. The nodes in the network are the people and groups while the links show relationships or flows between the nodes. SNA provides both a visual and a mathematical analysis of complex human systems."*

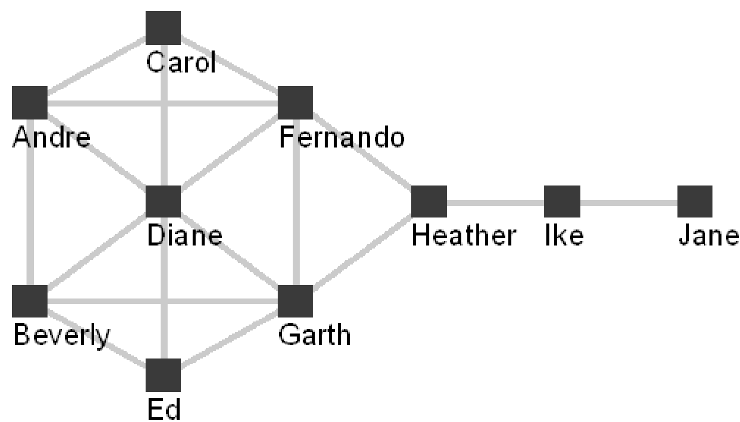

Figure 4: Social Network Analysis (Source: [Krebs02])

Social Network Analysis uses three measures to determine relations, degrees, betweenness and closeness. Krebs illustrates these concepts with an illustration (figure 4).

> **Degrees**
> Network activity for a node is measured by using the concept of degrees -- the number of direct connections a node has. In figure 3, Diane has the most direct connections in the network, making hers the most active node in the network. She is a 'connector' or 'hub' in this network.
>
> **Betweenness**
> While Diane has many direct ties, Heather has few direct connections -- fewer than the average in the network. Yet, in many ways, she has one of the best locations in the network -- she is between two important constituencies. She plays a 'broker' role in the network.
>
> **Closeness**
> Fernando and Garth have fewer connections than Diane, yet the pattern of their direct and indirect ties allow them to access all the nodes in the network more quickly than anyone else. They have the shortest paths to all others -- they are close to everyone else.

Social network analysis is a largely unexplored science.[123] Many organizations, OSP and CSP alike, would benefit from mapping out the information flows and connections among their members. Such an analysis has applications for human resources (recognizing and fostering crucial nodes and links), project management (monitoring information flows) and marketing (identifying efficient information conduits). Even

---

[123] http://www.sfu.ca/~insna/ is a good resource for Social Network Analysis.

without the benefit of social network analysis it is apparent that relations within OSP are varied, and their importance for the health of a project is often underappreciated.

Now that OSP have been defined, what are classical projects? What are their properties?

3.3 Defining Classical Projects

To better understand OSP it is beneficial to define the notion of classical projects and work out the differences and similarities between these two approaches. Therefore, it makes sense to apply the same structural elements to classical projects.

3.3.1 Resources

It is a safe bet to assume that classical projects utilize many of the same resources that are used in OSP. Classical projects might be more limited in the software they can use due to licensing issues, but they do have financial resources that OSP do not have. Classical projects have almost the same hardware requirements as OSP, but have other requirements for their staff.

3.3.1.1 Software

Classical projects try to reuse code to speed time-to-market, increase quality and reduce costs. Commercial off the shelf (COTS) software has been available to fill this demand for quite some time. However, it appears that reuse is still lacking: [Price99] “some organizations have an over-abundance of programmers who do not know how to estimate savings in reusing a piece of software let alone how long it will take to develop and test a piece of software from scratch." Often, licensing issues prevent easy customization of COTS.

Tools
Classical software developers use tools that are at least as good as their Open Source counterparts. Some components, like modeling software, are not available as Open Source. On the other hand, it can be very expensive to put together a world-class development environment. It is quite common to mix and match Open Source and proprietary components. Many companies use the GNU C compiler, for instance.

Components
The classical software industry has developed a large set of components that can be leveraged to build applications. There are components for graphical user interfaces, mathematical routines, business logic and many other fields. These components usually have to be acquired, and come with restrictions. On the other hand, many classical projects make use of Open Source code that falls under one of the more permissive BSD-derived licenses, which allows use in closed source environments. Microsoft, for instance, uses BSD code extensively in their operating systems. Source code access is still relevant after years of talk about component-oriented design, frameworks and libraries, because more often than not, COTS code does not provide what is needed.

Integrated Code
Licensing of source code for integration into proprietary products exists, but is not very common. The usage of such licensed code is heavily regulated and cannot usually be disclosed to third parties. Such licensing usually covers high-value source code for, say, video compression technologies where the source code provider enjoys patents on the techniques used.

3.3.1.2 Hardware

Classical projects tend to be more physically centralized than OSP. Their hardware requirements thus focus less on online collaboration as project members have the opportunity to meet in person. Contract work that is outsourced to India, for instance, shares some characteristics with OSP as it faces the same communication limitations. Classical projects do have more uniform hardware, because their participants usually work at the same company. This added compatibility can be an asset for some projects that develop software that interacts directly with hardware, because the project members operate in a well-known environment. The diversity in hardware seen in OSP can be a hindrance, when time and effort is spent to track down obscure technical glitches, or it can be an asset, when the goal of the project is to operate on a wide variety of hardware platforms.[124]

3.3.1.3 People

Classical projects do have all the tools and methods of management at their disposal to create well-performing teams. Models such as People Capability Maturity Models aim to improve software engineering by improving its participants. [Curtis95] However, Brooks's laws do apply, and as [Raymond99b] notes: "Open Source has been successful partly because its culture only accepts the most talented 5% or so of the programming population. My correspondent spends most of her time organizing the deployment of the other 95%, and has thus observed first-hand the well-known variance of a factor of one hundred in productivity between the most able programmers and the merely competent." The argument is that since classical projects are not self-selective they will not be able to attract the talent that OSP do. This argument may be true for some select OSP, but a casual look through some randomly chosen projects on Sourceforge.net will quickly reveal that Open Source participants are not more talented on average than participants in classical projects are. Newer methodologies like Extreme Programming try to apply peer review concepts that have worked so well for OSP to classical projects. As [Welch00] notes, management of OSP is difficult, more difficult than the management of classical projects. If an OSP tries to achieve the same level of quality as a classical project without the benefits of pay, steady resources and clear responsibilities, its management needs to compensate for these shortcomings. OSP leadership requires leading by example, and motivating volunteers to undertake tedious tasks. OSP leaders thus need to have very good social skills in addition to their technical skills, a rare combination indeed.

[124] http://www.netbsd.org/Ports/ The NetBSD Operating System has the goal to run on as many platforms as possible, and supports 52 hardware platforms.

3.3.1.4 Funding

Classical projects do have funding for all the resources that are needed for the completion of a project. Budgets are not unlimited though, and the question of proper allocation of resources crops up. Funding for a project may be suddenly reduced, which may serve other goals, but will usually result in delays or project failure. Funding is a mixed blessing, as it inevitably creates conflicts between the source of the funding, and the recipients. Projects that are paid for by clients do have to make technical compromises all the time, which typically irks developers. In fact, many developers are motivated to work on OSP in their spare time because they feel they can apply their craft without interference. Clearly, this view is not sustainable in classical projects, but it helps to know that it is there.

3.3.1.5 Service and Infrastructure

Services and infrastructure are usually available for classical projects (CSP). CSP do rely on the same services and infrastructure than OSP. Both types of projects require configuration management and communication infrastructure. CSP have greater liability for data loss since their resources are usually far more centralized (single point of failure) and project participants are not allowed to keep personal copies of their work on their private computing resources. On the other hand, classical projects usually have paid staff for maintenance and infrastructure, which allows the developers to focus on development. Classical projects do spend more time on administrative processes than OSP. Be it accounting for hours spent, attending meetings or performing other corporate functions, an interestingly large amount of time is spent on issues that do not directly relate to advancing the project.
With these resources at hand, how are they used? What considerations apply for coordination? How is work coordinated in classical projects?

3.3.2 Coordination

As in OSP, coordination for CSP is difficult. It is somewhat helped by the more centralized nature of most CSP. CSP do have less issues with time zone differences and subtle communication issues due to never having met in person. Still, only very few individuals in any given project are knowledgeable enough about the finer points of a project to coordinate it. These persons are usually overwhelmed with work. CSP do have leadership, and the benefits of coordination are widely accepted. The [Malone93] Coordination framework will be used for CSP to allow for direct comparison. Coordination entails shared resources, producer-consumer relationships, simultaneity constraints, and task dependencies.

3.3.2.1 Shared Resources

"Whenever multiple activities share some limited resource (e.g., money, storage space, or an actor's time), a resource allocation process is needed to manage the interdependencies among these activities." [Malone93] CSP do exhibit the following shared resources:

Human Actors
Coordination for human actors in CSP takes several forms. Besides the informal coordination and persuasion that is prevalent in OSP, there are also more formal methods. CSP management does have the authority to enforce tasks and monitor progress. Despite these additional tools available to CSP managers, most CSP are behind schedule and over budget. Many approaches for improving project productivity focus on human resources. [Curtis95] lists those efforts somewhat mockingly in his introduction to his People CMM:

Organizations have attempted to apply many different techniques in their efforts to move towards strategic human capital management. They combine downsizing with restructuring, apply reengineering or process improvement, improve information sharing, clearly communicate the organization's mission, institute employee involvement programs, establish formal complaint resolution procedures, institute gain-sharing or other incentive plans, emphasize the importance of training the workforce, formalize performance management and feedback processes, perform job or work analysis and design, support job rotation, begin to establish team-based work designs, retrain employees to meet changing demands, provide flexible work arrangements, address diversity issues, conduct formal mentoring programs, and align business and human resources strategies [Mirvis 97, Becker 98, Becker 96]. What many organizations lack is a framework for implementing these advanced practices.

Clearly, human resources are the most difficult area for project improvement, both for OSP and CSP. They do hold the biggest promise for advancing project management and software engineering, however, and merits attention.

Support Systems
CSP tend to be more centralized than OSP, making resource allocation more important. Many CSP benefit from existing infrastructure and are able to leverage corporate systems management resources. This frees the CSP participants from having to worry about the maintenance of support systems. This easy availability of support systems can lead to heavy processes, where support systems are introduced that have dubious value for completing a project. Many corporate environments suffer from an overabundance of support systems. OSP may work on a shoestring, but they are very wary to introduce new tools and systems for the sake of it.

Information
Ensuring timely access to information is crucial for project success. While information is an unlimited resource, provisions need to be taken to emphasize important information. Often, there is too much information that demands attention, and participants are overwhelmed[125]. Within a CSP it is usually easier to emphasize important information, as participants can meet in person and resolve

[125] The term "data smog" has been coined to describe the phenomenon. http://www.valt.helsinki.fi/comm/argo/anet00/data.htm

communication issues much easier. On the other hand distractions are much larger, and participants may spend a lot of time with informal communication that is irrelevant to the project.

3.3.2.2 Producer/Consumer Relationships

"[...] a situation where one activity produces something that is used by another activity." [Malone93] Producers may be part of a different project, or other team members within the same project.

Relations to Other Projects
CSP may have relations to other projects within the same organization, or outside of it. Depending on non-disclosure agreements and licensing issues, CSP participants may need to take extra care to not leak sensitive information to other projects. This contrasts starkly with OSP, where sharing of information is encouraged.

Relations inside the Project
As a project grows sufficiently large, dependencies develop within the project. Bottlenecks arise because tasks have interrelationships. It is the job of the project management to resolve these issues. CSP do have much more effective tools for dependency resolution at their disposal, which would indicate that CSP cope better with interrelationships than OSP do. Market forces may pressure CSP into settling with less than satisfactory solutions, though.

3.3.2.3 Simultaneity Constraints

"Another common kind of dependency between activities is that they need to occur at the same time (or cannot occur at the same time)." [Malone93] Two examples illustrate the problems with simultaneity constraints:

Real-Time Communication
Meetings, conferences, phone calls and other kinds of real-time communication must take place at the same time for all participants. Due to the relative ease of setting up meetings in CSP, there is a tendency to schedule too many meetings with poorly defined agendas. This can have a major productivity impact.

Document Modification
Most CSP use configuration management software to avoid conflicts arising from simultaneous edits of their source code. Configuration management can be very simple, like a locking system for files that are being edited. They can also be very complex and be integrated with other development tools, like issue tracking systems.

3.3.2.4 Task Dependencies

"[...] a group of activities [that] are all 'subtasks' for achieving some overall goal." [Malone93] Several persons working on the same task must coordinate their activities to

make sure that the results of their activities integrate. Three different methods to handle this dependency can be identified.

Top-Down Goal Decomposition
"[...] an individual or group decides to pursue a goal, and then decomposes this goal into activities (or subgoals) which together will achieve the original goal." [Malone93] In this method, significant resources are used to produce a plan of future actions; subtasks are and assigned to team members. This method is often used in CSP. Nonetheless it has its own problems:

1. Plans become outdated quickly (new requirements, new technologies, etc.).
2. Many problem domains have no clear-cut solutions

Bottom-Up Goal Identification
"[...] several actors realize that the things they are already doing (with small additions) could work together to achieve a new goal." [Malone93] CSP usually do not exhibit this type of goal identification. The concept of a "skunkworks" project comes closest:[126]

A 'skunkworks' is a group of people who, in order to achieve unusual results, work on a project in a way that is outside the usual rules. Typically, a skunkworks has a small number of members in order to reduce communications overhead. A skunkworks project may be secret

Concurrent Task Processing
Another possibility to coordinate goals and their subtasks is to let people work concurrently. This method is similar to the bottom-up method. The advantage of concurrent task processing is that participants have a lot of freedom, keep the control over their resources and still improve their productivity by collaboration. A common procedure is the following: Everyone starts working, produces some results, the group compares achieved (partial) solutions, discusses them and starts a new concurrent working cycle with the parts they agreed on as the most promising ideas. This approach is impractical for CSP for the most part since schedules are unpredictable.

CSP do arguably coordinate their activities much more than OSP. It would thus be logical to assume that CSP do have better established structures than OSP. What do these structures look like, and how do they differ from OSP?

3.3.3 Structures

The organizational structure of CSP has been studied extensively. The bottom line of these studies[127] is that no single structure is appropriate for all types of CSP, large or small. It therefore makes sense to reflect on structural elements more than on complete

[126] http://whatis.techtarget.com/definition/0,,sid9_gci214112,00.html
[127]http://www.informit.com/

structures. As in OSP, structure emerges out of activities; participants assume roles and are related to each other in complex ways.

3.3.3.1 Activities

CSP activities are mostly performed within organizations, and are therefore not generally available for research purposes. Decades of research into software engineering have nevertheless produced a rich body of evidence. Major activities in CSP are communication, decision-making, coordination, documentation and software production.

Communication
Project information is distributed among its participants. In order to work together and to achieve results, participants need to exchange a lot of information between each other. This creates a strong need for efficient communication facilities. Efficient and effective communication needs to answer these questions:

1. Which knowledge should be transferred?
2. Who should provide the knowledge?
3. Who should receive it?
4. How should knowledge be transferred?

Knowledge should be transferred by communication. Communication means to share the relevant knowledge of the project between all participants to support their work. As in OSP, the source code of a CSP often constitutes the most reliable knowledge store. CSP do have more formalized methods of knowledge dissemination that may or may not work. For instance, CSP usually do make sure that software is properly documented. CSP may have obstacles for knowledge dissemination that do not exist in OSP. Participants may withhold information purposely to "stay one step ahead" of their coworkers. Legal reasons may further restrict the amount of knowledge that is being circulated broadly.

Decision-Making
Setting up goals or priorities for the future, choosing between contradicting ideas, changing a project's policies are all part of decision making. One person, a group or all members of a project might be involved in the decision making process. Decisions can be taken formally following rules the members agreed on before or informally by a simple conversation. Decisions might be considered final or provisional. Collecting good data to make informed decisions is a time-consuming task. Decision-making draws heavily on other activities. For instance, the collection of required information is part of the communication activity. Many decisions are made subconsciously because the number of decisions each individual makes every day is enormous. These inherent decisions and hidden assumptions can lead to communication problems within a project. On the other hand, it is impractical to decide minor issues in the group. Balancing individual and collective decisions is therefore critically important. CSP have the advantage to make harder decisions than OSP, because they can enforce them. On the other

hand, OSP are usually less subject to external influences (political, business influences) and can take decisions based on their merit for the project.

Coordination
All activities of a CSP need some kind of coordination as they all depend on their context including other activities. Many of these dependencies are hidden. Coordination and communication are very closely related activities, and communication is the major means available to CSP for coordination. Even though CSP do have much better facilities available for solving coordination problems, their participants should still try to prevent dependencies. Recent human resources frameworks such as [Curtis95] emphasize the need for individuals to be empowered, and assuming responsibility for parts of their work. This helps to reduce dependencies and bottlenecks between project participants.

Documentation
In CSP, documentation has a direct impact on potential profits for the software vendor. Provide good documentation, and your support infrastructure will be less taxed by requests. This economic incentive leads to better documentation for CSP than OSP, on average. Methodologies exist for CSP that focus heavily on process documentation. These methodologies aim to capture the knowledge that is gained during a project to reuse it in other projects, and improve quality.[128] While some of those methodologies involve lots of paperwork, they do have tangible benefits:

- Bugs It is helpful to have a detailed change history to track down bugs. Additional documentation can be used to investigate less obvious bugs. Over time, it should be feasible to detect patters in the way bugs occur, and address them methodically.

- Compatibility Good process documentation helps to enforce consistency. Established processes make it easier for coworkers to develop additions to a project, and make it easier to integrate it later. In addition, good documentation lessens the impact of key developers leaving a project, and makes maintenance of software easier (or even possible).

Process documentation is usually longer-lived than the underlying source code because implementations may change, but fundamental ideas stay the same. Documentation is a scarce resource in any software project, so any little piece of it helps, even simple log messages.

Software Production
Software production is the most important activity and all others only serve to optimize it. Software production can be described as a Meta activity, as it includes other activities recursively. Some projects have their software production activity defined precisely; others only have vague goals. Cynical observers may note that some CSP do not have a clear purpose, and seem to exist merely for political

[128] The Personal Software Process (PSP) is one example.

reasons. OSP are more likely to have a purpose, as participants need to weigh their invested time against other uses of their spare time. This opportunity cost makes sure that OSP participants are more motivated and interested in their problem domain than CSP participants. [Raymond99b]:

> *If the conventional, closed-source, heavily-managed style of software development is really defended only by a sort of Maginot line of problems conducive to boredom, then it's going to remain viable in each individual application area for only so long as nobody finds those problems really interesting and nobody else finds any way to route around them. Because the moment there is open-source competition for a 'boring' piece of software, customers are going to know that it was finally tackled by someone who chose that problem to solve because of a fascination with the problem itself – which, in software as in other kinds of creative work, is a far more effective motivator than money alone.*

CSP perform mostly the same activities than OSP do. It is in the roles that participants play that you will likely find differences between the two approaches. What are those differences?

3.3.3.2 Roles

Even though CSP do usually plan the activities of their project, it is still difficult to identify clear-cut roles. Granted, role descriptions may exist for a CSP, even in written form and approved. Day to day activities are very likely to stray from these defined roles, however, and it is not uncommon to see CSP participants impersonate different roles over time. It makes sense to define the same roles that were defined earlier for OSP, namely developer, manager, maintainer, administrator and commenter.

Developer
A developer is responsible for implementing the projects goals. He participates mostly in the production and documentation activities. Developer is really a term that encompasses many functions in a CSP, such as architect, programmer, analyst and others.

Manager
A manager directs a project. Decision-making and coordination are his major activities. In CSP, a manager has authority to order other team members to perform tasks, unlike OSP. While most OSP recruit their managers from technical staff, CSP may have managers with a non-technical background. This can lead to acceptance problems when developers do not trust the judgment of a manager. Having a manager with a non-technical background can broaden the view of a project, and prevent a project from having an engineering bias.

Maintainer
A maintainer keeps track of issues with released components of a project and is responsible for their resolution. His major activities are coordination and

communication. CSP may have different teams for software under development and for software being maintained. Thus, maintainers may not have taken part in the original project.

Administrator
An administrator is responsible for a smooth operation of the project by maintaining project resources like centralized servers, communication facilities, and configuration management systems. In CSP, service personnel outside the project may take this role.

Commenter
Anyone who provides some kind of feedback and does not implement it himself in the production activity is a commenter. In CSP, this role could more aptly be named customer. In CSP, the customers assume the most influential role in a project, as they provide the funding, while as OSP often cater primarily to themselves, and accept little outside input.

Participants are somewhere in between these abstract roles and perform actions that belong to several different roles. In turn, most roles are occupied by several participants. Developers and maintainers provide the required software, managers manage a project and commenters review some activities and give feedback. The administrator is the invisible agent who supports all these activities. Figure 5 shows these roles and how they interact.

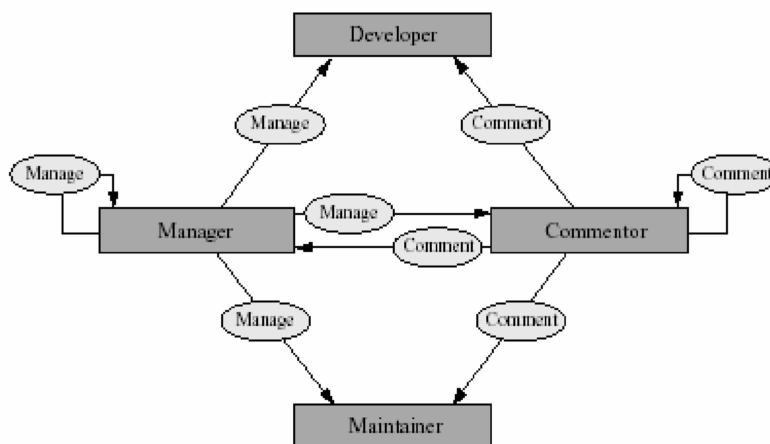

Figure 5: Relations between roles in CSP (Source: [Evers00])

CSP are just as varied as OSP. Hence, it is no surprise to find that roles within CSP vary to a large degree. It could be argued that roles within CSP are usually better defined than in OSP, and project participants have a better idea what their role is. With roles defined, how are relations in CSP structured?

3.3.3.3 Relations

CSP have very complex relations between their various constituents, and with their environment. These relations are however far more closely controlled, and often there are few relations that cross organizational boundaries. Many CSP forfeit a large number of possible relations by channeling them through points of contact, usually "customer representatives". This can help to reduce complexity, but carries the risk of losing valuable information that would otherwise be shared across organizations. Figure 5 illustrates such an abstracted relationship. Users interact solely with commenters inside the project, component providers sell ready-made software (COTS) to the project, and support system providers provide software tools or services for the operation of the project.

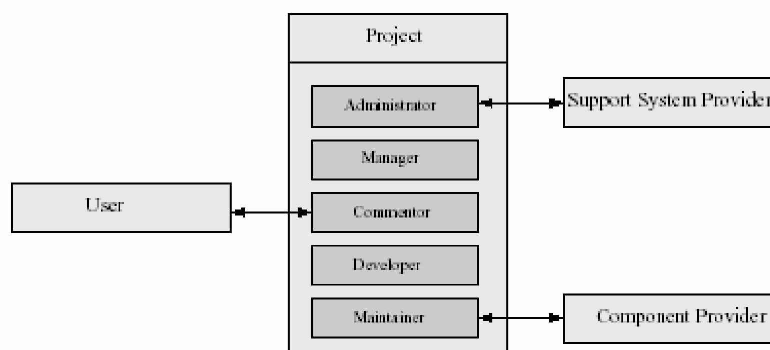

Figure 6: Relation of CSP with their environment (Source: [Evers00])

With CSP and OSP defined, it now makes sense to look at strengths and weaknesses of both approaches, and ultimately summarize them.

3.4 Strengths of Open Source Projects

Both literature and anecdotal evidence contain many accounts of the advantages and unique strengths of OSP. Release frequency, customer input and scalability are mentioned often.

3.4.1 Release frequency

A basic tenet of OSP is "release early, release often" [Raymond99b]. Frequent releases of software, combined with sensible policies for maintaining stable and experimental releases concurrently, increase the potential for feedback from a wide variety of users. OSP utilize their users in the testing process. Users of OSP are expected to report their findings about faults back to the project. High release frequencies are infeasible for production environments. For these types of uses, stable releases are provided, leaving the choice about tracking new releases in the hands of the users. Some OSP go to extremes to provide up to the minute updates for their project. Infrastructure exists to synchronize all changes in a project once a day to thousands of testers.[129] This process is

[129] The FreeBSD operating system project developed a tool CVSup which synchronizes the complete source tree. http://www.onlamp.com/pub/a/bsd/2001/08/16/Big_Scary_Daemons.html

good practice, and is also applied at Microsoft, where the build environment provides a full compile of the windows operating system once a day,[130]

3.4.2 Customer Input

OSP do exhibit short feedback loops between users and developers. Often, it is only a matter of minutes or hours from the point at which a bug is reported from the periphery to the point at which an official patch is supplied from the developers to fix it. Moreover, the use of powerful Internet-enabled configuration management tools, such as the GNU Concurrent Versioning System (CVS)[131], allows Open Source users in the periphery to synchronize quickly with updates supplied by the core. These quick response cycles encourage an open-source user community to help with the quality assurance (QA) process since they are "rewarded" by rapid fixes after bugs are identified. Moreover, because the source code is open for inspection, when users at the periphery do encounter bugs, they can often either help fix them directly or can provide concise test cases that allow the core developers to isolate the problem quickly. Thus, the effort of the user community extends the overall debugging effort and improves software quality rapidly.[132]

3.4.3 Scalability

OSP work by exploiting a loophole in Brooks Law that states, "Adding people to a late project makes it later." The logic underlying this law is that as a rule, software development productivity does not scale up as the number of developers increases. The culprit, of course, is the rapid increase in human communication and coordination costs as project size grows. Thus, a team of ~10 good developers can often produce much higher quality software with less effort and expense than a team of ~1,000 developers. In contrast, software debugging and QA productivity does scale up as the number of developers helping to debug the software increases. The main reason for this is that all other things being equal, having more people test the code will identify the defects much more quickly than having just a few testers. Thus, a team of 1,000 testers should find many more bugs than a team of 10 testers. QA activities also scale better since they do not require as much interpersonal communication as software development activities (particularly design activities) often do. To leverage the loophole in Brooks's law, therefore, most successful OSP have a "core" and "periphery" organizational structure. In this division of labor, a relatively small number of core developers (who may well be distributed throughout the world) are responsible for ensuring the architectural integrity of the project. These individuals review user contributions and bug fixes, add many new features and capabilities, and track day-to-day progress on project goals and tasks. In contrast, the periphery consists of the thousands of members of the user community who help with testing and debugging of the software released periodically by the core team. Naturally, these divisions are informal and individuals may fulfill different roles at different times during the life cycle of an OSP.

---

[130]www.usenix.org/events/usenix-win2000/ invitedtalks/lucovsky.ppt provides a very interesting account of the Microsoft build environment.

[131] http://www.cvshome.org/

[132] [Schmidt01] provides more details.

Strengths are balanced by weaknesses. OSP have plenty of weaknesses, and being aware of them can help to avoid false expectations.

3.5 Weaknesses of Open Source Projects

Open Source is no magic bullet, despite claims to the contrary in some popular explanations of the phenomenon. Most weaknesses ultimately boil down to a lack of formal organization or clear responsibilities.

3.5.1 Communication

Communication is crucial for all OSP, but the biggest challenge at the same time. The different cultural backgrounds, never having met in person, time zone differences all make it harder to communicate clearly and precisely.

