GOING OPEN:
GUIDELINES FOR COMMERCIAL ACTORS TO RELEASE SOFTWARE AS OPEN SOURCE

TOR ERIK EIDE
PER KRISTIAN SCHANKE

SUPERVISOR:
CARL-FREDRIK SØRENSEN

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering,
Department of Computer and Information Science
Abstract

This report presents the results of a combined explorative and descriptive study performed in collaboration with Keymind Computing AS in context of the European ITEA project COSI1. 

The purpose of this study is to explore commercial aspects of open source regarding development processes. This includes the use of externally developed open source components as well as the release of proprietary code as open source. We identify architectures and infrastructures of open source portals and open source tools. We also describe open source business models as well as licensing requirements and issues.

Based on our explorative research, we present a framework to explain what measures must be made by a company to successfully release code as open source. This framework should provide organizations with guidelines about the technological infrastructure, what license type(s) to choose, what can be done to attract developers, as well as discussing if opening the source code is indeed the best strategical move.

It has been found that several measures must be taken to achieve a successful open source project. The software architecture must be evaluated and adjusted to make it suitable for open source, and certain non-architectural properties should be present. We have also identified when the different business models and licenses should be chosen, and requirements that must be fulfilled regarding portal infrastructures and open source tools. We have created specific guidelines based on our findings, and discussed the rationale behind our suggestions. These guidelines have been conceptually applied to a specific company and their software product. Future studies can review these guidelines further. They can be used in studies targeting the releasement of open source projects, or in surveys to get further input. Such studies can shed more light on the guidelines’ applicability and help refining the guidelines.

Keywords: Open Source, Open Source Business Models, Open Licenses, Open source Portals

---

1Co-development using inner & Open source in Software Intensive products. Website: http://itea-cosi.org
Preface

This report is a part of the depth study at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The project is performed in context of the COSI project (Co-development using inner and Open source in Software Intensive products), which is an ITEA project.

We would like to thank Audun Jensvoll from Keymind Computing AS for valuable information about the needs they have and for information about their Keywatch system.

We would also like to thank our supervisor Carl-Fredrik Sørensen and NTNU research fellow Øyvind Hauge for valuable input on the process, research methods, and information we have used.

Trondheim, 2006-12-20

Tor Erik Eide
Per Kristian Schanke
# Contents

Abstract iii  
Preface v  
Contents vii  
List of Figures xiii  
List of Tables xv  

## I Prestudy  

### 1 Introduction  

1.1 Motivation ................................................. 3  
1.2 Project Context .............................................. 3  
1.3 Problem Definition ............................................ 3  
  1.3.1 Project Description ...................................... 4  
  1.3.2 Research Goal ............................................. 4  
  1.3.3 Research Questions ....................................... 4  
  1.3.4 Research Method .......................................... 4  
  1.3.5 Own Contribution ......................................... 4  
1.4 Report Outline ............................................. 5  

### 2 Open Source  

2.1 Historical Perspective ........................................ 7  
  2.1.1 The Early Days and the Research Environment .......... 7  
  2.1.2 UNIX, Intellectual Property Rights and the Free Software Foundation 8  
  2.1.3 Internet and the World-Wide-Web .......................... 8
## 2.1 Linux

- 2.1.4 Linux

## 2.2 Apache

- 2.1.5 Apache

## 2.3 Mozilla and The Open Source Initiative

- 2.1.6 Mozilla and The Open Source Initiative

## 2.4 Impacts of Open Source Software

- 2.1.7 Impacts of Open Source Software

## 2.2 Definition

- 2.2 Definition

## 2.3 The Open Source Community - Characteristics, Organizational Structures and Roles

- 2.3.1 The Community

## 2.4 Reasons for Participation

- 2.4 Reasons for Participation

## 2.5 Development Processes

- 2.5 Development Processes

## 2.6 Tools

- 2.6 Tools
    - 2.6.1 Tools for Communication
    - 2.6.2 Tools for Software Configuration Management
    - 2.6.3 Tools for Software Documentation

## 2.7 Portals

- 2.7 Portals
    - 2.7.1 Portals Where All Projects are Allowed
    - 2.7.2 Portals that Restrict the Content

## 2.8 Summary

- 2.8 Summary

## 3 Open Source in a Commercial Setting

### 3.1 Benefits from Introducing Open Source

- 3.1 Benefits from Introducing Open Source
    - 3.1.1 Reuse of Existing Components
    - 3.1.2 Releasing Code as Open Source
    - 3.1.3 End-Use of Open Source Products

### 3.2 Licenses

- 3.2 Licenses
    - 3.2.1 GNU General Public License (GPL v.2)
    - 3.2.2 GNU Lesser General Public License v.2.1(LGPL)
    - 3.2.3 (Revised) Berkeley Software Distribution License(BSD)
    - 3.2.4 Mozilla Public License version 1.1(MPL)
    - 3.2.5 Apache License (Version 2.0)

### 3.3 Business Models

- 3.3 Business Models
    - 3.3.1 Support Sellers
    - 3.3.2 Loss Leader
    - 3.3.3 Widget Frosting
    - 3.3.4 Accessorizing
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.5</td>
<td>Hybrid Model</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>Summary</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>Research Design</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Research Goal</td>
<td>29</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Research Questions</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Research Approach</td>
<td>30</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Explorative Research</td>
<td>30</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Descriptive Research</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>Collection of Data</td>
<td>30</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Literature Survey</td>
<td>30</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Identifying when it is Suitable to Release Source Code under an Open License</td>
<td>31</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Identifying required Deployment and Advertising Processes</td>
<td>31</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Identifying Infrastructural Requirements</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>Limitations to our Approach</td>
<td>31</td>
</tr>
<tr>
<td>4.5</td>
<td>Data Analysis</td>
<td>32</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Meta-Analysis</td>
<td>32</td>
</tr>
<tr>
<td>4.6</td>
<td>Validity Threats</td>
<td>32</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Threats to Conclusion Validity</td>
<td>32</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Threats to External Validity</td>
<td>33</td>
</tr>
<tr>
<td>4.7</td>
<td>Summary</td>
<td>33</td>
</tr>
<tr>
<td>II</td>
<td>Guidelines for Establishing Open Source Projects and Suitable Infrastructures</td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>Making the Decision to go Open Source</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>Identification of a Suitable Software Architecture</td>
<td>37</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Modular Architecture</td>
<td>37</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Adherence to Open Standards and Protocols</td>
<td>38</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Suitability of Software Components</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Generating Income Based on Open Source</td>
<td>39</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Gaining a Strategic Advantage</td>
<td>39</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Choice of Business Models</td>
<td>39</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Choice of Licensing Scheme</td>
<td>41</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Recommendations Regarding Specific Licenses</td>
<td>41</td>
</tr>
</tbody>
</table>
### 5.2.5 Recommendations Based on Choice of Business Model

- Page 43

### 5.2.6 Recommendations Based on the Environment

- Page 43

### 5.3 Summary

- Page 44

### 6 Attracting people to the project

- Page 45

#### 6.1 Necessary non-architectural software properties

- Page 45

#### 6.2 Making the Project Known in the Community

- Page 46

#### 6.3 Relationship Between Company and Community

- Page 47

#### 6.4 Summary

- Page 49

### 7 Infrastructure

- Page 51

#### 7.1 Portals

- Page 51
  - 7.1.1 Public Portal Solutions
  - Page 51
  - 7.1.2 Internal Portal Solutions
  - Page 52

#### 7.2 Usability

- Page 52

#### 7.3 Scalability

- Page 53

#### 7.4 Servers

- Page 53
  - 7.4.1 Company-owned Servers
  - Page 53
  - 7.4.2 External Servers
  - Page 53

#### 7.5 Iterations

- Page 53

#### 7.6 Summary

- Page 55

### 8 Applying the Framework

- Page 57

#### 8.1 Keymind

- Page 57
  - 8.1.1 About
  - Page 57
  - 8.1.2 Reasons for Going Open Source
  - Page 57

#### 8.2 Keywatch

- Page 58
  - 8.2.1 Description
  - Page 58
  - 8.2.2 Architecture
  - Page 58

#### 8.3 Applying the Guidelines to Keywatch

- Page 59
  - 8.3.1 Keywatch Suitability as an Open Source Product
  - Page 59
  - 8.3.2 Recommended Business Model(s)
  - Page 61
  - 8.3.3 Recommended Choice of License
  - Page 61
  - 8.3.4 Recommended Portal
  - Page 62
  - 8.3.5 Recommended Tools
  - Page 62

#### 8.4 Summary

- Page 63
III Evaluation and Conclusions

9 Project Evaluation

9.1 Evaluation of Research Questions .................................................. 67
  9.1.1 Research Question 1 ................................................................. 67
  9.1.2 Research Question 2 ................................................................. 69
  9.1.3 Research Question 3 ................................................................. 70
9.2 Evaluation of Contributions ............................................................ 70
9.3 Evaluation of Validity ..................................................................... 72
  9.3.1 Threats to Conclusion Validity ..................................................... 72
  9.3.2 Threats to External Validity ......................................................... 72
9.4 Summary ..................................................................................... 72

10 Conclusion and Future Work ............................................................ 73
  10.1 Conclusion .................................................................................. 73
  10.2 Future Work .............................................................................. 73

IV Appendix ....................................................................................... 75

A Bibliography ................................................................................... 77

B Glossary .......................................................................................... 83

C Guidelines ...................................................................................... 85
  C.1 Making the Decision ................................................................... 85
  C.2 Choosing the Business Model ...................................................... 86
  C.3 Choosing License ....................................................................... 86
  C.4 Advertising ................................................................................ 86
  C.5 Portal .......................................................................................... 87

D Evaluation of tools .......................................................................... 89
  D.1 Practical matters .......................................................................... 89
    D.1.1 Test personnel ......................................................................... 89
    D.1.2 Hardware ............................................................................... 89
  D.2 Operating Systems ..................................................................... 90
    D.2.1 Debian Linux ......................................................................... 90
  D.3 Concurrent Versioning Systems .................................................... 91
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.3.1 CVS</td>
<td>91</td>
</tr>
<tr>
<td>D.3.2 Subversion</td>
<td>92</td>
</tr>
<tr>
<td>D.3.3 Other</td>
<td>92</td>
</tr>
<tr>
<td>D.3.4 Conclusion</td>
<td>92</td>
</tr>
<tr>
<td>D.4 Documentation</td>
<td>92</td>
</tr>
<tr>
<td>D.4.1 LaTeX</td>
<td>92</td>
</tr>
<tr>
<td>D.4.2 MediaWiki</td>
<td>93</td>
</tr>
<tr>
<td>D.4.3 Code documentation</td>
<td>94</td>
</tr>
<tr>
<td>D.5 Communication</td>
<td>95</td>
</tr>
<tr>
<td>D.5.1 phpBB 2.0.21</td>
<td>95</td>
</tr>
<tr>
<td>D.5.2 Invision Power Board</td>
<td>97</td>
</tr>
<tr>
<td>D.5.3 vBulletin</td>
<td>97</td>
</tr>
<tr>
<td>D.5.4 Forum of choice</td>
<td>98</td>
</tr>
<tr>
<td>D.6 Tracking</td>
<td>98</td>
</tr>
<tr>
<td>D.6.1 Bugzilla 2.22.1</td>
<td>98</td>
</tr>
<tr>
<td>D.7 Portal solutions</td>
<td>100</td>
</tr>
<tr>
<td>D.7.1 SourceForge EE</td>
<td>100</td>
</tr>
<tr>
<td>D.7.2 eZ Publish</td>
<td>101</td>
</tr>
</tbody>
</table>

E Portal overview  103
List of Figures

2.1 Hosts on the Internet ......................................................... 9
2.2 Roles in open source projects [Hauge and Røsdal 2006] ..................... 12
2.3 CVS tools ............................................................................. 18

7.1 Menu items in an open source portal ........................................... 52
7.2 Suggested portal configuration .................................................. 54

8.1 The monitoring value chain ...................................................... 58
8.2 Architectural overview ............................................................ 58

D.1 phpBB Examples .................................................................... 96
D.2 The lifecycle of a bug in Bugzilla [Bugzilla Team 2006] ..................... 98
List of Tables

4.1 Journal databases and search engines .................................................. 31
5.1 Properties of some of the most used open licenses ................................. 42
7.1 Features needed in a portal .................................................................. 51
D.1 Server hardware ................................................................................. 89
D.2 Time requirements to install and configure Debian ............................... 91
D.3 MediaWiki 1.6.8 software requirements ............................................. 94
D.4 MediaWiki 1.8.2 software requirements ............................................. 94
D.5 Time required to install and configure MediaWiki 1.6.8 ...................... 94
D.6 phpBB 2.0.21 software requirements ................................................. 96
D.7 Time required to install and configure phpBB 2.0.21 ........................... 97
D.8 Bugzilla 2.22.1 software requirements .............................................. 99
D.9 Time required to install and configure Bugzilla 2.22.1 ....................... 99
D.10 SourceForge EE software requirements ........................................... 100
D.11 SourceForge EE hardware requirements ......................................... 101
D.12 Time required to install SourceForge EE .......................................... 101
D.13 eZ Publish software requirements ................................................... 102
D.14 Time required to install eZ Publish 3.8 ............................................ 102
E.1 A overview of the tools available at different portals ......................... 103
Part I

Prestudy
Chapter 1

Introduction

1.1 Motivation

Open source software has gained a large momentum in the recent years, with more and more commercial actors adopting open source as an important strategy. Several of these have released software as open source, or are considering such a move. When a commercial company is interested in releasing software as open source, it will always be necessary to assess the risks and costs associated with adopting such a strategy, making sure it will provide the company with a strategic advantage over alternate paths. The objective of this study is to provide a framework for commercial companies that are interested in going open source. This framework should help the companies in making the right choice regarding whether to go open source or not, as well as providing suggestions for how to conduct the release.

We will base our research largely on reviews of the open source literature. Based on the existing literature, as well as our own impressions gathered from communication with open source researchers and actors, we will try to identify the key aspects which lie behind successful use of open source as a development methodology in a commercial company, and provide a best-practice framework. The framework will be in the form of general guidelines that take several key attributes of the company into consideration.

1.2 Project Context

This project is written as a student project at the Norwegian University of Technology (NTNU), Trondheim. The project is an obligatory part of the Masters Degree, and provides 15 ECTS credits, which responds to 50% of the work load in the semester.

The performed in the context of the COSI project, which is an ITEA\(^1\) project aimed at making the techniques and processes of open source known in companies developing software.

1.3 Problem Definition

The purpose of this project is described in terms of a project description, and three research questions that we will answer in order to accommodate the assignment stated in the project description.

\(^1\)http://www.itea.org
1.3.1 Project Description

Describe processes, roles, and tools used in open source environments to establish state-of-the-art processes and necessary tools to establish and run an open source development project.

1.3.2 Research Goal

Our goal of this project is to provide guidelines for how to successfully take advantage of open source, with the main focus on how to release proprietary software as an open source project. We will describe a process that may guide a company in the strategic considerations in relation with the suitability of the software architecture, how to choose a business model and how to choose a suitable license. We will also consider what steps should be taken regarding technical aspects such as the necessary infrastructure and its architecture. Lastly, we will suggest certain measures that can be taken in order to increase the probability of successfully attracting users and developers to the project.

1.3.3 Research Questions

We will address the following research questions in our report:

RQ 1 When should source code be released under an open license?
   - RQ 1.1 What are the most suitable license(s)?
   - RQ 1.2 What are the architectural requirements to the project?
   - RQ 1.3 What are suitable business model(s)?
   - RQ 1.4 What are the necessary properties of the product?

RQ 2 What deployment and advertising processes should be followed to achieve a successful open source project?
   - RQ 2.1 What can be done to attract contributors to a project, i.e., to create a community?
   - RQ 2.2 How can the project be made known in the community?
   - RQ 2.3 How should a company relate to an open source community?

RQ 3 What infrastructural requirements must be addressed when releasing open source code?
   - RQ 3.1 Are there any requirements regarding what tools that must be provided?
   - RQ 3.2 Should an external portal like SourceForge be used, or is it preferable to provide an own solution?

1.3.4 Research Method

Our research method will mainly consist of conducting a literature survey and providing a concept implementation. A more detailed outline of our research approach can be seen in Chapter 4.

1.3.5 Own Contribution

We will in our research look into how to decide whether to release source code as open source or not. We will also point out different problems and what to consider when actually releasing software as open source, and present guidelines for the choices that should be made.
1.4 Report Outline

Part I contains a description of our research and background information about open source and its characteristics. In Chapter 1, we define our research. In Chapter 2, we present the history of the open source movement, the open source definition and information about the community, reasons for participating in open source development and development processes. We also describe the tools commonly used in open source efforts and popular open source portals. Chapter 3 contains information about how open source can be used in a commercial setting. We look into the benefits that can be achieved by adopting open source software, open source licenses and what business models that can be used with open source. Finally, we present our research approach in Chapter 4.

Part II contains our recommendations. In Chapter 5, we present what considerations to make before actually releasing software under an open source license. The importance of having a suitable architecture is discussed. We also come with recommendations regarding the choice of business model(s) and license(s). Chapter 6 follows up with recommendations on how processes that should be followed to attract a community. We describe important properties of the software, how the project can be made known and how a company should interact with the open source community. Chapter 7 discuss and come with recommendations regarding the infrastructures that lie behind a successful open source project. In Chapter 8, we provide information about Keymind and their Keywatch software. Then, we apply the guidelines to their situation and provide recommendations based on these.