Language
Although English is the language of the Open Source community; it is not the native language of all its members. This results in several problems. Misunderstandings crop up and people feel offended because communication can be interpreted in several ways. Aidan M. Humphreys writes,[133] "English is, to be sure, the closest thing we have to a lingua franca for software engineering. One of my involvements, the PHP-based PostNuke CMS Project, has over 200 developers from -- well, just about everywhere, with English as a common language. But there are many talented developers who, whilst quite happy to read the latest W3C spec or RFC, do not feel confident enough of their Franglais, Singlish, or Ginglish to hold their corner when flame wars break out."

Irrelevance
The content of messages on discussion groups might lead one to the wrong impression that the Internet is filled with junk and jerks. It is common for Internet users to complain bitterly about the lack of cooperation, good manners, and useful information. This is not completely true, but the *signal-to-noise ratio* is bad and getting worse. A casual trip through cyberspace will turn up evidence of hostility, selfishness, and simple nonsense (much like a random walk in the real world will yield evidence of hostility, selfishness and nonsense). For many, it is much easier to be hostile in an email discussion than face-to-face.

3.5.2 Redundant Efforts

OSP coordinate very little. Independent parties sometimes carry out tasks in parallel without knowing about each other. This consumes additional resources, but it also has the nice side effect that there are often several solutions to choose from. The choice between different alternatives helps to improve software quality. [Bezroukov99a] has noted this similarity between the OSP process and academia. Concurrent research sometimes leads to conflicts and passionate discussions when someone feels left out.

[133] http://www.onlamp.com/pub/a/onlamp/2002/06/12/modelixe.html

3.5.3 Lack of Priorities

Where quick, far-reaching decisions are needed, OSP fail. Due to their distributed nature and their lack of leadership, priorities are either nonexistent or severely skewed towards the personal biases of influential contributors. One area where this lack of focus is apparent is usability[134]. It has been argued that large-scale changes like making security the top priority at Microsoft[135] are not possible in OSP because no one has the authority to make these calls. In addition, OSP often are dragged down into endless arguments because no one can force his opinion on others, and end discussions. OSP users often claim a right to take part in technical decisions even though they neither have the necessary knowledge nor are they prepared to spend time to consider their requests in light of other issues.

3.5.4 Lack of Conventions

OSP normally do not have formal rules or written conventions. Instead, newcomers to a project are expected to gradually learn the hidden rules of the group, and are measured on their success at reacting to group clues. These tightly knit communities may improve communications for their members by sharing much cultural context, but make it harder for new arrivals to integrate themselves into a group. Since every OSP competes for attention and talent, these barriers to entry are very damaging to a project. [Edwards00] calls these phenomena “epistemic communities”. “Open source software development is a learning process where the involved parties contribute to, and learn from the community.” We will revisit epistemic communities later, as they hold one of the keys for improving OSP performance.

3.5.5 Lack of focus

According to the FLOSS study, around 70% of the participants in open source projects spend 10 hours or less per week on project work. This low level of participation introduces problems with communication as individuals fail to keep up with all developments in a project. These hours are most often spent on evenings or weekends, and may be scattered over the course of a workweek. These circumstances make it hard for contributors to focus on the project at hand. Very often, works is conducted piecemeal and is drawn out over a long period of weeks. The effort expended to stay on top of issues is often so great that little time remains for work on contributions. Many contributors lose interest in a project or are faced with other commitments, leading to many half finished projects.

3.5.6 Dependency on key persons

Jamie Zawinski[136]: “If you have a project that has five people who write 80% of the code, and a hundred people who have contributed bug fixes or a few hundred lines of code here and there, is that a '105-programmer project?'" [Jones00] argues that the bulk of the work is done by a few dedicated members or a core team -- what Brooks calls a "surgical team." This centralization of work and responsibility contrasts with [Raymond98a],

---

[134] http://www.sims.berkeley.edu/~sinha/opensource.html

[135] http://news.com.com/2100-1001-816880.html

[136] Zawinski is one of the original authors of the Netscape browser. http://www.jwz.org/

where he argues that open source projects are self-organizing and very distributed. Instead, we find that many projects critically depend on a few key persons. There are several plausible explanations for this phenomenon. The most obvious is the level of intimate knowledge that is required to understand all parts of a large software system. The effort to gain this knowledge is usually only undertaken by the most active contributors. A lack of documentation further reinforces this trend. Another explanation is the level of recognition individual contributors can expect to receive. The limited attention span of the Open Source audience dictates that only a very small number of people will become widely known for their contributions. Again, this tends to strengthen the role of core contributors. This dependency can become a liability if these key persons are unable to continue work on the project for some reason. It may be impossible to reconstruct the implicit knowledge of these persons from their artifacts (source code, documentation, notes, and emails) alone. This often leads to project failure.

3.5.7 Leadership

Success in OSP is largely dependent on good, charismatic leadership. In addition to the qualities needed from a software engineering perspective, Open Source leadership needs to address issues such as communication, marketing, political savvy, and motivation. Open Source leaders lead by force of persuasion alone. They do not have a mandate to lead, nor do their coworkers have a mandate to follow. Leaders are judged on their technical skills, on their vision and their ability to communicate. The Open Source landscape is much less forgiving with weak leaders. Even though many Open Source contributors do have a professional background in information technology and are more skilled than most, these requirements are so selective as to make it very hard to fill all Open Source leadership positions with qualifying personnel. It has been argued [Raymond99b] that the success of the Linux project was to a large degree due to the excellent leadership skills demonstrated by its founder Linus Torvalds. The scarcity of good leaders is a very serious issue for OSP and one of the factors that inhibit their growth.

3.6 Strengths of Classical Projects

CSP exhibit many strengths that should not be overlooked in the context of a comparison. For the purposes of this comparison, it is assumed that a classical project is well run and uses state of the art knowledge about software engineering and project management processes. The study of these strengths may lead to insights about projects in general that can subsequently be applied to OSP to improve them further.

3.6.1 Predictability

It is very often argued that one of the biggest strengths of CSP is their predictability. For instance, almost all CSP try to deliver results by a deadline. In contrast, open source projects do rarely feel compelled to achieve goals within a given period. Speaking in much generalized terms one could say that CSP follow a predictable development path. This means that they deliver the requested functionality and do not change course radically in mid-action. OSP, on the other hand, are subject to the whims of their developers in first approximation. Users do have an influence, but developer interests often prevail in controversial decisions.

3.6.2 Standards

Well-run CSP, (as defined by the capabilities maturity framework) use standardized methodologies in their software engineering processes. This allows them to operate at high efficiency levels and makes it easier for new project members to be acquainted with the mode of operations. Areas for standards include:

- Requirements Analysis
- Testing
- Verification
- Documentation

Standards in and of themselves are no panacea though. They need to be wisely applied, and sometimes it may be more sensible to forego standards to achieve results.

3.6.3 Documentation

In the case of shrink-wrapped software (admittedly only a very small part of CSP), good documentation is a major differentiator in the market. Documentation is written with a non-technical audience in mind. The incentive to reduce support costs by providing well-written documentation produces results that are far superior to the average quality of OSP documentation. In the case of internally produced software, standards often require the creation of documentation. In contrast, no such mandate exists for OSP.

3.6.4 Accountability

Software for specialized domains needs to guarantee the highest levels of availability and correctness. Clients of CSP use legal incentives to ensure compliance with requirements. The existence of a corporate body that can be sued if need be is reassuring to the business world. While end user agreements for shrink-wrapped software disclaim all responsibility, custom-made software is often accompanied by clauses that demand compensation payment if a set of conditions is not satisfied. OSP are mostly developed by informal groups, and accountability is therefore greatly reduced. In most cases, OSP disclaim all responsibility for possible damages resulting from the use of their software out of fear of personal liability.

3.7 Weaknesses of Classical Projects

Perceived or real weaknesses of CSP lead to the search for better solutions. OSP need to address these issues if they want to offer a credible alternative.

3.7.1 Customer Input

In some domains, customers will be or will have access to technical people who are capable of understanding a software system and contributing to it. Whether it is a library or framework that is sold to application developers, an operating system and development environment used by others, or embedded software used by system builders, when these types of users find bugs or need additional capability, they may indeed be able to make the change themselves. An open source model for the product would allow these customers to quickly solve their own problem, and contribute to the product as well. In

this scenario, the self-interest motivator in OSP is satisfied because the user will probably get the fix much sooner than in a normal CSP report-bug-and-wait-for-fix type process. The philanthropy motivator does not come into play because presumably they bought the system in the first place. Further monetary rewards could be given for fixes that are incorporated back into the selling product.[137]

3.7.2 Scalability

CSP projects are perceived as being hard to scale. This means that it is very difficult to increase the output of a project linearly with an increase in the number of staff. Brook's Law "Adding manpower to a late project makes it later" exemplifies these observations. CSP do have more formal interactions than OSP, and its participants work much closer together. OSP projects often have participants that work for weeks without communicating with other participants, and deliver results afterwards. They do not take attention away from other participants due to frequent communication. OSP that have contributors like these are able to scale to much higher numbers without the adverse effects observed by Brook.

3.7.3 Bureaucracy

In an effort to raise quality, many CSP apply intense processes with much paperwork. The effectiveness of such overzealous red tape is elusive at best. Additionally, being in the same physical building increases the likelihood for aimless meetings to be convened, where every project participant is required to attend. A tendency for micromanagement destroys productivity. CSP are more prone to these ills because micromanagement is almost impossible over the Internet.

3.7.4 Skill levels

It is no secret that the biggest single factor in programmer productivity is the individual. Well-established results point to at least two orders of magnitude difference in productivity, and some studies have shown that perhaps one out of five programmers actually produces *negative* work - that is, the other four would be more productive without that person. Certainly much of the success of open source software development is dependent on this fact. There is little doubt that those who undertake and succeed on the initial construction of an OSP are at the top end of the talent spectrum. Furthermore, open source software development naturally selects for its contributors the cream of the talent. It does not have to do this with hire and fire decisions as CSP do - lesser individuals can submit potential contributions, but only the best contributions will actually be incorporated. CSP, unfortunately (or not, depending on your viewpoint), cannot simply rely on the cream for its resources. It has always been a delicate balance just to get enough individuals interested in pursuing a technical career, much less eliminating most of them to concentrate on the "cream". The plain fact is that most CSP will be staffed with a team that reflects the normal distribution of technical talent. Often there is at least one person who can be a "hero" and lead the project through to success. Nevertheless, most of the team will be of average productivity. Hoping to magically

[137] [Schmidt02] discusses one of these potential reward systems.

reach the success levels of Open Source software development when using a different talent pool, is akin to believing in a silver bullet.

With both strengths and weaknesses addressed, it makes sense to directly compare OSP and CSP. Reducing complex issues to keywords obviously does not lend itself to subtlety, but it gives the interested reader a quick overview. These one-word summaries stand for whole concepts, and should be understood as such. What are the properties of OSP and CSP, in direct comparison?

3.8 The Properties Matrix

To conclude the comparison between OSP and CSP, various properties are directly compared, and summarized. The properties matrix necessarily reduces the subtleness of both OSP and CSP realities to a black and white comparison. With the disclaimer that the matrix does represent idealized states it is, however, a useful primer.

Property	CSP	OSP
Structure	hierarchical	networked
Scope	internal/closed	external/open
Resource focus	capital	human, information
State	stable	dynamic, changing
Direction	management commands	self-management
Basis of action	control	empowerment to act
Basis for compensation	position in hierarchy	competency level
Business Processes (development)	Linear	Parallel
Cost of development	High	Low
Cost of coordination	High (Brook's law holds)	Low (Brook's law doesn't)
Mode of organization	Centralized	Decentralized
Management	Hierarchical	Collaborative-community

Hierarchical layers	Several	Collaborative-community Four, but not in a bureaucratic fashion
Modularity	Low	High
Knowledge functions (access, sharing, diffusion, creation, exploitation)	Low	Massive
Organizational learning	Linear	Parallel
System	Closed	Open
Users-producers	Separated	Overlap
Number of participants	Limited	Unlimited
Product transparency	Absent (copyright)	Massive (copyleft)
Decision-making transparency	Low	Massive
Product innovation	Low	High
Organizational innovation	High	Massive
Cost of platform	High	Low
Flexibility of platform	Low	High
Average Hours / Week	40	14.4
Reuse	Low	High
Motivation	Monetary	Non-Monetary
Use of standards	Low	High

Table 3: OSP versus CSP properties

With the properties of both OSP and CSP defined, the major strengths and weaknesses outlined, the context is established to embark on a theory of Open Source. Attempts have

been made to characterize Open Source with theories before, but all of them remain flawed. Why is it so hard to arrive at a theory? Where did previous theories fail, and what areas need special consideration? Finally, how would one approach a unified theory of Open Source?

4. Towards a unified Open Source Theory

No comprehensive theory about Open Source has emerged in the literature yet. This chapter discusses the shortcomings of existing theories, looks at challenges that have prevented the emergence of a comprehensive theory and proposes a new approach that considers the acknowledged difficulties. First, it is interesting to note that all existing theories try to describe Open Source from the perspective of whatever notion the author is fond of, be it sociology [Raymond99b], economics [Goldhaber97], software engineering [Scacchi02] or psychology [Hertel02]. While all these theories provide valuable insight into Open Source, none of them bridges disciplines to embark on a wholesome approach. Where do existing theories succeed, where do they fail?

4.1 Limitations of existing theories

Several authors have developed theories to explain the Open Source phenomenon. Each of these theories contributes to the general understanding about Open Source, but all theories are incomplete and focus only on selected aspects. That said, it is always easier to pick on existing theories and take them apart then to come up with a new theory. The goal here is not to make fun of these earlier efforts, but rather to distill the core of each theory, and do away with the weaker parts of a theory. With luck, we will be able to assemble these pieces to a theory that goes beyond the individual pieces, and adds to the understanding of Open Source. The most famous theory is of course “The cathedral and the Bazaar”, and we will give it special attention because it has by its own contributed more to Open Source theory than all others combined. Also of note is Bezroukov’s assertion that there are strong parallels between academia and the Open Source community. Economic approaches to Open Source are plentiful [Nahm02], [Wegberg00], [Ghosh98], [Kenwood01], [Bessen02], [Hippel02], [Schmidt02], [Lerner01], [Iannicci02], [Rasch01], and [Edwards00b]. One explanation of this huge economical interest is that Open Source seems to defy many notions about economic motivations, and thus raises curiosity. Another promising venue is to look at the participants in Open Source, and research their motivations, their interactions and discover governing structures. Management theory has embraced the concept of virtual organizations for a while, it is thus not surprising to see them being applied to Open Source. Observing a virtual organization *is* rather fascinating, however, deducing how it works is an entirely different matter. To understand their workings, it is necessary to probe deeply into psychology, to find underlying motives, and levers for influencing project outcomes by appealing to the psyche of individuals.

4.1.1 The cathedral and the Bazaar

[138]The ideas represented in the Cathedral and the Bazaar (CatB) became a part of Open Source folklore; they are reproduced frequently in papers and interviews, and have been

[138] I am indebted to [Bezroukov99b] who made many of the observations presented here.

instrumental for a communicating the core aspects of the Open Source phenomenon. Many Open Source authors base their arguments on an implicit assumption that these ideas are true. Some of the most important ideas in CatB include:

- Brooks' Law does not apply to Internet-based distributed development;
- "Given enough eyeballs, all bugs are shallow";
- Linux belongs to the Bazaar development model;
- The OSP model automatically yields the best results;
- The Bazaar development model is a new and revolutionary model of software development.

All these ideas are vulnerable to varying degrees. Understanding the weak points of CatB helps to develop stronger, more comprehensive theories later. The remarks on Brooks' Law are among the most important statements in CatB.

4.1.1.1 Brooks' Law does not apply to distributed development.

One of the vulnerable ideas of CatB is that Brooks' Law is non-applicable in the Internet-based distributed development environment as exemplified by Linux. From CatB:

"In The Mythical Man-Month, Fred Brooks observed that programmer time is not fungible; adding developers to a late software project makes it later. He argued that the complexity and communication costs of a project rise with the square of the number of developers, while work done only rises linearly. This claim has since become known as "Brooks's Law" and is widely regarded as a truism. But if Brooks's Law were the whole picture, Linux would be impossible."

This belief that programmer time scales differently as soon as programmers are connected to the Internet and are working on OSP is repeated elsewhere in a different form:

"Perhaps in the end the Open Source culture will triumph not because cooperation is morally right or software "hoarding" is morally wrong (assuming you believe the latter, which neither Linus nor I do), but simply because the closed-source world cannot win an evolutionary arms race with open-source communities that can put orders of magnitude more skilled time into a problem."

The "Mythical Man Month" is a software engineering classic. Written almost 30 years ago, its basic ideas remain true. Any claim that its observations are wrong therefore has to be considered with appropriate caution. The real problem with this CatB statement is that due to the popularity of the CatB this statement could discourage the open source community from reading and studying *The Mythical Man-Month,* one of the few computer science books that remained current decades after its initial publication. Actually, the term "Brooks' Law" is usually formulated as "Adding manpower to a late software project makes it later". The term "mythical man-month" (or "mythical man-month concept") is used to identify the concept of diminishing output of multiple developers even if all work on a given project from the very start. Ray Duncan gave one

of the best explanations of this concept in his review[139] of *The Mythical Man-Month:* "What is a mythical man-month anyway? Consider a moderately complex software application from the early microcomputer era, such as the primordial version of Lotus 1-2-3, Ashton-Tate dBASE, or Wordstar. Assume that such a program might take one very smart, highly motivated, expert programmer approximately one full year (i.e., 12 'person-months'), to design, code, debug, and document. Imagine that market pressures are such that we want to get the program finished in a month, rather than a year. What is the solution? You might say, "Get 12 experienced coders, divide up the work, let them all flog away for one month, and the problem will be solved. It is still 12 person-months, right?

Unfortunately, time cannot be influenced so easily. Dr. Brooks observed that person-months are not "factorable, associative, or commutative". One programmer times 12 months does not equal 12 programmers times one month. The performance of programming teams, in other words, does not "scale" in a linear fashion any more than the performance of multi-processor computer systems. Dr. Brooks found, in fact, that when you throw additional programmers at a project that is late, you are only likely to make it much *later*. The way to get a project back on schedule is to remove promised-but-not-yet-completed features, rather than adding more resources. This phenomenon is easy to understand. There is inescapable overhead having programmers work in parallel. The members of the team must "waste time" attending meetings, drafting project plans, exchanging email, negotiating interfaces, enduring performance reviews, and so on. In any team of more than a few people, at least one member will be dedicated to "supervising" the others, while another member will be devoted to housekeeping functions such as managing builds, updating Gantt charts, and coordinating everyone's calendar. As the team grows, there is a combinatorial explosion such that the percentage of effort devoted to communication and administration becomes larger and larger. CatB assumes that Internet connectivity can improve performance for a project in comparison with, say, LAN connectivity or using the same mainframe. The Internet allows access to a potentially very large talent pool, but does not guarantee that these resources will materialize. Moreover, assuming the same level of developers, geographically compact teams will always have an edge over distributed teams connected via the Internet. Reducing the effects of distance does not eliminate other constraints under which OSP operate, but can dramatically increase the quality of the pool of developers. Brooks' law does not hold only for projects for which a fully functional prototype already exists and most or all architectural problems are solved. This may have been the case for Linux, which is essentially an open source re-implementation of UNIX. [Vallopillil98] points out: "The easiest way to get coordinated behavior from a large, semi-organized mob is to point them at a known target. Having the taillights provides concreteness to a fuzzy vision. In such situations, having a taillight to follow is a proxy for having strong central leadership. Of course, once this implicit organizing principle is no longer available (once a project has achieved "parity" with the state-of-the-art), the level of management necessary to push towards new frontiers becomes massive. This is possibly the single most interesting hurdle to face the Linux community now that they've achieved parity with the state of the art in UNIX in any respects."

[139] http://www.ercb.com/feature/feature.0001.html

4.1.1.2 Given enough eyeballs, all bugs are shallow

One of the most important ideas promoted by CatB is the phase attributed to Linus Torvalds - "Given enough eyeballs, all bugs are shallow". [Raymond99b]: "In the bazaar view, on the other hand, you assume that *bugs are generally shallow phenomena* - or, at least, that they turn shallow pretty quick when exposed to a thousand eager co-developers pounding on every single new release. *Accordingly you release often in order to get more corrections, and as a beneficial side effect you have less to lose if an occasional botch gets out the door.*"

The debugging of a complex system is much more difficult undertaking that simply getting a huge number of "eager co-developers" to analyze lines of code. For most complex projects for every second or third bug located and fixed another one may be introduced. [Schach02] noted that "it seems inevitable that, at some future date, the dependencies between modules induced by common coupling will render Linux extremely hard to maintain. It will then be exceedingly hard to change one part of Linux without inducing a regression fault (an apparently unrelated fault) elsewhere in the product." CatB assumes that several talented developers can successfully work on the same piece of code in parallel without any coordination other than email; one of them eventually will fix the code quicker than in the commercial environment with specially trained testers. Certainly, if enough talented developers try to find the same bug simultaneously it probably will be found eventually. But there are several problems with this idea of parallel debugging. Several questions arise:

Is this the best way to utilize talented developers?
Why waste their skills on debugging? In some commercial environments, professional testers provide an important edge. Synergy between volunteer developers and commercial developers and testers could be beneficial. Talented developers usually dislike debugging code not their own unless this is absolutely necessary; they want to create their own code. Moreover, if we assume that ten talented developers work constantly to isolate one given bug, this seems like a "water hose" approach. Improved project management could be more beneficial than assigning ten developers to a single bug could. Talented developers who understand a given system are extremely scarce and this sort of approach is extremely wasteful.

A bug is a bug is a bug.
No. There are at least three major types of bugs - code errors, logical errors and architectural problems. One large subclass is just normal coding bugs that are easy to find. Another subclass is errors in logic that are approximately one order of magnitude harder to find and fix. The most complex bugs are consequences of architectural flaws or limitations of tools used. Numerous architectural flaws of Linux are widely known and are being corrected over time. From a software engineering perspective, those flaws are similar to the architectural problems of any prototype that was converted into a production system. Initially Linux was expected by many to be a temporary system to be eventually replaced by GNU

Hurd[140]. Hurd has an arguably better architecture than Linux.[141] Many OSP stagnate by debugging software that is architecturally inferior rather than solving the underlying problems. Debugging does have a higher status in OSP than software architecture. This is mainly due to the macho roots of many OSP in the hacker culture.[142]

Is it easy to force gifted programmers to search for the same bug?
It depends. Usually forcing many talented developers to look at the same piece of code is like herding cats. It's neither easy, nor a rewarding task, unless the bug is really critical (technically or politically). Talented developers are first programmers not testers; they usually prefer making their own bugs to fixing bugs of others. Any situation where many talented developers actually work on the same segment of code is more of an exception than a rule. With the increasing complexity of a given project, this pooling of talent will occur very rarely and only for the most critical or politically important bugs. For any sufficiently complex project there will be never be sufficient "eyeballs" to locate and eradicate all bugs. Many large OSP are far more complex than any individual can handle, and hence the amount of developers even able to understand a piece of software with all its dependencies is very small.

Does it make sense to fix bugs in badly written code?
Usually it does not. If the code is badly written (and as Ken Thompson pointed out, some parts of Linux are[143]) additional bugs could be easily introduced by fixing an existing one. Rewriting, not fixing, is a more viable option here. The open source model, with its over-reliance on debugging, could be at a disadvantage. In the commercial environment, a talented manager could partially avoid this problem by exercising his or her judgment and power. In OSP, modules incorporated at early stages of a given project could outlive their utility in short order. Thanks to inertia and programming overload on key developers, there may be no effort to rewrite these modules until serious problems occur that justify the effort. Sometimes only when a problem becomes visible will you be able to attract a decent developer to rewrite code - a very unrewarding task indeed, as any programmer can attest. In this case, the publicity about the bug makes fixing it a worthwhile investment in the status game (often the situation with security bugs). The seemingly infinite number of bugs precludes the positive influence of random bug fixing on the product as a whole.

4.1.1.3 Does Linux belong to the Cathedral or to the Bazaar model?

Many have pointed out that the level of decentralization in the Linux world is open to review. The model in CatB is too simplistic. The metaphors for high centralization

---

[140] http://www.gnu.org/software/hurd/hurd.html
[141] http://www.cs.pdx.edu/~trent/gnu/hurd/hurd-paper.html argues this point.
[142] http://www.tuxedo.org/~esr/jargon/html/entry/hacker.html notes: “Hackers consider themselves something of an elite (a meritocracy based on ability), though one to which new members are gladly welcome. There is thus a certain ego satisfaction to be had in identifying yourself as a hacker”
[143] http://www.computer.org/computer/thompson.htm

(Cathedral) and no centralization (Bazaar) do not account for the size of a given project; its complexity, timeframe and time pressures and its access to resources and tools. They also do not specify whether a project produced core functionally (like the Linux kernel) or peripheral parts of the system. For large projects like operating systems, it is especially important that the core of the system be developed in a highly centralized fashion with a small core team. Peripheral parts of the system can benefit from a more relaxed, more decentralized approach. CatB fails to distinguish between these two types of activities, as the following quote demonstrates:

"In retrospect, one precedent for the methods and success of Linux can be seen in the development of the GNU Emacs Lisp library and Lisp code archives. In contrast to the cathedral-building style of the Emacs C core and most other FSF tools, the evolution of the Lisp code pool was fluid and very user-driven. Ideas and prototype modes were often rewritten three or four times before reaching a stable final form. And loosely-coupled collaborations enabled by the Internet, á la Linux, were frequent."

The Emacs C core and the Lisp code are different and should be examined with different models in mind. There are advantages in using mixed models other than the pure centralized (Cathedral) or completely decentralized (Bazaar) extremes. It is hardly surprising that in reality a mixed model dominates or that there is a place for highly centralized development in the Linux world. Linus Torvalds once noted: "*My workload is lower because I don't have to see the crazy ideas. I see the end point of work done for a few months or even a year by other people."*

One can immediately see elements that are foreign to the Bazaar style in the current stage of the Linux kernel development as described by the principal author of the kernel. It looks more like a highly centralized (Cathedral) development model. For example, you cannot communicate with Torvalds directly but need to supply patches to his trusted lieutenants. If the patch is rejected, there is no recourse. [Hubbard98] notes: *"Despite what some free-software advocates may erroneously claim from time to time, centralized development models like the FreeBSD Project's are hardly obsolete or ineffective in the world of free software. A careful examination of the success of reputedly anarchistic or "bazaar" development models often reveals some fairly significant degrees of centralization that are still very much a part of their development process."*

Of course, these arguments do not exclude the fact that some activities in Linux can be classified as belonging to the decentralized (Bazaar) model, especially development of drivers, utilities and small applications.

4.1.1.4 Does the OSP model automatically yield the best results?

CatB postulates that OSP deliver higher quality results than CSP: "Perhaps this should have been obvious (it's long been proverbial that "Necessity is the mother of invention") but too often software developers spend their days grinding away for pay at programs they neither need nor love. But not in the Linux world - which may explain why the average quality of software originated in the Linux community is so high."