Part III contains evaluations of the project and our results. In Chapter 9, we evaluate whether we have answered our research questions. We also evaluate our guidelines and discuss the validity of our results. In Chapter 10, we come with our conclusions and suggestions for further work.

Part IV contains appendices. Appendix A contains the bibliography. Appendix B comes with a glossary of words used in the report. Appendix C sum up the guidelines from Chapters 5, 6, and 7. Appendix D describes the tests of different types of tools used in our recommendations. Appendix E provides an overview of open source portals.
Chapter 2

Open Source

In this chapter, we outline several aspects of open source based on a review of the literature. We begin by describing the open source phenomenon as it has evolved historically, before we go on providing an explanation of the term “Open Source”. Furthermore, we establish several characteristical traits that can be identified in open source projects and give a description of typical tools used in conjunction with open source development efforts. We round off this chapter by providing an overview of some of the most popular open source portals\(^1\), including a brief explanation of what these portals can provide to the community.

2.1 Historical Perspective

2.1.1 The Early Days and the Research Environment

The history of open source software can be traced back to the early days of computing, where much of the software development was performed in academic and corporate research laboratories [von Hippel and von Krogh 2003, Lerner and Tirole 2002]. In these environments, it was considered normal practice to freely distribute the software being developed, a practice that made it possible for others to contribute to the development projects. These projects were therefore often a collaborative effort between different organizations, in which they built upon each other’s research.

The ARPANET\(^2\) was created in 1969, being the world’s first packet-switched network. This network, which is now widely recognized as the forerunner to the Internet [Mowery and Simcoe 2002], linked together several major computer facilities such as universities, research laboratories, and defense contractors [Lindman 2004]. The amount of interconnected institutions quickly increased, and during 1975 it reached more than 100 institutions, making this a natural medium for communication.

Together with the launch of the ARPANET, an informal correspondence process known as Request For Comments (RFC) was introduced. This quickly became the standard way of communicating ideas and for providing comments and refinements to existing proposals within the technical community of the ARPANET [Mowery and Simcoe 2002]\(^3\). Many of the organizations composing this community used incompatible hardware platforms, thereby leading to much focus on establishing platform-independent network technologies that could work across organizational borders.

One of these technologies was the Transmission Control Protocol (TCP). Initially created in 1973, it featured an open architecture, high reliability, and simplified communication over the existing protocol that was in

---

\(^1\)Portals where open source software is made available to the public
\(^2\)Advanced Research Projects Agency Network
\(^3\)The RFCs have many commonalities with open source software, e.g., they evolve as a collaborative effort with peer review
use in ARPANET and provided the means to connect physically distinct networks. TCP was published in the IEEE Transactions on Communication in 1974, and was then further refined in the RFCs the following years. The protocol proved to be an ideal “glue” for integrating networks built on a variety of platforms and protocols [Mowery and Simcoe 2002], and replaced the old communication protocol of the ARPANET in 1983\(^4\). Other influential protocols that evolved in the RFC processes include: the File Transfer Protocol (FTP) [Postel and Reynolds 1985], the Telnet Protocol Specifications (TELNET) [Postel and Reynolds 1983], the Simple Mail Transfer Protocol (STMP) [Postel 1982], and the Post Office Protocol - Version 3 (POP3) [Myers and Rose 1996].

### 2.1.2 UNIX, Intellectual Property Rights and the Free Software Foundation

During the 1970s, much resources were also directed towards the platform-independent operating system; UNIX [Lerner and Tirole 2002]. Unix was initially developed by Dennis Ritchie and Ken Thompson in 1969 at the AT&T’s Bell laboratories, and was distributed freely to other organizations which helped developing the software further [Lerner and Tirole 2002]. One of these organizations was the University of California (Berkeley), which established the Berkeley Software Distribution (BSD) of Unix in 1977. The Berkeley distribution became very popular, and became the operating system of choice in order to link together the ARPANET-nodes. Additionally, an implementation of the TCP/IP protocol was provided along with the Berkeley-distribution. This contributed heavily to the acceptance of the TCP/IP protocol as the communication protocol in the ARPANET [Mowery and Simcoe 2002].

During these early days, there were typically not spent any resources defining property rights or to restrict the reuse and distribution of software. This changed in the 1980s when a trend towards proprietary source code started taking form. AT&T’s Bell laboratories began enforcing their claimed intellectual property rights on Unix, resulting in a court battle with BSD over copyright violation [Feller and Fitzgerald 2002]. At about the same time, MIT\(^5\) licensed some of the code developed by their hackers to a commercial firm, restricting access to the source code even for those who had participated developing it [von Hippel and von Krogh 2003]. As a response to this, Richard Stallman, a programmer based at the MIT laboratories, founded the Free Software Foundation (FSF) in 1985. The FSF was established to “promote computer users’ rights to use, study, copy, modify, and redistribute computer programs” [Free Software Foundation 2006]. To achieve this goal, they created the GPL license\(^6\). This license, being quite revolutionary at that time, gave all users the right to use the software at no cost as well as to freely modify and redistribute the software, in order to preclude commercialization of cooperatively developed software.

### 2.1.3 Internet and the World-Wide-Web

During the 1980s, a large number of improvements to the ARPANET infrastructure were conducted. The Domain Name Server(DNS) was introduced in 1984, mapping domain names to the numerical network address scheme utilized by TCP/IP. Additionally, the network capacity was greatly increased, with the network backbone being upgraded from 56Kbit/s in 1985 to 1.5 Mbit/s in 1988 and 46.1Mbit/s in 1991 [Mowery and Simcoe 2002]. These network speed enhancements, as well as the reduced cost of personal computers during the late 1980s and early 1990s, caused a large scale increase in the use of the ARPANET, as can be seen in Figure 2.1.

---

\(^4\)Technically, the TCP protocol had been split into the Transmission Control Protocol (TCP) and the Internet Protocol (IP), making it by now commonly known as the TCP/IP protocol [Mowery and Simcoe 2002]

\(^5\)Another university that had participated in source-code sharing

\(^6\)GNU General Public License, which is explained in Section 3.2.1
Following the introduction of the World Wide Web in 1991, this diffusion evolved even further, resulting in a widespread increase in the contributions to freely shared development projects, as well as the initiation of many new such projects [Lerner and Tirole 2002].

2.1.4 Linux

Perhaps the best known example of an early open source project actively spread on the internet, is the Linux Operating System. Linux was initiated by Linus Torvalds during his studies at the University of Helsinki. He began working on Linux in 1991, with the objective to create a Unix-like operating system for the IBM PC 386 series [Feller and Fitzgerald 2002]. In this process, he openly sought help with the project, succeeding in attracting a great deal of worldwide support. Hence, the Linux development evolved into a large-scale collaborative effort, characterized by almost completely relying on Internet-based tools [Tuomi 2001]. Torvalds also introduced an entirely new development model in that unstable versions of the project were released early and often [Raymond 2002].

2.1.5 Apache

Another highly successful project adopting distributed development with free sharing of the source code, is the Apache HTTP Server. The development of the Apache project started in February 1995, due to the discontinuation of the development efforts of the already existing NCSA\(^7\) web server. The Apache development was based on a series of patches to the NCSA server, and was performed by a group of volunteers, known as the Apache Group. This group then worked together to coordinate the distribution of the patches. The Apache Group has since 1999 been known as the Apache Software Foundation, a non-profit organization that today is the host of several other high-profile projects in addition to the web server [The Apache Software Foundation 2005].

The HTTP server, as of November 2006, has a market share of 60.32\% [Netcraft LTD 2006], clearly being the most widely used alternative, having a higher adoption than commercial alternatives such as the Microsoft IIS.

\(^7\)National Center for Supercomputing Applications
2.1.6 Mozilla and The Open Source Initiative

In 1998, Eric Raymond released his book, “The Cathedral and the Bazaar” [Raymond 2002], describing the nature of free sharing, and why it was so successful. After reading this text, Netscape as the first commercial actor, decided to release the source code of their proprietary software. This was the source code of their well known Netscape Navigator browser, which was adopted into the new Mozilla browser. The Mozilla browser has been widely adopted, and is today the second most popular web-browser, only preceded by the Microsoft Internet Explorer.

In conjunction with this move, several proponents of the free software movement, including Eric Raymond himself, decided to adopt a more commercialized stance than the ideologies of the Free Software Foundation. According to Bruce Perens, they “realized it was time to dump the confrontational attitude that has been associated with “free software” in the past and wanted to sell the idea strictly on the same pragmatic, business-case grounds that had motivated Netscape” [Open Source Initiative 2006]. Several people went together and coined the term “Open Source”. With this, they hoped to get the corporate world to listen to their claims regarding the superiority of an open development process. The Open Source Initiative (OSI), [Open Source Initiative 1998], was launched the following week.

2.1.7 Impacts of Open Source Software

In the years following the founding of OSI, the impact of open source software has been continually increasing. Several commercial actors have followed in the footsteps of Netscape and decided to release their proprietary software as open source. Additionally, the adoption of open source components into commercial development efforts has been a well-known phenomenon. Open source software is also gaining terrain in the “end-use” arena, both by individual persons and in many organizations. Indeed, open source is today often referred to as a development methodology in itself, that complements, and in some cases even competes with with traditional, commercial development practices [Hecker 2000]

National governments are also beginning to recognize the value of open source software, and are worldwide adopting open source solutions across the public sector. This is much due to the recent focus on open standards8, which are generally accepted as being better supported by open source software than their proprietary alternatives [Schmitz 2001]. Other factors, such as cost savings, independence from proprietary vendors, and improved security can also be mentioned. Additionally, developing economies of which have had problems adhering to intellectual property rights are on many occasions using open source software as a means to reduce this problem [Weber 2003].

Here in Norway, open source software and open standards have been given significant funding in the recent budget proposal [FAD 2006], showing that the open source phenomenon has gained widespread acceptance here as well, causing an increased emphasis on technological openness in the Norwegian public sector.

2.2 Definition

The concept of open source is interpreted in different meanings depending on the context in which it is considered [Wang and Wang 2001, Gacek and Arief 2004]. Some use the term about all software where the source code is publicly available. However, adhering to the definition provided by the Open Source Initiative [Perens 2006], software to be considered open source, must be released under a license that conforms to certain criteria9:

---

8Open standards are publicly available and implementable specifications, not locked to any proprietary vendor
9These are not directly quoted from the Open Source Definition, but have been rewritten somewhat in order to be more easily read. For the original, we refer the OSI homepage [Perens 2006]
• The license shall not restrict any party from selling or giving away the software as a component containing programs from several different sources, and it shall not require any royalty or other fee for such a sale.

• The program must allow distribution of the source code as well as in compiled form, and the source code can not be deliberately obfuscated.

• 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.

• Author integrity of the source code must be maintained, and the license may require derived works to carry a different name or version number from the original software.

• The license cannot discriminate against any person or group of persons, nor restrict the use of the program in any particular field of endeavor.

• The rights attached to the program must apply to all to whom the program is redistributed without any additional license imposed by those parties.

• The license cannot be specific to any product, causing the rights attached to the program to depend on the program’s being part of a particular software distribution.

• The license cannot place restrictions on other software distributed alongside the licensed software.

• The license must be technology-neutral.

2.3 The Open Source Community - Characteristics, Organizational Structures and Roles

Even though open source projects have a less formal organizational structure than what is typical in inhouse development projects, certain organizational traits within the open source communities can be identified. We will here provide some information gathered about open source communities as well as typical roles and governing structures within these projects.

2.3.1 The Community

Much of the open source literature focus on open source development being performed in a highly collaborative environment, in which the developers are distributed across many different locations around the world [Mork 2005]. Although this highly collaborative environment can be identified in the most successful open source projects, such as Linux and Apache, several empirical studies suggest that this notion is not typically the case.

Based on a review of projects in open source portals such as SourceForge, it has been shown that the majority of open source software only have one or two developers [Capiluppi et al. 2003, Krishnamurthy 2002], being a counterexample to the claim about collaborativeness. Furthermore, these same studies found that the number of active participants within open source communities also was generally low. Only 1% of the projects at SourceForge had more than 100 subscribers and only 10% could claim more than 20%.

Still, the open source community comprises a wide variety of participants. Development is performed both by computer science students and professional developers as well as others with technical knowledge. According
to a visitor survey performed by the Open Source Technology Group (OSTG)\textsuperscript{11} in March 2006, 95% of their visitors are men, with the average age of 32. About half of their visitors have a college degree or higher [Open Source Technology Group 2006]. Similar data is found in [Hars and Ou 2002], a study that also discovered that 58% of the total working hours were performed by professional developers, while the rest were students (14%) and hobbyists (28%). This can be related to a similar survey in which approximately 80% of the open source developers either were professionals or were studying in a related field [Ghosh et al. 2002]. The majority of developers were also found participating in more than one project [Hars and Ou 2002], although the time spent on open source development was found to be generally low [Ghosh et al. 2002]. This study also found that the great majority of open source developers were from Western Europe or North America. When it comes to those that are not actively participating in the development efforts, due to, e.g., lack of technical knowledge, these users can in many cases provide important input to the project, such as bug reports, feature requests and reporting usability concerns.

2.3.2 Organizational Structure of Projects and Roles within the Community

Due to the voluntary nature of participation, there are no formal programs for controlling an open source project, and no authoritative leaders that monitor the development such as can be found in traditional development organizations [Yamauchi et al. 2000]. However, the organizational structure is still quite rigid, typically being organized in a hierarchical fashion. This hierarchy is often referred to as a meritocracy, in which status is gained according to the contributions participants make, and their merits within the community [Bonaccorsi and Rossi 2003].

![Diagram of roles in open source projects](image)

Figure 2.2: Roles in open source projects [Hauge and Rosdal 2006].

Figure 2.2 gives an overview of this meritocracy and the distinguishing roles in an open source project. Anyone can join an open source project, and these are often referred to as “joiners” or “newcomers” [Bonaccorsi and Rossi 2003]. The passive users are those that use the software, but do not participate actively by providing anything back to the community. People who are new to the project but are participating actively are deemed of higher value. Such participation can consist of reporting bugs, providing fixes for bugs or even providing source code that implements new features. These contributions are then reviewed by the developers and added to the code base if found worthy. By providing several high quality contributions, one can rise further and gain a developer status, which can include the rights to add source code to the repository and accepting contributions from others. The core developers are typically a small group, consisting of the project initiator(s) and others that have proven invaluable to the success of the project. The meritocratic structuring means that the hierarchy is not strictly assigned from the beginning, but is evolving as the time goes by. However, the initiator(s) will generally stay in the top of this hierarchy as long as they are remaining active in the project.

\textsuperscript{11}Owner of SourceForge.net, Slashdot.org,freshmeat.net and more
Even though the hierarchical authority often relates to actual decision making power, it is mostly related to how much people listen to your opinions [Bonaccorsi and Rossi 2003]. It should also be noted that these authorities only are valid as long as the rest of the community agrees on their legitimacy [Scacchi 2004]. If, e.g., the leadership is not accepted by the other participants, they may abandon the project or bring the code base along in order to create a new project with a new leadership\(^{12}\).

### 2.4 Reasons for Participation

Economic theory tells us that a programmer participates in a project only if a net benefit can be derived from engaging in the activity [Lerner and Tirole 2002]. While the traditional aspect of economic compensation explains proprietary development, other motivational factors must be considered regarding the open source phenomenon. It should be noted, that also open source development increasingly is being paid for by commercial actors [Fitzgerald 2006], but what about all those that contribute for free?

There are several motivational aspects that can be identified in the literature [Krishnamurthy 2002, Hars and Ou 2002, Lakhani and Wolf 2003]. An open source developer can typically relate to several or all of these points:

**Participation in an Intellectually Stimulating Activity**

Quite a few people find coding itself fascinating and want to participate in open source projects to challenge their intellect and come up with creative solutions to difficult problems. With this motivation, programming is in itself a hobby, and participation in open source projects are a way of enacting this hobby. Indeed, it was mentioned as the top reason for joining an open source project by [Lakhani and Wolf 2003].

**Learning**

Some open source participants join an open source project to achieve knowledge, either related to programming skills or how the open source community works, being mentioned as high as 70-80% in two open source surveys [Ghosh et al. 2002, Hars and Ou 2002]. As we have already seen, many of the participants are students that use open source as a compliment to their studies, or directly related to school work. As the code quality found in open source projects is generally high [Dinkelacker et al. 2002], this can help identifying good coding practice and style. Still, the mentioned studies comprised many professional developers describing the learning aspect as important, thereby making it a universal motivational factor.

**Future Monetary Rewards**

It has been shown that the Core Developers in high-profile projects receive higher salaries in their daytime job than other programmers. When someone participates in open source projects and can provide good contributions, they both prove their skills to future managers and gain specific knowledge of these open source projects that outsiders do not possess. This causes a self-marketing effect; improvement of job opportunities was mentioned as an important motivator by 24% of the respondents in one survey [Ghosh et al. 2002]. In addition to proving ability by open source participation, it is also a possibility of being recruited by a commercial actor if they perceive someone as influential in a project in which they take interest. Future earnings can also be perceived by providing related products and services to the project that one participates in [Hars and Ou 2002], or by the networks established by such participation.

**Belonging to a Community**

The feeling of belonging to a community has been mentioned as a motivational driver [Hars and Ou 2002, Lakhani and Wolf 2003]. Some find the participation in an unknown form of cooperation interesting [Ghosh et al. 2002], and thereby join open source communities based on, e.g., social gratification. When belonging to an open source community, the building and further enhancement of social and commercial networks are

---

\(^{12}\)Such a move is usually referred to as “Forking”, and can also be due to other reasons such as disagreement related to how to project should evolve.
quite common. Additionally, by being a good developer in an open source community, one might gain a high reputation and a feeling of “power”. The rapid, constrictive feedback of contributions that is common in such communities, also work in a self-reinforcing way [Hars and Ou 2002], that among other things can increase the perceived self-worth and self esteem.

**Need of the Software**
This has often been described by Eric Raymond as “scratching an itch” [Raymond 2002], and relates to solving a problem that there are not any valid solutions for yet. This can be a driving force for initiating an entirely new open source project, as well as for joining a project that the person finds having the potential for fulfilling his need with some adjustments. This can also take form in providing a fix for an annoying bug that has been discovered, or personal customization of the software. These needs can either be in relation with software for personal use or work-related functionality. Mostly, it is professional developers that mentions this a reason to join an open source project [Hars and Ou 2002].

**Altruism**
Some people have such strong beliefs in the free sharing of source code that they participate in projects based on altruistic reasoning. By increasing the welfare of others, one can increase self-esteem and feel more fulfilled as a person, thereby relating this to charity work performed in other arenas. Even though the whole motivation can be “giving to others”, there might also be underlying factors such as gaining similar services back from the community. This can be related to the gift economy, and is discussed further in [Bergquist and Ljungberg 2001]. In [Ghosh et al. 2002] 30% of the respondents mentioned opposition to proprietary software as a motivational factor, a view that can drive them into contributing to open source development by somewhat altruistic reasons.

### 2.5 Development Processes

In this section, we will try to explain the typical development processes in open source projects and relate them to closed-source practices.

**Conceptualization**
open source projects typically start with the unfolding of an idea, being either in the mind of a single person or as a group effort. This will often relate to fixing a perceived problem, such as an identified lack of some software that the person(s) would like to have. The project initiator(s) will generally become the owner of the project, taking on responsibility of managing and organizing the project into (hopefully) a successful one [von Hippel and von Krogh 2003].

This can be contrasted to commercial actors, which typically receive a specific request from external stakeholders, and then act accordingly to meet the needs of the customers.

**Requirement Specification, Planning, Analysis and Design**
As the conceptual idea is invented by the single person or group, there is typically no formal phase of requirement specification in open source projects. The requirements are taken as generally understood, due to the fact the developers are also the users [Fitzgerald 2006]. Such a lack of documentation can provide a significant entry-barrier to newcomers, making it harder to gain an overall understanding of the underlying architecture [von Krogh et al. 2003].

However, this release of internally developed source code under open licenses, leads to differences from the natural design that occurs in projects evolved within the community. First of all, the commercial product will typically have evolved within the company by traditional development processes. This means that their initial requirements and the software architecture might be provided along with the source code. In these cases, the existing architecture and code base must be taken into account. Although the documentation provided might make it easier to contribute to the project, the existing code base might not adhere to the
community’s wishes. As exemplified by the Mozilla project, the existing architecture and code base from the Netscape Navigator did not suffice, requiring an entire rewrite of certain modules after being released as open source [Reis and de Mattos Fortes 2002]. Commercial organizations will also have an interest in leading the projects as they see fit, i.e., with their choice of licensing schemes. Still, it can be argued that the involvement of commercial actors in open source projects leads to more user friendly product packages, with an increased emphasis on the planning, analysis, and design phases in the projects [Fitzgerald 2006].

In the case of proprietary development, requirement elicitation from the customers can be an involved process. The customers typically provide an initial problem, and the software company must then analyze this, provide a design solution and then return to the customers with a proposal. This process will often consist of many cycles, before the requirements have been elicited properly. There are also in certain cases strict formalities that must be followed, which can have a lack of popularity among the developers.

Writing Code and Submitting it to the Community for Review
When a person has written a contribution they believe is of interest to the community, they will submit it to the community for review. The review process is important to avoid having faulty code integrated into the code repository. This can vary from the somewhat informal posting of the code on a mailing list, and adding it to repository if no one objects within a certain amount of time, to an involved process requiring authorities to accept the code, as is the case in the Mozilla project [Reis and de Mattos Fortes 2002]. Often, the project leaders will request certain features, and will then receive several submits sent for review. The best solution is then chosen, typically based on a voting system. The actual committing of source code to the central repository will in most cases be done by only a few trusted developers, who serve as “gate keepers” for the code submitted as contributions [von Krogh et al. 2003]

Implementation
When the initiator(s) have coded a prototype based on their idea, the project is put forth on the Internet to attract interest from external users. If successful, people begin to download and use the program. Hopefully, some of them will begin to make modifications to the code or provide additional features, giving it back to the community. The implementation phase in open source projects is thereby highly iterative, with new modifications and new features implemented at a fast pace [von Hippel and von Krogh 2003]. Due to the nature of open source projects and the large extent to which one choose what work to perform, these contributions are typically in accordance with ones own expertise and interests. This can be opposed to the specific task-assignment more commonly enacted in commercial environments. With voluntary participation and choice of tasks, the open source model probably causes an increased motivation over being assigned a task in the job environment.

Pre-Commit Test
It is important to ensure that a contribution is tested properly to not break the build. Projects can include many testing activities, including ad-hoc volunteer testing, smoke tests, and even contributed functional tests [Reis and de Mattos Fortes 2002]. For mature projects, an involved smoke testing procedure is important to ensure stability for all supported platforms.

Development Release
Typically, the lifecycle consist of rapid implementation with frequent releases. These are the so-called development releases. Several projects conduct a nightly build in which accepted contributions are included. The large number of potential debuggers on different platforms and system configurations help in discovering bugs/defects quickly in these releases. This can be related to the so-called Linus’ Law: “Given enough eyes, all bugs are shallow” [Raymond 2002]. Having frequent development releases which includes code that have recently been submitted, can also provide a significant motivation for the developers [Fitzgerald 2006].

Production release
Many open source projects do not, as opposed to commercial projects, follow a strict deadline in which the product must be finished. This causes the development to last for as long as the project authorities seem
necessary, before the release is conducted. The production release is performed when a thoroughly tested, stable version of the product which include the required functionality is finished.

2.6 Tools

Development tools such as having an Integrated Development Environment (IDE), are of essential importance when developing software. However, when working with an open source project, some additional tools must be included due to the distributed setting in which these projects are conducted. We will in this section look at such tools, focusing on those used in relation with communication, configuration management, and documentation.

2.6.1 Tools for Communication

Asynchronous Communication

Asynchronous communication encompasses all forms of communication that do not require a recipient to be available when a message is sent. Commonly used asynchronous tools in open source projects include:

- E-mail/Mailings-lists
- Forums

These provide inherent logging capabilities, and are suitable for presenting information to a large group of people. In an open source project, the community is distributed across the world. Thus, asynchronous communication arguably can be said to be the most viable form of information exchange in these environments. Different time-zones or working-time will not pose any problem when asynchronous tools are being used. Additionally, the information will continue to stay available after it has been read, making it a suitable source of information for future usage. Typically, all important messages will be posted to the mailing list. Broadcasting all communications makes the development work transparent, providing awareness in the community of what others are doing [Yamauchi et al. 2000].

Synchronous Communication

With synchronous communication, we consider all forms of communication where two or more parties are available at the same time. Tools that support this form of communication include:

- Telephone
- Instant Messenger
- Internet Relay Chat (IRC)

In an open source setting, usage of such tools are primarily viable when a decision must be made quickly. It might also provide a more informal way of socializing within the community, i.e., through project-related IRC channels. Since these tools require the participants to be available at the same time, they are not suitable as the only channel for information exchange in open source projects\(^{13}\). They should therefore only be considered as a supplement to the asynchronous communication; indeed, several high-profile projects rely solely on mailing-lists and forums.

\(^{13}\)With tools such as IRC and instant messenger, the recipient can retrieve information in a somewhat asynchronous fashion if he is away (Commonly referred to as being AFK, away from keyboard)
2.6.2 Tools for Software Configuration Management

Software Configuration Management (SCM) is concerned with controlling changes to software products [Conradi and Westfechtel 1998], and is an important area of concern in open source projects. Several tools exist to improve the configuration management processes, some of which will be presented in this section.

Revision Control

Revision control systems\textsuperscript{14} are applications designed to synchronize work and to keep track of changes in the source code done by developers working on the same files [von Krogh et al. 2003].

Files, changes, and user comments are stored in a central repository, which is kept separate from users’ local versions. People can download the contents of the repository, making it possible for developers to work on the same set of files. These can then be added to the central repository when changed, and in most cases these systems will merge conflicting changes, i.e., when two users have revised the same document.

In open source projects, there are generally only a few trusted developers that have write-access to this repository\textsuperscript{15}, however everyone are allowed to view and download the repository, making it possible for them to contribute even though they can not make changes to the repository.

There are several different revision control systems in existence. Some of these are commercial and not typically used in open source projects, while others such as SVN and CVS are used by the absolute majority of developers. Other systems, such as GNU Arch are also used in OS development\textsuperscript{16}.

Concurrent Versions System (CVS)

CVS was first developed in 1986, based on the already existing Revision Control System (RCS). CVS is able to keep track of large projects [Berliner 1990], as opposed to RCS which only operates on single files, and is based on a client-server architecture, where the server stores the project and related information\textsuperscript{17}. Clients have to connect to the server to make changes, or download the most recent copy of a project to his local machine. This can be done either with a command-based interface (Figure 2.3a), or with the aid of a graphical client, e.g., TortoiseCVS\textsuperscript{18} (Figure 2.3b).

Several users can work on the same project concurrently, since their changes are made to their own local versions. If someone then tries to upload a modified file, it will only be accepted by CVS if it is a modification to the most recent version existing in the repository. This is a security mechanism implemented to avoid version conflicts, with each new upload the version will be incremented.

The history of a project will be stored by the server. This can be taken advantage of in several different ways, e.g., to compare the changes between different versions, or to change back into an older version.

Subversion (SVN)

SVN is a project that started in 2000, intended as a replacement for CVS. Subversion generally provide the same features and interfaces as CVS, but fixes some of the problems identified in its forerunner. As opposed to CVS, Subversion provide versioning of directories, renaming, and metadata. It also makes sure that commits are atomic, i.e., no part of the commit takes places until the entire commit has succeeded. Several other changes have also been implemented. For a complete listing, we refer to the Subversion homepage\textsuperscript{19}.

Subversion has gained a large momentum these last years, being adopted by several notable projects, e.g., Apache.org ([The Apache Software Foundation 2006]), Google Code ([Google 2006]), and Tigris.org
Bug Tracking Systems

When developing software, it is of vital importance to keep track of the problems that emerge. Bug Tracking Systems are applications that aid in this process by keeping track of bugs or other issues as they are discovered. These systems contain a database where information about reported bugs is stored and tracked, which can include:

- The time of discovery
- Who reported the bug
- How the bug manifests itself
- How to reproduce the bug
- Severity of the bug

There exist several such systems, both as proprietary solutions and as open source.

Bugzilla

Bugzilla is probably the most widely used bugtracking system in open source projects\textsuperscript{20}. This software is used in several mature projects, e.g., the Mozilla Browser, The Apache Project, The Linux Kernel, Eclipse, OpenOffice, Gnome, and KDE [Mozilla.org 2006b].

It includes several configuration management features in addition to keeping track of bugs. It can also be used to track code changes in general, to submit and review patches, as a communication tool and as an aid in managing quality assurance [The Mozilla Organization 2006].

\textsuperscript{20}Developed by the Mozilla Foundation for use with their software; \url{http://www.bugzilla.org/} (Accessed 2006-11-10)
2.6.3 Tools for Software Documentation

Even though software documentation can be considered to span from architectural documents to how-to documents, we will here only consider the tools commonly used in open source projects.

End-User Documentation consists of all documents intended for helping the end-users with the use of a program. They can exist in many variants, and usually several variants are provided simultaneously. Typical user-documentation includes help-files, how-tos, frequently asked questions, tutorials and troubleshooting.

Documentation Generators

Documentation generators are commonly used as an aid to automatically extract documentation from the source code. This documentation can include, e.g., descriptions of the APIs, algorithms and datastructures, which is formatted into an output such as HTML, XML, or PDF to provide easily readable documentation. There are several tools that can aid in this process, with Javadoc probably being the most well known.

**Javadoc**

Javadoc is a tool for Java used to generate API documentation in HTML format. It is freely supplied along with the Java SDK, and is widely used in both open and proprietary development efforts. Javadoc works by automatically extracting information about package structures, classes, method signatures and more [Leslie 2002]. Its benefits can be greatly improved when the developers include some explanatory comments about the properties that are to be extracted.

**Wikis**

Wikis have lately emerged as a new commonly used tool for writing documentation. It is implemented as a website-component, a CGI script or similar technology [Aronsson 2002], and provide an easy way to write documentation as a collaborative effort. This relates both to technical documentation as well as documentation for end-users. Visitors can access the information in the wiki as a normal HTML page, and can then view or modify the existing content as well as adding new information or deleting what already exists\(^\text{21}\).

2.7 Portals

There are many different sites and portals related to open source projects. We will in this section describe the following:

- [http://www.sourceforge.net](http://www.sourceforge.net)
- [http://www.freshmeat.net](http://www.freshmeat.net)
- [http://www.mozilla.org](http://www.mozilla.org)
- [http://www.apache.org](http://www.apache.org)

All these sites serve as an external interface for multiple projects, providing several tools and services that are of importance when publishing a project. We have distinguished these portals into two categories; those in which all open source projects are allowed, and those that restrict projects according to certain criteria.

\(^{21}\)Depending on the access rights, some sites such as Wikipedia are free for everyone to modify, while others might require registration or exist only within an internal network.
2.7.1 Portals Where All Projects are Allowed

The two most popular open source portals that do not restrict their services to any particular projects, are SourceForge\(^{22}\) and Freshmeat\(^{23}\). Due to their accessibility, these portals contain a large amount of projects, many of which are inactive.

**SourceForge**

SourceForge.net is today the largest open source portal, hosting a total of 137,080 projects (as of 2006-12-18). It is owned by the Open Source Technology Group, and provides a wide range of tools and services. [SourceForge.net 2006](http://sourceforge.net) support forums, mailing lists, version control systems (CVS and SVN), and a bugs- and feature requests tracker.

**Freshmeat**

Freshmeat.net is probably the largest index online, and is also owned by the Open Source Technology Group. It does not host any projects directly, but rather acts as an information-repository. The Freshmeat index contains 41,977 projects (as of 2006-12-18).

Freshmeat maintains rankings of projects according to a multitude of parameters. In this way, good and/or popular projects can attract publicity from the open source community, both from users and developers.

2.7.2 Portals that Restrict the Content

As a contrast to the portals mentioned above, there are also several portals that will only allow certain projects. To be accepted into a portal like this, you have to oblige to some kind of rules set up by the community, and the project must be deemed worthy. This is often a matter of a vote; either within a managing board, or throughout the community.

Limited inclusion portals have the advantage that accepted projects will usually have better guarantees for being maintained as the ownership is usually transferred to the main-project. The board of the portal may then take the wheel of a project when it is adrift.

**Mozilla.org**

Mozilla was set up by Netscape [Mozilla.org 2006a](http://mozilla.org) in 1998 when they released the source code of the Netscape browser as open source. Together with the source code, they also supplied an open source license, namely the Mozilla Public License [Mozilla.org 2006c](http://mozilla.org). This license is used with all projects hosted by Mozilla.

The Mozilla Firefox browser is today the second most widely used web-browser, only preceded by the Microsoft Internet Explorer [Net Applications 2006]\(^{24}\). Additionally, Mozilla also develops the e-mail client Thunderbird, the tracker tool Bugzilla, a calendar application, and a huge amount of other projects more or less associated with the main objectives of Mozilla; developing tools for keeping the web open.

\(^{22}\)http://sourceforge.net (Accessed 2006-10-05)

\(^{23}\)http://freshmeat.net (Accessed 2006-10-05)

\(^{24}\)The average person tends to use the Internet Explorer as it is provided alongside the Microsoft Windows Operating Systems...
Apache

Apache hosts a large amount of projects. It originally started with the Apache Web Server, but has since evolved into a large amount of high-profile projects. They work in a slightly different way than the other large repositories. New projects go through an incubation period where the Apache community decides whether it should be an Apache project or not. The tools available depend on the project, but tools like Bugzilla, CVS, and SVN are common.

To avoid lawsuit for breaches of copyright and patents, every developer must agree to give Apache unlimited licensing rights to use the material provided to the project. If the developer does not have the ability to give these rights, this becomes a problem between the developer and the copyright/patent holder; not the Apache Foundation.

2.8 Summary

This chapter has given an overview of the history of open source and how it as evolved to what it is today. We have provided a definition of open source, and described the characteristics of organizational structure and development processes. We have also explained the reasons people participate in open source projects, tools commonly used and the major open source portals. The information presented here have provided information about open source, its processes and infrastructures and provides important background information for creating the guidelines.
Chapter 3

Open Source in a Commercial Setting

3.1 Benefits from Introducing Open Source

Several potential benefits can be associated with using open source, and it has been reported that adoption of open source software has resulted in significant savings in some organizations [Fitzgerald 2004]. Possible beneficial areas exist both regarding the reuse of existing components in internal development as well as when releasing code as open source. Additionally, the use of open source software as a fixed software solution can prove feasible on many occasions. However, there are certain risks associated with open source software, some of which will be discussed in this section.