Looking at sourceforge.net, one of the biggest repositories of OSP, one gets a completely different impression. 95% of all projects are of exceptionally bad quality, or do not even have any results to show. An anonymous jokester aptly summarized it with it his advice how to develop open source software: [144]

> "Open Source Development HOW-TO (Score: 5, Funny)
> by Anonymous Coward on Tue July 09, 14:17 (#3851076)
>
> 1. Introduction
> As everyone knows, Open Source software is the wave of the future. With the market share of GNU/Linux and *BSD increasing every day, interest in Open Source Software is at an all time high. Developing software within the Open Source model benefits everyone. People can take your code, improve it and then release it back to the community. This cycle continues and leads to the creation of far more stable software than the 'Closed Source' shops can ever hope to create.
>
> So you are itching to create that Doom 3 killer but do not know where to start? Read on!
>
> 2. First Steps
> The most important thing that any Open Source project needs is a Sourceforge page. There are tens of thousands of successful Open Source projects on Sourceforge; the support you receive here will be invaluable. OK, so you have registered your Sourceforge project and set the status to '0: Pre-Thinking About It', what's next?
>
> 3. Don't Waste Time!
> Now you need to set up your SourceForge homepage. Keep it plain and simple - do not use too many HTML tags, just knock something up in VI. Website editors like FrontPage and DreamWeaver just create bloated eye-candy - you need to get your message to the masses!
>
> 4. Ask for Help
> Since you probably cannot program at all you will need to try to find some people who think they can. If your project is a game, you will probably need an artist too. Ask for help on your new Sourceforge pages. Here is an example to get you started:
>
> "Hi there! Welcom to my SorceForge page! I am planing to create a Fisrt Person Shooter game for Linux that is going to kick Doom 3's ass! I have loads of awesome ideas, like giant robotic spiders! I need some help thouh as I cant program or draw. If you can program or draw the tekstures please get in touch! K thx bye!"

[144] http://slashdot.org/comments.pl?sid=35668&op=Reply&tid=99 &pid=3851076

> Thousands of talented programmers and artists hang out at Sourceforge ready to devote their time to projects so you should get a team together in no time!
>
> 5. The A-Team
> So now you have your team together you are ready to change your projects status to '1: Pre-Bickering'. You will need to discuss your ideas with your teammates and see what value they can add to the project. You could use an Instant Messaging program like MSN for this, but since you run Linux, you will have to stick to e-mail.
>
> Do not forget that YOU are in charge! If your team does not like the idea of giant, robotic spiders just delete them from the project and move on. Someone else can fill his or her place and this is the beauty of Open Source development. The code might end up a bit messy and the graphics inconsistent - but it is still 'Free as in Speech'!
>
> 6. Getting Down To It
> Now that you have found a team of right thinking people you are ready to start development. Be prepared for some delays though. Programming is a craft and can take years to learn. Your programmer may be a bit rusty but will probably be writing "hello world" programs after school in no time.
>
> Closed Source games like Doom 3 use the graphics card to do all the hard stuff anyhow, so your programmer will just have to get the NVidia 'API' and it will be plain sailing! Giant robot spiders, here we come!
>
> 7. The Outcome
> So it has been a few years, you still have no files released or in CVS. Your programmer cannot get enough time on the PC because his mother will not let him use it after 8pm. Your artist has run off with a Thai She-Male. Your project is still at '1: Pre-Bickering'...
>
> Congratulations! You now have a successful Open Source project on Sourceforge! Pat yourself on the back, think up another idea and do it all again! See how simple it is?

With all these issues that manifest themselves with Raymond's theory, what parts of it remain as valid contributions? The observation that debugging can be parallelizable under some circumstances has merit, likewise for the assumptions made about individual motivations to contribute to Open Source. Above all though, Raymond's accessible style led to a wide dispersal of his ideas, and influenced many companies and individuals in their thinking towards Open Source. Any theory that aims to appeal and entice its readers to apply its findings ought to be written in such an attractive style. This runs against the grain prevalent in academia, where precision stands above all else. It is an illusion to believe that a precise definition of Open Source will be possible any time soon. Rather, it makes sense to accept the incompleteness and weakness of any given theory, and instead

focus on applying the insights that have been gained to date. It is my belief that a pragmatic theory, or more precisely, a framework can do more to advance general understanding about Open Source, and make a contribution to the field than a purely academic theory.

Interestingly, the Open Source community bears a more than passing resemblance to the academic community. Many of the same processes that dominate in academia are also prevalent in Open Source. Peer reviews, status competition, credits as a major currency are some of the similarities. What does this comparison between the two communities reveal about Open Source?

4.1.2 Open Source as academic Research

Nikolai Bezroukov developed an alternate theory for Open Source, called "Open Source as academic research". [Bezroukov99a] introduces his theory thus:

"First of all I would like to stress that the Internet can significantly reduce the costs of providing some types of software like OS, compilers or utilities. The Internet makes it possible to produce an infinite number of remotely accessible perfect copies of a computer program, multimedia presentations, or interesting e-mail discussions. It is generally accepted that the fact that the cost of duplicating a computer program is close to zero creates important differences between computer programs and other consumer products. These differences are to a certain extent ignored or suppressed in the conventional "shrink-wrap" software distribution model. I think that the creation of a program is similar to the creation of applied theory. I would like to classify programming as a special kind, or at least a close relative, of scientific activities. The second important difference between software and other consumer products is that minor modifications are easy. Software can be more adaptable. Here again the important analogy with science holds. In both science and programming, those involved are not in it for the money. Most of the Open Source developers are doing it to chase a dream, not to build up their bank balances. They can be motivated for a variety of reasons but simply there are many with great programming abilities that often are underutilized in their current (often corporate) environments. Some want a particular tool and found that is either unavailable or too expensive so they try to build one themselves. Again, they are chasing a dream, not the competition."

This generalization seems too broad. As others have pointed out, economic incentives play a major role for OSP. In addition, Bezroukov notes the case of shrink-wrap software, which only accounts for a very small part of software. Most software is produced for internal use in organizations, and distribution costs do no matter in that context. Bezroukov claims that minor modifications are easy. That does not necessarily hold, since most software is not modularized enough to allow modifications without side effects.

"*Due to the Internet, it is now possible to create a virtual team for software development, parallel to the interests in science where several researchers interested in a particular phenomenon create a virtual scientific community to resolve a question or problem.*

Team structure and responsibilities can be a dynamic process. Software will be produced not by an isolated person or a group of persons, but by a deeply interwoven network of actors. I would like to stress that virtual teams probably resemble historical informal communities of scientists that used hand-written letters to share achievements, ideas and criticism. Yet little is known about how Internet-based virtual teams (IVT) really operate and what problems develop in that sort of cooperation. Some evident problems are:

- *overload and subsequent burnout of leading developers due to excessive load, with significant loss of interest in a given product;*
- *a conservative approach to architecture (it's really difficult to change the architecture of a product after its development has started)*
- *E-mail based written communication to some degree tends to distort meaning and invite fights and flame wars*."

Bezroukov fails to draw the parallels to the scientific community here. [Cox99] points out that most interchanges between developers do have a very low signal-to-noise ratio because many non-contributors take part in the discussion, but do not offer useful comments. These "lurkers" can be very damaging for a project to the extent that they obscure the communication of contributors. The scientific community, by contrast, does arguably have much less to worry about non-contributors clogging their communication, as interest in the scientific discourse is usually limited to the scientific community. This "elitism" of the scientific process has disadvantages, of course. It may preclude feedback from stakeholders that would participate if they would only understand the language being used, for one.

"*There are several other visible problems with open source projects that were pointed out by others. Understanding these problems probably can help current and future Open Source projects. Again, I strongly believe that the problems of Open Source are by and large the same as those that confront academic culture and are better understood in this context. From a theoretical point of view, participants in an Open Source project should probably be considered as a special kind of academic workers. Solutions for typical problems developed in academic community are directly applicable to Open Source and its use can probably provide important benefits to the Open Source development model*."[145]

The jury is still out on whether these similarities are relevant enough to allow application of scientific solution to OSP problems. Areas that are not yet well established in OSP are peer review and a system for giving credits. Peer review does happen, albeit in very informal ways, and the results are unpredictable. Having ways to solicit peer reviews would help to increase quality across the board. [Ghosh02] mentions how difficult it is to attribute credit in OSP. Seeing how credits are the major benefits participants get out of OSP, this is a major deficiency.

---

[145] See also [Scacchi02b], page 4

The financing of OSP is very similar to the financing of applied science. Sometimes scientific research is funded indirectly, because individuals employed in a particular institution become interested in a particular phenomenon and research with existing funds. A considerable number of open source software developers work either in academic institutions or large corporations. Large corporations usually provide a very good environment for open source projects as they often contain niches where gifted developers are partially unemployed or underemployed due to various internal issues. The initial development of UNIX at Bell Labs is a good example of such possibilities. Recently several large corporations used a second avenue of financing of open source projects. When a given open source project can promote a strategically important hardware component, programmers can be assigned to it. This activity becomes a standard "loss leader". Digital contributions to Linux seem to belong to this scheme. Current Intel and IBM interest in Linux seems to fall into this category.

[Raymond00] does have a more extensive treatment of OSP funding models. Raymond mentions cost-sharing, risk-spreading, loss leader and widget frosting as possible OSP funding models. To summarize, OSP do have some similarities with the scientific process, but it is far too early to conclude that they closely follow it. Further areas of research should include peer review in OSP and attribution of credit.

The notion of giving credit to contributors underlies many economic approaches. In the absence of monetary flows, economists have tried to discover alternative wealth-distribution channels to frame Open Source in well-known economic terms. It is generally assumed that individuals act like the proverbial "homo oeconomicus", meaning that they are ultimately motivated by profit. Discovering the source of profit, and ways to allocate it efficiently are at the center of these economic approaches.

4.1.3 Economic Approaches

Various scholars have attempted to explain OSP with economic theories. While their theories offer important insights into motivational aspects, they are incomplete, and fail to explain OSP on their own. The most obvious criticism is that Open Source business models have largely failed in the marketplace. How can economic theories of Open Source be sound if they lack empirical evidence (read: thriving businesses) to back them up? To be fair, this down-to-earth look at economic theories does not do them justice. Research into economic motivations is very important for the long-term health of the Open Source community. Most participants would not mind at all if sustainable business models would emerge eventually.

One of the earliest theories that apply to Open Source is Goldhaber's notion of the attention economy.

4.1.3.1 Attention economy

[Goldhaber97] established the notion of the "Attention Economy" which positions attention as the scarce resource, and builds an economy around it.

So, at last, what is this new economy about? Well if the Net exemplifies it, then you might guess it has less to do with material things than with the kinds of entity that can flow through the Net. We are told over and over just what that is: information. Information, however, would be an impossible basis for an economy, for one simple reason: economies are governed by what is scarce, and information, especially on the Net, is not only abundant, but overflowing. We are drowning in the stuff, and yet more and more comes at us daily. That is why terms like "information glut" have become commonplace, after all. Furthermore, if you have any particular piece of information on the Net, you can share it easily with anyone else who might want it. It is not in any way scarce, and therefore it is not an information economy towards which we are moving. What would be the incentive in organizing our lives around spewing out more information if there is already far too much?

Well, my title gives it away, of course. There is something else that moves through the Net, flowing in the opposite direction from information, namely attention. So seeking attention could be the very incentive we are looking for. Parenthetically, I have now rejected both parts of the conference title; no economics in the conventional sense, and not digital information either. You might conclude I am speaking at the wrong conference. I would rather say it has the wrong title. Except the title did serve its purpose. It did get your attention, and that was something, in fact a lot.

Ignoring for a while that Goldhaber plays semantic games with his audience, his notion of "spewing out more information" as an action spurred by the desire for attention is valid.[146] Goldhaber goes on to explain:

Attention, at least the kind we care about, is an intrinsically scarce resource. Consider yours, right now. You are reading this paper. You have a certain stock of attention at your disposal, and right now, a large proportion of the stock available to you is going to me, or to my words. If you are just reading this, assuming it gets printed in a book, the fact that your attention is going to me and not just to what I write may be slightly less obvious. Now this might not matter if attention were not desirable and valuable in itself, but it is. In fact, it is a very nice feeling to have respectful attention from everybody within earshot, no matter how many people that may include. We have a word to describe a very attentive audience, and that word is "enthralled." A thrall is basically a slave. Right now, it should be evident that having your attention means that I have the power to bend your minds and your bodies to my will, within limits that in turn have to do with how good I am at enthralling you. This can be a remarkable power. When you have superb control over your own body, so that you can perform great athletic feats, it feels great; likewise, it feels good when your mind feels focused and powerful; how much more wonderful then to be able to have the minds and bodies of others at your disposal! On the rather rare occasions when I have felt I was holding an audience "in the palm of my hand, hanging on my every word," I have very much enjoyed the feeling, and of course others who have felt the same have reported their feelings in the same terms. The elation is independent of what you happen to be talking about, even if it is to decry something you think is horrible.

[146] [Hannemayr99] researches attention inside the hacker community.

Did this account get your attention? In all likelihood, it did. Now what? It is a long way from getting someone's attention to use it for your own advantage. The attention economy may be happening for a select few who are in the public eye, but it fails to account for countless unknown contributors that stand very slim chances of ever getting their due recognition. OSP typically only support a handful of participants in a very public role. Most participants will not partake in the benefits of being recognized, and "getting attention" cannot compensate them for their efforts.

The attention economy seems far-fetched. Ultimately, it might well come to pass if more and more of the value creation in society happens in the services sector, and physical goods lose importance. In such a future scenario, attention gains in value, and indeed, there are first signs that such an environment is beginning to take shape.[147] More immediately useful is the notion of Cooking-pot markets by Ghosh.

4.1.3.2 Cooking-pot Markets

[Ghosh98] developed the concept of "Cooking-pot Markets" where participants contribute resources to a "pot" and take out other resources. The model works because software can be copied without loosing the original, and hence "taking out of the pot" will not adversely affect other participants.

Unlike the markets of the "real world", where trade is denominated in some form of money, on the Net every trade of ideas and reputations is a direct, equal exchange, in forms derivative of barter. This means that not only are there two sides to every trade as far as the transaction of exchanging one thing for another goes - which also applies to trades involving money - there are also two points of view in any exchange, two conceptions of where the value lies. (In a monetary transaction, by definition, both parties see the value as fixed by the price.) When I buy your book about cats, it's clear that I am the consumer, you the producer. On the Net, this clear black-and-white distinction disappears; any exchange can be seen as two simultaneous transactions, with interchanging roles for producer and consumer. In one transaction, you are buying feedback to your ideas about cats; in the other, I am buying those ideas. In the "real world" this would happen in a very roundabout manner, through at least two exchanges: in one, I pay for your book in cash; in the next, you send me a cheque for my response.

The core idea here is that idea exchanges on the Internet are barters, and hence not subject to a price in the classical sense.

As soon as you see that every message posted and every Web site visited is an act of trade - as is the reading or publishing of a paper in an academic journal - any pretense at an inherent value of economic goods through a price-tag is lost. In a barter exchange, the value of nothing is absolute. Both parties to a barter have to provide something of value to the other; this something is not a universally or even widely accepted intermediary such as money. There can be no formal price-tags, as an evaluation must take place on

[147] http://www.metaverselab.org/

the spot at the time of exchange. When you barter you are, in general, not likely to exchange your produce for another's in order to make a further exchange with that. Unlike the money you receive when you sell something - which you value only in its ability to be exchanged for yet another thing - in a barter transaction you normally yourself use, and obviously value, what you receive.

The fallacy of this argument becomes clear when you consider typical producer-consumer relationships in OSP. The author of software may gain very little from the feedback a random user is able to provide. In such a case, the barter is very uneven and unsustainable over time. Barter exchanges may explain incentives for fellow developers who use software of their peers, but does not address end users who most likely are unable to contribute back. As tools become more powerful, it might be feasible for less technically inclined people to contribute back. Often, these people would very much like to contribute, but they lack the tools to transform their ideas into workable solutions.

Just as the existence of the thousands of independent Linux developers are valuable to the newspaper because they are also users of the product - and may face similar problems - other Linux developers welcome the Times of India because how it faces its problems could help them as Linux users. As Torvalds says, " [t]here are lots of advantages in a free system, the obvious one being that it allows more developers to work on [Linux], and extend [Linux]." However, "even more important" is that making Linux free brought "in one fell swoop ... a lot of people who used it" - not just reporting problems, but also playing a crucial role in the further development of the system. Torvalds notes that a single person or organization "doesn't even think of all the uses a large user community would have for a general-purpose system" - so the large user base of Linux was "actually ... a larger bonus than the developer base."

This again assumes that users are vocal, and take the steps to report their experiences and wishes back to a project. Collecting feedback does not scale. As the user community for a project grows, most feature requests tend to be duplicates. The author has been involved in an OSP. In July 2002, there were 473 open feature requests out of 837[148]. At least a third of those were duplicates. Another problem was the very low quality of most feature requests. They were either too vague, too specific or had other deficiencies, making it difficult to turn them into requirements, and eventually, working code. Coming back to the Linux example, there is very little feedback from end users to the authors of the Linux kernel. Instead, most is filtered through software vendors or other developers before it reaches the kernel authors. Thus, it is a bit of stretch to claim that a barter is happening between users and kernel developers. As a project grows, it is less and less appealing for users to contribute because they figure someone else will contribute instead. Large OSP fall prey to the tragedy of the commons, with all its symptoms.

A personal experience by the author may help to illustrate this. For the time period from July 2001 to August 2002, I contributed to the PostNuke project.[149] I spent about 40 hours a week helping to develop the core architecture of this community-oriented Content

[148] http://sourceforge.net/tracker/?atid=392231&group_id=27927&func=browse
[149] http://www.postnuke.com

Management System. A very enjoyable experience and I learned a lot in a short amount of time about software engineering, interacting with large virtual groups and conducting a dialogue across language barriers and time zones. The project, meanwhile, exploded in popularity, and attracted thousands of users. To date, the software has been downloaded over 500'000 times, and been installed on tens of thousands of web sites. This popularity led to an interesting phenomenon. The more users the project attracted, the more newcomers that knew little or nothing about the conventions of the Open Source community assaulted the social fabric that had held the project together. The common courtesy among fellow developers made room for clamors for more, better, faster, and appreciation declined. Cries of conspiracies emerged, and many developers grew increasingly frustrated with the monster that they had created. Andy Varganov, of Manchester, UK summed the feelings of many longtime contributors up as "users are a pain in the ass". The situation got so bad that most contributors quit in the same week. What had led to this rapid burnout?
It turned out that many newcomers to Open Source mistakenly thought that their ideas would necessarily have to be considered by virtue of them stating these ideas. Open Source was understood as a "free for all" with mob rule. Technical decisions were challenged because some constituents felt it was all a big conspiracy, and that their voice had not been considered. They failed to understand that software engineering needs to make hard choices, and that an amalgamation of all ideas does not lead to a workable product. Despite a total lack of understanding how software production works, they felt compelled to opine. This in turn frustrated the few contributors in the project with actual software engineering expertise. Rather than submit to the whims of random people, they preferred to quit the project and rebuild it out of the public view. They had gone through a very quick maturation process, and after one year, they concluded that the cathedral style was more appropriate for the future of their project than the bazaar it had started with. Outlining one of the strongest motivators for developers, Carl Corliss, of Martinez California, stated, "I am only participating in this project if it is fun again". It will be interesting to watch whether the PostNuke project will be able to survive without most of its contributors, and whether the "band of rebels" that decided to quit PostNuke will be able to correct the mistake that have been made in the past.

The mechanics of groups undoubtedly play a central role in OSP. It is not astonishing to find various theories tackling Open Source from this angle. Dafermos establishes the concept of virtual decentralized networks, and contrasts them with earlier, more rigid models of organization to arrive at some very interesting conclusions. How and why do virtual organizations out-compete traditional organizations, and what does that mean for the Open Source phenomenon?

4.1.4 Virtual Decentralized Networks

[Dafermos01] uses the example of the Linux project to develop a theory of virtual decentralized networks. His theory aims to explain the phenomenon of the Linux project by looking at the organization behind it. Some main tenets can be identified, namely that motivation is a crucial element; that even virtual organizations do have some sort of boundaries and that virtual organizations may be more successful because they utilize

knowledge better. Dafermos arrives at some fairly broad conclusions that, while useful, do not readily apply to concrete projects.

4.1.4.1 Motivation is the source of sustainability

Traditional organizational structures are based on deadlines, payroll roles, fixed positions, etc. and this is unlikely to attract creative people [78]. Creative people are excited by fluid structures that provoke competition among peers. The incentive is to differentiate yourself by rising above the standard level (through achieving excellence) and gain the recognition of your peers that regard you as a leader. This is the ultimate form of this motive and recognition by your boss in a bureaucratic environment is the least.

Motivation alone sustains nothing, as motivation neither puts food on the table, nor does it satisfy all intellectual cravings. Motivation is definitely involved in OSP, but it is far too broad a concept to explain OSP.[150]

4.1.4.2 The Virtual Roof

*The Linux Project is characterized by flow of information within the entire organization, implying among the thousands of developers, Linus Torvalds and the "trusted lieutenants". All of them have access, can engage into all occurring conversations and can communicate directly with every other member-implementer through discussion forums, mailing lists, and newsgroups. But the flow of information is not restricted only within implementers but it extends to the global community reaching virtually everyone interested, including commercial companies (i.e. Cygnus Solutions, SuSe, Red Hat, VA Linux) that provide support services and packaged distributions, computer scientists that may have not been involved directly (as implementers), companies that consider adopting open source software for their own internal use, users that need help from experts and anyone interested or curious enough to observe or even participate in any given conversation. Access to the various open source-related Web sites, discussion forums, etc. is open to the public and all interested parties. Figure 7 shows that communication and information flow is so pervasive that spreads equally towards all directions and is diffused throughout the virtual roof. The lines and the arrows represent information flow and specific project functions (transfer of patches-source code) whereas information flow pervades the virtual roof and is diffused towards all direction*s.

The Virtual Roof fails to account for the communication bottlenecks that become apparent by looking at Figure 7. Communication is not pervasive if it has to be routed through a small number of individuals. Nevertheless, the virtual roof captures the element of modularization very nicely. Modularization is the key for scalability. By relaying only select messages between modules, the strain is limited to the persons intermediating between modules.

[150] [Lancashire01] dismisses motivation as a primary driver of Open Source activity.

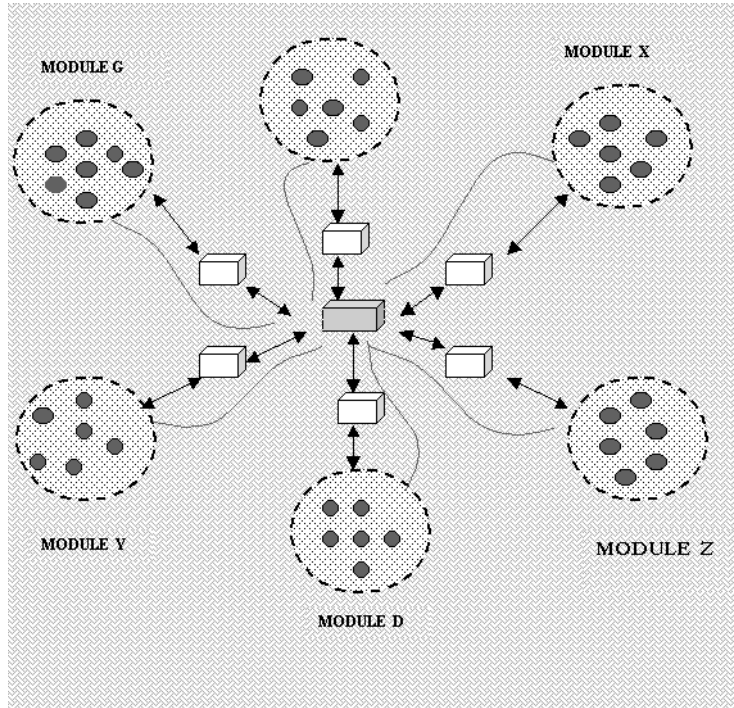

Figure 7: The Virtual Roof (Source: [Dafermos01])

4.1.4.3 Knowledge is the competitive advantage

To date, organizational thinkers had been mainly concerned with how the 'virtual organization' will carry out the same identical functions-activities with the 'physical organization'. Literally, how can the virtual organization duplicate exactly what the physical organization does? This is a wrong question to start with. Perhaps, a more appropriate question would be what could the virtual organization do that the physical organization cannot (replicate)? The answer lies on the fact that by bringing the organization (implementers), the surrounding global community - industry and the end users - customers together, it can generate massive knowledge and exploit it in the most effective way (Figure 8).

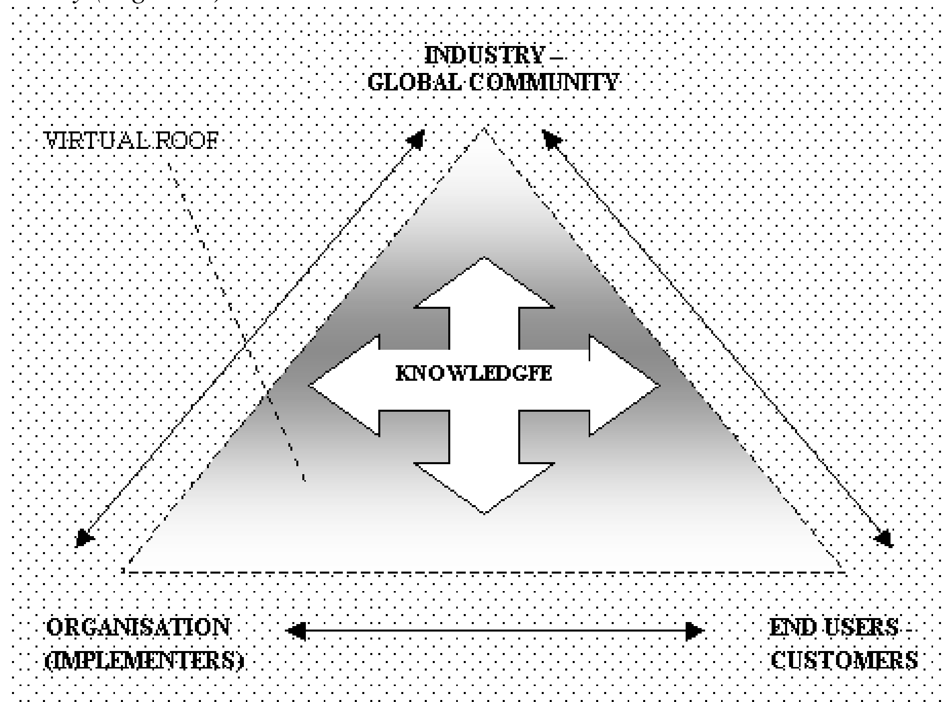

Figure 8: Knowledge exchange in a virtual organization (Source: [Dafermos01])

It remains to be seen if a virtual organization is superior in collecting and diffusing knowledge than physical organizations. Physical organizations do have access to the same knowledge stores that virtual ones do, primarily the Internet. Organizations that are able to bring their members together face to face are able to circumvent misunderstandings that online communication would entail. To phrase it differently, speaking with a person in the same room does have a significantly higher communication bandwidth than sending an email does.

The Dafermos model is mainly useful to raise awareness for some attributes of the virtual organization that are beneficial to OSP success. Once familiar with its concepts, it may seem obvious, but it definitely helps an organization if its members are aware of the opportunities and pitfalls of their organizational structure. Especially the notion of channeling information through special conduits to reduce network complexity helps to understand why some project members are notoriously busy, and can incite responsible

behavior in project members. Made aware of these limitations, project contributors tend to think twice before sending material that is of little interest to the group to these key persons.

If theories succeed in fostering sensible behavior in project participants, they have succeeded far beyond their goals. Many OSP are faced with participants that do have few clues as to how their project runs on a sociological level, or what psychological processes are contributing to decision-making in OSP. A few researchers have tried to identify the psychological processes.