3.1.1 Reuse of Existing Components

Most open source software is obtained from external sources [dal Molin 2003], and the reuse of existing components is probably the most common way to take advantage of open source in software development\(^1\). With this approach, most of the work is already performed by others, hence resulting in less internal development efforts. This can especially relate to less time spent on common programming tasks and easier adoption of new technologies. The usage of external workforces also makes an effective use of specialists and has been said to reduce the process risk as well as providing increased reliability [Adams et al. 2005].

Even though the use of external components can provide reduced costs and time to market while at the same time providing products of increased quality, it must be emphasized that this approach also brings along certain problems and risks. Much efforts can be spent trying to find suitable components where none exists, and even when suitable components are identified, correctly choosing one component over another can prove a difficult task. The use of off-the-shelf components will also in many cases require company-specific adaptations, possibly causing increased maintenance costs. One might in these cases consider whether to freeze the component as-is. Such a strategy will however make the company unable to take advantage of new releases. By staying updated with each new version, the maintenance work might in some cases prove a heavy burden. The lack of documentation, testing and field support in many open source projects [Li et al. 2005], can also cause problems in initially getting to know the source code, thereby making the learning curve steep. This will in turn cause additional indirect costs. Because of the problems mentioned, one should not consider open source as free software, as it certainly can require substantial investment before it can be deployed into the marketplace [Ruffin and Ebert 2004]. Indeed, in some cases it can be more worthwhile to buy a component over choosing a similar open solution, due to reduced total cost of ownership.

\(^1\)These components are often referred to as Off-The-Shelf(OTS) components
3.1.2 Releasing Code as Open Source

These days, an increasing number of commercial organizations decide to release their proprietary code as open source. Even though this initially might seem paradoxical, going against common economical traditions, there are certainly several benefits related to such a move.

One obvious benefit is the possibility of contributions from external developers. If the community takes interest into the project, a plethora of skilled developers might provide detailed bug fixes, security fixes, and even new features for the project. The peer-review will in this way help making the product better, more reliable, and more secure [Wheeler 2005, Ruffin and Ebert 2004]. In addition to direct contributions to the program, development of plug-ins and add-ons might take form, which in turn increases the total value of the product. These can also be provided from the company using normal license sales (as long as licenses are adhered).

When the product is given away free of charge, it may also gain a highly increased user-base, possibly increasing the company’s popularity. The company may thereby derive additional benefits related to their other products and services, as well as reducing the market share of competitors. An increased user-base will also cause the program to be tried out on a higher amount of platforms and hardware configurations, revealing compatibility issues and other issues that must be addressed.

Negative aspects with such a move mainly relates to the lack of licensing-revenues from the customers, although this can be taken somewhat into account by providing hybrid licensing schemes. In addition, there will never be any guarantee that a large mass of people will take interest in the project. If this does not happen, the anticipated benefits cannot be derived, while at the same time the code is left open for everyone.

3.1.3 End-Use of Open Source Products

Benefits regarding the end-use of open source products, mainly relate to increased savings, i.e., not having to pay licensing fees. Much open source software provide good implementations, making it unnecessary to spend money on commercial software. Indeed, open sourced software has in many cases proven equal or, arguably, exceeding commercial alternatives considering quality and reliability, as well as security, [Ruffin and Ebert 2004], making such software the natural choice. Open source is also considered to provide better interoperability due to its association with open standards. There are no vendors that try to protect their market-share by locking the software to proprietary standards.

The main arguments against using open source from an end-users perspective relate to the low usability and lack of support that are often found in open source software [Nichols and Twidale 2003, Lerner and Tirole 2002]. Still, some of these issues seem to be given more considerations with open source software entering the commercial mainstream [Joseph Feller 2000], possibly reducing the validity of such an argument. Additionally, considering the widespread use of commercial software such as Microsoft Word and Microsoft Excel, exchanging documents between the proprietary and open source solutions can be troublesome due to the adherence to a proprietary format in the former solution. In exceptional circumstances, the usage of open source tools in development activities may also cause the resulting artifacts to be considered derivative work, thereby giving the copyright holder certain rights [Ruffin and Ebert 2004].

3.2 Licenses

There exists a large amount of different open source licenses. Most of these are based on the earliest ones, such as the GNU General Public License, the GNU Lesser General Public License and the Berkeley Software Distribution License(s). We will in this section provide a brief overview of these three licenses, which are especially for the non-technical user
also the most commonly used ones\textsuperscript{3} [Freshmeat.net 2005, dal Molin 2003, Lerner and Tirole 2005b]. We will also describe the Apache License and the Mozilla License, due to the influential position these projects have in the community. All the licenses we describe here are in adherence with the Open Source Definition, fulfilling the criteria stated in [Perens 2006].

3.2.1 GNU General Public License (GPL v.2)

The GPL license was introduced by the Free Software Foundation, and is today the most commonly used open source license. The General Public License was first written by Richard Stallman in 1989, but was superseded by the GPL v.2 two years later.\textsuperscript{4}

The license is in accordance with the FSF philosophy, intending to “guarantee the freedom to share and change software” [Free Software Foundation 1991], by providing certain rights:

- The right to freely distribute copies, both as-is and in modified versions
- The right to modify the source-code
- The right to run the program for any intended purpose

However, the General Public License is probably more known for being highly restrictive. In order to conform with the ideology of the FSF, it imposes that all derivative works of software licensed under GPL, also will be effected by this same license\textsuperscript{5}. In the case of the GPL, this means that all derivative works must be distributed in source code along with the binaries, efficiently making it impossible to incorporating a GPL-program into proprietary software for closed source commercial distribution. This is commonly known as the copyleft property or the “viral effect” that transfers the rights over to the community. However, the license does allow anyone to charge a price for this distribution (although in practice such fees are generally very low), as well as providing warranty protection in exchange for a fee. There is no warranty for programs licensed under the GPL, and neither the copyright holder(s) nor other parties that distribute the program can be held responsible for any liabilities, unless otherwise stated in writing.

3.2.2 GNU Lesser General Public License v.2.1(LGPL)

The LGPL was introduced alongside the GPL v.2\textsuperscript{6} and although being less restrictive than the GPL, it generally impose the requirements. It differs from the GPL in that it permits the linking of libraries into non-free programs [Stallman 1999]:

A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a “work that uses the Library”. Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

The Free Software Foundation has stated that the reasoning for the introduction of this less restrictive version of the GPL was to encourage a larger user base for certain of their GNU libraries, e.g., in order for the library to gain status as a de-facto standard. However, if someone makes changes directly to LGPL-licensed libraries, the license will come into effect. This includes that:

\textsuperscript{3} There are a large amount of licenses somewhat similar to the Berkeley License. These are often referred to as BSD-type licenses. In this report, we will only look at the revised version of the BSD.
\textsuperscript{4} At the time of this writing, a third version of the GPL is under work.
\textsuperscript{5} It is allowed to create derivative works for personal or organization-internal use, meaning that this clause only relates to distribution.
\textsuperscript{6} It was initially released as the “Library General Public License”, later changing the name to “Lesser General Public License”.

25
• The modified work must itself be a software library.
• The files modified must carry prominent notices stating that the files have been changed and the date of the change.
• The whole work must be licensed at no charge to all third parties under the terms of the license.

3.2.3 (Revised) Berkeley Software Distribution License (BSD)

The BSD license was written at the University of California at Berkeley for use with their early Unix Distributions. This original version of the license included an advertising requirement, in which an acknowledgement to the original source was required\(^7\). The requirement became unpopular by many, and was, e.g., deemed incompatible with the GPL by the Free Software Foundation. After receiving a letter from Richard Stallman, it was officially rescinded in 1999 [University of California 1999], leading to a rewrite of the license to what is now known as the Revised BSD\(^8\).\(^9\).\(^10\)

The BSD License is much less restrictive than the GNU Licenses. It grants the right to use of source code and binaries in every possible way. This includes the right to create a proprietary product for commercial distribution where the source code is not available\(^11\). However, the license imposes a requirement that a predefined template must be included with all redistributions. This includes a disclaimer, denying all liability for copyright holds and contributors as well as a copyright notice that cannot be used to “endorse or promote products […] without prior written permission” [University of California 1999].

3.2.4 Mozilla Public License version 1.1 (MPL)

The Mozilla Public License (MPL) was created by Netscape as a part of the release process of the source code for their Netscape Communicator browser. This license explicitly permits MPL-licensed code to be combined with separate proprietary source code to create a proprietary work for further sales. Still, if any modifications are made to the code itself, these modifications will be affected by the license. It can be said that MPL in this regard is somewhat influenced by GPL, meaning that to avoid source code to be affected by the license, it must be kept in separate files from the MPL code\(^12\). If these demands are not fulfilled, the source code must be made freely and publicly available, just as is the case with the GNU Licenses. This license is mostly used with the Mozilla software, although it has been adopted in a few other projects as well [Freshmeat.net 2005].

3.2.5 Apache License (Version 2.0)

The Apache License, although not commonly used [dal Molin 2003], is quite well known due to its usage in the Apache applications. A licensee of Apache licensed software can copy, modify and distribute the software in source and/or binary form. Thereby, the license permits the licensed code to be included in other projects, including those that are of commercial nature. The license has a distinguishing feature in which adds an explicit grant of patent rights [The Apache Software Foundation 2004]. This also includes a termination

---

\(^7\)Which in time could evolve to a significant amount of “sources”, becoming a highly impractical requirement in projects with a high number of contributors
\(^8\)This also made the BSD compatible with the GNU GPL
\(^9\)This license also goes by the name of the New BSD
\(^10\)The Original BSD is still in use, the FreeBSD project being an example. Both FSF and OSI disencourage the use of this old license
\(^11\)Several commercial actors have implemented BSD-licensed software in their solutions, e.g., Microsoft has publicly admitted the usage of BSD-code in their Windows operating system
\(^12\)By some referred to as being weakly copyleft
clause in which the license rights are revoked if the licensee sues over patent infringement\textsuperscript{13}. The reasoning for this is to ensure that all contributors take on the responsibility for possible patent infringements.

3.3 Business Models

As opposed to the more traditional way of developing and selling proprietary software, open source software sets the stage for generating income by other means. These models for generating income are typically based on providing value-added services, although direct license sales can be a coexisting possibility. We will here explain some of the business models that are applicable in relation with open source software.

3.3.1 Support Sellers

In this model, the application is given away for free and revenue is generated by other means. The revenue can be based on distribution, branding and after-sale services, and has been successfully accomplished by well-known companies such as Red Hat and eZ Systems. To be successful with this business model, it is important to differentiate from the competitors. This can be done by providing better and easier-to-use distributions than competitors, or by providing similar services at lower costs. In relation to this, the first mover advantage should be mentioned. This means that the first company that manages to attract a large customer base will be difficult to outcompete, something that has been an important factor in the success of, e.g., eZ Systems. It is also possible to provide services such as support, warranties and sale of liabilities to generate income.

3.3.2 Loss Leader

In this business model, an open source product is offered free of charge with the intention that this product will increase the sales of other of the company’s products that are following a traditional business model. This can work in several ways. One example is that the open source product is used to build the brand and reputation of a company. By releasing the product freely, the user base of the product can increased, hopefully leading to increased future sales of other products. Another approach is to release software for free that complements other software, leading to increased value of these products which are sold under commercial licenses.

3.3.3 Widget Frosting

This business model is a possible choice for companies where the revenue mainly is generated through the sale of hardware. By supplying open source software such as drivers, complete applications or even operating systems that support the hardware, this can lead to increasing the sale of the main product, hardware.

3.3.4 Accessorizing

This business model is intended for companies that distribute books and other accessories associated with open source software. For example, companies that are selling books explaining how to program in Perl or how to setup Linux distributions are following this business model. Since the software is licensed as open source, this also means that the software can be bundled together with the book adding to the book’s value at a low cost.

\textsuperscript{13}Making also this license incompatible with the GPL
3.3.5 Hybrid Model

The hybrid model involves software distributed under licenses that are not considered true open source, nor as restrictive as traditional proprietary licenses. The idea is to take advantage of open source while still being able to create revenues directly from the sale of licenses. This is generally accomplished by either limiting the availability of the source code or by treating the users differently according to the context in which the software is to be used. It is often referred to as dual licensing, a concept that has been successfully adopted by several companies. This can be done by providing one restrictive open source license that makes the software free of charge to everyone, as well as a commercial license that must be bought to include the software into other commercial products. Some examples of companies that have used this approach are Trolltech and MySQL.

3.4 Summary

This chapter has described aspects related to open source in the commercial environment, and will be used as background material later in the report. We started by explaining some benefits that can be derived from the use of open source software or from existing closed code under an open license. We also related these potential benefits with associated dangers. It has been shown that although open source in many cases can prove a good solution, this will not always be the best option. The choice of whether to adopt open source or not is an important strategic choice. Certainly, one can never guarantee that such a move will prove successful. We also briefly described some of the licenses that are used with open source software, and the associated risk that must be taken into account.
Chapter 4

Research Design

4.1 Research Goal

The goal of this project, as stated in the project description, is to:

Describe processes, roles, and tools used in open source environments to establish state-of-the-art processes and necessary tools to establish and run an open source development project.

4.1.1 Research Questions

To fulfill the assigned task, we have designed the following research questions:

RQ 1 When should source code be released under an open license?
   
   RQ 1.1 What are the most suitable license(s)?
   
   RQ 1.2 What are the architectural requirements to the project?
   
   RQ 1.3 What are suitable business model(s)?
   
   RQ 1.4 What are the necessary properties of the product?

RQ 2 What deployment and advertising processes should be followed to achieve a successful open source project?

   RQ 2.1 What can be done in to attract contributors to a project, i.e., to create a community?
   
   RQ 2.2 How can the project be made known in the community?
   
   RQ 2.3 How should a company relate to an open source community?

RQ 3 What infrastructural requirements must be addressed when releasing open source code?

   RQ 3.1 Are there any requirements regarding what tools that must be provided?
   
   RQ 3.2 Should an external portal like SourceForge be used, or is it preferable to provide an own solution?
4.2 Research Approach

Given the definition of our assignment, we have found that our research approach will be of both explorative and descriptive nature.

4.2.1 Explorative Research

In explorative research, neither the nature nor the dimensions of a topic are well known [Kleining and Witt 2001]. Explorative research can used as a pre-study for a more through investigation of a topic to assure that important issues are foreseen [Wohlin et al. 2000].

4.2.2 Descriptive Research

Given the definition of our task, we are to describe processes, roles, and tools used in open source development. Descriptive research can be performed to provide a systematic description that is as factual and correct as possible, and such an approach will be taken to answer our research questions. That is, we will provide descriptive guidelines based on our exploration of factors that influences the success of open source projects.

4.3 Collection of Data

We have chosen to use a literature survey as the main method for collecting data. Additionally, we will gather information about open source tools by direct involvement, i.e., installing and testing such tools manually.

4.3.1 Literature Survey

Conducting a literature survey is a good way to get a better understanding of the field that is being researched. When we initially started our research, we did not have much general knowledge about the open source phenomenon, and conducting a literature survey therefore seemed like the natural approach. We will continue with this approach to provide answers to the research questions. We believe that this is a good option within the time-limit of this depth-study. We considered performing a quantitative study, such as performing surveys or an experiment. However, the data collected with such methods within the limitations and resources provided by this project might be difficult to generalize to specific guidelines, i.e., lacking external validity. It is also difficult to construct a survey since open source projects are very heterogeneous.

Search Strategy

To collect information for our research, we decided to taking an initial holistic approach to surveying literature. This is due to the small amount of knowledge we have about the subject, and the need for getting a better understanding of the subject early in the course of the project. As we have a course that is directly relevant to the subject at hand, TDT10 Software Technology: COTS and Open source software, we began with reviewing the curriculum of the course. We then extended our search to the large journal databases as presented in Table 4.1. The search in the databases are targeted directly at our research questions as well as to relevant new information we identified as the project went along. Additionally, we looked at relevant references provided in our articles as well as reviewing the advices we received from our supervisor and fellow students.
To gain a better understanding and overview over certain topics, we used Google\(^1\) and Wikipedia\(^2\). We are aware that such sources can be unreliable, and is therefore necessary to verify sources and cross-check information with the related research literature.

<table>
<thead>
<tr>
<th>Journal Database</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>The ACM Digital Library</td>
<td><a href="http://portal.acm.org/portal.cfm">http://portal.acm.org/portal.cfm</a></td>
</tr>
<tr>
<td>ScienceDirect</td>
<td><a href="http://www.sciencedirect.com/">http://www.sciencedirect.com/</a></td>
</tr>
<tr>
<td>IEEE Xplore</td>
<td><a href="http://ieeexplore.ieee.org/Xplore/dynhome.jsp">http://ieeexplore.ieee.org/Xplore/dynhome.jsp</a></td>
</tr>
<tr>
<td>Google Scholar</td>
<td><a href="http://scholar.google.com">http://scholar.google.com</a></td>
</tr>
<tr>
<td>Blackwell Synergy</td>
<td><a href="http://www.blackwell-synergy.com/">http://www.blackwell-synergy.com/</a></td>
</tr>
</tbody>
</table>

Table 4.1: Journal databases and search engines

### 4.3.2 Identifying when it is Suitable to Release Source Code under an Open License

*RQ 1*, is also mainly answered by use of relevant research literature. Based on the literature, we specify what *we* find to be the most important issues.

### 4.3.3 Identifying required Deployment and Advertising Processes

*RQ 2*, is answered mainly by use of the literature. Our specific guidelines are based on the findings from the literature study.

### 4.3.4 Identifying Infrastructural Requirements

One of our stated research assignments consists of identifying the demands regarding infrastructure when “going open source”, and this has been translated almost directly into first research question, *RQ3*. To answer this question, we are reviewing the literature that explores open source portals, as well as studying several of these portals ourselves. Our own study consists of gathering statistical material from such portals, as well as studying the infrastructure both related to the hardware and software configurations they provide.

We take a qualitative approach by testing some of the available tools that is often used in open source software development by ourselves. The goal is to identify features provided by the tools, their ease of use and considerations a company should make when considering such tools.

### 4.4 Limitations to our Approach

As we are approaching the research with the means of a literature survey, we will be limited to the information found in the existing literature. We will also be subject to potential bias in this literature as well as the limitations the primary studies themselves were subject to. We are basing our research on existing sources, thus we will not be able to provide any new revelations such would be possible in an experiment or by performing a case study. Instead, we will try to provide a good overview of the state-of-the-art of open source development, as well as create general guidelines that in our opinion should help companies that are interested in adopting an open source strategy.

\(^1\)http://google.com
\(^2\)http://en.wikipedia.org
4.5 Data Analysis

To analyze the data properly, we both review the sources, references, and to a certain extent the research methods used in the primary studies. We compare studies of similar nature and create guidelines based on the results gathered from these studies. It can be said that we are using a meta-analytical strategy for these purposes, although we do not review the primary studies’ research design to the extent that would be required to call our study a meta-analysis per se.

4.5.1 Meta-Analysis

Meta-analysis is a reviewing method that aims to balance the literature reviews more properly than traditional literature studies. A meta-analysis is used “[…] to summarize the results from several studies into one analysis […]” [Wohlin et al. 2000; pp. 38]. Our guidelines can be viewed as such a summary, trying to assemble the results found in many different studies into one coherent package.

4.6 Validity Threats

It is important to be aware of threats to the validity early in the project, as these threats must be accounted for in the research process. It is also important for others to be able to view what threats might impact this research if they are to use it for any practical purposes.

4.6.1 Threats to Conclusion Validity

Threats to conclusion validity are concerned with issues that affect the ability to draw correct conclusions [Wohlin et al. 2000]. This means that our guidelines might be invalid if these threats are not properly accounted for.

Inclusion of Poor Quality Studies

A comprehensive literature search may yield many articles that are not relevant to the question under investigation or that may be methodologically weak leading to invalid information [Oxman and Guyatt 1988]. If we use studies that are of poor quality, this might lead to our guidelines being based on invalid assumptions and conclusions. To negate such effects as much as possible, we try to make sure that the studies we review are objective, properly conducted and that similar results can also be found in other studies. We mainly using studies published in well-knowned journals as a basis of our project, with the hope that such studies have been quite thoroughly reviewed.

Citation Bias

It is a possibility that the literature we are reviewing will contain citations that are from biased sources. We have also found that certain papers are heavily cited, leading to several studies being based on these sources. Indeed, as we too will be using citations that are commonly referred to for our investigations, this validity threat might be even higher.

Researcher Bias

Biased opinions found in the literature we are studying as well as our own, may be a threat to the validity. We have identified that open source is a phenomenon comprising many strong opinions. In our view, the open source community and the researchers studying it, might provide information based on philosophical grounds as well as biased opinions. Our knowledge of the open source community initially was quite limited, reviewing a high amount of research literature where biased opinions are expressed might lead to similar opinions of our own. As our own opinions as well as the findings we do will impact the result of this study,
this might be a threat to the validity of our conclusions. We hope to eliminate this threat as much as possible by keeping an objective view, by trying to identify biases and by comparing studies. We also hope that reviewing of our report by ourselves, our fellow students, and our supervisor might help identifying bias.

Regarding the tools that we will study, our findings here might also be impacted by our own biased opinions as well as those of others. First of all, we only study the tools that are most commonly in use with open source software development. Additionally our own knowledge of tools and use of them might impact our results.

**Improper Combination of Findings**

When we are studying a large amount of literature and trying to combine the findings, it is a possibility that our combination of findings will be improperly performed. This might be related both to the bias we already described, as well as wrong interpretations of the findings. If our combination of results are not done properly, our guidelines might be invalid, thus leading to wrong conclusions.

### 4.6.2 Threats to External Validity

Threats to external validity are conditions that limit the ability to generalize the results [Wohlin et al. 2000]. The major threat related to our project will be to base our conclusions on a population that is not representative of the entire community. We do think that this threat is negated as much as possible with the wide variety of literature that we study. As this literature comprise many different domains in which open source is used across a wide variety of organizations, we believe that our ability to generalize the results will be high. Still, the generalizations we make must be accounted for in every single company as factors impacting any individual company, always will be present.

### 4.7 Summary

This chapter describes the way we plan to perform our research. We describe the what we expect to get out of our research, and how to get it. Our main source of data is a literature study. The data we get from our sources will be subject to a kind of a meta-analysis. We also consider what our limitations are, and what validity threats we face.
Part II

Guidelines for Establishing Open Source Projects and Suitable Infrastructures
Chapter 5

Making the Decision to go Open Source

In this chapter, we present some strategic considerations to be considered when planning to release code as open source.

5.1 Identification of a Suitable Software Architecture

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements and the relationships among them [Bass et al. 2005].

To make the process of releasing code as open source successful, it is important that the existing architecture is suitable for this purpose. The following sections present a list of related aspects that we consider as most important.

5.1.1 Modular Architecture

What is of first and foremost importance is to have a modular architecture. Having a modular architecture means that each module in the software has well-defined areas of functionality, and that the architecture should make it clear how these modules interact. Software development in general benefit greatly from such an architecture, but in an open source project it can be viewed as an absolute necessity. Indeed, software featuring high modularity makes distributed functionality possible, allowing for extending the functionality of the product without changing the existing one. From a pragmatic viewpoint, this allows a company to maintain control of the original product, while allowing the functionality to extend and scale with the community [O'Reilly 1999]. The loose coupling associated with a modular architecture also makes the software more maintainable, leading to reduced ripple effects caused by maintenance work.

Modularity is also important for attracting newcomers by allowing a gradual learning curve. A modular architecture causes increased transparency, which makes it easier to take advantage of specialized knowledge [von Krogh et al. 2003]. Additionally, it lowers the domain knowledge that is required for participating in the project, i.e., it will not be necessary to have full understanding of the entire architecture in order to provide beneficial contributions. This is especially important related to open source development. Outside contributors can choose from a wide variety of projects, and will typically contribute in an area which they consider themselves knowledgeable. Thereby, an expert in a certain area can provide contributions in relation
to his area of expertise. The modularity of the architecture should also accommodate for the development of plug-ins and add-ons to the product, which both are popular within the community and value-adding for the product [Scacchi 2002].

If the system is not modular, it can generally be said that it is not suitable for open source development [Kogut and Metiu 2001]. Therefore, if modularity is not present in the Architecture, the necessary modifications must be made in order to make it so, although a careful consideration of such costs should be made. If found unfeasible, then the product have low probability for succeeding in the open source world, and such a strategy should be abandoned.

5.1.2 Adherence to Open Standards and Protocols

One of the important requirements to sustain innovation, is to adhere to open standards and protocols [O’Reilly 1999]. Open source software is well known to have a high rate of adherence to open standards. This allows for easy connection between new and existing software, as well as easier porting across platforms. First of all, by adhering to open standards, the software is easier accepted by the open source community. Secondly, the simpler porting and combining of software solutions can cause an increased user base as well as the development of coexisting value adding software.

5.1.3 Suitability of Software Components

There are also certain considerations related to software components that must be made when releasing software as open source. If it has been written entirely inhouse, meaning that no commercial or open source Off-the-shelf components are being used, then this section can be skipped. If not, then each of these questions should be assessed.

Does your Software Contain any Commercial Components?

COTS\(^1\) components that are used in the system can easily make it impossible to release the software as open source. These components will often include a non-disclosure clause stating that redistribution of the source code and/or the interfaces is not allowed. The possibility of finding another suitable component to use is of course an alternative that can be considered in these circumstances, although intensive cost-benefit evaluation might be necessary.

Does your Software Contain any Open Source Components?

First of all, if your software has been previously sold without violating any of the components’ licenses, e.g., only comprising BSD licensed software, then distribution of the entire product as open source should not cause any problems from a legal point of view.

If you have used the software for internal purposes only, and would like to release it as open source, then the licenses’ intercompatibility becomes an issue. This is because open source licenses come into effect only when software is distributed in any way. If the software comprises several open source components, then the licensing scheme of each and every component must be assessed considering compatibility. Especially, the GPL License has a large amount of open source licenses of which it cannot be combined with. To check this, we refer to their compatibility list at: \url{http://www.gnu.org/philosophy/license-list.html}. Additionally, open source software is often a compilation of code from many different sources, making it hard to detect whether parts are protected by intellectual property rights. Therefore, an assessment of the components must be taken regarding property rights. The legal exposure should be minimal if a component has a strong open source community, is mature, is widely adopted and have a clear legal status [Ruffin and Ebert 2004].

\(^1\)Commercial Off-The-Shelf
When the Licenses are Not Compatible
If the licenses are not compatible, certain measures must be taken. In general there are four options:

- Receive a written permission from the copyright holder if an adjusted license is suitable.
- Find another component with a compatible license.
- Develop a similar component internally.
- Abandon the open source strategy of this software.

When the Licenses are Compatible
When the licenses are compatible, release of open source software seems like a viable option. Still, there are certain considerations to be made. The first is whether the components should be locked as-is, or whether the company should follow new releases as they become available. The first option will cause less need for future adaptation of the components. However, the benefits provided by newer versions will be lacking, and certain compatibility issues might arise as the time goes by.

Secondly, the company should be aware of the impact the components might have on the future business strategies. The licenses might, e.g., determine what business models that can be chosen, due to the restrictions they impose. In particular, going for hybrid solutions might be impossible without receiving written permission from the licensors of the Off-The-Shelf software.

5.2 Generating Income Based on Open Source

5.2.1 Gaining a Strategic Advantage
When a company is considering to release software as open source, it is important that the company carefully evaluate the open source strategy up against continuing with a proprietary one, and only adopt the open source strategy after careful consideration. Generally, it can be said that the main driver for a commercial actor is to increase its revenues. Therefore, proprietary software should only be released as open source in the case that such an effect can be obtained either directly or indirectly. This can relate to increasing the company’s current or future earnings as well as decreasing the market share of a competitor, making the total monetary gain higher than what would have been the case if the software had not been released under an open license. Actually measuring these gains may prove difficult, especially when related to indirect effects such as adversely affecting a competitor or the resulting advertising effect that leads to higher revenues from other products.

5.2.2 Choice of Business Models
If all the above aspects have been considered, and the company has concluded that going open source seems like the right thing to do, the next step is to consider how the release of the source code can help in creating increased revenues. It will be necessary to outline an overall business plan, taking both the company, the product, intended users and the market into account. We will in this section come with certain suggestions regarding choice of business model. We will only consider the business models applicable for software-providers, meaning that we will suggest either the ‘support seller’ model, the ‘loss-leader’ model, the ‘hybrid’ model, or a combination of these.
The Company

It is important to take the company’s profile into account when considering the open source strategy. The initial evaluations that should be taken comprise:

- How do you intend to govern the project?
- What amount of influence do you want?
- Do you have the capability to hire external developers?
- Does the evolution of the project have impact on the company and other business-strategies?

Large Companies

Large companies have available resources that they can put into the open source venue. They possess the capability of putting efforts into R&D, which should include open source experimenting. Large companies generally run a much smaller risk by failing the open source strategy. They should probably not go full-scale into the open source community by releasing their main source of income openly, but should start with supplementing software they develop. Such companies will also have a huge opportunity to negatively effect competitors, taking away their market share. Additionally, they possess the resources to organize and help with the development project to a large degree as well as hiring external contributors that are of value to the project. The possibility of steering the project as the company sees fit is therefore quite good, however, it will still be necessary to listen to the community and users.

The huge amount of news reports that will follow such a strategy from major companies, will also minimize the risk of not attracting a community. It will also cause a massive advertising effect for the company. The choice of license type must still be taken into account, and should not be too “proprietary”\(^2\).

These companies can choose all possible business models. Large companies will often have a large amount of software under development, making the loss-leader strategy ideal in creating value for the other products. Providing services by following the support-seller model will still be a viable option. We do not recommend the use of hybrid strategies for large companies as we think such resources might be better spent enhancing complementary segments within the company. However, if the company has few projects that will be influenced by a loss-leader strategy, then a hybrid solution can be chosen. The same applies if the project has high suitability for other companies, making them willing to pay for an alternative license.

Middle-Sized Companies

Middle-sized companies will not have as many options as the large ones, and giving general advice in this regard is difficult. The company’s other products will have to be assessed, as well as the available human resources. A Loss-leader strategy is suitable if the company has complementary products that might gain increased value of such a move. It might also be suitable in certain cases to adopt the loss-leader strategy by developing a professional version with added benefits, including adoption of the support seller model. However, we generally believe that such a company will benefit most by of adopting a hybrid strategy as long as they have full copyrights of the code base. In this way, they can derive benefits from the open source community, while still gaining revenue from selling licenses. Open sourcing of software in which the company is too small to compete commercially, is also a possibility, hence pursuing alternative strategies instead.

Small Companies

Small Companies will have the greatest risk when releasing software under an open license. They should therefore not opt for going fully open source unless the software is outside the segment from which they derive most of their revenues. We recommend adoption of a hybrid if they have full ownership of the code, and a loss-leader strategy if they plan to develop a professional version. Support Selling can be used as an alternative source of income, but should not be relied on as the sole strategy.

\(^2\)When Netscape tried to use the Netscape Public License with their software, they met massive resistance
The Product and Intended Users

The product itself must also be considered, both regarding suitability for open source and how it relates to the company’s strategy, its intended users, and complementary software. If most of the company’s income is based from a certain product, then releasing it as open source is not recommended. If the product is not business-critical, then adoption of an open source strategy might be viable. As an example, this can relate to software used internally within the company but that they are not gaining much revenues through sales. The release of such software as open source will possibly contribute to the evolution of the software, indeed giving benefits to the organization which itself is a user. Additionally, this might open up the possibility for new business strategies, might improve the reputation of the company and thereby increase future sales of their other products.

The target users for the software should be identified, as the business strategy should evolve around the users. If the product is intended for end-use, then both the loss-leader or hybrid model is possible options. Development of a professional version providing additional features might also be a possibility related to the loss-leader model. If the intended users are other commercial developers, then the hybrid model should be the strategy of choice. If the users are mostly consisting of commercial actors or others that are willing to pay money for support services, then such should also be provided.

5.2.3 Choice of Licensing Scheme

After having considered the compatibility of components and chosen a business model, a suitable license must be selected. The choice of license type might be one of the most important issues to consider when going open source. Certainly, the choice of using an open source license should be carefully examined. If the current business model works well, then there should be no immediate need for open sourcing the software. Also, if the issue is that the community might gain a more favorable view of the software if they can view the license, there exist options that can allow people to view or even contribute with modifications while not releasing the source code under a true open source license.

The choice of a licensing scheme might impact the community; who is attracted to it as well as the end-users of the software. It might also influence other projects that compete with, or complement your product, as well as commercial vendors and support providers [Lerner and Tirole 2005b]. All of these factors must be taken into consideration.

In this section, we will provide our recommendations of licensing in the form of general guidelines. Note that these guidelines will be based on the licenses we have discussed earlier in the report, and that other more suitable licenses for any specific project might exist. Additionally, these guidelines are based on our own perceptions as well as those found in the literature. The final evaluation of license suitability and choice must therefore be left up to each and every company and to their specific situation.

Choose a Well Known License

This is advisable as the open source community generally will have a good notion about how these work. Selecting such a license will also give the community a good pointer as to what the company’s intentions are. Some properties of the different licenses can be seen in Table 5.1.

5.2.4 Recommendations Regarding Specific Licenses

The GNU General Public License

This is a license that is preferred by many open source developers, and is the most commonly chosen license. It should be chosen to show the community good intentions as of why the source code is released, or when the

3 Adhering to the OSD
source code is not to be used by other commercial actors. It helps protecting the work of the community from commercial hijacking; however only value-adding strategies are viable with the code licensed with GPL. The project is more likely to attract a large amount of developers with the use of GPL, and it should be used when accelerating innovation, attracting contributors, and users are the most important issues. The greater inclusion of users builds relationships and loyalty. GPL is especially well-acknowledged within the POSIX community, and software developed exclusively for such platforms might benefit for the use of GPL.