4.1.5 Psychological Models

A group at the University of Kiel, Germany conducted a study about the psychological factors in the development process of the Linux kernel [Hertel02]. The study concluded with the following results:

- *Successful Linux (subsystem) developers consider their project to be important (valence) for Linux.*
- *Reward motives are the main reason to further increase efforts.*
- *"Soft" criteria (social motives and trust) determine the satisfaction and contribution to organizational issues but not productivity.*
- *No influence of idealistic collective motives was found.*
- *The Linux Kernel is a product of a few very hard working developers.*
- *Developers expect (and receive) rewards from the involvement in Linux development.*

[Hertel02]: "This motivation can be based either on feelings of personal challenges to improve existing software, on competitive motives to compete with other developers (either within OSS or between OSS and commercial software projects), on simple fun to program, or on motives to build a reputation that might be helpful for the further career. Unfortunately, apart from these (plausible) speculations, there are no empirical data available that explore the motives of developers in OSS projects more systematically."

The results of this study contrast sharply with accounts that OSP are motivated by "the greater good". Instead, a sense of purpose for individuals seems crucial. These conclusions seem to strengthen the economic argument that project participants ultimately act out of egoistic motives, and places emphasis on satisfying these needs.

[Weber00] reflects on the oft-cited reputation as a main currency for Open Source developers.

Reputation is a powerful motivating force for open source developers. But there are strong reasons to believe that reputational concerns by themselves cannot explain successful collaboration. If reputation were the primary motivation, we should be able to tell some version of the following story. Assume that project leaders like Torvalds receive greater reputational returns than general programmers do. Then programmers should compete to become project leaders, particularly on high profile projects, of which there

are a limited number. This competition could take at least two forms. We would expect to see a significant number of direct challenges to Torvald's leadership -- but in fact there have been few such challenges, none serious. Alternatively, we could see 'strategic forking'. A strategic forker would fork a project not for technical reasons per se, but rather simply to create a new project that he or she could lead. The problem of how to attract other programmers would be managed by credibly promising to maximize other programmer reputations on the new project -- for example, by sharing some of the gains with anyone who joins. In that case, a new programmer would be motivated to join the forked project rather than Linux.

It seems that there are strong forces at play in OSP that prevent such disruptive behavior. Successful OSP do have very strong shared values, as [Edwards00] points out. Looking at individual psychology is not sufficient; instead, it is necessary to consider group dynamics.[151]

Clearly, existing theories are not able to explain Open Source satisfactorily, nor are they able to offer much support for the budding OSP manager or participant. A new theory needs to be constructed, based on the best of existing theories, and with new additions. Coming up with new theories poses considerable challenges though, especially in an area like OSP that defies precise measurement. A new theory of Open Source will have to face the same challenges that all social science theories do. What are these challenges, and do they explain why relatively few theories have emerged?

4.2 Challenges for Theory Formulation

Developing theories about Open Source is faced with various challenges. Among these challenges are data collection, a proper determination of the subject of study, and ways to verify a given theory. Acknowledging these challenges helps to determine proper scopes and methodologies for describing the Open Source phenomenon.

4.2.1 Data Collection

The absence of empirical, factual and verifiable data on a large scale is clearly a major disadvantage to most kinds of research into the OSP phenomenon. Previous experience from the very few extensive surveys carried out so far[152] suggest that quantitative and qualitative empirical survey methods can be useful. However, surveys can introduce biases that can be difficult to calculate. Tools and systems to analyze the traces left behind by collaborative activity in the form of software source code or discussion archives can be much more useful in finding hard facts. The OSP community is much talked about, though little hard data on this community and its activities has been collected. This is mostly due to the fact that it is very time-consuming to aggregate data from various online repositories, and normalize them. Each discussion group unfortunately has its own format, and needs to be handled separately. Some preliminary tools for analyzing and displaying these relations have been developed. Figure 9 displays connections between different postings on Usenet, the largest discussion system on the

[151] [Kaisla01] explores constitutional dynamics in OSP

Internet. Netscan might, with some extensions, be used to map OSP relations in the future.

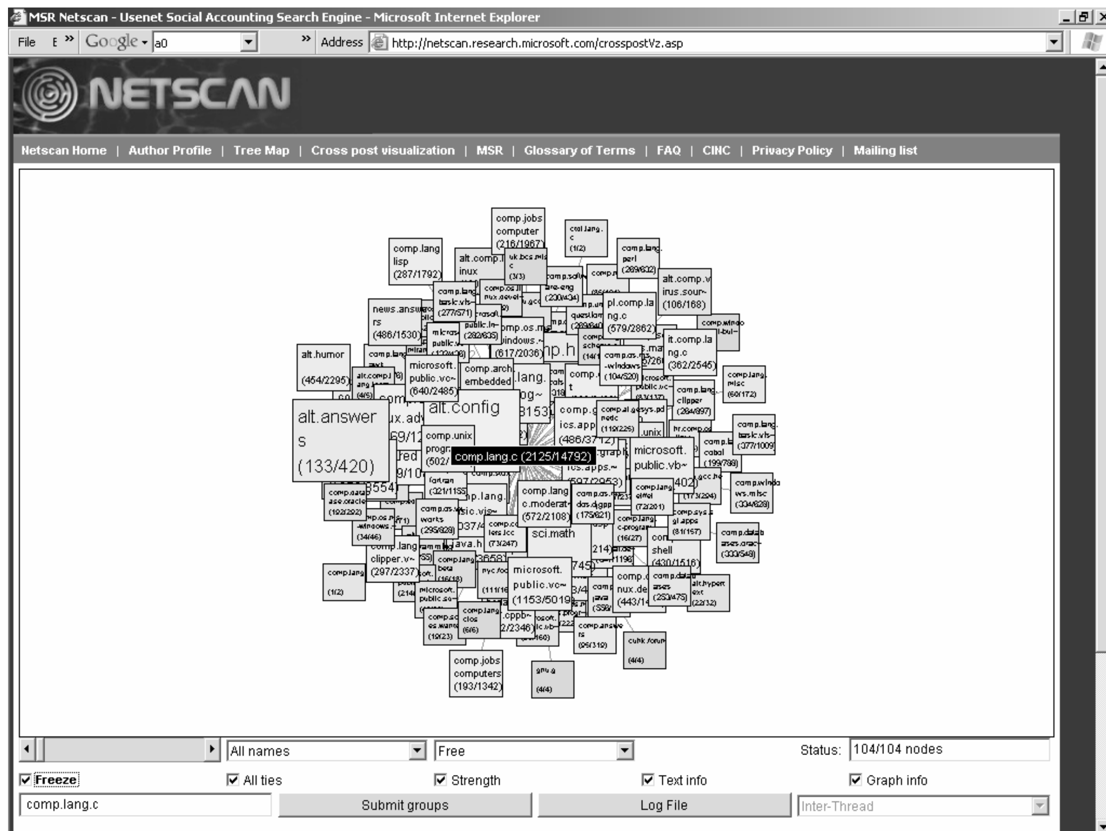

Figure 9: Visualization of Usenet postings

Nevertheless, collecting relevant data remains a challenge. Free software and Open Source are considered competing definitions of this community or phenomenon. For researchers into this phenomenon, especially economists, the fact that software is available free is what makes analysis difficult since tools for measurement without the use of money are not sufficiently advanced.

Even if appropriate tools and techniques were available, the question of what to conduct research on lingers.

4.2.2 Subject of Study

The sheer number of OSP makes it very difficult to determine any but the most superficial similarities. Projects have one to several thousand developers, are days old or have been going on for decades. Projects are in different phases of their live cycle. Dozens of legal definitions of "Open Source" exist. It is unlikely that a theory will be

[152] Among the studies conducted so far are:
BCG02: Survey conducted by the Boston Consulting Group; see www.osdn.com/bcg
FLOSS02: Background/project outline see http://www.infonomics.nl/FLOSS/
WIDI (2001): Who Is Doing It, survey conducted by Technical University Berlin. See http://widi.berlios.de

able to encompass all these differences, let alone precisely define what "Open Source" means. Open Source touches many different fields of academic study, and no one field is able to represent it on its own. Where should one focus his research? Establishing terminology for a single discipline is difficult enough, and doing the same for interdisciplinary studies presents additional difficulties.

Finally, validating a theory in the field of social sciences is not an easy matter either.

4.2.3 Theory Validation

Building up enough evidence to validate any theory is a lot of work. Theories that deal with human behaviors are notoriously difficult to validate. The necessary statistical confidence requires extensive and precise data collection. Many of the necessary techniques for data collection and surveys have had little time to mature and adapt to the internet environment. It is therefore unlikely at this point that a definitive theory of Open Source will emerge. It makes much more sense to collect the existing knowledge about Open Source into a framework, and offer practical advice rather than trying to build a huge theoretical foundation for Open Source.

With those weaknesses of theory development and validation acknowledged, how does one approach a framework? What are necessary components, what is a viable methodology?

4.3 A framework approach

In light of the shortcomings of earlier theories, and the acknowledged challenges for theory formulation, a new approach is called for. This approach is based on a framework that will be incomplete as it is started, but will provide facilities to integrate more aspects as the common knowledge about Open Source increases. Over time, the framework should grow to integrate newer insights, and add more tips for budding OSP participants. The framework has its own goals, is fed by various inputs, makes some assumptions, and follows a methodology.

What goals does the framework strive towards?

4.3.1 Framework Goals

Primarily, the framework aims to be more useful than earlier theories to explain Open Source, and offer advice in its application. To reach this goal, the following properties seem helpful.

Expressiveness
The framework should provide new insight into OSP phenomena, and make useful predictions. It should aim to boil issues down to easily identifiable concepts and trends.

Extensibility
It is anticipated that the framework will be incomplete. Hence, methods to extend it need to be provided. This will allow the framework to acquire new knowledge

about OSP and integrate it. New knowledge should be added in a controlled matter to prevent the framework from becoming a hodgepodge of different theories. The framework needs to be generic enough to allow for a broad range of concepts to be integrated, yet needs to be rigid enough to set some structure for the whole framework.

Applicability
The framework aims to offer applicable advice to OSP participants in addition to being a theoretical foundation for the study of OSP artifacts. Each explanation should be followed by practical advice. The benchmark for the addition of new material to the framework should be whether this material adds to the understand and the application of the framework.

Which inputs would help to achieve these goals? What major areas of previous theories merit inclusion and what areas have been neglected to date?

4.3.2 Framework Inputs

As outlined earlier, many existing ideas make sense and will be included in the framework. In addition to prior research, two new inputs enrich the framework. The People CMM will add a human resources perspective, and the Epistemic Community will explain group processes that are at work.

4.3.2.1 People CMM

[Curtis95] developed the People Capability Maturity Model as an extension to the CMM in 1995. The People CMM (P-CMM) is described as follows:

"The People Capability Maturity Model® (People CMM®) is a tool that helps you successfully address the critical people issues in your organization. The People CMM employs the process maturity framework of the highly successful Capability Maturity Model® for Software (SWCMM®) [Paulk 95] as a foundation for a model of best practices for managing and developing an organization's workforce. Based on the best current practices in fields such as human resources, knowledge management, and organizational development, the People CMM guides organizations in improving their processes for managing and developing their workforce. The People CMM helps organizations characterize the maturity of their workforce practices, establish a program of continuous workforce development, set priorities for improvement actions, integrate workforce development with process improvement, and establish a culture of excellence."

The P-CMM promises to address a crucial area of OSP that has been largely neglected: Its participants. In the absence of most other resources normally associated with CSP, OSP rely even more on the quality of their participants to achieve results. Improving these qualities is expected to have immediate repercussions on the quality of OSP results. That said, OSP are not formal organizations, and many processes do not apply. Care needs to be taken to ensure only relevant aspects of the P-CMM will be incorporated into the framework. The large body of research that contributed to the P-CMM will be helpful to further OSP. It will not be possible to include the full P-CMM into the framework due

to its enormous size. Instead, the main concepts of the P-CMM will be integrated into the framework.

4.3.2.2 Epistemic Communities

Not all OSP participants are alike. Research has found [Jones00] that productivity of participants varies greatly, and that most projects are heavily dependant on key contributors. The aim of each OSP should thus be to do everything possible to ensure key contributors find a supportive environment, and to nurture participants to make them (future) key contributors. The notion of "Epistemic Communities" and "Situated Learning" are very helpful to understand this crucial transition from bystanders to lead developers. [Edwards00] explains the parallels between OSP and "Epistemic communities" thusly:

"Open Source software projects share the same characteristics as do the definition of epistemic communities. Contributors in Open Source software projects share a set of normative and principled beliefs, which provide a value based rationale for contributing to the project."

Edwards goes on to note that epistemic communities do not account for the entry of new members into a community, and proposes a process of "situated learning" to explain the transition from onlooker to active participant:

"The process of entering and becoming a member of an epistemic community is, as noted, not readily explained in the epistemic communities approach. This is an important process as it is the key to understanding the population and reproduction of the epistemic community. This understanding may also be used to ease the process of becoming a member. Many 'wannabe' developers and newcomers to Open Source development have difficulties understanding how to contribute to the development."

Each framework needs to make some basic assumptions to ground itself in a solid base of theorems. These assumptions form the bedrock for the rest of the framework. What assumptions need to be made for this particular framework?

4.3.3 Framework Assumptions

This framework tries to formulate current best practice of OSP management. The following assumptions influenced the framework in many ways.

4.3.3.1 OSP Software Engineering is directed evolution

OSP very rarely follow a clearly laid out plan. More often than not, their features evolve during the development process. Linus Torvalds[153] calls this process directed evolution.

The impressive part is that Linux development could _look_ to anybody like it is that organized. Yes, people read literature too, but that tends to be quite spotty. It is done

[153] http://kerneltrap.org/node.php?id=11

mainly for details like TCP congestion control timeouts etc – they are _important_ details, but at the same time we're talking about a few hundred lines out of 20 _million_.

And no, I'm not claiming that the rest is "random". But I _am_ claiming that there is no common goal, and that most development ends up being done for fairly random reasons - one persons particular interest or similar. It's "directed mutation" on a microscopic level, but there is very little macroscopic direction. There are lots of individuals with some generic feeling about where they want to take the system (and I'm obviously one of them), but in the end we're all a bunch of people with not very good vision.

And that is GOOD.

Accepting that OSP follow a directed evolution throws away many established software engineering principles. This is no step to be lightly taken. For the purposes of this framework, it is therefore assumed that an OSP is large enough to exhibit the features of directed evolution:

- Many particular interests, no single “true path”
- A diverse development group
- No “design” that specifies every detail
- A constant review of ideas

Projects that exhibit these features are interesting because they are unpredictable. No one can say what the Linux kernel will look like in five years, and no one can predict the requirements that new applications and hardware will pose. A design that adapts to new requirements, as the “directed evolution” model has a longer lifespan. Smaller OSP that are below this critical mass may find that their development processes do follow conventional patterns, and in fact single individuals or very small groups develop most OSP. These groups form because their members have very similar requirements, and the resulting software is thus very specialized to this task, and may not be reusable.

4.3.3.2 Personal Involvement is crucial

Managing an OSP is much more demanding than a CSP.OSP leaders lead by persuasion, not by fiat, and therefore require very good interpersonal skills in addition to their software engineering skills. People who exhibit all these qualities are rare, and most OSP will not have such outstanding human resources at their disposal. That said, personal involvement in OSP is crucial. It is assumed that a person that visibly identifies with the goals of the project is leading an OSP.[154] The framework places great emphasis on human resources by introducing the P-CMM model with its various processes. Earlier theories and frameworks took staff development for granted, and human resources has thus been an underappreciated area of OSP.

[154] See [Ehresman01]

4.3.3.3 Small Teams work: The buddy system

Compared to a conversation that takes place face-to-face, communication bandwidth on the Internet is limited. Due to limitations of technology, but also written language, misunderstandings happen much more, and do have much more serious consequences. Special efforts need to be made to balance the weaknesses of Internet communication. One way to overcome communications issues is to form small teams that share common notions and have friendly relations among team members[155] It is assumed that an OSP is fostering such friendly bonds between its participants beyond the duties of the project. The framework emphasizes the flexibility of roles and actors, and assumes that participants switch frequently and quite effortlessly between them. This requires a level of trust which only personal bonds and shared values can provide.

4.3.3.4 Market your Project

OSP participants are a scarce resource. All OSP compete for those same resources, and for every interest, there are more than likely several projects. As [Vallopillil98] notes, the winner takes all. Projects that are able to attract attention will drive resources away from less outspoken projects. It is therefore assumed that a project makes efforts to let the world know about itself. Granted, not all domains attract the same number of participants, but vocal projects do have an edge over obscure ones. Marketing is added to the framework because it plays an important role to reach critical mass for a project. OSP failures are very often due to an inability of the project to attract an audience.

4.3.3.5 No one likes to administrate

OSP are volunteer-driven. [BCG02] established that most participants are IT professionals who work on OSP in their spare time. Many of these seek to run things differently than in their day jobs. Rather than worry about red tape, they would like to get things done, and solve challenging problems. As soon as a group of people starts to interact, communication overhead builds up. Some amount of administration is inevitable for every OSP, but most try to keep it minimal. This rules out the application of complicated processes for OSP, as no one has the inclination to work on bureaucracy in their spare time. An OSP framework needs to consider this, and tailor its processes to its volunteer audience. The tools section of the framework acknowledges the need to automate as many repetitive tasks as possible to free contributors from these tedious tasks.

With these assumptions outlined, the framework needs a methodology to develop the various components, and piece them together in a consistent way.

4.3.4 Framework Methodology

Open Source is influenced by many trends from different areas. Economical, social, psychological and technical forces each contribute to Open Source identity. It seems hopeless to unite these different forces under a common metaphor without creating a metascience. The approach taken maps out the various areas and highlights their intersections. The central organizing structure of the framework is a matrix that maps

[155] [Cavalier98] emphasizes the importance of small teams for review and instant feedback.

these intersections explicitly. The matrix is extensible to accommodate yet to be discovered forces that shape OSP, to integrate additional roles, actors and new tools. The framework then describes the various roles, actors, areas, processes and tools that are involved in an OSP. The aim of the framework is to explain these different forces; it is not a recipe for creating and operating an OSP. Such practical advice is provided in an appendix.

Formulating theories that pass the test of time is nontrivial. How does the Open Source framework stack up? What new elements does it add to the academic discussion, what are its major insights? How do its elements help to understand the many attributes of Open Source? Does the framework succeed in its aim to offer practical advice to Open Source practitioners?

5. A framework for Open Source Projects

5.1 Overview of the framework

The OSP framework is quite extensive. Based around the notions of roles, areas, processes, and tools, it brings together a wide range of topics. The framework draws from a rich existing literature, and tries to acknowledge as many diverse ideas as possible. This presents practical difficulties for the integration of these concepts into a coherent whole. To better integrate these ideas, and highlight relations between them, the framework uses a matrix as its central organizing element. What elements compose the framework?

5.1.1 Framework Components

Four components make up the OSP framework. Roles, areas, processes, and tools provide structure for the collection of ideas that make up the framework.

Actors and Roles
Based on the notions described in [Edwards00], Actors are either users or contributors. As interested people make headway into the epistemic community of an OSP, their skills increase, and they adapt the social norms of the group. If they feel inclined, they may start to contribute to the project, which marks their transformation from user to contributor. As their experience grows, they may take on more and more roles, and accept additional responsibility. The culmination of responsibility is the role of project manager.

Areas
OSP have very complex structures. Areas are the coarsest entities in the OSP framework. They group related processes together. One area might be software engineering, with processes like requirements engineering, testing, review, and prototyping. Areas do overlap, and are intended as a simplifying model only. Existing theories would usually deal only with select areas of OSP. The focus with this framework is especially to highlight the connections between areas, as those remain the least understood phenomena.

Processes
All work activities within OSP can be abstracted as processes. Processes are not intended to have the connotation of red tape and formal structure. Formal structures and written rules are almost non-existent in OSP; instead, processes signify recurring units of work. Again, it is the interaction of processes from different areas that provides new insights into the complex structure of OSP.

Tools
Processes are supported by tools. Due to the virtual nature of most OSP, tools take on a crucial role to facilitate the goals of an OSP. Interestingly, many OSP concern themselves with the creation of tools for OSP processes. While tools by themselves do not enable OSP to work, they are indispensable for the day-to-day operation of all OSP, and have been mostly neglected in previous studies. Granted, only wisely applied tools with a clear purpose add to the bottom line of a project, but more often than not OSP are not even aware of the existing tools that might help their projects.

These components have many interrelationships. While it would be futile to try to describe all these relations in detail, it is nevertheless important to acknowledge them as much as possible. A matrix allows one to display many relations in a structured manner, and gives a good overview at the same time.

5.1.2 The Framework Matrix

As a conceptual aid, the components of the framework are arranged into a matrix. This allows conceptualizing their interrelationships and allows understanding the major parts of the framework very quickly. Actors and roles are mapped on the horizontal axis of the matrix. Both skill level and involvement in the epistemic community increase from left to right. Areas are mapped on the vertical axis, with areas that have less overlaps on top. Project management is the area that overlaps with all other areas, and is hence mapped at the bottom. At the intersection are processes. Processes can appear at more than one intersection, and are marked with numbers. Tools are marked with letters, and can appear on multiple intersections as well. Interactions between processes are not being considered within the matrix, they do however appear in the detailed descriptions.

< Increasing involvement in OSP / **Areas**	Increasing involvement in OSP >	**Actors / Roles**								
		Users			Contributors					
	Areas / Roles	Users	Tester	Supporter	Administrator	Analyst	Maintainer	Developer	Architect	Manager
	Marketing	14, 18, g, c, j	14, 18, a, j	14 ,18, a, c, g, j	14, 18, a, j	14, 18, a, c, j	14, 18, a, c, j	14, 18, a, c, j	14, 18, a, c, j	14, 18, a, c, j
	Human Resources		15, g, k	15, g, k	15, b, c, k	15, b, c, g, k	15, b, c, g, k	15, b, c, g, k	15, b, c, g, k	3, 15, a, b, c, g, k
	Systems Mgmt	5	5	5	1, 2, 3, 5, h, l, m	5	1, 2, 5, m	5, m	1,5, m	2,5, m
	Software Engineering	7, f	7, 8, 10, 15, 16, d, f	7, 10, b, f	4, 6, 9, e, f	7, 8, 10, 15, 17, g, f	3, 4, 6, 9, 16, d, e, g, f	3, 4, 7, 8, 10, 15, 17, a, d, e, g, f	7, 8, 10, 15, 17, a, d, e, g, f	7, g, f
	Project Mgmt				h, i	h, i	11, 12, a, b, i	11, a, b, h, i	13, a, i	11, 12, 13, a, b, h, i, k

Table 4: The OSP framework matrix

Processes

1) Backup
2) Access Control
3) Logging
4) Version Management
5) Communication
6) Release Management
7) Requirements Analysis
8) Prototyping
9) Deployment
10) Documentation
11) Scheduling
12) Tracking
13) Allocation
14) Project Promotion
15) Staff Development
16) Bug Triaging
17) Reviews
18) Advocacy

Tools

a) Asynchronous communication
b) Real Time communication
c) Web Log
d) Unit Testing Framework
e) Source Configuration Management (SCM)
f) Bug Tracker
g) Content Management System
h) Issue Tracker
i) Calendaring
j) Online fundraising
k) Skill Matrix
l) Backup
m) Authentication System

5.2 Actors and Roles

5.2.1 Actors

Actors are a loosely defined term in this framework. Every individual that interacts with a project is an actor. For the purposes of this framework, we distinguish between users and contributors. Users are consumers of the results of an OSP; contributors invest efforts to improve the product in various ways. The notions used draw heavily upon work done by [Edwards01], especially epistemic communities and situated learning. All individuals involved with a project are users, but not all users are contributors.

5.2.1.1 Users

Users are persons who want to use the services provided by a project and do not intend to contribute to the product. Although users are per definition non-vocal in a project, they are an important resource. As time goes by some users begin to take an interest in the project and voice their opinion, and thereby they become contributors. This could happen in a situation where a new version is released and a user realizes that the new version of his favorite software no longer runs on his system. This situation often motivates a user to become a contributor and post error reports to the project mailing lists in order to get the problem fixed[156]. The user may also posses programming skills himself, in which case he may review the source code and fix the bugs. The user has a choice whether to fix the bugs on his personal system or to send the fixes to the project mailing list for inclusion in the project code base. The advantages of just fixing the problem on his personal system is a short term gain, since the problem could be fixed immediately without having to interact with the project mailing lists. However, if the bug fixes are not included in the project code base, the user will have to make these fixes again next time the project releases a new version. The new version may have changed to such a degree that the fix no longer applies and has to be modified to fit the new version. Thus, there is an incentive to include the fix in the project code base. Contributing code to a project also provide status and recognition among peers in the project, which is rewarding in itself.

5.2.1.2 Contributors

Contributors are people who take an interest in a project, follow the discussions on the mailing lists, and voice their opinion. Contributors need not contribute actual code to the project; they may just participate in discussions and stimulate debate within the community. Some contributors answer questions about problems related to general use and contribute to the community in this manner. Contributors are also persons who ask questions about a project and by doing so provide valuable information about the usability of the project software and documentation. Becoming a contributor is a learning process that spans multiple stages. The learning process itself is crucial for the cultural context of a project. By learning the notions of a community, and passing them along among to new members of a community, the community is kept alive and thriving.

[156] See also [Hippel02]

Learners
Learners are persons who wish to become part of a project and participate in it. Learners do not receive abstract knowledge de-coupled from any practical use. Learners become contributors in a project by being part of its community and acting as apprentices. This requires that active contributors of the community view apprentices as legitimate peripheral contributors or onlookers. Contributors must allow learners to participate at a learner's level. By observing and participating, the learner acquires knowledge, skill, and understanding of the social sphere of which he is a part. In short, learners want to become insiders.

Insiders
Insiders are members of the community who are active in the community and participate in the activities of the community. Insiders are contributors who function in the community. The process of becoming an insider requires the learner to learn the ways of the community, how they (inter)act. This is not something that can be taught, and descriptions of how to enter a community miss the finer points. The learning process has an individual bias depending on the learner, which is affected (influenced) by the learners' relations in the community. Becoming an insider can only be learned, and this is a situated process where the learner participates in the actual and practical life of the community.

From Learner to Insider
By participating, the learner observes the practice of the other members of the community. What is more important is that when the learner participates, he tests different strategies or ways of interacting with the other members of the community. In the course of this process, the learner begins to form his own impression of the inner workings of the community. Completion of this process and transformation of the learner into an insider requires that the insider must allow the learner legitimate peripheral participation. Insiders must acknowledge the fact that learners are just learners. Learners cannot be expected to perform at the level of the insiders, and insiders must respect their effort. It must be legitimate for learners to participate in the practice of the community. Learners are bound to a peripheral position when participating. Learners do not have the skill of the insiders, and must accept a position where they are allowed to participate at their own level and observe. The learning process in OSP takes place in two spheres; the private and the collective sphere.

1) The Private Sphere
Learners spend many hours studying documentation, reading source code, installing new software releases in an effort to learn the details of an OSP. These efforts are not noticed by the community and happen completely independently and on the initiative of the individual.

2) The Collective Sphere

After a while, the learner may start to interact with the epistemic community using mailing lists, chat, news groups, and other means of communication. By doing so, the learner enters the collective sphere, and becomes noticeable by other community members. Questions, answers, and discussions are relayed back and forth between the community and its members.

These two spheres are by no means separate but rather exist side by side. Many learners and contributors have a keen eye on what is happening on different mailing lists and in chat rooms while they are working. The author, for instance, always has his email and chat clients running while working. Every half an hour, or more often, depending on the level of concentration needed, the author would interact with others via either email or chat. These means of communication allow one to stay in touch with the community, and indeed every day acquaintances would greet the author upon entering a chat room. The learning situations of the two spheres are different but both require understanding of the inner workings of a community. The private sphere requires understanding of the practice of coding - the coding style – in a project. Most projects have a special way of writing code. Adhering to the project coding style ensures readability of the code by other members of the community, a crucial property to ensure reviews and lower the barriers for other contributions. Many of these rules are quite formal and can thus easily be learned without having to engage in the community.
Becoming a contributor in the collective sphere, on the other hand, cannot be taught. Situated learning and legitimate peripheral participation are important in this sphere. Learners engage in the practice of the collective sphere, and by doing so they become contributors who should be allowed legitimate participation in the project. Becoming a contributor and going from learner to insider is the process by which OSP recruit new members, a process that requires the learner to actively take part in the project and contribute. It is also a process where the learner must expect that becoming an insider happens gradually. The transition from learner to insider is subtle, and the learner will often be the last to call himself an insider. In the process, the learner adopts the language, the coding style, and other social conventions of the community. The learner has then become an integral part of the community.

Every member of an OSP, be it a user or a contributor, fills various roles.

5.2.2 Roles

The constituents of OSP do fill widely varying roles. Roles overlap and it is difficult to assign clear roles to individuals. Therefore it is rather difficult to identify abstract roles in specific projects and even harder to find suitable roles for OSP in general. The roles presented here are by no means complete, but they cover major activities. From Tester, supporter to administrator, analyst and maintainer, to architect and project manager, each role carries increasingly more responsibility for the project.

5.2.2.1 Tester

Tester is a very general term due to the nature of OSP. All users of Open Source software are testers to varying degrees. It has been argued that testing is the price that has to be paid for Open Source or free software despite its availability at no charge [Liu01]. In a more specific sense, testers install and run software to actively look for deficiencies, and make efforts to report their results in a useable format, usually with the help of bug tracking software. [Liu01] writes:

In practice, testers' jobs are sometimes more subtle than simply producing bug lists. I once asked a test lead from a large software company what his most important responsibility was. The answer was quite surprising to me at that time: the most important thing was to know the status of the software product at all times. After I thought about it, the idea became quite reasonable. Clearly, when both the program-under-test and the description of the problem are changing everyday, it is not feasible to produce a comprehensive bug list for each daily build. Nor is it necessary. It is more useful to the development team if testers can provide constant and rapid feedback on the status of the current builds. Overview information is as important as individual bug reports. This "service" view of software testing focuses on the need for rapid feedback and the evolving nature of the program-under-test. Just as with many other services such as phone services, the need for rapid responses is paramount. When a person picks up a phone, she expects to talk right away; when a development team gets a build done, they expect feedback right away.

5.2.2.2 Supporter

A supporter assists users of software by various means. Supporters may answer questions on mailing lists, on discussion groups, in live chats, or even in person (at a gathering, for instance). Supporters maintain frequently asked questions, file bug reports, and spread the word about a project. Hence, supporters are the talent pool from which most eventual developers are selected. In their quest to help others (and themselves) resolve issues, they learn more and more details about a project, and may even start small modifications for customization or to fix a bug. Supporters are crucial for the long-term health of a project, as [Moorman00] points out:

General project supporters come in many flavors. Some aid in the propagation of the software, assisting in large implementations of the software or installfests, or providing inexpensive physical media (such as CDs) containing the software. Others provide themselves as a support resource for the product, helping new users with the installation or implementation of the software, responding to concerns on mailing lists, and providing direct support via IRC for other end-users. Project supporters play an important role which reduces the amount of time developers and administrators need to invest in support, the foundation of all good projects.

5.2.2.3 Administrator

OSP are critically dependant on shared resources. Administrators make sure these resources are operational, concern themselves with security issues, and are in charge of the supporting tools for a project. Their role is that of an enabler. Their actions may not

be obvious at first glance (doing daily backups is not linked with glory), but are nonetheless essential. Overload and burnout are frequent symptoms with administrators, and projects need to make sure to not tax their administrators more than necessary since their departure from a project could often entail serious issues. Many administrators donate server resources to OSP, and withdraw them once they leave a project.

5.2.2.4 Analyst

Analysts process the feature requests for a project or propose their own. The goal of their work is to produce useful recommendations for developers (specifications, priorities, feature lists) to allow developers to focus on developing. As with testers, all users are analysts to some degree, and many users submit feature requests if given the opportunity. However, most of these feature requests are badly thought out, unrealistic, or do not fit well within the scope of the project. The most important function analysts can provide to a project is to provide reasons to reject proposed functionality. Feature creep is a problem with OSP as well, as everyone wants their favorite function integrated, and hence it should be actively counterbalanced. Analysts mainly work with bug tracking software and may use mailing list and content management systems to organize their work. Authors such as [Schmidt01] recognize the potential of Open Source for analysis:

Open Source development techniques can also help improve software quality by enabling the use of powerful analysis and validation techniques, such as white box testing and model checking. Although this benefit of Open Source is not widely employed today, the next-generation of testing tools and model checkers will be more effective since they can instrument and analyze large-scale systems where many components and layers (e.g., network drivers, OS, and middleware) are Open Source.

5.2.2.5 Maintainer

Once a project reaches a certain maturity, it may be necessary to pursue multiple parallel development tracks[157]. The usual distinction is between a stable track, which contains ready to use software, and an unstable track which contains the newest changes, and may run or not run at any given time. Such projects often assign responsibility for the stable track to a maintainer. This person oversees all changes to the stable track in order to prevent regressions. The major activities of this role are communication, reviews, bug tracking and planning. Interestingly, a maintainer does not hold power of other developers to assign them tasks. Rather, the maintainer is passive until approached with an (implemented) change [Asklund01]:

In OSS, the evaluation of change proposals is not explicit, if it is there at all. Anyone can propose a change and most often changes are not even proposed before an implementation is submitted directly. Change proposals might be prioritized implicitly or explicitly, but an OSS project cannot assign tasks to developers – everyone works on what he chooses. Two slightly different processes exist depending on whether contributions have to be sent to a moderator or if you can apply your changes directly to the repository through your write access. In both cases, however, it is the same overall

[157] [Schach02] outlines the maintenance problems inherent in OSP.

process that is followed. An idea for a change is conceived, it is implemented and tested, it is submitted as a patch or applied directly on the repository, and finally the implementation (and sometimes the change idea itself) is evaluated through testing, review and discussion. The final evaluation may result in the patch being rejected by a moderator or a change to the repository being reverted by a co-coordinator. Usually write access to the repository is given only to trusted developers, so cases where a change to the repository is reverted are rare.

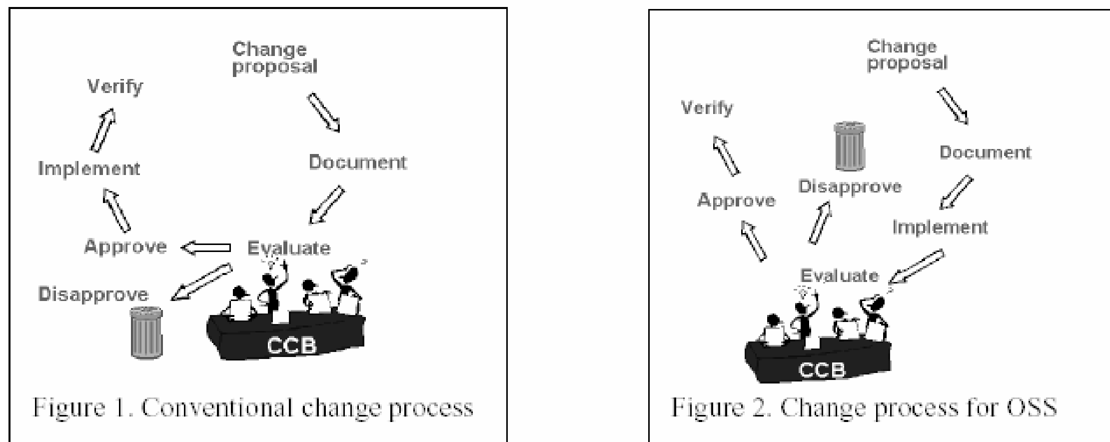

Figure 10: CSP versus OSP change process (Source: [Asklund01])

Figure 10 illustrates the difference between CSP and OSP change processes. CSP evaluate, and then implement; OSP implement, and then evaluate.

5.2.2.6 Developer

Another term for developer might be producer. Developers are primarily concerned with software engineering. It is through their actions that a project takes shape. Many other roles solely exist to facilitate the work of developers, like architect, maintainer, analyst and administrator. Projects with many non-developer contributors may develop conflicts about the future direction. These conflicts are usually resolved by the actions of developers, who influence the project by their changes to the software. This "rule of the developers" contrasts sharply with CSP. Even the manager of an OSP cannot rule by fiat and has to bow to the accumulated knowledge and judgment of developers. This does not mean that there are no OSP where a central person has authority to make technical decisions. [Kuwabara00] describes the position of Linus Torvalds in the Linux Kernel project:

Torvalds is the undisputed leader of the Linux project. He oversees the project as a whole, keeping track of major developments and reviewing numerous patches of code submitted by his developers, from which he builds new versions of the kernel. Torvalds is also an obvious authority on the Linux system. On any typical day, he receives around 200 e-mail messages directly from Linux developers. His activity on linux-kernel reflects a small fraction of his presence in the actual project. Torvalds works closely with the so-called "Inner Circle" of technical advisors, or "lieutenants," immediately around him. They are core developers who have established their status as competent programmers

and proven their expertise valuable to the project through years of involvement. The Inner Circle is neither handpicked specifically by Torvalds nor clearly defined by the community. It is a group that formed naturally, without external intervention but simply by virtue of their involvement and expertise acknowledged by the community. For example, Torvalds devotes most of his attention to the experimental version of the kernel, whereas Alan Cox is currently responsible for the maintenance of the stable, non-experimental version of the kernel for general users.

5.2.2.7 Architect

The architect is responsible for the overall technical vision of a project and its successful execution. Contrary to popular belief as exemplified by the Bazaar approach, having a coherent vision for a project strengthens it enormously. Technical debates can often only be resolved by the decisive power of one individual. Empirical evidence suggests that design by committee, as practiced by standards bodies like the W3C[158], lead to inferior results due to the desire to accommodate many constituencies, and hence the tendency for compromises. Large OSP are sometimes impossible to oversee for a single individual. In that case, the project is divided into smaller modules with well-defined interfaces, and shared responsibility. The architect can then focus on the interaction of modules and keep the big picture in mind without getting lost in details. Having at least two levels of developers within a project is common practice, and allows projects to scale, and key contributors to focus on the long-term viability of the project. Modularity is one of the most important architectural features of such OSP, as [Jones00] explains:

"In most Open Source projects," says Zawinski, "there is a small group who do the majority of the work, and the other contributors are definitely at a secondary level, meaning that they don't behave as bottlenecks." Zawinski goes on: "Most of the larger Open Source projects are also fairly modular, meaning that they are really dozens of different, smaller projects. So when you claim that there are ten zillion people working on the Gnome project, you're lumping together a lot of people who never need to talk to each other, and thus, aren't getting in each others' way." Brian Behlendorf of Apache and of Collab.net agrees. "We don't consciously think about it, but I think that the philosophy of keeping things simple and pushing out almost anything extraneous or nonessential to external modules has been followed fairly carefully in Apache. We've also been fairly successful (I think) in 'federalizing' the Apache process to sister projects."

5.2.2.8 Project Manager

Sufficiently large projects have many contributors besides developers. A project manager tries to coordinate the efforts of the various roles, may provide a vision for the project (project managers are often founders and have been with a project since the beginning), and resolves conflicts. A humble approach works best, as volunteer participants have no obligation to bow to orders issued by the project manager (or anyone, for that matter). A project manager may also be involved with advocacy for a project, may try to secure

[158] Interestingly, W3C members strongly object to this assessment. http://www.w3.org/People/Bos/DesignGuide/committee.html has an interesting commentary on the issue.

funding for key infrastructure, and serve as a spokesperson for the project. The special demands placed on a OSP manager are described in [Dafermos01].

It is increasingly the role of management to break free from any restraints related to bureaucratic regimes, and empower the customers (both internal and external) of the organization to participate in all crucial decisions. Issuing orders and direct exercise of control must be replaced by communication of vision and direction. Management becomes leadership. This explains why a project such as Linux, which operates and grows organically under no central planning, needs a leader like Linus Torvalds - to initiate change, communicate vision and "create a organizational mindset" where network communication is fostered.

No real project will have only these roles, nor will OSP participants exclusively work on any given role. Instead, roles are blurred, each participant may assume several roles, and multiple persons may occupy each role. Roles will be revisited in the discussion of processes, and the contributions of each role to a process will be discussed. OSP are very heterogeneous. Rather than focusing on software engineering alone, they always exhibit attributes of other disciplines as well, such as human resources. To fully understand OSP, it is necessary to outline these broad areas, and then discover the processes at play in each of them.

5.3 Areas

The OSP framework is composed of various areas that were determined by studying the existing literature and sampling select OSP. The framework can easily be extended by introducing additional areas. In fact, it could be argued that some areas of OSP have only been present for a short time. One example would be marketing, which has taken on a more prominent role now that Open Source programs are increasingly being used by non-programmers. In increasing order of relevance to the final product of an OSP there are marketing, human resources, systems management, software engineering, and project management.

5.3.1 Marketing

Marketing is not often mentioned in conjunction with software development. Indeed, in CSP there is often a noticeable rift between marketing and technical personnel. OSP, on the other hand, depend on marketing to attract volunteers for their project. The first Open Source programs were written by programmers to solve problems they encountered while performing their work. The audience of these tools was mostly other programmers, with the exception of some recreational programs. Later, as more and more non-programmers began to use Open Source programs, for instance communication programs, it became more important to market Open Source. Service companies were founded to support popular Open Source programs[159], and the web brought a large influx of designers, writers, and artists to the Open Source community. The contribution of these individuals is different from programmers, and to appeal to them and entice them to contribute requires different approaches than a simple list of the technical features of a program.

[159] [Kenwood01] contains an Open Source business case study.

5.3.2 Human Resources

Volunteer-based work often overlooks the needs of its constituents in the areas of personal development, coaching, and training. Preliminary evidence suggests this to be a major success factor for OSP and attests to its large growth potential. The human resources area of the framework tries to identify the needs of the participants of a project and offer advice for improvement in this crucial area. The frameworks draws on the experience of the People Capabilities Maturity Model and Epistemic Communities to address issues that have been largely neglected to date. Human resources on the Internet are still largely taken for granted. People appear seemingly out of nowhere and start to contribute, and they often disappear just as quickly. Many OSP wrongly assume that resources are infinite, and that high churn rates are not a problem, since "someone else will pick up the slack". This notion is critically flawed. Every experienced participant that is lost takes a lot of knowledge with him that is not documented anywhere, and each gap in a project needs to be replaced by other participants. A lot of redundant effort is thus invested into re-training new volunteers and frantically reshuffling responsibilities within a project, which could be much better spent by focusing more on the human resources aspect. Why is the churn rate of OSP often very high? What motivational measures can be undertaken to increase morale in a project, or "put the fun back" into an activity that is supposed to be recreational for most participants?

5.3.3 Systems Management

This area is a prerequisite for all other areas. Systems Management makes sure that all necessary resources for a project are available and is concerned with issues such as data integrity, communication, and access control. OSP has to take systems management into account because unlike CSP, its resources cannot be taken for granted. In fact, missing resources are a key vulnerability for OSP. System management permeates all areas of OSP due to a heavy reliance on tools. Tools are a necessary, but not sufficient ingredient of OSP, and systems management is often underappreciated. Most OSP participants know enough about computers to operate their own infrastructure proficiently, and hence often assume that the infrastructure requirements of an OSP can be managed with the same slack that is appropriate for personal systems. This results in downtime for the project, lost data and frustration. OSP would do good to give full credit to the persons outside the limelight that keep the servers running.

5.3.4 Software Engineering

OSP concern themselves with software production and are therefore subject to the rules of software engineering. Not only is success in OSP heavily dependent on the correct usage of software engineering methodologies, but Open Source has new impulses for the discipline of software engineering itself. Open Source is no software engineering panacea, and it is not surprising that most OSP do disregard software engineering best practice. It is often argued that since OSP are performed in the spare time of the participants, they should be "fun", and some of the more formalized software engineering methods are accordingly being dismissed. Another barrier to software engineering in OSP is the macho culture that cherishes technical prowess above all else, but has little respect for careful design. A large part of OSP participants are attending higher education and their involvement with OSP is often their first exposure to large projects. OSP thus offer

a great learning environment to gain real world engineering experience. OSP are not perfect examples of applied software engineering because they are very much learning grounds for their participants. Projects with excellent software engineering practice are often led by very experienced programmers, while junior programmers usually work on smaller projects with less prestige.

5.3.5 Project Management

Project Management is as crucial for the success of an OSP as it is for a CSP. OSP project management faces different challenges however. Due to the non-committal nature of most contributions, planning can only concern itself with short-term goals. Project management in OSP is leadership foremost. OSP are delicate social structures that need the right stimulants from their leadership to prosper. An OSP leader does not usually assign, he rather suggests. [Dafermos01] researched these new challenges for management, and summarized them:

Management should ensure that the organizational and project design maximizes organizational learning and empowers big teams to collaborate digitally. The organizational design must enable the human intelligence that resides within the organization and its environment to get networked. With creative use of the technology, knowledge will be disseminated throughout the network towards all the participants (organizational learning), and not just exchanged among few nodes (individual learning). As the Linux Project proves, massive parallel learning is not just a matter of technology but of design and management. The project design must be as modular and simple as possible to facilitate digital collaboration (between an incredibly big project team). It is more important that the design is simple and modular than error-free.

Ultimately, all areas are composed of work activities, or processes. It is within those processes that an OSP takes shape.

5.4 Processes

The activities of OSP are termed processes in this framework. For clarity, processes are assigned to areas, even though they do overlap just as much as roles do. By focusing on processes, the framework aims to offer advice for carrying out these activities both effectively and efficiently. Each process is described, the participating roles mentioned, and supporting tools listed.

5.4.1 Marketing

Marketing processes involve the broadest subset of OSP participants. It is very easy to get involved with OSP marketing; all it takes is to spread the word about a particular project. The boundaries of participation are vague. Marketing is the area that new participants come in contact with first, usually, and plays a crucial role in attracting initial attention. Two broad processes can be distinguished: project-specific promotion, and general advocacy.

5.4.1.1 Project Promotion

Involved Roles	All
Involved Tools	Asynchronous Communication, CMS, Weblogs

The mechanics of popularity are poorly understood. There is no recipe for making a project popular, but getting the word out about a project is widely understood as a key ingredient for its success. Most projects do not promote themselves in any way, and wait for users (and future contributors) to find them instead. This lack of marketing can often be explained with the technical background of most contributors, and their negative experiences with corporate marketing. [Vallopillil98] argues that successful projects will starve other projects out of resources, and that projects should try to become the "category killer", meaning that they address most needs for a given problem, and attract the most users. Project promotion is really a task that all members of an OSP are responsible for, maybe the one task that they all share.

5.4.1.2 Advocacy

Involved Roles	Users, Supporters, Developers and Manager
Involved Tools	Asynchronous Communication, CMS, Weblogs

Advocacy, or "the act of pleading or arguing in favor of something, such as a cause, idea, or policy; active support"[160] has taken on a more prominent role in OSP lately. OSP do not exist in a vacuum, they are very much a product of their environment, and depend on intellectual property laws, copyright and a host of other regulations. The political climate for Open Source has recently become more hostile with the introduction of new legislation that limits usage rights for copyrighted materials and restricts computer usage to only carry out "allowed" operations.[161] With the widespread popularity of the Internet and file sharing, engineering concepts have entered the realm of legislation, such as reverse engineering. Open Source has thus become more political, and many OSP feel a need to take a political stance to defend their rights to carry on with their projects. Another related issue is government sovereignty. Many governments have launched inquires into Open Source [EC02] to determine if increased usage of Open Source would guarantee the sovereignty of a foreign government from (mostly American) companies. These issues rarely affect OSP directly. Instead, they affect the environment in which OSP operate, and advocacy thus spans OSP. All members of an OSP are involved in advocacy, and use the communication channels available to them to spread their message. More effective, but often neglected, is political action. As Lawrence Lessig, a well-known law professor notes:

But if you don't do something now, this freedom that you built, that you spend your life coding, this freedom will be taken away. Either by those who see you as a threat, who then invoke the system of law we call patents, or by those who take advantage of the

[160] http://www.dictionary.com/search?r=67&q=advocacy
[161] http://www.chillingeffects.org/

extraordinary expansion of control that the law of copyright now gives them over innovation. Either of these two changes through law will produce a world where your freedom has been taken away. And, If You Can't Fight For Your Freedom . . . You Don't Deserve It. But you've done nothing.

5.4.2 Human Resources

Human resources are a largely unresearched area for OSP. While the model of Epistemic Communities and Situated Learning [Edwards00] explains the human resources aspects of OSP from the perspective of the individual, the People CMM model [Curtis95].explains them from the perspective of the collective. Due to these close interrelationships between human resource processes, they are presented here as one process rather than many. This allows one to gain an overview of this complex topic without getting lost in details.

5.4.2.1 Staff Development

Involved Roles	All
Involved Tools	CMS. Asynchronous communication, Web Logs

OSP are competing for talent, and the talent pool is apparently shrinking. As the knowledge required to build an OSP increases, the retention of experienced contributors becomes critical to improving a project. The ability of an OSP to compete is directly related to its ability to attract, develop, motivate, organize, and retain talented people. Traditional organizations have attempted to apply many different techniques in their efforts to better manage their human capital. They apply reengineering or process improvement, improve information sharing, clearly communicate the organization's mission, institute employee involvement programs, establish formal complaint resolution procedures, institute gain-sharing or other incentive plans, emphasize the importance of training the workforce, formalize performance management and feedback processes, and many more initiatives. However most organizations and OSP in particular, lack a coherent strategy to deal with human resources. Many OSP fail to do any staff development, citing the transitory nature of most contributors involvement as a reason to do nothing. While many of the more formal methods for human resources management may be inadequate for the virtual environment of an OSP, it is still beneficial to apply some lessons from the rich literature on human resources. The People Capability Maturity Model [Curtis95] "helps organizations characterize the maturity of their workforce practices, establish a program of continuous workforce development, set priorities for improvement actions, integrate workforce development with process improvement, and establish a culture of excellence." The People CMM (P-CMM) consists of five maturity levels that establish successive foundations for continuously improving individual competencies, developing effective teams and motivating improved performance. By following the maturity framework, an OSP can avoid introducing practices that its contributors are unprepared to implement effectively. The application of the P-CMM to the OSP environment is challenging. With minimal structures present in most OSP, many of the formalized approaches proposed by the P-CMM do not work. The core concept of increasing levels of maturity is applicable if the coexistence of these

levels is allowed for. OSP may be at multiple levels of maturity simultaneously, with new contributors joining a project, and veteran ones taking a leave.

Level	Processes
Repeatable	Work Environment Communication Staffing Performance Management Training (documentation, one on one) Compensation
Defined	Competency Analysis Competency Development Competency-Based Practices Participatory Culture
Managed	Mentoring Team Building (Network of peers) Team-Based Practices Organizational Competency Management Organizational Performance Alignment
Optimizing	Personal Competency Development Coaching Continuous Workforce Innovation

Table 5: CMM Levels (Source: [Curtis95])

Table 5 lists the levels of the P-CMM, and the processes that make sense in the context of OSP.

Work Environment
The purpose of Work Environment is to establish and maintain physical and virtual working conditions and to provide resources that allow individuals and workgroups to perform their tasks efficiently and without unnecessary distractions. An OSP should make the resources for collaboration available, and try to minimize distractions, for instance by filtering communications by relevance.

Communication and Coordination
The purpose of Communication and Coordination is to ensure timely communication across the organization and that the contributors have the skills to share information and coordinate their activities efficiently. OSP should make sure that individuals or groups are able to raise concerns and have them addressed by management. OSP should ensure that activities are coordinated.

Staffing

The purpose of Staffing is to establish a process by which committed work is matched to resources and qualified individuals are recruited, selected, and transitioned into assignments. Individuals or workgroups are involved in making commitments that balance the workload with staffing. OSP should base staffing decisions and work assignments on an assessment of work qualifications and other valid criteria. Individuals should be transitioned into and out of positions in an orderly way. Even posting a notice to a mailing list may be appropriate.

Performance Management
The purpose of Performance Management is to establish objectives related to committed work against which group and individual performance can be measured, to discuss performance against these objectives, and to continuously enhance performance. OSP should document performance objectives. These objectives could be requirements, design standards, deadlines etc. The performance of committed work should regularly be discussed to identify actions that can improve it. Code reviews are one way to discuss performance. Poor performance should be criticized, outstanding performance rewarded. OSP may feel leery of criticize each other's work, because it was done voluntarily. In the interest of the project, and to improve individuals these deficiencies should be discussed though.

Training
The purpose of Training and Development is to ensure that all individuals have the skills required to perform their assignments and are provided relevant development opportunities. OSP should provide individuals with timely training. Training for OSP may mean encouraging newcomers to read certain documentation, and answering their questions either by email or in IRC (Internet Relay Chat).

Compensation
The purpose of Compensation is to provide all individuals with benefits based on their contribution and value to the organization. Compensation strategies and activities should be planned, executed, and communicated in the open to prevent accusation of nepotism. Compensation should be equitable relative to skill, qualifications, and performance; in essence individuals with the same performance should receive the same compensation. Compensation in OSP may mean the respect of peers, special mention[162] or endorsements for job applications or other opportunities. [Raymond98b] lists the many faces of recognition: *"First and most obviously, good reputation among one's peers is a primary reward. Secondly, prestige is a good way to attract attention and cooperation from others. Thirdly, reputation may spill over and earn one higher status in other realms"*.

Competency Analysis
The purpose of Competency Analysis is to identify the knowledge, skills, and process abilities required to perform the organization's goals. OSP should think

[162] The KDE project features a different developer each week. http://www.kde.org/people/people.html

about the competencies that they need to perform their goals. This is rarely done unfortunately, and the thousands of OSP that went nowhere attest to this failure to formulate a projects needs (and a failure to meet these needs, obviously) As an OSP evolves, its required competencies change. Correspondingly, OSP should regularly revisit their competencies to assess if they still possess the necessary skills to achieve their goals. Due to the very open nature of the open source ecosystem, it may be possible to attract other contributors which exhibit the required competencies, or two OSP may decide to collaborate.

Competency Development

The purpose of Competency Development is to constantly enhance the capability of contributors to perform their assigned tasks and responsibilities. OSP should provide opportunities for individuals to develop their capabilities, and encourage them to share their knowledge with others. Besides the traditional means of web sites and mailing lists, weblogs[163] may be a more appropriate tool to support this activity. The collective knowledge of an organization is arguably its biggest asset. This holds true especially for OSP, which have few common points of reference other than their collectively acquired knowledge. The process of knowledge acquisition in groups is poorly understood, especially for very loose groups typical in open source. Traditional knowledge management does concern itself too much with "quality control" to be of practical use for OSP. Instead, a bottom-up approach, or "Knowledge Sharing" is called for, with much more emphasis on enticing individuals to share their knowledge than on making sure the knowledge meets quality standards. Traditional knowledge management has completely failed because it provides little incentives for contribution, and places far too high barriers to entry.