### Revised BSD

The Revised BSD is very permissive and should be used in cases where this is needed. It might encourage commercial users to adopt your product. It can be combined with GPL, making it a possible alternative in the case that commercial use is intended while still trying to leverage the GPL-community that might want to use your software. If trying to establish a standard, then the use of an unrestrictive license such as BSD makes sense. If a certain project has strong appeal and contributors might benefit considerably from signaling incentives, or the licensors are well trusted, then the use of BSD might also be a good choice [Lerner and Tirole 2005a]. The main negative point with the permissive property of this license is the risk of project hijacking.

### Mozilla Public License

This license can be referred to as being weakly copyleft, as it requires new source files to be separated from the MPL licensed code to avoid becoming MPL licensed code itself. It enhances open competition among commercial and non-commercial developers, and is a suitable alternative for keeping future business paths open. It is not compatible with GPL, hence use of MPL leads to neglection of the benefits such compatibility can provide. It should probably not be the license of choice if the target Operating System of the software is POSIX-based. It does, however, encourage contributions back to the community in a larger extent than the Revised BSD.

### LGPL

This is recommended to use if the program is a library, as it is both compatible with the GPL and can be combined with proprietary works. As the licensed code otherwise works as the GPL, it can, e.g., help promoting standardization of a library. Of course, adherence to the licensing terms of the LGPL will in such cases be necessary, meaning that eventual plugged-in proprietary code must be properly connected.

### Apache Public License

This license is very similar to the Revised BSD. Indeed, the patent infringement clause is generally the only difference. We recommend that the Revised BSD is used instead of this one, as we find the GPL-compatibility to be of great benefit. If the added patent infringement security that APL provides is deemed necessary, then this license can be chosen. The same goes if much of the software is heavily based on software already licensed under the Apache license.

---

4 Unless you have copyright over the entire code base

5 Collection of standards that define the API for software compatibility with UNIX
5.2.5 Recommendations Based on Choice of Business Model

Support Seller
To attract the largest amount of users to sell support to, as well as gaining as much contributions as possible we will in this business model recommend the use of GPL. This will ensure that the company does not have any hidden agenda and creates trust within the community.

Loss Leader
With this business model, we recommend the use of GPL, BSD, or MPL depending on certain factors. If the main objective is to attract many users and create trust, then GPL should be chosen. Generally, if the negative effects that GPL imposes on the company will not impact the company’s other strategies in a negative way, then GPL is the best choice. If the main objective is to gain an increased user base, but with a wish of incorporating the received contributions into other software that is promoted through the loss leader strategy, then BSD should be chosen. MPL can be chosen with similar objectives, and might be a good alternative if future strategies have not been properly decided yet.

Hybrid
If going for a hybrid solution with a dual-licensing arrangement, then we recommend that the openly licensed source code is GPL or BSD depending on the hybrid strategy. Along with the proprietary license, GPL should be chosen when the target is developers that want to use the software in their own software. By choosing GPL in this case, you may give to the community and receive the benefits that GPL provides while still being able to charge commercial actors for the proprietary code, which is then sold as COTS. Such a strategy will require full ownership over all of the source code, and will make it impossible to use the community’s contributions in the software that is sold commercially.

The use of BSD is viable when you want to incorporate the direct contributions in your own product. However, you will in this case need to stay ahead of other commercial competitors – that can also use the code base for similar purposes – by developing further inhouse enhancements to the software. Such a strategy, however, might make it more difficult to attract the community to aid with the development process.

If the shared code is a common library with defined APIs, then the library can be released under LGPL. This makes the library open source, while the proprietary code can still make use of it.

5.2.6 Recommendations Based on the Environment

The Licensing of Components
If you have GPL-licensed software incorporated in the software, the project have to be GPL-licensed.

Competitors and Complementary solutions
If the company is being outcompeted, then licensing the source code under GPL might be a possibility to try to steal away market share from the competitors. MPL and BSD is not very viable solutions here, taking into consideration that the software then might be exploited by the competitors.

If the competing open source software is using less restricting licenses such as the BSD, then your own software should probably also stick to the same license. This is due to complementary strategies and network externality effects [Lerner and Tirole 2005b], and is generally a good rule.

Attracting the Target Users
The target audience can have an impact on the license choice. If the target audience mainly comprises users of a UNIX based operating system, then GPL is probably the choice that will attract most contributions. If the target audience comprises non-technical users, and the software is mainly intended for end-use, then the

---

6An entire rewrite of the submitted code might be a possibility as copyright only protects the expression and not ideas.
choice is more open. Especially related to end-use for Windows platforms, GPL probably is not necessarily the way to go [Krishnamurthy 2003] and MPL, BSD or other licenses not covered here should be considered. If the target is developers that want to use the software commercially, then GPL is unsuitable, however in these cases a hybrid model should be used if possible.

5.3 Summary

This chapter has presented some of the strategic considerations that have to be done when planning to go open source. We started by identifying architectural requirements for open source projects. That the project already use open standards and open protocols will always be considered a good thing in the open source community. We also considered ways of generating an income from releasing your code open source. We provide information on the best business models and licenses to choose considering the business environment and already used licenses.
Chapter 6

Attracting people to the project

A successful establishment of an open source community depends on many different factors. There are technical, psychological, and legal aspects to consider before attempting to go open source. We will now consider some of these aspects and describe some actions that can be performed.

6.1 Necessary non-architectural software properties

Some non-architectural properties should be present before releasing software under an open source license, and are of importance to attract contributors to a project.

The Product should be runnable.
The product should be in a runnable condition when it is released as open source. As stated by Raymond, “Your program doesn’t have to work particularly well. It can be crude, buggy, incomplete, and poorly documented.” [Raymond 2002]. Having a product that can be tried out, is important for outsiders to evaluate the product and find out whether it is of use or not. As we have already seen, the developers in open source projects are typically also the users, making this an important factor. When outsiders consider the product worthwhile to use, they can also identify certain problems, additional features that should be implemented etc, which they then either can identify to the project developers or develop themselves.

Of course, an internal project that is currently being sold under a closed license will fulfill such a requirement, making this mostly a necessity with projects that are early in the product’s lifecycle. Generally, if the project is not currently runnable, the release as open source software should be delayed until this criteria is fulfilled.

The project must be of value to outside developers

Even though the system does not have to be fully functional, it must have future potential for outside developers to take interest. This relates to the “scratching of own itch” metaphor, i.e., if a developer sees the necessity in a product, he may become interested in partaking its further development. When considering this, it seems that one important factor in attracting developers is to make sure that the project can be of interest to a large group of users. To achieve this, the project should either fill an existing hole, such as, e.g., Linux and Apache, or be of more value to the users than the existing alternatives.

The project should be properly documented

Although documentation has traditionally been neglected in open source projects, as well as given little emphasis by authorities such as Raymond, [Raymond 2002], we agree with Tim O’Reilly in that providing good documentation is of high importance in open source projects [O’Reilly 1999]. Documentation of the architecture and APIs is important in order for outsiders to quickly gain an understanding of the project. By
making it possible to concentrate on a single well-documented module of interest, the possibility of attracting developers increase. When regarding the documentation, it is also important that no embarrassing comments are disclosed with the open source product. If the code base has been developed inhouse and intended for internal use only, then all documentation and comments should be scrutinized to eliminate unsuitable commenting. Providing good documentation for the End-Users is also important to gain an increased, and happy user base, although this is probably not of a main concern early in the open source project.

**The project should be of general benefit**
The project should be of interest to many different domains, being beneficial to many different people. This means that software that has been tailored for a highly specific market might not be suitable as open source software [Kogut and Metiu 2001]. Such code will be too specific and few people will have the necessary domain knowledge and interest in such a product to want to participate in the development.

**There must be interesting tasks for outside developers to partake and a well-defined extension mechanism**
It is important that the project seems doable and provides merit for outside developers [Lerner and Tirole 2002]. If the project shows future promise, it might be easier to attract developers since the early contributions will be very visible in a successful project [Lerner and Tirole 2002]. The description of the project should also be generated with potential developers in mind. This mean that its attributes and features should be highlighted. The company should also provide interesting tasks that need further work, i.e., by suggesting challenges to the developers. These should still be simple enough to attract users, but challenging enough to be of interest [Dahlander and Magnusson 2005]. Such a strategy was exemplified by Linus Torvalds, where he explicitly highlighted the extent to which creative programming was required to achieve full functionality [Lerner and Tirole 2002]. Another way to provide tasks is in the form of TODO-lists. This is an effective way to manage dispersed work [Yamauchi et al. 2000], and can make it easier for outsiders to identify what needs to be done. The company should also be aware that as outsiders can pick tasks that they find of interest, more mundane tasks such as documentation might be up to the company itself. A company should not rely on the more non-technical user-base to provide such documentation [Fitzgerald 2004].

### 6.2 Making the Project Known in the Community

As far as we know, there have not been performed many studies regarding specific marketing strategies for open source software. The adoption of open source software and its diffusion has rather been said to be influenced by its perceived intrinsic value and externality effects1 [Khalak 2000, Bonaccorsi and Rossi 2003].

For an open source product to be self-sustained, it must gain a “critical mass” of the market share [Khalak 2000, Bonaccorsi and Rossi 2003]. Achieving such a critical mass might prove extremely difficult, however, the first step will be to create an awareness within the outside community of the existence of that product. As is the case with both Open and Closed software, a company cannot know beforehand what the demand for the product will be. While companies selling proprietary software will often stick to direct advertising strategies, open source products should probably rely more on indirect marketing and the community to spread the word. A list of our suggestions in order to create such an awareness is presented below:

1. **Present the Project**
To gain some awareness about the project before it is released, it will be suitable to present the project, rationale, and what it will contribute to the community at related conferences. This can help creating initial hype, and might also gain the interest by news sites and papers, if the project comes forth interesting enough.

2. **Create a Proper Web page**
This webpage should be easily reachable by all major search engines, when people search for related software.

---

1Negative externality effects include those from the dominant standards, while positive externality effects come from, e.g., the community of programmers
The project, the intentions behind it, licenses, complementing software, and the components it comprises, should be easy to identify. We will discuss the web page more in detail later in this chapter.

3. Post the Project on all Major Open Source Portals
The project should be posted on SourceForge, Freshmeat, and other portals where open source developers flourish. All important info about the project should be posted, along with tasks that needs to be done. Freshmeat will also list the newly published projects on their main page, which might be viewed by many potential contributors.

4. Post the Project on Software Portals
To attract more users the project can also be posted on download sites such as download.com. This is especially suitable for software targeting end-use at Microsoft platforms. Such software must of course be viable to use in order to keep the users interested.

5. Let News-Sites be aware of the Project
If articles about the project can be published in media such as online news-sites, a large amount of users might be attracted. This means that a press release should be sent to all news papers that might be interested in posting about your newly released software. Of course, it will be highly important that the software is usable with much functionality already finished to keep those users. For most companies, the most realistic initial option will be to get it published in local or national online news-sites, e.g., digi.no might be a good option for Norwegian open source solutions. However, if the company is very large and releasing formerly proprietary software as open source, a major news will probably come automatically. Perhaps the major news site for open source developers is slashdot.org, and if an article can be printed here, it will be possible to create awareness among a large amount of potential contributors. An article about a new unknown open source product to this site might, however, prove a high chance of rejection.

6. Use Wikipedia
An article about the product could be created at Wikipedia. However this article must be written in an as objectively manner as possible. Linking to the product’s homepage where it seems viable, e.g., where other similar products are linked to, might also be an option. Writing an article on your own product on Wikipedia may conflict with the rules of Wikipedia.

7. Newsletters
If your company already posts a newsletter about the company and its products, including information about the open source product in such newsletters will be a good idea. If not already having such a newsletter, it can be a good marketing strategy if it seems viable that many people will take interest in such an option.

6.3 Relationship Between Company and Community
The inter-relationship between open source companies and communities seems to comprise a set of tensions and inconsistencies that can lead to certain managerial issues [Dahlander and Magnusson 2005]. Such issues must be properly resolved to attract and maintain a community, and particularly relate to how much resources the company are willing to spend to create and maintain a community, how much influence the company want, and the organizational structuring of the project.

With a commercial organization as the initiator of a project, it might be even more important than in community-evolved projects, that the leadership is practiced properly. Leadership by a commercial entity may not internalize enough of the same objectives as the open source community [Lerner and Tirole 2005a].

\[\text{This site requires 15 days processing time if sticking to their free service, however it does not hurt}\]

\[\text{Especially the following: Neutral point of view (http://en.wikipedia.org/wiki/WP:NPOV), and Notability (companies and corporations) (http://en.wikipedia.org/wiki/WP:CORP)}\]
If the company maintains too much control, it is quite possible that the amount of interest generated is low. Too much control can thereby lead to the community abandoning the project, or to the creation of competing forks. However, if the control is too little, the effect for the company may be small or even counterproductive [Dahlander and Magnusson 2005].

Even though leaders in an open source project have no formal authority over the community, they play a key role in formulating the initial agenda, setting goals as the project evolves, and resolving disputes [Lerner and Tirole 2005a]. The company should therefore provide a vision for the future of the project, and suggest tasks that can be performed. However, the community must still be considered as peers, letting them direct the future direction of the project as much as possible. As open source governing structures typically are of meritocratic nature, higher influence should be given to those in the community that are important for the project, and credit should be given for contributions.

It seems self-evident that the company will initially retain a leading position. The company should have initial control over the architecture and steer the direction of the development. They should not allow direct contributions to the code-repository without first reviewing the submissions, working the same as other open source projects where the leaders serve as gate-keepers. However, outside participants must still be taken heavily into consideration and be allowed into the core group if they provide many, high quality contributions to the project. Still, company involvement might to some extent obstruct the possibility for a community to have the desired ownership [Dahlander and Magnusson 2005], and a proper balance must be maintained. The proper balance can be identified as having a win-win situation between the company and its community.

**Having a Win-Win Situation**

The main target for any company should be to create a win-win situation between the company and the community, in which both derive benefits from the relationship. This has been related to having a “symbiotic relationship” [Lerner and Tirole 2005a, Dahlander and Magnusson 2005], and implies that the firm tries to co-develop both itself and the community.

To achieve such a situation, the company must be willing to spend much resources on the development of the community itself. The company must actively attempt to both create a large community as well as be willing to spend resources in maintaining it. This includes that resources must be spent for building and setting up a well-functioning infrastructure as well as through direct involvement in the development efforts. It will be necessary to take the open source community into consideration when all related decisions are taken. This means adherence the to community’s norms, rules, and its organizational structure. Legitimacy to influence the community cannot be obtained by having a formal role in the company, but rather by the status gained by providing good contributions. This means that to successfully influence the direction of the project, the company itself must develop code for the project. This also means that the community must be allowed to actively influence the direction of the project. Giving proper credit for community-acquired contributions is also extremely important.

There will however always be a possibility for tensions between the open source community and the commercial actor. It might for example be hard to gain acceptance for using community-developed software in commercial applications. This can be somewhat negated by providing certain incentives for the community, such as providing them with beneficial deals and access to the company’s resources. By organizing social events for the community, face-to-face meetings can improve the social relations and make it easier to gain acceptance for commercial use of the knowledge created by the community [Dahlander and Magnusson 2005]. To relieve tensions, it will also be important to be clear on what pieces of software that is used in conjunction with internally developed code [Dahlander and Magnusson 2005].

**Having a Win-Neutral Situation**

Another approach is to keep direct involvement within the community to a minimum. Such an approach means that the company only provides the benefits to the community that is necessary to keep them from fleeting the project. The company still adheres the norms and regulations in the open source community, but keep direct development efforts provided back to the community to a minimum. By having such a relationship, it might be significantly harder to gain acceptance for using community-developed software in
commercial activities, as well as to avoid conflicts. There is a large possibility that the community will gain the impression that the company is free-riding, and perceiving the relationship as a win-lose situation.

**Having a Win-Lose Situation**

It might be possible for companies to only focus on its own benefits while not adhering to the community in any ways. Such an approach should always be avoided, and the norms and rules of the community should be followed. If the company ends up in such a category, it will create a negative advertising effect for the company due to loss of reputation. If the company is perceived as a free-rider, the community will almost certainly create a forking project if they find the project of value. When a project forks due to such a relationship between the company and the community, it might also work as a motivational factor for the community to create software that outcompetes the original product. If no forks are created, the community will instead abandon the project completely if they find the relationship unsatisfactory, leaving the company with no communal benefits.

### 6.4 Summary

This chapter has given an introduction to what could, and should, be done to make the project known. We start by defining some properties about the projects that should be present before going open source. To have any chance of getting successful, your project must be runnable, documented, and of value. Without these properties, it is unlikely that many users will come. We also consider what steps should be made to advertise for the project. Getting information published in many different sources will attract more users. We also describe what should be done to get in a win-win situation between the company and the community.
Chapter 7

Infrastructure

Setting up a viable infrastructure is essential for success. Such an infrastructure should include a good portal solution providing the features and tools required to establish and keep an open source community. We will in this chapter discuss portal solutions, issues regarding the backbone of such infrastructures, and how the portal should be rolled out.

7.1 Portals

Table 7.1 presents the features that an open source portal should provide. We will here discuss the choices that have to be made regarding portal solutions, and whether to use a public solution, an internal solution, or a combination of those.