Competency-Based Practices

The purpose of Competency-Based Practices is to ensure that all workforce practices are based in part on developing the competencies of the workforce. This means that OSP should strive to achieve their goals in ways that provide the biggest possible learning experience for contributors. One way to do this is to not only encourage and reward immediate results (source code), but also the efforts of persons who help others to develop their skills. To capture these efforts which may often happen in private email conversations, and thus not visible for the project at large, rating systems could be used that allow recipients to rate the advice they have received from others.

Participatory Culture

The purpose of a Participatory Culture allows the organization to recognize and exploit the full capabilities of its participants. OSP should go to great lengths to encourage information sharing beyond the immediate requirements of the projects goals. Forming friendly bonds between participants allows making the project more aware of the capabilities of its participants. Informational exchanges between individuals tend to be much more revealing and instructive to assess each

[163] http://jrobb.userland.com/2001/10/05.html#a405 gives a good definition.

other's skills than formal systems to capture skills. OSP should spread information that is relevant to all participants in appropriate ways, and not hoard information. On the other hand, OSP have to make sure that participants are not overwhelmed with information. A rule of thumb could be to make the information available to interested parties rather than broadcast it to all project participants. OSP should delegate decisions as much as possible to avoid bottlenecks. If OSP fail to do this, they may be prone to micromanagement, and key persons will be completely inundated with work, while blocking progress for everyone else. OSP with large user communities should encourage participation and feedback from the user community while not falling prey to individuals who only want to make themselves heard without providing useful contributions. This distinction can be quite challenging, and can result in accusations of elitism if an OSP is perceived to not listen to its user community to a sufficient degree.

Mentoring

The purpose of Mentoring is to transfer experience from seasoned contributors to newcomers. OSP should consider setting up mentoring programs that assign mentors to newcomers. These mentors provide guidance and support to individuals or teams, and .help them to get on the right track. Furthermore, mentors can quickly summarize complex issues due to their experience, and thus provide shortcuts for newcomers. Very often, the knowledge of an OSP is much cluttered or not available in written form, and mentors are the only way to quickly gain familiarity with a project.

Organizational Performance Alignment

The purpose of Organizational Performance Alignment is to enhance the alignment of performance results across individuals and teams with organizational performance objectives. OSP that reach such high levels of synergies between contributors are a rare exception, and are usually the results of a stable group of collaborators that have worked together for years. Such tightly knit OSP should try to understand how improving the performance of individuals helps the overall performance, and what specific actions are required to better both individuals and the group as a whole.[164]

Continuous Workforce Innovation

The purpose of Continuous Workforce Innovation is to identify and evaluate improved or innovative workforce practices and technologies, and implement the most promising ones throughout the organization. OSP should stay alert to new processes and technologies that may benefit their projects. OSP should also be welcoming to change instead of falsely insisting on "stick with what works". Due

[164] The OpenBSD project may be such a closely knit group. Oliver Friedrichs illustrated this with an example: "*OpenBSD has been "victim" to one of the most stringent software source code reviews (that is, looking for and discovering security flaws), that I know of, occurring in any operating system. Many individuals worldwide have spent much of the last year piling through source code, identifying and fixing security problems*"

to the non-commercial nature of OSP, they provide ideal test beds for innovative forms of collaboration, and OSP should leverage this flexibility.

5.4.3 Systems Management

Operating the infrastructure for an OSP is an unglamorous necessity. Systems management processes should be automated as much as possible to free resources from repetitive tasks, and to increase reliability. Backup, access control, logging, and maintaining the communication infrastructure are the main systems management processes.

5.4.3.1 Backup

Involved Roles	Administrator, Maintainer, Developer
Involved Tools	SCM, Mirroring Software, Backup tools

Computer systems are vulnerable to loss of data due to various causes.

System Failures
Computer systems fail. Hardware problems, subtle errors in operating systems and applications or may occur at any time.

Operator Error
Most data losses are due to accidental actions by the user, either through careless operation or failure to understand software functionality.

Unauthorized Access
The data of compromised computer systems may be vulnerable, depending on the intentions of the attacker.

All these problems are not specific to OSP, but to the extent that OSP rely on shared resources connected to the Internet, they are more vulnerable to data loss. Backups are thus a very important activity to preserve the work of an OSP. Besides backups of the shared resources, which are performed by administrators, there is also a second source of backups. As users and developers download copies of their project, they indirectly hedge against data loss at the central location. Tools like CVS (to retrieve copies of source code) and site mirroring (to retrieve copies of web sites) help with this task. Central backup is usually performed with the tools that come with the operating system.

5.4.3.2 Access Control

Involved Roles	Administrator, Manager
Involved Tools	Authentication, PKI[165]

Most information in an OSP is by its nature public, and does not need special protection. Access control is mostly related with controlling write access to shared repositories for code and content, and performed by administrators, or the project manager. Some

[165] Public Key Infrastructure http://www.pki-page.org/

projects have started to build a PKI infrastructure to prevent attacks on their source code.[166]

5.4.3.3 Logging

Involved Roles	Administrator, Maintainer, Developer, Manager
Involved Tools	Facilities of the Operating System

Collaborative work is greatly helped by logging, as it allows the ability to determine which individual contributed which changes. In the absence of “water cooler” discussions, logging allows fellow participants to keep track of changes, and exposes actions that are controversial or damaging for a project. As OSP are meritocracies, logging also provides raw data to spot the major contributors, and allows the project manager to reward these individuals with more status.[167]

5.4.3.4 Communication

Involved Roles	All
Involved Tools	(A)synchronous Communication, Web Log, CMS

Information in its various forms (software, documents, URLs, etc.) is the most important resource for OSP. The amount of information grows daily. Distributing it efficiently and effectively poses some challenges:

1. Who might be interested?
2. What is the best way to notify interested persons?
3. How should information be archived for later retrieval?
4. How should relevant information be separated from noise?

The Internet has evolved several means of communication like email, newsgroups, and IRC. Nevertheless, most of the four challenges remain unsolved. Better tools and processes for communication would improve OSP tremendously, as communication is very often the bottleneck. Some interesting features for future communication systems would be:

1. To join/leave single discussion threads instead of entire mailing lists
2. To open / close discussions
3. Notification about new discussions
4. The possibility to view all open discussions
5. Easy retrieval from an archive

OSP use many different communication channels. Each channel has its unique properties and strengths:

---

[166] The Debian project set up a PKI infrastructure to sign all packages. This allows to verify that packages are unmodified and therefore prevents insertions of malicious code into the Debian system through third parties. http://lists.debian.org/debian-dpkg/2000/debian-dpkg-200007/msg00044.html

[167] The Debian project has developed a karma system that shows the level of contribution for each project participant. http://master.debian.org/~edd/karma.txt

Meetings
OSP participants are spread across the globe, making meetings an expensive and rare event. Meetings usually happen during conferences, and address major issues for a project. One notable example is the Kernel summit, which is an invitation-only event that brings the top 60 Linux kernel developers together to discuss the future of Linux. Besides being able to resolve difficult issues in hours what would take weeks or months with mailing lists, it also allows participants to bond.[168] Having personal relations helps tremendously with communications, as it reminds participants that there are humans at the other end of an email.

Documentation
OSP participants and users learn about a project at different times. Having asynchronous communication is therefore crucial to transfer knowledge to new contributors. Documentation is the result of informal knowledge exchange, often collected as FAQ (frequently asked questions) and later refined into documentation.

Source Code
Due to the openness of OSP, the source code itself is one of the most important channels of communication. Source code is the canonical document in technical matters, and reading source code can greatly expedite questions for seasoned programmers.

Log files, Bug trackers
Investigating the log messages of the version management system (e.g. CVS), bug reports of bug tracking systems or information archived by some other tools can be helpful. These log messages contain the complete trail of all changes made to a project, and are very informative if you are interested in the historical context of some functionality.

Traditional Communication
The telephone and written letters are rarely used in OSP. Especially telephone conversations can be a very efficient means of getting to know a fellow participant, and converging on solutions quickly. Long distance telephone charges prevent this means of communication from being used more widely. Meanwhile, some voice chat systems exist that allow for voice communication over the Internet.

Next Generation Communication
Video-conferencing, shared white boards, and remote access solutions may soon enter the Open Source scene.

Generally, more advanced tools have too many requirements to allow their widespread use, and Internet connectivity is still limited to dialup speeds in many parts of the world.

[168] http://www.freebsd.org/events/2002/usenix-devsummit.html

Until broadband Internet access becomes widely available worldwide, mailing lists remain the most appropriate communication channel for OSP.

5.4.4 Software Engineering

The various software engineering processes are at the center of OSP activity, and are heavily dependent on each other and processes from other areas. Ultimately, the degree of quality with which these processes are performed determines the resulting quality for the products of an OSP. Requirements analysis, prototyping, testing, version and release management, bug triaging, deployment, and documentation make up software engineering in OSP.

5.4.4.1 Requirements Analysis

Involved Roles	Testers, Analysts, Developers, Architects
Involved Tools	Asynchronous Communication, Bug Tracker, CMS

Requirements Analysis is concerned with collecting and commenting on user requirements for given software. OSP have strong requirements analysis processes because it is so easy for users to make their voice heard. The downside of this manifests itself in the glut of requirements that many projects have, and a lack of priorities to respond to these requirements. Requirements analysis starts with users submitting feature requests, or reporting bugs. These requests are of very low quality on average. Since all feature requests are publicly visible, interested parties can voice their support for a feature. This allows gauging interest in a feature. Some projects, like the Mozilla project use voting to assign priorities to features. This collaborative requirements gathering process may be supported by analysts who aim to combine similar requests, reword requests to make more sense, and start the thought process about integration of a particular feature. Developers then use this refined requirements lists to assess feasibility of features, and assign priorities to them. If they feel so inclined, they may start on the implementation of some requests, or come up with proposals for farther-reaching changes. Local changes are usually the domain of individual developers, while global changes and major additions require the approval of the architect.

5.4.4.2 Prototyping

Involved Roles	Testers, Developers, Architects
Involved Tools	Asynchronous Communication, SCM

The process of prototyping can be observed in most OSP. Many projects use extreme programming (XP) methodologies, often unknowingly. The open source mantras “Release early, release often” and “Show me the code” are good examples of this behavior. Developers are encouraged to work on experimental approaches, and if testers provide positive feedback their work may be integrated into the main line of development. Rapid prototyping is probably one the strongest attributes of OSP. [Nishinaka01] explains: “We never tried our project in XP style. The word "XP" did not exist when we started the project. We preceded our project with XP style in consequence. XP can be a good methodology for Open Source software projects if:

- it is not a big project
- there are one or more core members
- there are some contributors
- a project is not for profit
- a project is not for any specific customers”

Others, such as Linus Torvalds[169], hold the more radical view that *“nobody else "designed" Linux any more than I did, and I doubt I'll have many people disagreeing. It grew. It grew with a lot of mutations - and because the mutations were less than random, they were faster and more directed than alpha-particles in DNA.”*

Torvalds argues that software is never designed, but evolves. *“And I will go further and claim that _no_ major software project that has been successful in a general marketplace (as opposed to niches) has ever gone through those nice lifecycles they tell you about in CompSci classes. Have you _ever_ heard of a project that actually started off with trying to figure out what it should do, a rigorous design phase, and a implementation phase?*

Dream on.

Software evolves. It is not designed. The only question is how strictly you _control_ the evolution, and how open you are to external sources of mutations.

Too much control of the evolution will kill you. Inevitably, and without fail. Always. In biology, and in software.”

5.4.4.3 Testing

Involved Roles	Users, Testers, Developers, Architects
Involved Tools	Asynchronous Communication, SCM, Bug Trackers

Most OSP do not use formalized methods of testing. This major drawback is only to a small extent compensated by the large number of testers for popular projects. Notable exceptions were researched by [Halloran02], who provides this summary:

”*We also note unusual project-specific quality practices beyond personnel practices (i.e., limiting commit privileges) and the obvious elements of “micro-process” (particularly the ubiquitous nightly build) .The surveyed projects focus on people as the locus of Quality Assurance (QA). Clearly, responsibility for quality rests on those who have code commit privileges. Any code received from a programmer without commit privileges (usually attached to a bug report in patch format) must be reviewed and accepted by a programmer with commit privileges. In addition, more elaborate processes such as Mozilla’s review and super review and NetBeans’ high resistance ensure that lead programmers review code changes.*
All projects use nightly builds. The nightly build ensures the source code still compiles after the day’s changes. Most projects include some regression testing in the build

[169] http://kerneltrap.org/node.php?id=11

process, and some projects also generate daily binary builds for public download. The Mozilla and Perl projects use the Tinderbox tool to help assure portability by building the code on multiple platforms after any code change; if the build breaks on any platform, then all subsequent code changes are limited to fixes.
A publicly visible bug and issue tracking tool is used by nearly all the projects we examined (one of the projects allows bug submission but not public status viewing). Users post bugs and enhancement requests. Each such post becomes, in effect, a tiny public mailing list focused solely on that issue. Some of the discussions are resolved rapidly (e.g., "invalid; not a bug") while others can last for weeks and include tens of messages. The bug and issue tracking tools provide the vehicle for contributions of source code from programmers without source code commit privileges. In addition, the issue tracking tool has a role in project management and tracking, enabling, for example, specific issues (performance, functionality, bug repairs) to be linked to a "stable release issue," thus setting a threshold for a subsequent release."

Providing easier and more efficient ways for participants to run tests will improve software quality tremendously. Many projects hesitate to simplify this process out of fears that less than able contributors might add many false alarms to their bug tracking systems. This risk is overrated though. Combined with efforts to regularly prune bogus bug reports widespread testing does uncover many issues that do not manifest themselves in the limited tests run by developers. A good testing policy needs to be tied in with sensible change management policies. Tools like CVS allow a wide range of processes to be implemented, from the four eyes commits rule up to highly complex review and approval processes.

5.4.4.4 Version Management

Involved Roles	Administrator, Maintainer, Developer
Involved Tools	SCM

Version management records all modifications to project data, and is capable of reverting modifications to any former state. Version management (sometimes also called Configuration management) does have many uses in OSP. Some of them are:

1) It allows the ability to experiment with source code, and submit changes, since evolutionary dead ends can easily be corrected by reverting changes.
2) Version management systems normally provide a version history and the opportunity to store log messages. These logs can be used as documentation for the development process, and are often the most accurate description of a projects state.
3) Concurrent revision control systems like CVS help to coordinate and merge distributed work. Merging changes from dozens or hundreds of contributors by hand is nearly impossible.

Version management is one of the main activities of developers, and many tools have evolved to support it. Version management is integrated into most development environments, and is available on all platforms. CVS is the most used tool for version

management, but has its limitations for very large projects. Correspondingly, better systems have been developed.

5.4.4.5 Release Management

Involved Roles	Maintainer
Involved Tools	Mirroring Software, SCM

OSP release many versions of their software, heeding the motto “Release early, release often”. Although it is normally easy to distinguish different versions by their version number, it is sometimes very hard to figure out what the version number means. For instance, one release of Linux was described as: “It turns out the 2.4.0-test1 is actually 2.3.99-pre10-pre3 [...]”. Despite the strange numbering schemes adopted by some projects, some common threads emerged:

Numbers
Most projects use a numbering scheme to name their releases. A number like 2.3.20 usually means major version 2, development branch 3, and point release 20. Many projects do have multiple lines of development happening at the same time. In that case, even numbers at the second position indicate stable branches and odd numbers indicate development branches.

Development Branches
Many projects have different branches of development and name their branches ’stable’ (intended for productive usage), ’experimental’ (intended for developers). Some projects even use three branches, such as the Debian project, which has stable, testing and unstable branches.

Work Status
OSP often use common terms to indicate the work status of a project: alpha (in development) beta (feature-complete), pre (possible release candidates), RC (release candidate), final (official release).

Code Names
Some projects use code names for their major releases, e.g. ’potato’ instead of 2.2. Code names are a source of entertainment or respect for most projects, and great efforts are invested to search for suitable code names.

Release Dates
Some projects use the release date as the version number. Due to varying date formatting conventions, only a scheme like “20020703” will be unambiguous, and care should be taken to use such a scheme.

Custom versions
Major contributors of some projects sometimes make their own versions available and mark them with their initials. The purpose of such experimental versions is to

pursue side projects that may or may not be integrated with the official release at some point.

5.4.4.6 Deployment

Involved Roles	Administrator, Maintainer, Developer
Involved Tools	Mirroring Software, SCM, CMS

Deployment is term that encompasses packaging, dependencies management, upgrades policy and installation. Many OSP exhibit weaknesses in those processes, often due to their catering to sophisticated users that are expected to be able to deal with such issues. Poor deployment is a large barrier to entry for open source software. Projects with make the extra effort to provide easy installation do attract more users, and in turn, more developers. Historically, many OSP do have an elitist attitude and would rather not be bothered with mere users of their product. Instead, they hope for well-designed contributions to come out of the void. In doing so, these projects forego many potential users, and a bigger niche in the software ecosystem for their project.

Managing dependencies between packages is a major issue in OSP. Due to the very high reuse inherent in open source, most packages rely on dozens of others to work correctly. Various solutions have been developed for these problems, with APT by the Debian project probably being the most sophisticated. APT defines standard formats to specify package dependencies, and allows for automated updates of a complete system.[170] Due to the great care taken by the Debian project, it is one of the best managed projects in terms of dependency conflicts.

5.4.4.7 Documentation

Involved Roles	Developers, Supporters
Involved Tools	Asynchronous Communication, CMS

Comprehensive and qualitative documentation is a key for adoption of a project by other volunteers. Even more than in CSP, OSP need to make sure that collective knowledge about the domain at hand is collected and made accessible in the documentation. Good documentation can compensate for a lack of access to informal knowledge by new entrants to a project. Most documentation grows organically, and originates in comments in the source code, or answers to questions on a mailing list. Many projects do have very inadequate documentation, or do not make the effort to write documentation that is actually useful. Non-programmers have very different needs for documentation:

Internal and on-line
Non-programmers insist that context-sensitive, on-line help must be provided with an application. Non-programmers want screen-shots in the on-line help. They do not care if it increases an application's file size. Non-programmers utilize on-line help as a quick reference, so indexes and search functions are important.

[170] http://www.debian.org/doc/debian-policy/ch-archive.html contains the full Debian policy, which forms the backbone of APT.

Non-programmers will go through an on-line tutorial, if one is provided as part of the application. Non-programmers will look at a "Tips and Tricks" dialog box, if one is provided. Non-programmers ignore the printed manuals bundled with off-the-shelf software. Non-programmers would never buy a book about an application. They say technical books are for programmers.

Simple
Non-programmers do not want detailed explanations. They want simple answers. Non-programmers hate too much detail. Non-programmers prefer short, systematic instructions. Non-programmers prefer information that answers the question "How do I do X?" (where X is a common use of the application). Non-programmers do not want to see information about how a feature was implemented.

Complete, correct and up-to-date
Non-programmers assume that on-line help will be updated in each new version of an application. If part of the on-line help is obsolete or missing, non-programmers will not use any of it. If a non-programmer cannot find an answer in the on-line help, they will either call tech support or use another application.

That said, there are also voices that advise against spending too much effort on documentation (considering the scarce resource of able technical writers), because users do not read the documentation anyway. One of them summarized this sentiment: "*Non-programmers do not read any documentation that contains more than about 5 words. Non-programmers have arbitrarily weird ideas on how something should be performed. If it does not work, they will first blame it on the computer, then on the program, then on the admin. They will not listen to any explanation longer than five words. If you stand besides them to help them, they want you to do it for them. If you do, they will see no need to watch what you do, they will rather ask you every time to do it. In no case are they able to substitute anything in the documentation. If a login documentation contains the word "insert-your-username-here", they will type exactly that. If a screenshot for "Save as..." shows the correct image, but the "path" field contains "/home/admin/", they will detect the difference in their dialog, type the given path into it, and then complain the computer is broken. Anything written besides, below or onto the screen shot as explanation is ignored.* "

Writing documentation can be a business model, as [Raymond00] points out. Many OSP fund themselves by selling their documentation. This may seem hypocritical at first, selling something closed to be able to produce something open, but it works. Furthermore, special licenses exist that bridge the gap between commercial exploitation and keeping information freely accessible.[171]

[171] One such license, the GNU Free Documentation license, is described at http://www.fsf.org/licenses/fdl.html

5.4.4.8 Bug Triaging

Involved Roles	Testers, Developers, Supporters
Involved Tools	Bug Tracker

Most software has far more defects than can be corrected with reasonable amounts of time and resources. Bug triaging tries to identify the most critical bugs that provide the most value if they are fixed. Determining the most important bugs can be done in a variety of ways. Collecting statistics about the most common bugs is the starting point for a good assessment [Spolsky01] argues that only the bugs with a positive net present value stemming from their fixing should be fixed. This is a very basic economic argument that is often overlooked in the open source world. It is often a matter of pride to try to fix all known bugs, when time would be much better spent on other things, like improving documentation or adding new features. It is often (incorrectly) argued that open source software should support every arcane configuration under the sun. This is nonsense. Economic principles apply to the open source world just as they do to other areas. In fact it is one of the weaknesses of many open source projects to assume that resources are unlimited and can therefore be spent at will. Measuring the value of a bug being fixed can be challenging. One approach is to assess its likelihood and assign a severity to it. Bug fixing efforts can then start from the likely and critical bugs and fix as many bugs as possible within the allotted time. The Mozilla project operates a very large bug database; at the time of this writing, it contained 25899 bugs.[172] The following rules have been established by Mozilla to help triage bugs:[173]

> **Can this bug report easily be reproduced?**
> Or does the bug report lack accurate steps to reproduce, along with clear expected results and actual results? Is the bug's description ambiguous like, "Mozilla crashes" or "browser problem", or does it accurately describe an actual problem?
>
> **Does the bug report identify one specific problem?**
> Less experienced bug reporters try to save time by writing a single bug report that raises many different, unrelated issues. A bug report should contain no more than one issue, for a multitude of reasons: Distinct bugs enable work to proceed in parallel. When a bug is fixed, it must be verified against a specific test. Bug reports that list multiple conditions and problems are difficult to verify. Bug reports that list multiple bugs may mean that some problems noted do not get addressed. Bug reports that list a single bug are more likely to be resolved in a timely fashion.
>
> **...and can an engineer actually fix it?**
> Less experienced bug reporters will submit bugs that are general complaints, and which do not distill to a fixable bug.
>
> **Does a bug look like a duplicate report of a known issue?**

[172] http://bugzilla.mozilla.org/
[173] http://www.mozilla.org/quality/browser/prescreening.html

Many bugs are reported many, many times. If you think it might be a duplicate --- but are not sure --- just add your comments into the bug report. Someone down the road will be thanking you for your thoughtfulness.

5.4.5 Project Management

Project management processes mainly support the project manager and interested parties to keep an overview of a project and steer its long-term direction. Most near-term activities in OSP are performed ad hoc, and it is generally not a good idea to try to fit those activities into a strict project management framework. In other words, micromanagement is lethal for OSP. Project managers also have to take the notions of the project participants into account. If they appear to manage too much they will be met with a backlash, as most OSP participants prefer to work unencumbered from timesheets and deadlines. Scheduling, tracking and allocation are to be seen as supporting processes only.

5.4.5.1 Scheduling

Involved Roles	Developers, Manager
Involved Tools	Asynchronous Communication, Issue Tracker, Calendaring

It has been argued that OSP are immune to Brook's law ("Adding manpower to a late project makes it later") because the concept of deadlines or schedules does not apply to them. This is a flawed notion and is only true on a very superficial level. Well-managed project do you use scheduling to raise effectiveness of their collaborators. The Mozilla project has used a very sophisticated scheduling for years[174]. A roadmap specifies target dates for feature freezes and releases. Special rules apply for the time periods between releases.

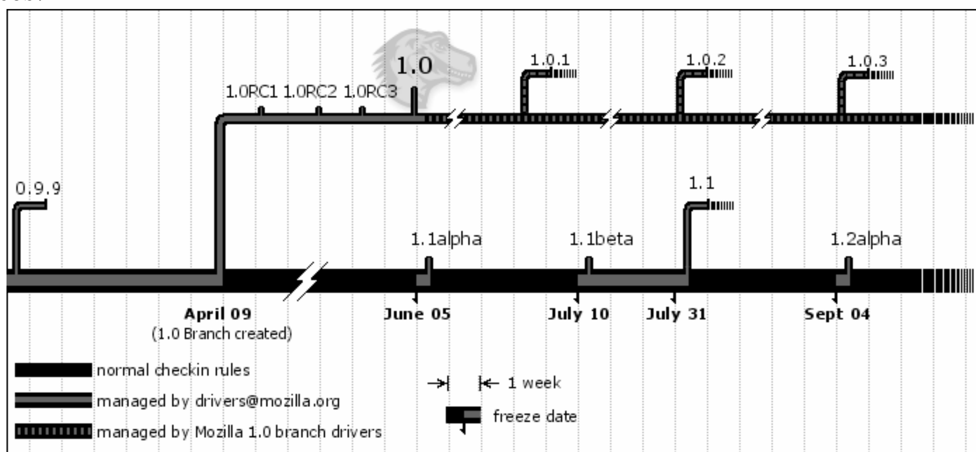

Figure 11: Mozilla Roadmap

In the experience of Brendan Eich, who has managed the Mozilla project for several years, sticking to a schedule proved to be very challenging: "*Just before 2001 began, I wrote that useful and relevant (defined by the community) extensions are always welcome, provided that they don't have a high opportunity cost in terms of contributors*

[174] http://www.mozilla.org/roadmap.html

who otherwise could and would have helped hack on 1.0. But by the fall of 2001, as noted in the Mozilla 1.0 manifesto, the opportunity costs of features and extensions had grown to the point where such "non-1.0" work jeopardized a 1.0 milestone that fit into any achievable schedule."

Contributors to OSP are primarily motivated by seeing their code being included in a project at the earliest possible time. Imposed schedules conflict with this goal, and are thus strictly opposed by most contributors. It is the duty of the project manager to impose deadlines in the interest of the project, while still considering contributor biases. Interestingly, most work is performed as soon as feature freezes are announced, as everyone scrambles to include his favorite feature before the freeze is in effect.

5.4.5.2 Tracking

Involved Roles	Developers, Manager
Involved Tools	(A)synchronous Communication, CMS, Issue Tracker

Closely related to scheduling, tracking concerns itself with the oversight of ongoing and upcoming work to spot issues as early as possible, and introduce corrective measures. Tracking in OSP is much complicated by the distributed nature of its contributors, their voluntary and therefore unreliable commitment and by communications issues. Nevertheless, many OSP make the effort to track progress, and some formalize the process by writing weekly newsletters or holding online meetings. One such project where the author is involved, PostNuke, conducts weekly meetings on IRC (Internet Relay Chat) to discuss the issues of the week, and set the agenda for the following week.[175]

5.4.5.3 Allocation

Involved Roles	Manager
Involved Tools	Asynchronous Communication, CMS. Calendaring

Most OSP face severe staff shortages during their lifecycle. Even successful, popular projects find it very hard to allocate resources to specific task. Reasons for these difficulties include the inability to force assignments on volunteers, lack of knowledge about the skill sets of contributors, lack of oversight about the open tasks. Improvements in allocation hold the promise of much better utilization of available resources. One problem with allocation is that the individuals with the oversight to be able to allocate are usually inundated with work, and have to decide if they want to spend their time instructing others, or working on their own task. In the experience of the author, most contributors like to be given specific tasks instead of vague goals. Specific tasks make it easier for contributors to "do the right thing;" especially given that many contributors feel that they do not know enough about a project to contribute more substantially. The need to provide detailed instructions puts even more of a strain on the key persons in a project. The PostNuke project, for instance, has seen over 250 developers join the project over the course of a year. New developers demand attention from senior contributors. Existing

[175] http://chaynez.dyndns.org/meetlogs/

contributors are faced with a choice: Either spend time introducing new contributors to the details of the project, or contribute to the project themselves. Pledging help for a project is quickly done and very non-committal. As a result, only about one out of ten "contributors" to a project actually comes through and justifies the initial time investment by other contributors to educate him. The reasons for this very low success rate are manifold. Many projects do have a very strong culture that may be hard to get into.[176] Another reason is that new contributors to a project fail to find instant gratification, especially with larger projects. It is sometimes almost impossible to make a meaningful contribution to a project without an extensive study of the projects peculiarities. With so many projects around, many contributors are not ready to invest the necessary time, and may start their own projects instead. Case in point, sourceforge.net contains the carcasses of thousands of such projects that never went anywhere beyond the initial, vague idea. As projects grow, it becomes harder for individual contributors to gain status within the project because so many other contributors compete for the glory of widespread recognition. Most projects sustain only around a dozen key contributors who are in the public eye. Other contributors are barely noticed.