<table>
<thead>
<tr>
<th>Essential</th>
<th>Recommended</th>
</tr>
</thead>
<tbody>
<tr>
<td>Download</td>
<td>Forum</td>
</tr>
<tr>
<td>Mailing list</td>
<td>Documentation</td>
</tr>
<tr>
<td>Bug/feature tracking</td>
<td></td>
</tr>
<tr>
<td>Information</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Features needed in a portal

7.1.1 Public Portal Solutions

Public portals like, e.g., Sourceforge, provide a predefined portal solution where open source projects can be published. They might not provide all the features that are of importance, but setting up a project on such a portal is very easy. We recommend publishing the project on such portals early in the process of going open source. Public portals provide the most essential features, and are a good option for making the project available with little efforts. They also provide a good search capability that developers can use when looking for interesting projects.

The disadvantages with public portals are mostly concerned with the lack of customizability. It is not possible to select tools and features that the portal does not support, or to configure the portal according to specific needs. Thus, an internal portal solution should also be made available early on. The public portals should be used to make your project more accessible, to provide a description of the project and for linking to your internal portal solution.
7.1.2 Internal Portal Solutions

Pre-Configured

There exist several pre-configured portal solutions, that should provide a suitable portal configuration for most companies. Such portal solutions are to some degree customizable, and will probably be a good choice for the majority of companies. The main strength of the pre-configured solutions lie in the complete integration of the tools, leaving little configuration to the company itself. It may be possible to use a pre-configured solution as the basis solution, and to provide additional features that are set up manually. Such an option does not give as much control as a complete manual setup, but it requires much less work than doing everything yourself. A pre-configured solution might also make it difficult to provide a fully integrated solution. Thus, if such options are required, a complete manual implementation of the portal solution might be necessary.

Manual Implementation

Creating an internal implementation of the entire portal is a possibility for companies where full control and customizability are required. There are many existing tools that can be combined to create a suitable portal solution for your project. Companies selecting the tools themselves, can tailor the features and functionality of the portal to their own specific requirements. Performing a complete, manual integration of the portal can be both time-consuming and resource demanding, particularly related to maintenance issues. When new versions of the tools are available, the choice must be made whether to keep your current solution or to implement the new version in the portal. If the first option is selected, the company will miss out on bug fixes and new features, while the second option will require adaptation of the new version to the portal. Because of this, we only recommend choosing a complete manual portal implementation when a large amount of customizability is required, and the other options are found not to fulfil the company’s needs.

7.2 Usability

Portal solutions contain a large amount of tools and features, and it is important that the portal is easy to both navigate and use. The product should be the main focus, meaning that the most important information should be found on the front page. Such information should include the name of the product, a description about the product and its features, and an easily accessible link to the download page. The front page is intended for the casual user, and should not contain technical details that are of little importance to end users.

The developer tools such as version control, bug handling, and mailing lists, should be found under a specific menu item. The reason for this is to provide easy access to these features while at the same time avoiding to confuse casual users with technical details. The forum and contact information should also be separate menu items, as these can be of importance both to casual users and the developers.

Figure 7.1 gives an overview of a possible setup.

![Menu items in an open source portal](image)

Figure 7.1: Menu items in an open source portal
7.3 Scalability

It is important to consider scalability issues when creating an open source portal. The portal should be configured with a modular architecture, making it possible to divide the workload on additional servers if necessary. The initial design must allow for the database to be spread over several servers, as databases might be subject to a large amount of queries and become a bottleneck in the system. Tools like forums and wikis can also be very resource-demanding, and should be possible to transfer to other servers.

The project will only need a single dedicated server and a backup in the initial phase. If the project becomes too popular for this server to handle, the first step would be to mirror the most popular services (such as download) to new servers. Other services that can be transferred without causing problems, should also be moved to other servers.

7.4 Servers

When choosing anything but a public portal, the company will have to decide whether to use servers of their own or external servers provided by, e.g., web-hotels and similar hosting services.

7.4.1 Company-owned Servers

Providing servers owned by the company gives complete control of the system. All tools can be run and configured internally, and the company is not limited by load limits set by an external provider.

However, there are several drawbacks with such an option. The company will have to both configure and set up the servers themselves. It will also be necessary to pay for keeping the servers going, and to provide enough bandwidth and hardware to handle the load on the servers.

We do not believe that using company-owned servers for hosting the majority of services is a good early choice. Only those services that external servers cannot provide, requires physical access, or are found too costly, should be hosted by the company. If the company-owned servers are hosted by the same ISP or are located at the same office, there might be downtime errors due to power failure or problems with the ISP. We therefore recommend keeping them spread, e.g., at server parks to the extent this is possible. If the open source project is highly successful and the company generates income based on their open source product, a larger amount of company-owned servers can be justified. It might in these cases be necessary to employ personnel dedicated to maintenance of the servers.

7.4.2 External Servers

We believe that most of the services should be hosted on external servers. Most of the services initially required can be hosted on a normal web-hotels. This web-hotels should be chosen carefully based on factors such as cost, load limitations and guaranteed uptime. Some services might, however, require shell-access, something that most web-hotels do not provide. Thus, these services must be hosted internally or on external servers where the company has the required access permissions.

7.5 Iterations

Open source projects should be rolled out in an iterative manner. This is a strategy that minimizes the risks and costs associated with releasing the software as open source. We will here provide suggestions for iterative
steps that can be followed, where we explain how the portal solution can be established. An example of the final configuration can be seen in Figure 7.2.

Figure 7.2: Suggested portal configuration

**Iteration 1**

The first step is to establish the absolute minimum of services. It is necessary to create a proper webpage that provides information about the product and links to the download page. Both the binary and the source code must be available, and a subscribable mailing-list should be provided. Contact information should also be made available at this step. Having a download counter might also be a good option, as this can become a nice advertising point later on.

Iteration 1 is an important initial step to make the product and source code available as early as possible. The mailing list is crucial, thus problems, suggestions and code contributions can be posted. We do not recommend staying at this step for more than a couple of weeks as this should be enough time to set up a larger, more feature-rich portal solution.

**Iteration 2**

Even though there is no traffic at the site the first couple of weeks, Iteration 2 is of essential importance. At this step, it should be possible to report bugs in a bug-tracker and to browse the code repository. A forum should also be made available for discussions about both the product and off-topic issues. Forums are more user-friendly than the mailing lists for casual users, and are important to give the users and developers a feeling of belonging.

Design is also important at this point and should be emphasized heavily. Features that already exist must be made easily accessible, and the portal must be able to scale without ruining the design. Additionally, a log-in function should be provided. This is important to set access rights to the bugtracker, and to let outside developers advertise their names.

With this step, most of the important features are up and running. It will now be necessary to listen to the users for their feedback and make active use of the mailing-list and the forum. All internal discussions regarding the project should be posted on the mailing-list to include the external developers in the process. Moving on to the third iteration should be done when the community is growing larger. This should also be done when the outside developers are providing valuable contributions and should be given higher access
rights.

**Iteration 3**

At this step, a wiki should be provided to increase the documentation and support resources. Important participants in the community should be given higher access rights, i.e., given commit rights to the code repository. This would also be a suitable time for conducting automatic nightly builds that are uploaded to the download page as “unstable” builds. Scalability might become an issue at this point, and measures must be taken to avoid downtime and to make sure that all the services work properly.

### 7.6 Summary

The infrastructure is an important aspect of going open source. You have to include the users from the first day; giving them the opportunity to test out the product and feel included. To have an arena where the participants can communicate freely, is important to attract users and developers and to let outsiders feel that they belong to a community. It is important to build the portal solutions in iterative steps, offering extra qualities and possibilities when a new tool is launched. Lastly, it is important to use the portal actively and keep project-related discussions on the mailing-lists.
Chapter 8

Applying the Framework

One of the reasons for doing this project is to be able to advice the company Keymind about how to release Keywatch as open source. So far have we made some generic guidelines that everybody may follow, but we will now take those guidelines and apply them to Keywatch. The results will be a theoretical recommendation on what they should do to actually release Keywatch as open source; what considerations to make, what kind of resources they should have available, and which process they should have available.

8.1 Keymind

Information about Keymind Computing AS is found on their webpage: http://www.keymind.no.

8.1.1 About

Keymind is a small IT company based in Oslo and Lillehammer. They were founded in 1998 to build products and solutions for the Health Care business.

One of their main customer is the Cancer Register in Norway where they deliver a nationwide IT infrastructure for the Breast Cancer Screening program. They are responsible for the operation of the infrastructure and the software on both regional and centralized servers.

8.1.2 Reasons for Going Open Source

As a part of their operational services, they have developed a system for service monitoring called Keywatch. This system is developed using existing open source components and open standards.

Keymind is currently a part of the COSI project which has as a goal to create awareness of the use of open source software components and distributed development techniques in the industry. As a part of their work in COSI, Keymind has identified Keywatch as an suitable project to release as open source.

Keywatch is developed using the OSGi standard which is an open standard. OSGi is a component model that handle the lifecycle of the components it is set to handle.
8.2  Keywatch

The information about Keywatch and images are taken from their technical whitepaper, [Keymind Computing 2006].

8.2.1  Description

Keywatch is a system designed to monitor servers, applications, and networks. The main focus of Keywatch is to support information collection, analysis, and presentation. The value chain of monitoring can be seen in Figure 8.1

Monitoring systems have traditionally been large expensive systems that are hard to configure and hard to use. As the base components connected with distributed communication and other monitoring technologies have become commodity software, it is a technology that have reached a broader market. Smaller companies are now able to afford monitoring software as there are open source and free systems on the market. The differentiating technology today is concerned with the analytical and presentational features of the system.

Keywatch is designed as the backbone of a monitoring system. The system is built to be easy to use, install, and adapt into different situations.

8.2.2  Architecture

The Keywatch architecture is based on the OSGi component model that allows for an easy and lightweight model for system building. As we can see in Figure 8.2, the architecture of Keywatch is modular and easy to add functionality to.

Figure 8.1: The monitoring value chain

Figure 8.2: Architectural overview
Keywatch is supplied with plug-ins for different types of use.

**Event server**

The event server is the core of the Keywatch system. The events that are handled by the server are information about what happens in areas of concern and do all the information carrying throughout the system. Events are created by separate components that register with the event server and get access to manipulate, create, and delete events.

**Client**

Keywatch come with a client that connects to the system through a website using Ajax technology. By following this strategy, there is no need for installing anything extra to run Keywatch as the client run through any JavaScript enabled browser.

**XML-RPC Provider**

The XML-RPC provider give agents\(^1\) a connection point to Keywatch. The XML-RPC provider connects to the event server and create events based on the reports from the agents. There are currently implemented a Perl agent, and a Java OSGi agent.

**Rules**

The rules in Keywatch do the analyzing. When an event gets in, the rules are applied so that the necessary actions can be determined. The rules interact with the system as users of the event server.

### 8.3 Applying the Guidelines to Keywatch

#### 8.3.1 Keywatch Suitability as an Open Source Product

This is our evaluation of the suitability of Keywatch\(^2\) as an open source product based on our guidelines from Chapters 5, 6 and 7.

**Architectural Suitability**

**Is the architecture modular?**

- Yes

**Does the software adhere to open standards?**

- Yes

**Does the software contain any commercial components?**

---

\(^1\)Monitoring units on remote hosts

\(^2\)Note that this evaluation is based somewhat on the impressions we have gained, as we do not have access to the source code.
Does the software contain any open source components?

- Yes

Are the component’s licenses compatible?

- Yes

  Keymind has stated that the core of their system incorporates Hibernate and a relational database, which they prefer to be MySQL. Hibernate is licensed by LGPL and should not pose any problems as long as the linking is performed according the restrictions of the license. MySQL are using a dual-licensing strategy with a modified GPL license and proprietary ones. This modified GPL license allows for work to be combined with other open source software as long as they are licensed under a license adhering the open source definition. As long as this derived work can be reasonably considered as being independent and separate works in themselves, they do not fall under the terms of this modified GPL$^3$. The rest of the components are either internally developed or licensed under Apache 2.0.

The architectural suitability for release of open source seems to be fulfilled according to the criteria discussed in our guidelines.

Non-Architectural Suitability

Is the product runnable?

- Yes, the program is today both functional and runnable, fulfilling this first criterium and making it possible to evaluate for both users and potential developers.

Is the product of value to outside developers?

- Yes, in our consideration it seems of value to outside developers. According to Keymind [Keymind Computing 2006], other similar systems are either highly expensive proprietary solutions or open source solutions that are hard to configure and hard to use. Taken into consideration that the target domain of the software is the one with the highest amount of open source developers, Keywatch should be of interest to outside developers.

Is the project properly documented?

- The architecture is well documented. Regarding user-documentation, there are only help-files for installation purposes. We are not aware of how the API’s are documented, but we take such documentation for granted.

Is the product of general benefit?

- The product targets network administrators, and is suitable for all domains in which server, application, or network monitoring software is required. Such monitoring services are of necessity in many organizations, and therefore the project should be of general benefit.

Are there interesting tasks for outside developers to partake in with a well-defined extension mechanism?

$^3$The exception statement: http://www.mysql.com/company/legal/licensing/foss-exception.html
Keywatch encourages the open source community to develop plug-ins, although to our knowledge they are not at this date specifically suggesting specific plug-ins or other features in the system that needs improvement.

We think the non-architectural suitability of Keywatch is high. To increase the possibility of success even further, we recommend that they come with direct challenges regarding additional tasks that should be performed. They should highlight where functionality is lacking, come with suggestions for what plug-ins that can be developed and suggest areas where declarative programming might be necessary.

We also recommend that Keymind provides an additional Windows-friendly installation bundle to the extent this is possible without violating licensing agreements. Providing direct links for the necessary packages in a help-file is also recommended, and supported versions of these packages should be made clear in this help-file.

Finally, we suggest that the API documentation is provided easily accessible along with the software. More user-friendly documentation regarding installation procedures and how the program is used should also be provided, but we do not find this to be of necessity early in the release process.

8.3.2 Recommended Business Model(s)

Keymind is a small company which does not have any existing supplementary software for which release of KeyWatch will increase the value. By this falls one of the motivational factors of pursuing a loss-leader model.

The company do not possess full ownership of the entire code base since the software comprises several components. Even though the software initially seems like a good choice for the hybrid model, the lack of ownership means that a hybrid solution with dual-licensing is inviable (and also impossible with GPL). Still, the software is today only used internally in the company, and is not business critical or a major source for income. We therefore recommend a combination of the loss-leader model and the support seller model.

Support seller is an option they can pursue, as the software is quite technical, and its users might require professional technical support. We also suggest that they should pursue the loss-leader model in the form of developing plug-ins for commercial sale, or potentially a professional version of the software that incorporates certain additional features.

8.3.3 Recommended Choice of License

Based on the nature of the software and our recommended choice of business models, we will initially consider the license choice to be either the Revised BSD or the Mozilla Public License. However, the software uses both the Google Web-Toolkit and Jetty which are licensed under Apache 2.0, making the choice a little more difficult. Indeed, based on our general recommendations, we stated that software should be licensed under similar licensing models as complementing software. We thereby see no reason for using the revised BSD, as its benefits will to a large extent be neglected by the Apache Licensing used in part of the software. The Mozilla License might however also be beneficial in that it is more restrictive regarding how software can be used in proprietary solutions, and might therefore lead to more contributions being given back to the community.

Our final recommendation will thus, be to use the Apache License. This makes everything easier as most of the software sticks to only one license, while still remaining suitable for further commercial development.
8.3.4 Recommended Portal

Choosing a portal for Keymind depends mainly on what kind of ownership they want over the code. Keymind has said that they want to keep control of the development of Keywatch as they are going to use it themselves.

We would recommend building a portal based on existing tools. This is a relatively cheap solution that offers high adaptability to the needs of Keymind.

The reasons for recommending a self built portal is that it gives Keymind full control of what happens to their product. As they probably are going to be the major contributor to the code and are going to use the product actively in their daily use, it is natural to keep an as high level of ownership as possible.

When setting up the portal for the first time, we do recommend to just use the default skins and use the resources on fixing the functionality.

8.3.5 Recommended Tools

We would recommend Keymind to start up with a baseline of simple tools. The first step they should make is to just put up the code and product on their webpages with some information and a feedback form.

Within a short time they should start extending the functionality to include a bugtracker, mailing list, and an automated upload of source code and product. We would recommend using Bugzilla for bugtracking and what their service provider offers as mailing list to control the mailing lists. The first part of tool customization should also be done now to allow users to login and control both their bug account and mailing lists. There should probably be a separate database where all user information stored. It is a good idea to store more information than the current tools need so that the users would not have to add information when more tools are integrated.

This phase will require some resources installing and integrating the tools, but if the login is handled in a good way, this will save resources later on. For simplicity, the source code should be read by an internal client once a day and be posted automatically by ftp to the server. This reduce the internal resource needs as there is no need for a dedicated uploader, but still get a nightly build.

The next phase will add more features for the user. Our first suggestion is to install phpBB as this is a good solution for forum functionality. Installation of MediaWiki to handle documentation will be a natural next step.

As long as the login integration with Bugzilla was done in a clean way, it should not be to hard to extend this to phpBB and MediaWiki either. There will be some need for style changes of the tools, but the time needed for the integration is not going to be too long.

At this time, Subversion should also be made available. Exactly how to do this depends on the opportunities at the webhotel Keymind use. If there are no possibilities to run it on the webhotel, they will have to either run it internally in the company\(^4\) or find an external service that may host this tool.

When everything else is up and running, the users should be able to start up their own projects and extensions on the server. These extensions should follow some rules put down by a governing board, but it should not be many limitations in the beginning. These projects should get some features like a dedicated forum, bug handling, mailing list, and a Subversion tree. With the tools suggested so far, this should not be a problem to offer..