Many, if not most OSP processes are performed with a variety of tools. While tools might not be of academic interest on their own, their usage in conjunction with software engineering and other processes raises many interesting questions. A survey of existing tools also allows OSP participants to judge for themselves if they use the right tools for a particular task.

5.5 Tools

OSP without tools are unthinkable. As outlined earlier, OSP have only become feasible since Internet connectivity is widespread and cheap. It is therefore not surprising that many OSP concern themselves with developing better tools to conduct OSP, and many of the Internet innovations started as a small project to improve collaboration or communication. Tools and processes form a symbiotic relationship. New processes beget new tools, and new tools beget new processes. The number of tools at the disposal of the budding OSP contributor is immense. For the purposes of this framework, only a small number is examined, and their application for OSP is shown. For each tool, a sample is demonstrated.

5.5.1 Marketing

Tools for marketing are essentially the same tools that are used for communication. The internet is by far the largest marketing channel, and internet communication tools abound.

5.5.1.1 Content Management System

Tools for managing web sites abound. There are hundreds of tools, from simple news scripts to full-featured content management. The needs of each project are different, but some considerations apply to most projects. OSP would want to make sure to follow guidelines established by Jakob Nielsen.[177]

[176] FreeBSD is a project that is often accused of such elitism. http://bsdvault.net/hubbard.html
[177] http://www.useit.com/alertbox/20020512.html

Include a One-Sentence Tagline
Start the page with a tagline that summarizes what the site or company does, especially if you are new or less than famous.

Write a Title with Good Visibility in Search Engines and Bookmarks
Begin the TITLE tag with the project name, followed by a brief description of the site.

Emphasize the Site's Top High-Priority Tasks
Your homepage should offer users a clear starting point for the main one to four tasks they'll undertake when visiting your site.

Include a Search Input Box
Search is an important part of any big website. When users want to search, they typically scan the homepage looking for "the little box where I can type," so your search should be a box. Make your search box at least 25 characters wide, so it can accommodate multiple words without obscuring parts of the user's query.

Show Examples of Real Site Content
Don't just describe what lies beneath the homepage. Specifics beat abstractions, and you have good stuff. Show some of your best or most recent content.

Begin Link Names with the Most Important Keyword
Users scan down the page, trying to find the area that will serve their current goal. Links are the action items on a homepage, and when you start each link with a relevant word, you make it easier for scanning eyes to differentiate it from other links on the page.

Offer Easy Access to Recent Homepage Features
To help users locate key items, keep a short list of recent features on the homepage, and supplement it with a link to a permanent archive of all other homepage features.

Don't Over-Format Critical Content, Such as Navigation Areas
You might think that important homepage items require elaborate illustrations, boxes, and colors. However, users often dismiss graphics as ads, and focus on the parts of the homepage that look more likely to be useful.

5.5.1.2 Online fundraising

Most OSP do not raise funds, but instead rely on the direct donations of individuals to operate their project. As the scope of OSP increases, many participants find it hard to contribute due to their skill set that might differ from the one needed in a project. These supporters often do not know how they could possibly contribute to a project, and many subsequently do not contribute at all. Online fundraising, combined with a rating system for contributors, allows participants to express their appreciation by donating to the cause

of the project. Affero[178] maintains such a system. It allows to track the reputation of individuals (a very important ingredient of successful OSP), and offers an easy way for users to contribute to an OSP. Figure 12 illustrates the basic concept of Affero.

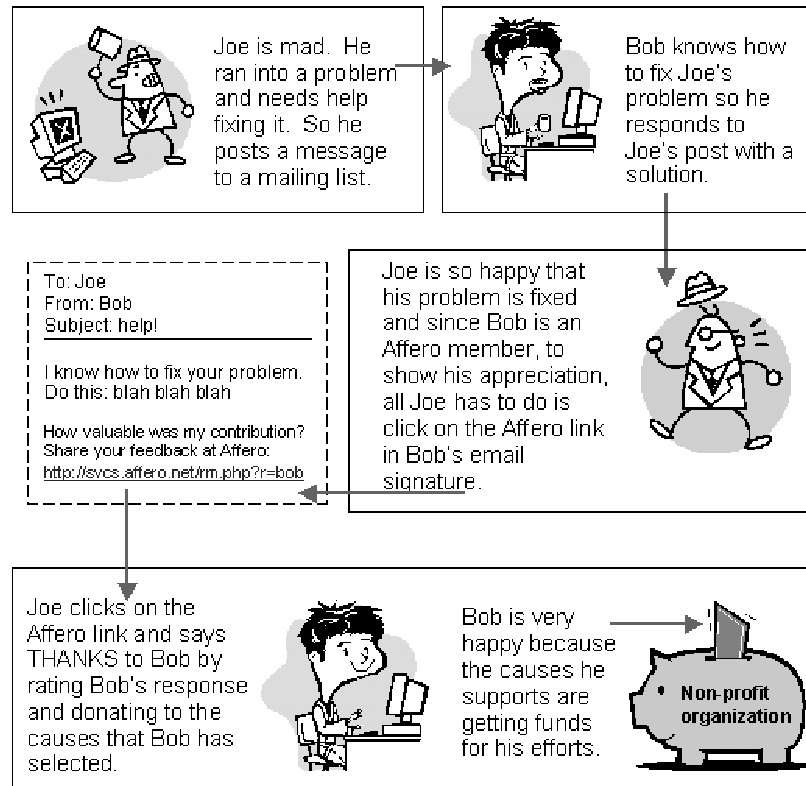

Figure 12: Affero online fundraising system

5.5.2 Human Resources

Human resources processes are mostly communication, and communication tools support staffing and knowledge acquisition. The phenomenon of virtual teams is quite recent, and tools to support it have only recently sprung up. These include knowledge management and skill matrices.

5.5.2.1 Knowledge management

OSP produce enormous amounts of data on mailing lists, web sites, repositories, in online communications, and other media. This data is very often dispersed across multiple sites, and no one maintains it. At the same time, most OSP suffer from a lack of good documentation. Several systems to address these problems have been in development. The major issues are the wide variety of media where content originates, and the widely varying quality. Traditionally, knowledge management tools have been touted as a solution for these problems. 70% of knowledge management projects fail[179], mostly due to efforts to "build the grand database in the sky" to house all the knowledge of an

[178] http://www.affero.com
[179] http://www.computerworld.com/softwaretopics/software/story/0,10801,46693,00.html

organization. What works is to entice collaborators to share their information. This is best done with Weblogs[180]. “Weblogs are often-updated sites that point to articles elsewhere on the web, often with comments, and to on-site articles.” John Robb proposes to use Weblogs for Knowledge Management:[181]

”What is a K-Log (Knowledge Management Weblog)? This is still a new concept, but it will be huge. Weblogs will potentially become the first new widely adopted desktop productivity tool since the browser. Here is the feature set:

1. *The ability to post to the Web through a browser.*
2. *Automatic organization of posts in a time based format.*
3. *Automatic archiving of posts in a calendar.*
4. *The ability to post documents and files to the cloud.*
5. *The ability to develop directories of resources (links and posted documents/files).*
6. *The ability to subscribe to other people's Weblogs (RSS newsfeeds).*
7. *The ability to view community stats (top Weblogs by traffic, referrers, search, and other data).*
8. *Integration with e-mail (to be able to post via e-mail or forward directory resources and posts).”*

Figure 13 shows one K-Log product, Radio Userland. A list of newsfeeds is collected from sites across the web, and can be commented on, and posted to the weblog of the author. The news aggregation, rating and commenting infrastructure of weblogs is continually being enhanced. As a recent article demonstrates[182], weblogs do have a very high impact on the search results of the number one search engine, Google. Due to their ability to disperse information much more effectively than other media, weblogs will play an important role for OSP. OSP depend on a constant influx of talent and ideas, and weblogs can provide it.

[180] http://newhome.weblogs.com/historyOfWeblogs
[181] http://groups.yahoo.com/group/klogs/message/33
[182] http://www.microcontentnews.com/articles/googleblogs.htm

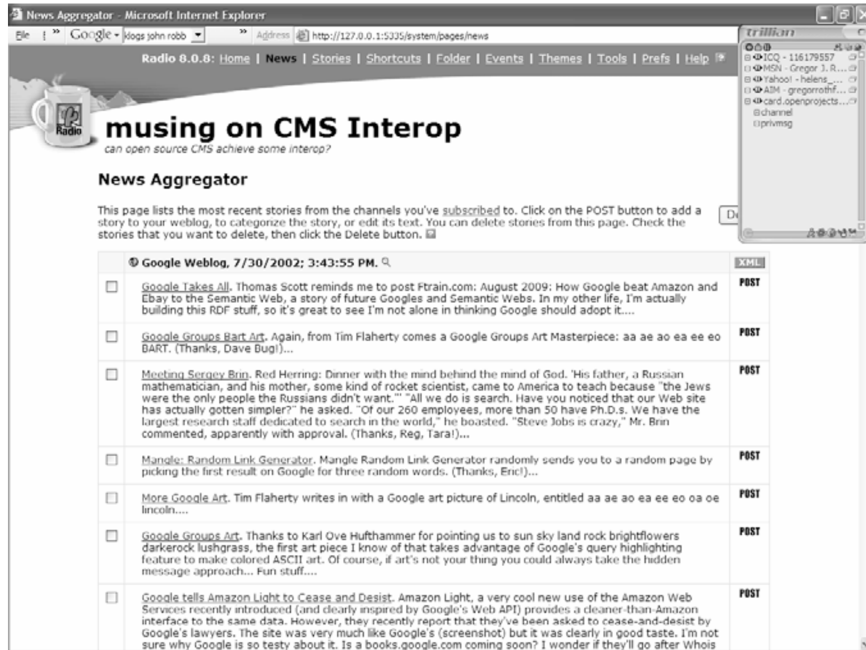

Figure 13: Radio Userland news aggregation

5.5.2.2 Skill Matrices

Most OSP have a very fluctuating base of contributors. This makes it hard to assign tasks to participants, as the project manager barely knows the skills of volunteers. Skills matrix software can help. It allows contributors to classify themselves on a wide range of skills, and optionally allows other participants to rate their skills. Applied consistently and ideally globally as opposed to single projects, skill matrices are a very valuable tool to assess and plan volunteer resources. Currently, skill matrices are not very widely deployed, and no central skill matrix exists. Sourceforge does provide a skill matrix for projects hosted on its site, as shown in figure 14. While that particular matrix leaves a lot to be desired such integration with peer ratings, it is a very interesting first step.

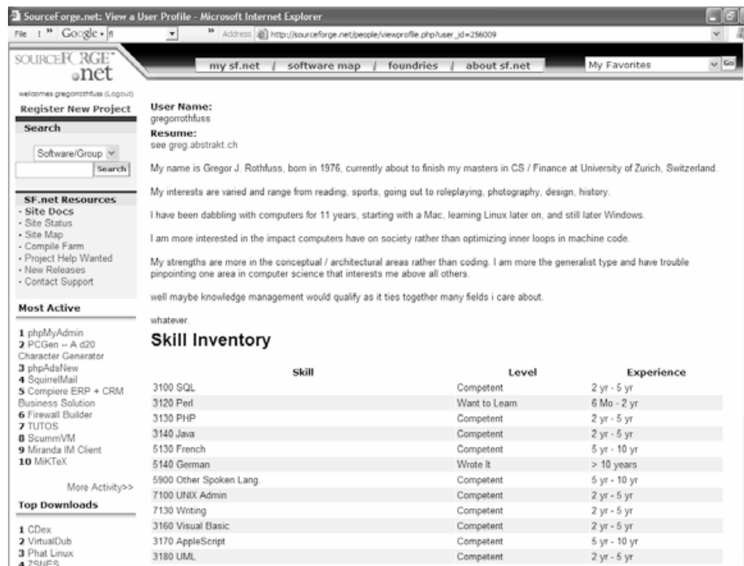

Figure 14: Sourceforge skill matrix

5.5.3 Systems Management

Traditionally, a rich variety of tools for systems management existed for OSP. This can be traced back to the common background of many of these projects. Very often, they got started by people in a systems position to assist them in their daily work routine[183].

5.5.3.1 Backup

Backup is probably the most unglamorous task imaginable. Yet it is remarkably important, as is often found out too late. Backup tools span a range from the very simple, manual copying of files to highly sophisticated automated continuous processes across several servers. Backup tools for OSP should leverage automation and the distributed nature of project participants by mirroring content to various servers around the world. This is commonly done for the source code of a project (members keep local working copies), and to some extent for communication (members keep personal archives).

5.5.3.2 Asynchronous communication

Most communication in OSP are performed asynchronously. Project members are distributed all over the world, and not all project members can participate in the project on a daily basis. The Internet evolved several tools for asynchronous communication.

Mail Servers

Originally developed for an Internet where shared code of ethics were the norm, mail servers now suffer from architectural weaknesses that make them vulnerable in today's more hostile Internet environment. Beyond reliable mail delivery, mail servers are now expected to protect against spam and scan incoming mails for viruses.

[183] As Larry Wall, the inventor of Perl, recounts, he developed Perl for personal use at his place of work. http://history.perl.org/PerlTimeline.html

Mail Clients
OSP participants receive far more mails on a daily base than the average Internet user. Special strategies and techniques are required to handle the increased load.[184] Smart filtering can help to organize large volumes of emails

Mailing Lists
The most common way of communication for OSP are mailing lists. Mailing lists work like a broadcast medium. They reach a lot of people, but noise increases with the number of participants. The concept of signal-to-noise ratio is often mentioned as a key indicator of mailing list quality. Proposals to raise mailing list quality through collaborative filtering [Greant02] try to address those issues.

News Servers
Conceptually very similar to mailing lists, news servers do have superior threading support, and can store discussions indefinitely. News (NNTP) is a mature protocol, and many excellent news clients do exist.

Web Forums
Web forums are another possibility to conduct online discussions asynchronously. While they might be easier for casual users they do not cope well with heavy discussion loads, and are cumbersome and slow to operate.

5.5.3.3 Authentication systems

Authentication in distributed environments can be quite a challenge. Most OSP participants have never seen each other face to face, and still need to work together in a trust relationship. Projects need to protect themselves against impersonation attacks, must keep sensitive information private, and must protect their software from being tampered with. Public key cryptography is a good way to secure the data of a project. The Debian project uses PGP keys to digitally sign their communication and their software releases.[185]

5.5.3.4 Real time communications

Real time communications have proven to be an invaluable tool for OSP. One popular real time communication program that is widely used is IRC (Internet Relay Chat)[186]. OSP participants use IRC to discuss development issues with fellow developers, or provide support for users. IRC is also used to conduct meetings, and serves an important role to foster social exchange between project participants. Popular OSP do have hundreds of participants in a chat room at any one time, around the clock. Besides IRC, many other instant messaging networks have been established in recent years. Those networks are usually not interoperable, which makes them less than useful for OSP. One way around such limitations are multi-protocol instant messaging clients such as

[184] http://www.acm.org/sigchi/chi96/proceedings/papers/Whittaker/sw_txt.htm
has many notes on email overload.
[185] http://www.debian.org/devel/

Trillian.[187] Trillian allows connections to five different instant messaging networks simultaneously. It supports notifications on events, for instance if certain keywords are detected in a chat room, and detects if a user is idle for extended periods of time.

Figure 15: Trillian Multi-Protocol Instant Messaging Client

5.5.4 Software Engineering

The collection of software engineering tools is very rich because many OSP cater to fellow programmers and software engineers. Special care should be taken in the selection of software engineering tools as these tools affect project productivity immediately. Depending on project goals, technologies used, and the skill sets of its participants a project may employ more or less software engineering tools. Bug trackers, SCM, and a unit testing framework make sense for almost all OSP though, and can be considered a minimal requirement.

5.5.4.1 Bug Tracker

Most OSP employ some form of ticketing system to keep track of open issues, bug reports, and feature requests. Ticketing systems provide a good framework for assigning tasks in virtual teams, and allow OSP participants to learn quickly about the state of a project. Often, ticketing is used extensively in the release process to drive the number of open issues as far down as possible, and keep track of dependencies between software defects. Many Open Source ticketing systems exist, and most only support a very

[186] http://www.irchelp.org/
[187] http://www.trillian.cc

rudimentary feature set. Request Tracker[188] is a very sophisticated ticketing system. It supports web, email and command line interfaces, stores its data in a relational database, allows integration with CVS to query and close open tickets, and has sophisticated access control.

Figure 16: Request Tracker

188 http://www.bestpractical.com/rt/

5.5.4.2 SCM

CVS is a version control system that records the history of source files by only storing the differences between versions. It is by far the most popular Source Configuration Management (SCM) tool, and is extensively used for OSP. CVS allows OSP to conduct work in parallel, and supports conflict resolution if two persons edited the same file at the same time. CVS has many client implementations, with TortoiseCVS probably one of the most comfortable.

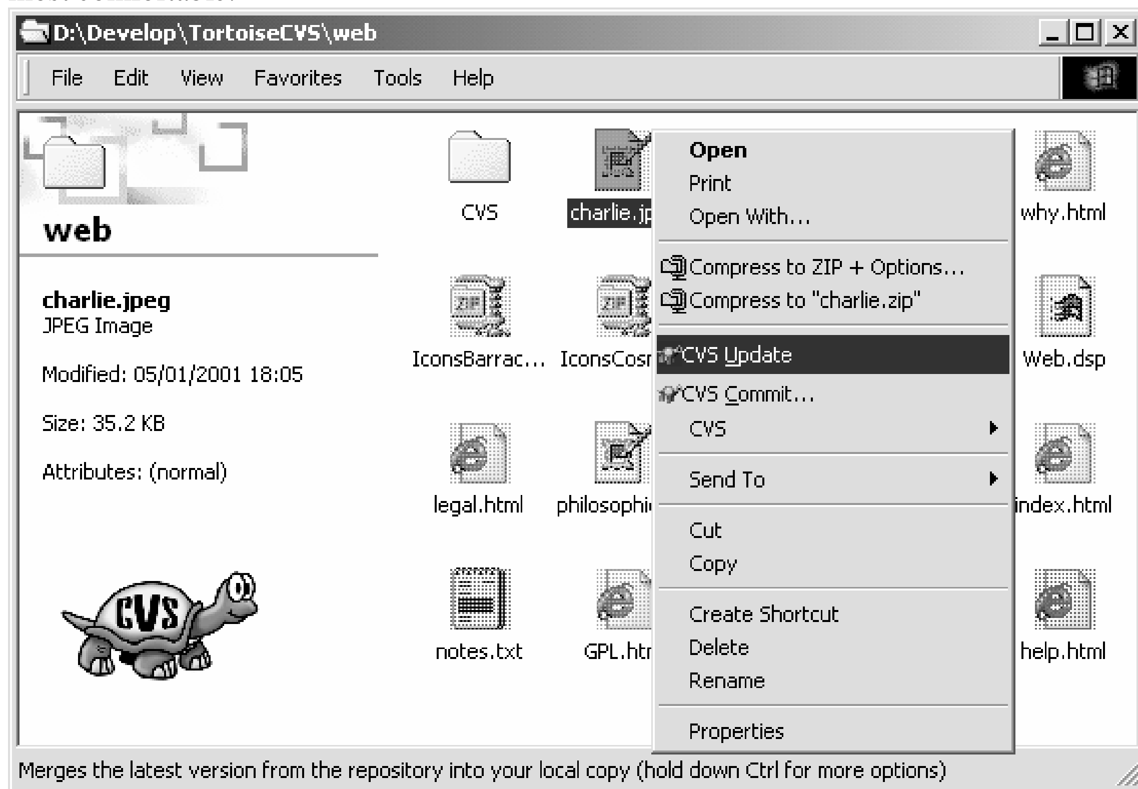

Figure 17: TortoiseCVS integration into the Windows explorer

5.5.4.3 Unit Testing Frameworks

Unit Testing is a new development style that is part of the extreme programming school of thought. It postulates that no code goes into a product unless it has associated tests, that these tests be written before the code, that these test determine what code needs to be written, and that all tests be maintained and run after each change in the source code of a product. The goal of unit testing is to catch errors early, and prevent their reappearance at a later stage. As [Nishinaka01] explains, many OSP could benefit from the principles of unit testing to raise quality in their projects. Unit testing is only feasible if it is applied consistently. Various unit testing frameworks have been developed to reduce the manual labor required to maintain unit tests. JUnit[189] is a well-known unit test package for the JAVA language. JUnit features include assertions for testing expected results, test fixtures for sharing common test data, test suites for easily organizing and running tests, and graphical and textual test runners.

[189] http://www.junit.org

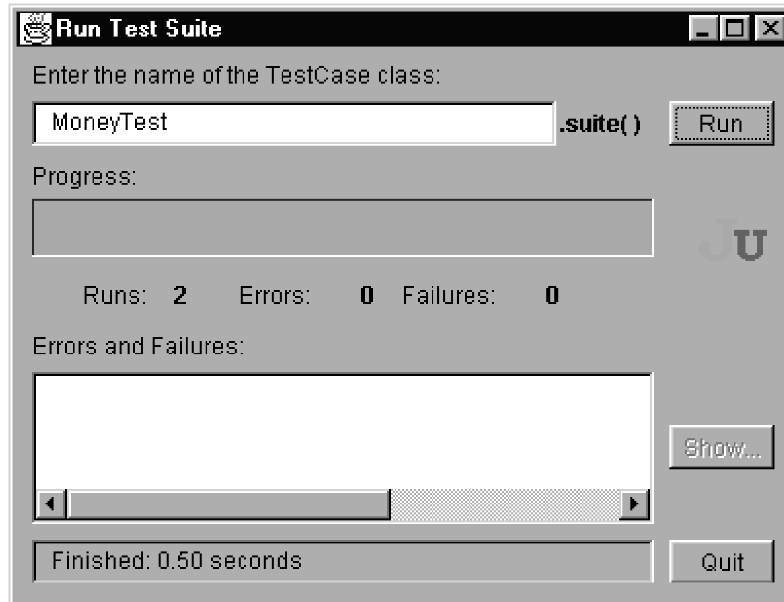

Figure 18: JUnit framework

5.5.5 Project Management

Project management tools for OSP are scarce. This can mostly be traced back to their questionable usefulness in most projects. Since amateurs run so many OSP in their spare time, they are usually not run like classical projects, and project participants are wary of introducing elements that may take the fun out of the project. Often these individuals are subjected to project management in their professional lives and would rather avoid the red tape that more often than not comes with project management. It is a widely held belief that OSP manage themselves and therefore need little tool support. This belief stems from a rejection of everything that smells "corporate" in OSP circles. Many participants do not want to submit to the same rigorous management in their spare time that they do during work hours. There are some tools that help OSP in a non intrusive way though, such as Calendaring and Issue trackers.

5.5.5.1 Issue Tracker

Issue tracking is very similar to bug tracking, and can be accomplished with the same tools given that these tools allow the ability to prioritize issues, attach them to a due date, and send reminders emails to the assignees.

5.5.5.2 Calendaring

OSP participants need to juggle their project involvement with their day jobs and many other activities. Often, a bit of scheduling would actually help to remind participants to finish some work that others in the group are waiting for. Various Internet-enabled calendaring tools can help with this task. vCard and vCalendar are two Internet standards that allow the ability to share appointments ands schedules across programs and platforms.

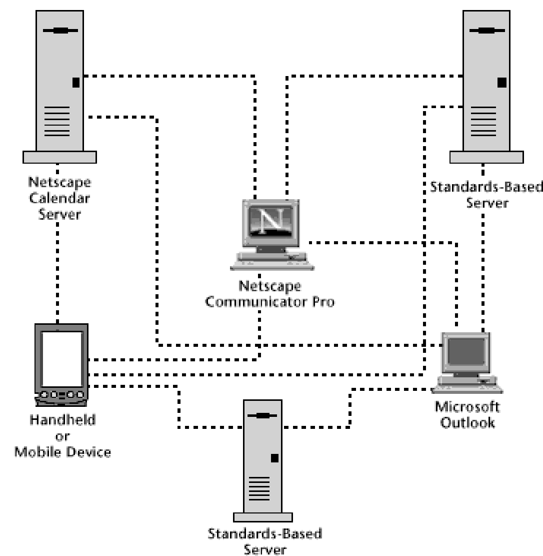

Figure 19: Calendaring across platforms

Tools provide valuable services to OSP, but should not be used for their own sake. Many OSP exhibit a over fondness for tools and try to solve issues with tools that should be resolved with processes instead. Tools support processes, but cannot replace them. One pitfall is to set up dozens of data entry tools to gather user comments, bugs, status reports, summaries, documentation, and then wonder why no one finds anything anymore. It is often better to use fewer tools more extensively than many tools only superficially. The ease to setup new tools masks the real problems that only crop up later: duplicate information, data silos that gobble up information but never release it again.

6. Conclusion

The OSP framework ties together roles, processes and tools to explain Open Source development both from a software engineering perspective as well as in the context of a social phenomenon. The major finding of the framework is that OSP cannot be explained by a singular discipline. Instead, an interdisciplinary approach is being called for. The knowledge about OSP is advancing rapidly. As a result, current explanations will leave a lot to be desired in a few years, and the framework considers those by being extensible. Another goal of the framework is to go beyond explanations, and provide advice for OSP participants and project leaders. Much work remains to be done in the coming years to complete the understanding of Open Source.

This paper looks at the Open Source landscape as it presented itself during summer 2002. Starting with a historical overview, the paper then goes on to develop the major concepts of Open Source, and put them into the context of classical (closed source) projects. Both strong and weak aspects of Open Source are examined, and the existing literature is sampled. The comparison leads to a search for theories that are able to explain the phenomenon, and finds existing theories lacking. Finally, a framework is developed that draws together the various strands of different theories, and adds in new elements. Over the course of writing this thesis, some areas showed promise for further research.

6.1 Areas for further Research

At the time of this writing, over 80 Open Source papers have been published. .A complete treatise of all these papers would far exceed the space allotted for this paper, but some interesting trends stand out.

Reputation and Trust
As [Raymond98b] has pointed out, reputation among peers is one of the driving forces behind OSP. Becoming a celebrity among some of the most talented software engineers worldwide is a goal few reach, but many attain. The mechanics behind reputation are poorly understood. How are decisions made in large collaborative groups? Why do some participants have proportionally greater influence than others do, and what are the factors that contribute to their reputation? Closely related to reputation is trust. Most OSP participants never physically meet each other. Yet their social networks are strong enough to place large amounts of trust into their fellow participants. Most OSP do have at least ten people talking for every single person doing actual work. How can metrics of trust, such as the excellent work done by [Levien02] be used to assign relevancy to information?

Economic Models
Participation in OSP is ultimately motivated by several factors. While a lot has been said and written about the "Noosphere" and other motivational concepts, most theories fail to have an economic grounding. At the same time, OSP struggle for financial support, and most Open Source startups fail to find a sustainable business model. For OSP to succeed longer term, and grow beyond a pastime for software engineers, valid economic models for Open Source need to be found. Finding common ground between Free Software idealists who oppose closed source software on philosophical grounds and commercial software developers, who write software for a living, will not be easy. It remains to be seen if the idealistic culture that permeates the Open Source community will be sustainable as the financial context becomes more lucrative, and more companies start to operate under an Open Source model. The commercial nature of Open Source is a too recent phenomenon to determine its long-term viability. For instance, it is unclear at this point how and if Open Source can master the tremendous marketing and development challenges associated with bringing Open Source products to a mainstream audience. Further research is needed to determine if individual contributors benefit financially from their participation in OSP, and by

which means. For instance, [Il-Hom02] found a correlation between high Open Source rank and higher wages, apparently due to companies making inferences about productivity differences based on the rank of a contributor. The author believes that economic incentives can lead to prevalence of Open Source software under the right circumstances. This would need a reexamination of patent laws, copyright laws and other areas of the common commercial code.

Implications for Software Engineering

OSP have succeeded despite (or because?) their defiance of conventional software engineering wisdom. Acknowledgements by [Torvalds01] that software is not architected, but grows with directed evolution question fundamental beliefs. Other authors have noted similarities between OSP and the concepts of extreme programming. Rapid releases with constant feedback have helped many OSP to reach very high levels of quality. Can these concepts be applied for projects with a less technical audience that is unable to provide technical feedback? How can testing and debugging be "smartened up" from its current colossal waste of programmer resources? Is requirements engineering parallelizable in the same way as testing is? Many projects employ massive parallel efforts to find faults in the software, leading to lots of duplicated work. How can having access to the source code be made more relevant for less technical users? Could newer, higher level programming languages allow a broader participation by non-programmers? Tool support for OSP software engineering is continuously increasing. How does the availability of new tools affect software engineering best practice in OSP?

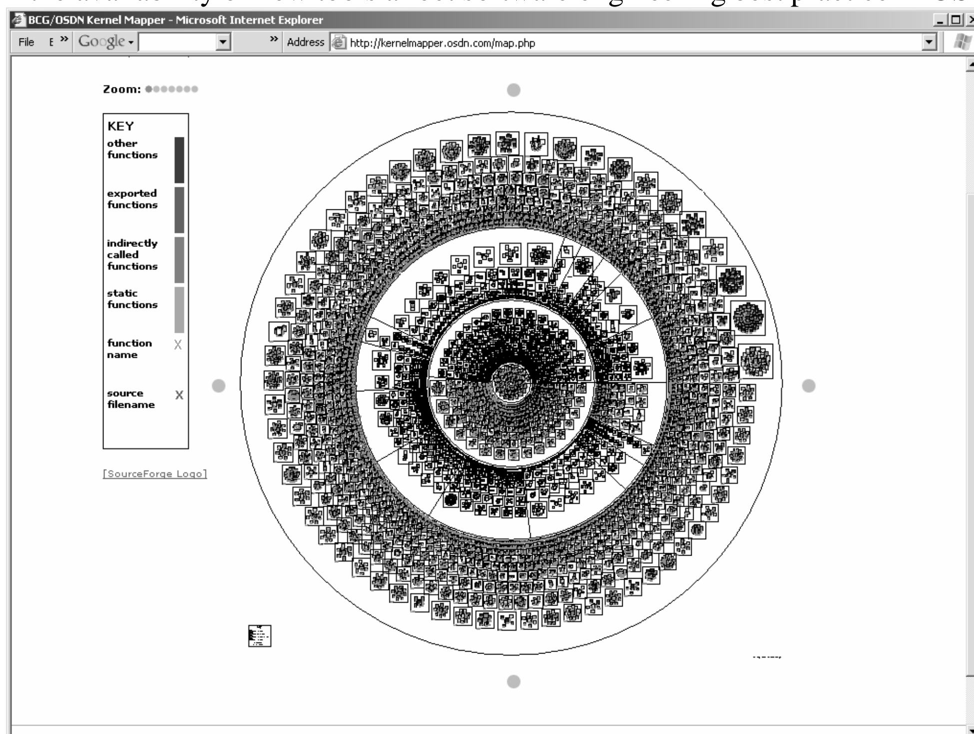

Figure 20: Modularity of the Linux kernel

Appendix A: Using the OSP framework

The OSP Framework is not a ready-made recipe for conducting Open Source projects. Each project is different, and no methodology fits the needs of all projects equally well. The OSP Framework provides the necessary data points for a detailed analysis of existing or yet to be created projects. The interested reader should therefore understand how to read the framework matrix, and how to conduct research that is more detailed based on his findings.

The framework matrix displays an idealized project where all activities, roles and processes are present. Real projects will likely differ substantially from the matrix. It is suggested to copy the matrix, and fill out the parts of the matrix that apply to a particular project. It is educational to compare the resulting matrix with the one provided here to spot differences, and ask why a project deviates from the idealized project. It is likely that various archetypal patterns within the matrix could be established by analyzing several hundred existing projects. Once the matrix for a project is filled out, one can then proceed to study the individual elements of the matrix that compose this particular project. Staffing and resource decisions should then be a logical consequence.

It may also be helpful to revisit the matrix periodically, as project maturity influences the composition of the matrix. More mature projects are more likely to have a solid technical decision-making structure, and place more emphasis on change and release management than emergent projects.

The OSP framework remains at a rather abstract level and does not outline the detailed procedures to start and manage an OSP. The best way to gain implementation experience is to observe and participate in existing OSP. Large OSP hosts like sourceforge.net provide a good starting point.

Another interesting area that is not explicitly covered by the OSP framework is the assessment of existing projects. The following questionnaire from [Kienzle01] might help in that endeavor.

Project Success

- Growth - of developer/contributor base
- Growth - of user base (non-developers)
- Number of contributors
- Developer activity - and is it visible?
- Openness of project - allows others into the development process
- Openness of project - communicates/indicates progress
- Quality of documentation - web site, changelogs, manuals
- Number of discussion/mailing list posts
- Number of websites discussing/linking
- Number of downloads
- Number of page views

- Number of other projects basing their project on it
- Number of commercial entities basing their business on it
- Adoption and support/investment from commercial entities - ranges from lip service to actual direct support
- Rate of stable release
- Number of bugs reported/resolved - robust under unexpected uses - stability (# of bugs reported dependent on project popularity)

Personal Success/Outcomes

- Learning - extended programming skills
- Enjoyment/fun - increased
- Employment outcomes
- Reputation outcomes

Factors contributing to project success

- Active marketing/promotion of project -an evangelist is helpful?
- Nice Website
- Willingness of founders to grant commit access to others
- Acceptance of ideas/viewpoints of others - and incorporate if appropriate
- Architecture of the project prevents coordination problems - embedded scripting language, plug-in architecture
- Leadership - technically proficient (a good programmer?)
- Leadership - personable
- Usefulness of software - fills a niche or is better than current
- Competent technical core of developers
- A sense of community
- Stability
- Documentation and support
- Sex appeal - allows creativity and growth, excitement about the software and end product, some attraction.
- Ownership by developer community
- Communication among developers
- Communication to user community
- Open Development Cycle
- Low Barrier to entry for developers - easy to get in
- Redundant Developer roles
- Clear dispute resolution mechanism - not a management hierarchy
- No Flat organization
- User base size

Bibliography

[Anderson02] Anderson, Ross: How to Cheat at the Lottery (or, Massively Parallel Requirements Engineering)
University of Cambridge Computer Laboratory, 2002

[Anderson02b] Anderson, Ross: Security in Open versus Closed Systems - The Dance of Boltzmann, Coase and Moore
University of Cambridge Computer Laboratory, 2002

[Arief02] Arief, Budi and Bosio, Diana: Dependability Issues in Open Source Software
Technical Report CS-TR-760 (February 2002), Department of Computing Science,
University of Newcastle upon Tyne

[Arief01] Arief, Budi and Gacek, Cristina: Software Architectures and Open Source Software – Where can Research Leverage the Most?
Position Paper, First Workshop on Open-Source Software Engineering, 23rd International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

[Asklund01] Asklund, Ulf and Bendix, Lars: Configuration Management for Open Soure Software
Position Paper, First Workshop on Open-Source Software Engineering, 23rd International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

[Asundi01] Asundi, Jai: Software Engineering Lessons from Open Source Projects
Position Paper, First Workshop on Open-Source Software Engineering, 23rd International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

[Bessen02] Bessen, James: Open Source Software: Free Provision of Complex Public Goods
Research on Innovation, June 2002
http://www.researchoninnovation.org/opensrc.pdf

[Bezroukov99b]Bezroukov, Nikolai: A Second Look at the Cathedral and the Bazaar
First Monday, volume 4, number 12 (December 1999)
http://firstmonday.org/issues/issue4_12/bezroukov/index.htm
l

[Bezroukov99a] Bezroukov, Nikolai: Open Source Software Development as a Special Type of Academic Research (Critique of Vulgar Raymondism)
First Monday, volume 4, number 10 (October 1999),
http://firstmonday.org/issues/issue4_10/bezroukov/index.htm
l

[Blase02] Blase, Paul: A Software Company's Dilemma
DiamondCluster International White Paper, 2002
http://www.diamondcluster.com/work/Wpapers/WPSoftware.
asp

[Boldyreff02] Boldyreff, Cornelia and Nutter, David: Open-Source Artefact Management
2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.

[Brusehafer01] Brusehaver, Tom: What's Wrong With The Way Things Are Done?
Freshmeat.net Editorial, May 31st, 2001

[Capiluppi02] Capiluppi, Andrea and Lago, Patricia: Characterizing the OSS process
Position Paper, 2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.

[Cavalier98] Cavalier, Forrest J.: Some Implications of Bazaar Size
Note, August 11th, 1998
http://www.mibsoftware.com/bazdev/

[Chu-Carroll02] Chu-Carroll, Mark C. and Shields, David: Version Control: A Case Study in the Challenges and Opportunities for Open Source Software Development
Position Paper, 2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.

[Chuang98] Chuang, John: Economies of Scale in Information Dissemination over the Internet
PhD Dissertation, November 1998

[Connell] Connell, Charles: Open Source Projects Manage Themselves? Dream On
Sun

[Cook01] Cook, Jonathan E.: Open Source Development: An Arthurian Legend
Position Paper, First Workshop on Open-Source Software Engineering, 23rd International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

[Cox99] Cox, Alan: Cathedrals, Bazaars and the Town Council
Slashdot.org October 13th, 1999

[Crowston02] Crowston, Kevin and Scozzi, Barbara: Exploring the Strengths and Limits of Open Source Software Engineering Processes: A Research Agenda
Position Paper, 2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.

[Cubranic99] Cubranic, Davor: Coordinating Open-Source Software Development
Proceedings of the 7th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 1999

[Cubranic01] Cubranic, Davor: The ramp-up challenge in open-source software projects
Position Paper, First Workshop on Open-Source Software Engineering, 23rd International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

[Curtis95] Curtis, Bill and Hefley, William E. and Miller, Sally: People
Capability Maturity Model (CMU/SEI-95-MM-002)
Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, September 1995.

[Dafermos01] Dafermos, George N.: Management and Virtual Decentralized Networks: The Linux Project
First Monday, volume 6, number 11 (November 2001),
http://firstmonday.org/issues/issue6_11/dafermos/index.html

[Debian02] Debian: Debian Social Contract
The Debian Proiect, 2002
http://www.debian.org/social_contract

[Dinkelacker01] Dinkelacker, Jamie and Garg, Pankaj K.: Corporate Source: Applying Open Source Concepts to a Corporate Environment
Position Paper, First Workshop on Open-Source Software Engineering, 23rd International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

[Drakos00] Drakos, N. and Driver, M.: Debunking Open-Source Myths: Origins and Players
Gartner Group Research Note, January 18, 2000

[EC02] European Commission, DG Enterprise: Pooling Open Source Software
Feasibility Study, June 2002

[Edwards00] Edwards, Kasper: Epistemic Communities, Situated Learning and Open Source Software Development
Department of Manufacturing Engineering and Management, Technical University of Denmark, 2000

[Edwards00b] Edwards, Kasper: When Beggars Become Choosers
First Monday, volume 5, number 10 (October 2000)
http://firstmonday.org/issues/issue5_10/edwards/index.html

[Edwards01] Edwards, Kasper: Towards a Theory for Understanding the Open Source Software Phenomenon
Department of Manufacturing Engineering and Management, Technical University of Denmark, 2001

[Ehresman01] Ehresman, Luke: A Year of Learning
Freshmeat.net Editorial, March 10th, 2001

[Evers00] Evers, Steffen: An Introduction To Open Source Software Development
Diploma Thesis, Technische Universität Berlin, August 13th, 2000

[Feller00] Feller, Joseph; and Fitzgerald, Brian: A Framework Analysis of the Open Source Development Paradigm
Proceedings of the 21st Annual International Conference on Information Systems, Brisbane, Australia, 2000

[FLOSS02] Free/Libre and Open Source Software: Survey and Study
European Commission
http://www.unimaas.nl/FLOSS/

[Fielding02] Fielding, Roy and Roberts, Jeff: Why Do Developers Contribute to Open Source Projects? First Evidence of Economic Incentives
Position Paper, 2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.

[FSF91] Free Software Foundation: GNU General Public License
June 1991
http://www.fsf.org/licenses/gpl.html

[FSF01] Free Software Foundation: The GNU Project
June 2001
http://www.fsf.org/gnu/thegnuproject.html

[FSF01a] Free Software Foundation: GNU General Public License Frequently Asked Questions
June 2001
http://www.fsf.org/licenses/gpl-faq.html

[FSF02b] Free Software Foundation: Various Licenses and Comments about them
June 2002
http://www.fsf.org/licenses/licenses.html

[German02] German, Daniel M.: The evolution of the GNOME Project
Position Paper, 2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.

[Ghosh98] Ghosh, Rishab: Cooking pot markets: an economic model for the trade in free goods and services on the Internet
First Monday Vol.3 No.3 - March 2nd. 1998
http://firstmonday.dk/issues/issue3_3/ghosh/index.html

[Ghosh00] Ghosh, Rishab and Prakash, Vipul Ved: The Orbiten Free Software Survey
First Monday, volume 5, number 7 (July 2000),
http://firstmonday.org/issues/issue5_7/ghosh/index.html

[Ghosh02] Ghosh, Rishab: Clustering and Dependencies in Free/Open Source
Open Source Workshop, Toulouse, June 20-21, 2002

[Goldhaber97] Goldhaber, Michael H.: The Attention economy and the Net
First Monday, volume 2, number 4 (April 1997)
http://firstmonday.dk/issues/issue2_4/goldhaber/

[Halloran02] Halloran, T.J. and Scherlis, William L.: High Quality and Open Source Practices
Position Paper, 2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.

[Il-Hom02] Il-Horn, Hann, and Roberts, Jeff: Delayed Returns to Open Source Participation: An Empirical Analysis of the Apache HTTP Server Project
Graduate School of Industrial Organization, Carnegie Mellon University, March 2002

[Hannemyr99] Hannemyr, Gisle: Technology and Pleasure: Considering Hacking Constructive
First Monday, volume 4, number 2 (February 1999),
http://firstmonday.dk/issues/issue4_2/gisle/index.html

[Hassan01] Hassan, Ahmed E. and Godfrey Michael W.: Software Engineering Research in the Bazaar
Position Paper, First Workshop on Open-Source Software Engineering, 23rd International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

[Hertel02] Hertel, Guido; and Niedner, Sven: Motivation of Software Developers in Open Source Projects: An Internet-based Survey of Contributors to the Linux Kernel
University of Kiel, 2002 (in review)

[Hindle01] Hindle, Stephen: Best Practices for Open Source?
Advogato.org, March 21, 2001

[Hippel02] von Hippel, Eric: Open source software projects as user innovation networks
MIT Sloan School of Management, June 2002

[Hissam01] Hissam, Scott A. and Weinstrock, Charles B.: Open Source Software: The Other Commercial Software
Position Paper, First Workshop on Open-Source Software Engineering, 23rd International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

[Hubbard98] Hubbard, Jordan K.: Pulling on one end of the rope
Freshmeat.net Editorial, July 13th, 1998
http://freshmeat.net/articles/view/114/

[Iannacci02] Innacci, Federico: The Economics of Open-Source Networks
London School of Economics, Department of Information Systems, 2002

[Jones00] Jones, Paul: Brooks' Law and open source: The more the merrier?
IBM DeveloperWorks, May 2000

[Kaisla01] Kaisla, Jukka: Constitutional Dynamics of the Open Source Software Development
Department of Industrial Economics and Strategy, Copenhagen Business School, May 2001

[Kelsey99] Kelsey, John and Schneier, Bruce: The Street Performer Protocol and Digital Copyrights
First Monday, volume 4, number 6 (June 1999), http://firstmonday.dk/issues/issue4_6/kelsey/

[Kenwood01] Kenwood, Carolyn: A Business Case Study of Open Source Software
MITRE Corporation, July 2001

[Kienzle01] Kienzle, Rene: Project Success and Project Success Factors
Queensland University of Technology, 2001
http://www.osstrategy.com/article.php?sid=11

[Kircher00] Kircher, Michael; and Levine, David L.: The XP of TAO eXtreme Programming of Large, Open-source Frameworks
1st International Conference on eXtreme Programming and Flexible Processes in Software Engineering, Cagliari, Italy, June 21-23, 2000
http://www.cs.wustl.edu/~mk1/xp2000.pdf

[Kishida01] Kishida, Kouichi: Conceptual Sociological Model for Open Source Software
Position Paper, First Workshop on Open-Source Software Engineering, 23rd International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

[Krebs02] Krebs, Valdis: Introduction to Social Network Analysis
Cleveland, 2002
http://www.orgnet.com/sna.html

[Krishnamurthy02] Krishnamurthy, Sandeep: Cave or Community? An empirical Examination of 100 Mature Open Source Projects
First Monday, volume 7, number 6 (May 2002)
http://firstmonday.org/issues/issue7_6/krishnamurthy/index.html

[Krawitz00] Krawitz, Robert: Free Source Project Management
Advogato.org, November 4th, 2000

[Kuan02] Kuan, Jennifer: Open Source Software as Lead User's Make or Buy Decision:
A Study of Open and Closed Source Quality
Stanford Institute for Economic Policy Research, 2002

[Kuwabara00] Kuwabara, Ko: Linux: A Bazaar at the Edge of Chaos
First Monday, volume 5, number 3 (March 2000), http://firstmonday.org/issues/issue5_3/kuwabara/index.html

[Lancashire01] Lancashire, David: Code, Culture and Cash: The Fading Altruism of Open Source Development
First Monday, volume 6, number 12 (December 2001), http://firstmonday.org/issues/issue6_12/lancashire/index.html

[Lerner01] Lerner, Josh; and Tirole, Jean: Some Simple Economics of Open Source
Journal of Industrial Economics, July 24, 2001

[Lerner02] Lerner, Josh; and Tirole, Jean: The Scope of Open Source Licensing
Paper for “Open Source Software: Economics, Law and Policy”
Toulouse (France), 20-21 June 2002

[Lessig00] Lessig, Lawrence: Code and Other Laws of Cyberspace
July 2000, IBSN: 0465039138
http://cyberlaw.stanford.edu/lessig/content/books/

[Levien02] Levien, Raphael: Attack resistant trust metrics
Doctoral Thesis, University of California at Berkley, 2002
http://www.levien.com/thesis/thesis.pdf

[Liu00] Liu, Chang: Is Software Testing Production or Service?
Freshmeat.net Editorial, February 19th, 2000

[LPF91] League for Programming Freedom: Against Software Patents
February 28, 1991
http://lpf.ai.mit.edu/Patents/against-software-patents.html

[Madey02] Madey, Greg and Freeh, Vincent: Understanding OSS as a Self-Organizing Process
Position Paper, 2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.

[Martins00] Martins, Lalo: Why do Free Software projects fail?
Advogato.org, July 20th, 2000

[Massey02] Massey, Bart: Where Do Open Source Requirements Come From (And What Should We Do About It)?
Position Paper, 2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.

[Miller02] Miller, Robin: The Open Source way to fame and fortune
Newsforge, July 2002
http://newsforge.com/newsforge/02/06/29/2127239.shtml?tid=3

[Minnihan02] Minnihan, John: Build and Release Management
Freshmeat.net Editorial, February 9th, 2002

[Mitchell99] Mitchell, Russ: How to Manage Geeks
Fast Company issue 25, page 174, June 1999

[Mockus02] Mockus, Audris and Herbsleb, James D.: Why Not Improve Coordination in Distributed Software Development by Stealing Good Ideas from Open Source?
Position Paper, 2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.

[Moglen99] Moglen, Eben: Anarchism Triumphant: Free Software and the Death of Copyright
First Monday, Volume 4, Number 8 (August 1999)
http://firstmonday.dk/issues/issue4_8/moglen/

[Moorman00] Moorman, Jacob: The Importance of Non-Developer Supporters in Free Software
Freshmeat.net Editorial, April 22nd, 2000

[Mundie02] Mundie, Craig: Security: Source Access and the Software Ecosystem
Microsoft Corporation, June 2002

[Nahm02] Nahm, Jae: An open system and its effects on R & D
Department of Economics, Hong Kong University of Science and Technology, Clearwater Bay, Kowloon, Hong Kong, April 2002

[Nakakoji01] Nakakoji, Kumiyo and Yamamoto, Yasushiro: Taxonomy of Open Source Software Development
Position Paper, First Workshop on Open-Source Software Engineering, 23rd International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

[Nishinaka01] Nishinaka, Yoshiyuki: Open Source Software Developments in XP Style
Position Paper, First Workshop on Open-Source Software Engineering, 23rd
International Conference on Software Engineering, Toronto, Canada, May 15, 2001

[OSD02] Open Source Initiative: The Open Source Definition
Open Source Initiative, 2002
http://www.opensource.org/docs/definition_plain.php

[Port01] Port, Dan: Introducing a “Street Fair” Open source Practice Within Project Based Software Engineering Courses
Position Paper, First Workshop on Open-Source Software Engineering, 23rd International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

[Price99] Price, Eric: Production Models Drive Reuse Readiness
Position Paper, Ninth Annual Workshop on Software Reuse. 7-9 January 1999. Austin, Texas, USA

[Rasch01] Rasch, Chris: The Wall Street Performer Protocol: Using Software Completion Bonds To Fund Open Source Software Development
First Monday, volume 6, number 6 (June 2001),
http://firstmonday.org/issues/issue6_6/rasch/index.html

[Raymond99] Raymond, Eric S.: A Response to Nikolai Bezroukov
First Monday, volume 4, number 11 (November 1999),
http://firstmonday.org/issues/issue4_11/raymond/index.html

[Raymond00] Raymond, Eric S.: The Magic Cauldron
August 2000
www.tuxedo.org/~esr/writings/magic-cauldron/

[Raymond99b] Raymond, Eric S.: The Cathedral & the Bazaar - Musings on Linux and Open Source by an Accidental Revolutionary
1999, O’Reilly & Associates Inc., Sebastopol

[Raymond98b] Raymond, Eric S.: Homesteading the Noosphere
First Monday, volume 3, number 10 (October 1998)
http://firstmonday.dk/issues/issue3_10/raymond/index.html

[Ritchie79] Ritchie, Dennis: ’The Evolution of the Unix Time-sharing System’ in ’Lecture Notes in Computer Science’, 1979
http://cm.bell-labs.com/cm/cs/who/dmr/hist.html

[Robbins02] Robbins, Jason E.: Adopting OSS Methods by Adopting OSS Tools
Position Paper, 2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.

[Roberts00] Roberts, Michael: Workflow toolkit: Case study of an open source project
Sponsoring open source development can benefit everyone involved
IBM DeveloperWorks, July 2000

[Rosenberg02] Rosenberg, Donald K.: The Coming Software Revolution
Position Paper, 2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.

[Salus94] Salus, Peter H.: ’A Quarter Century of UNIX’
1994, Addison-Wesley

[Scacchi02] Scacchi, Walt: Is Open Source Software Development Faster, Better, and Cheaper than Software Engineering?
Position Paper, 2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.

[Scacchi02a] Scachi, Walt: Understanding the Social, Technological, and Policy Implications of Open Source Software Development
Position Paper, NSF Workshop on Open Source Software, Arlington VA, January 2002

[Scacchi02b] Scacchi, Walt: Understanding the Requirements for developing open source software
IEEE Proceedings on Software, Volume 149, Number 1, February 2002

[Scacchi01] Scacchi, Walt: Software Development Practices in Open Software Development Communities: A Comparative Case Study
Position Paper, First Workshop on Open-Source Software Engineering, 23rd International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

[Schach02] Schach, Stephen R. and Offutt, A. Jefferson: On the Nonmaintainability of Open-Source Software
Position Paper, 2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.

[Schmidt01] Schmidt, Douglas C. and Porter, Adam: Leveraging Open-Source Communities To Improve the Quality & Performance of Open-Source Software
Position Paper, First Workshop on Open-Source Software Engineering, 23rd
International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

[Schmidt02] Schmidt, Klaus M; and Schnitzer, Monika: Public Subsidies for Open Source? Some Economic Policy Issues of the Software Market
University of Munich, CEPR, June 2002

[Spangler01] Spangler, Arnold: Open Source Development: A suitable Method to introduce a standardized communication protocol?
Position Paper, First Workshop on Open-Source Software Engineering, 23rd
International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

[Spolsky00] Spolsky, Joel: Where do These People Get Their (Unoriginal) Ideas?
Joel on Software, April 2000
http://www.joelonsoftware.com/articles/fog0000000068.html

[Spolsky01] Spolsky, Joel: Hard-assed Bug Fixin'
Joel on Software, July 2001
http://www.joelonsoftware.com/articles/fog0000000014.html

[Spolsky02] Spolsky, Joel: Strategy Letter V
Joel on Software, June 2002
http://www.joelonsoftware.com/articles/StrategyLetterV.html

[Torvalds01] Torvalds, Linus et al: Software Development as directed Evolution
Linux Kernel Mailing List, Dezember 2001

[Vallopillil98] Vallopillil, Vinod: Open Source Software: A (New?) Development Methodology
Microsoft Corporation, August 11th, 1998

[Varian02] Varian, Hal R: System Reliability and Free Riding
University of California, Berkeley, May 2002

[Weber00] Weber, Steven: The Political Economy of Open Source Software
Department of Political Science, University of Berkeley, California, June 2000

[Wegberg00] Wegberg, Marc; and Berends, Peter: Competing communities of users and developers of computer software: competition between open source software and commercial software
NIBOR Working Paper, May 2000

[Welch00] Welch, Rod: Management of Open Source projects is difficult.
Notes, November 1st, 2000

[Werry99] Werry, Chris: Imagined Electronic Community: Representations of Virtual Community in Contemporary Business Discourse
First Monday 1999

[White02] White, Terry and Goldberg, Joel: The Open Source rEvolution - A Pragmatic Approach to Making the Best of It
Position Paper, 2nd Workshop on Open Source Software engineering, May 25, 2002, Orlando, Florida.