Requirements

The server that is used will need to support the following technologies:

- PHP 4

\(^4\)Not desirable as this opens the internal network for attacks
8.4 Summary

This chapter makes a theoretical consideration of Keymind and Keywatch within the guidelines of this report. We start by giving an overview of Keymind and Keywatch. We then apply the guidelines we have arrived at in our research to recommend Keymind in what business model and license to choose, and how to choose and set up a portal.

---

5To be able to run the necessary commands to install Bugzilla and set the access rights on files
Part III

Evaluation and Conclusions
Chapter 9

Project Evaluation

This chapter contains our evaluation of the project. We first evaluate the research questions, and whether we have provided an answer to these. Then we evaluate our contributions and results, before we go on discussing the validity of these results.

9.1 Evaluation of Research Questions

We have in our studies defined three research questions. We will in this section evaluate whether we reached an answer to these questions.

RQ 1 When should source code be released under an open license?
   RQ 1.1 What are the most suitable license(s)?
   RQ 1.2 What are the architectural requirements to the project?
   RQ 1.3 What are suitable business model(s)?
   RQ 1.4 What are the necessary properties of the product?

RQ 2 What deployment and advertising processes should be followed to achieve a successful open source project?
   RQ 2.1 What can be done in to attract contributors to a project, i.e., to create a community?
   RQ 2.2 How can the project be made known in the community?
   RQ 2.3 How should a company relate to an open source community?

RQ 3 What infrastructural requirements must be addressed when releasing open source code?
   RQ 3.1 Are there any requirements regarding what tools that must be provided?
   RQ 3.2 Should an external portal like SourceForge be used, or is it preferable to provide an own solution?

9.1.1 Research Question 1

The reasons to actually release your source code under an open source license may be many, although the wish of generating increased revenues will be a main motivation for a commercial actor. We provide information
about the possible business models early in the report. By adopting the ones that are suitable for a software
developing company, we have provided guidelines for choosing the most beneficial business model. We have
also described the most commonly used licenses and when a specific license should be chosen, relating these
licenses to the choice of business model and the profile of the company. We also provide a descriptions of
necessary architectural and non-architectural properties a project should provide. These can be used as a
checklist for identifying the project’s suitability for release as an open source project.

RQ 1.1
Finding the most suitable license(s) can be done by following these guidelines:

- Business model
  - **Support seller** GPL
  - **Loss leader** GPL, BSD, MPL
  - **Hybrid** GPL, BSD, LGPL

- Environment
  - **Competitors** GPL
  - **Complementary** Same license as environment

- Attracting users
  - **Unix based** GPL
  - **Commercial** MPL, BSD

You will have to adjust for other parameters like licenses already used on components, but these guidelines
should at least give a pointer.

RQ 1.2
The architectural requirements for an open source project can be summed up in the following questions:

- Is the architecture modular?
- Does the system adhere to open standards and open protocols?
- Are the components suitable?
  - Are any of the components commercial?
  - Are any of the components open source?
  - Are all licenses compatible?

If you can answer yes to the three main questions, the architectural requirements should be fulfilled.

RQ 1.3
Choosing a suitable business model depends on many factors. The following guidelines summarize some of
the considerations to make:

- **Company size**
  - **Large** Loss leader, Support seller, Hybrid
  - **Medium** Hybrid, Loss leader
Small Hybrid, Support seller

- Intended users

End use Loss leader, Hybrid

Commercial developers Hybrid

Commercial Support seller, Hybrid

RQ 1.4
We have found certain properties to be the most important, and they can be summed up in the following questions:

1. Non-architectural software properties
   - Is the product runnable?
   - Is the product of value to others?
   - Is the product properly documented?
   - Is the product of general benefit?
   - Are there interesting development tasks in the product?

2. Strategic considerations
   - Is the product commodity software?
   - Will you get a competitive edge by releasing open source?
   - Is the product non-crucial to your company’s income?

9.1.2 Research Question 2
To answer the second research question, we have reviewed the literature thoroughly. Based on the literature, we have identified commonalities in successful projects and their deployment processes. We have also used recommendations found in the literature, reviewed and cross-checked such recommendations with other studies. Combining these findings, we came up with guidelines for deployment processes. These describe how to make the project known within the community, how to attract the community to the project and how to relate to this community to keep a good relationship, i.e., achieving a win-win situation for all parties. Regarding the advertising processes, we found a lack of related studies of open source software and such processes. Thus, these guidelines might be lacking in validity and are subject for further studies.

RQ 2.1

RQ 2.2
To get people to know the project, there are many steps you can take. The following is an outline of the most important steps in the process.

1. Present the project
2. Create a proper webpage
3. Post the project on a major open source portal
4. Post the project on software portals
5. Notify news-sites
6. Use Wikipedia
7. Post newsletters

RQ 2.3

9.1.3 Research Question 3

The third question is related to the infrastructure that should be provided when releasing open source software. We found the answered this question by performing both a literature study and a direct evaluation of tools that should be provided for in such an infrastructure. Based on this, we provided the following descriptions regarding choice of portal solution(s) and tools.

RQ 3.1
There are certain possibilities a portal must provide. These are:

- Easily accessible download.
- Mailing list.
- Bug/Feature tracking.
- Proper information about the project.

There are also some tools a portal should provide, but are not as essential as the others.

- Forum.
- Documentation.

RQ 3.2
We have found that it is preferable for a company to use their own portal for hosting the project. Use of public portals should be done to make users aware of the product.

9.2 Evaluation of Contributions

Our main contribution in this project has been identifying what should be done to have a successful open source project. This comes in forms of certain guidelines and questions a company should go through, and measure up against their own situation. Each guideline is briefly discussed, providing a description of the guideline and why it should be followed.

Our first guidelines relate to identifying a suitable architecture. The guidelines are as follows:

- The architecture should be modular.
- The software should adhere to open standards and protocols.
- The suitability of components must be examined.
Then we go on describing suitable business models according to a variety of company profiles such as size, resources and their other projects. We describe when certain licenses should be used relating our suggestions to choice of business models.

- The licenses should be well known

Further, we provide a description of non-architectural properties that should be present, and provide the following checklist:

- The project should be runnable.
- The project should be of value to outside developers.
- The project should be properly documented.
- The project should be of general benefit.
- The project should identify interesting tasks for further development.

We then describe measures that can be made in order to create awareness of the project:

- Present the project at conferences.
- Create a proper web page.
- Post the project at major open source portals.
- Post the software for download at software portals.
- Use Wikipedia.
- Provide a news-letter.

Further, we analyze what options there are for choice of open source portals. We discuss maintenance, scalability, and server issues, and describe features and tools the open source portals must and should provide:

*Essential*

- Easily accessible download.
- Mailing list.
- Bug/Feature tracking.
- Proper information about the project.

*Recommended*

- Forum.
- Documentation.

Finally, we go through our guidelines, using them directly in a specific company case, Keymind, and their software planned for release as open source; Keywatch.
9.3 Evaluation of Validity

We have during the project tried our best to negate threats to the validity of our summary. Following is our evaluation of such threats, taken in retrospect.

9.3.1 Threats to Conclusion Validity

Inclusion of Poor Quality Studies
We believe that we quite effectively have negated this threat. We have reviewed the literature for objectivity and have almost exclusively used literature from well-recognized publications. Probably, we have not validated the sources regarding design and methodology to the extent we should have, meaning that some poor-quality studies might have been included in our research.

Citation Bias
Our literature selection process has to a large extent been based on heavily cited sources. This might be a threat to the validity of the project. Mainly, certain articles heavily cited are written by proponents of the FSF, which in our view probably is the largest risk in this regard.

Researcher Bias
To negate the risk of researcher bias, we have compared related literature and tried to identify biases. We have also tried to take an objective view, using more objective statements in our own project. Thereby, we do not believe the risk here lies directly within the objectivity of the literature we have studied, but rather in the impact such articles might have had on our own view. To negate this problem, the project has been reviewed both by ourselves and our supervisor, and hopefully this has eliminated such bias.

Improper Combination of Findings
We believe this threat has been negated as much as possible, given our experience as researchers. We have studied several sources to identify commonalities. Based on these commonalities we have provided our guidelines. Thus, if our combination of findings are improper, such should be due to unconscious bias influencing our guidelines.

9.3.2 Threats to External Validity

We believe that this threat has been properly negated. The total amount of studies we have examined and the literature reviewed, covers a wide variety of companies and populations. Thus, the external validity should be high. Still, a company will have to relate the guidelines to their specific situation, although such would have been necessary no matter what measures we had taken.

9.4 Summary

We have in this chapter evaluated the guidelines we have provided, how we have answered the research questions, and the validity of our results. We have come to the conclusion that given the limitations of our research approach and time available for conducting this study, the final results should be a beneficial reference for companies that are considering releasing their source code under an open source license.
Chapter 10

Conclusion and Future Work

10.1 Conclusion

The purpose of this project was to identify processes and infrastructural requirements when releasing source code under an open license. We performed a literature study and described characteristics and processes in open source development. With a basis in this description and the literature, we generated guidelines a company can use when releasing software under an open source license. These guidelines can be used to evaluate the suitability of software for release as open source. They will also help a company by describing considerations that should be made when choosing a business model and an open source license. We also identify aspects as to how to attract a community, non-architectural features that should be provided, and issues related to infrastructural requirements.

Our guidelines are then applied to a specific company and one of their products that is to be released as open source software. We evaluate the suitability of their product as open source software, and come with suggestions for choice of business model and license related to their specific situation.

10.2 Future Work

Our research has taken a theoretical approach to identification of open source processes. Although we believe our guidelines can be of help to companies considering to release software as open source, there is a need for validation of our guidelines. We here come with suggestions for future studies, and argumentation why such studies might be of necessity.

Verification of the Guidelines

Even though our guidelines in our opinion should be helpful to a company considering releasing software as open source, there might be a need for further verification. One suggestion is to perform surveys or qualitative interviews with several commercial actors to gain their opinions regarding the guidelines. Such approaches might help further specification of the guidelines as well as in identifying weak points or errors.

Applying the Guidelines

Our guidelines are based on the literature, but have yet to be applied. As the conclusion validity of our project might be lacking in certain areas, we suggest performing case studies. Such case studies should apply the guidelines to specific companies and situations in a real setting. They could be performed either over a short period of time, to identify how following the guidelines might impact initial contributions, as well as being of longitudinal character. The results should be measured against a baseline where a comparable
project has been performed. It would also be possible to experiment with the guidelines, using them as independent variables.

Marketing of Open Source Software
We identified a lack in the literature related to marketing of open source software. What we found mainly consisted of certain notions regarding motivational factors and that the spreading of open source software will come automatically if the product is of high enough quality and some infrastructure is provided. Although this might be true to a certain extent, it seems quite reasonable that further work should be done in this area. We believe that the following two approaches will be suitable to provide further answers.

- Conduct a large-scale survey identifying what measures commercial actors made related to marketing and how successful they were
- Perform case-studies and/or experiments where our guidelines regarding advertising processes are followed. This should be performed as longitudinal studies, whereas each guideline is used as independent variables and the results are measured.
Part IV

Appendix
Appendix A

Bibliography


Appendix B

Glossary

**Ajax**  Asynchronous JavaScript and XML - The technology combining JavaScript and XML to get dynamic webpages

**API**  Application Programming Interface

**ARPANET**  Advanced Research Projects Agency Network - Developed by the US Department of Defense as a military and research network - Predecessor of Internet

**ASF**  Apache Software Foundation

**CMS**  Content Management System

**COSI**  Co-development using inner & Open source in Software Intensive products

**COTS**  Commercial off the Shelf component

**cron**  Terminal program at Linux that runs commands at defined times

**CVS**  Concurrent Versions System

**DARPA**  Defense Advanced Research Projects Agency - Agency for the US Department of Defense responsible for the development of new technology

**DNS**  Domain Name Server - Backbone of the Internet, translating domain names to IP addresses

**FLOSS**  Free/Libre/Open Source Software - A definition of Open Source software that covers both the Free software of FSF and the Open-Source software of OSI

**Free software**  Definition used by FSF for software that can be used, copied, studied, modified and redistributed with little or no restriction.

**FSF**  Free Software Foundation

**Hackers**  Skilled programmers that create and modify programs

**IM**  Instant Messenging - A synchronized communication. MSN, AIM, and ICQ are popular tools for IM

**IPR**  Intellectual Property Rights

**IRC**  Internet Relay Chat - Protocol for communication in groups on internet

**ISP**  Internet Service Provider - Company that offers connection to the internet, or services on the internet
LAMP Linux, Apache, MySQL, and PHP/Perl - Popular setup of web servers

Linux Collection of Operating Systems based on the Linux kernel developed by Linus Torvalds

NCSA National Center for Supercomputing Applications

Open Source Software[OSS] The definition used by OSI to define what Open Source Software is

Open Standard A definition of a format or protocol that is available freely to anyone

OSI Open Source Initiative

OTS Off The Shelf component

POSIX An API that defines what is needed to be software compatible with UNIX

RFC Request For Comments - System for developing open standards, especially for use on the internet

SDK Software Development Kit

SotA State of the Art

SVN Subversion

tarball Archive of files packaged together with the tar program

UNIX An Operating System originally developed AT&T Bell Labs in the 1960s

webhotel A service that offers hosting of a website on the Internet

wiki A technology for allowing users to easily add, and modify the contents of a website

XML eXtended Markup Language
Appendix C

Guidelines

In this appendix will we provide some checklists based on our results in Chapters 5 and 6.

C.1 Making the Decision

1. Architectural software properties
   - Is the architecture modular?
   - Do the system adhere to open standards and open protocols?
   - Are the components used in the project suitable?
     - Are any of the components commercial?
     - Are any of the components open source?
     - Are all licenses compatible?

2. Non-architectural software properties
   - Is the product runnable?
   - Is the product of value to others?
   - Is the product properly documented?
   - Is the product of general benefit?
   - Are there interesting development tasks in the product?

3. Strategic considerations
   - Is the product commodity software?
   - Will you get a competitive edge by releasing open source?
   - Is the product non-crucial to your company’s income?

If you can answer yes to all of these, you should consider to go open source. If all strategic and architectural properties are met, you should really consider fixing up the non-architectural properties and go open source. Not all these points need to be available to go open source, but most of them should.
C.2 Choosing the Business Model

You should pick the type of business model best considering this list:

- **Company size**
  - **Large** Loss leader, Support seller, Hybrid
  - **Medium** Hybrid, Loss leader
  - **Small** Hybrid, Support seller
- **Intended users**
  - **End use** Loss leader, Hybrid
  - **Commercial developers** Hybrid
  - **Commercial** Support seller, Hybrid

C.3 Choosing License

Picking the license to use depend on many things. This is a short checklist to give an idea of which license to chose.

- **Business model**
  - **Support seller** GPL
  - **Loss leader** GPL, BSD, MPL
  - **Hybrid** GPL, BSD, LGPL
- **Environment**
  - **Competitors** GPL
- **Attracting users**
  - **Unix based** GPL
  - **Commercial** MPL, BSD

C.4 Advertising

This is a list of the most important steps when marketing the system.

1. Present the project
2. Create a proper webpage
3. Post the project on a major open source portal
4. Post the project on software portals
5. Notify news-sites
6. Use Wikipedia
7. Post newsletters
C.5 Portal

This is a summary of what tools a portal should provide: *Essential*

- Easily accessible download.
- Mailing list.
- Bug/Feature tracking.
- Proper information about the project.

*Recommended*

- Forum.
- Documentation.
Appendix D

Evaluation of tools

In this chapter we go through installation, configuration, and maintenance of many different tools used in open source projects. We have tried to evaluate as many parameters as possible, and have made a summary on each.

D.1 Practical matters

The tools we want to test are all tools used in open source communities today. We have tried to install at least one tool from each category, and we have also evaluated demos of others. Most of the tools are free of charge. We have also tested some proprietary solutions that cost money. We have not tried to actually integrate the tools with each other, but we have looked into such possibilities.

D.1.1 Test personnel

The testers of the installations are the authors of this paper. Neither of us have any extensive experience with installing anything on Linux, but we consider ourselves as accomplished computer scientists. We intend to use any resource available to help us with the installations.

We will list all resources used and all problems we had during the installation and running the software.

D.1.2 Hardware

The hardware specifications can be seen in Table D.1. The computer is not a powerful one, but as we plan to mainly perform feature testing, and not subject it to stress tests, the hardware will be adequate for our use.

<table>
<thead>
<tr>
<th>Type of equipment</th>
<th>Brand/size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motherboard</td>
<td>Pentium III (Coppermine) 1GHz</td>
</tr>
<tr>
<td>CPU</td>
<td>251 MB</td>
</tr>
<tr>
<td>RAM</td>
<td>10/100Mbit</td>
</tr>
<tr>
<td>Network interface</td>
<td></td>
</tr>
</tbody>
</table>

Table D.1: Server hardware
D.2 Operating Systems

D.2.1 Debian Linux

Linux is an open source operating system based on the kernel initially developed by Linus Torvalds. The development started in 1991, and the current kernel version is 2.6.19. The rationale for choosing the Debian distribution is that it was recommended by Linuxguiden\(^1\).

Installation

We downloaded Debian Linux\(^2\); specifically the network install\(^3\).

The initial installation was easy. We burned the downloaded cd-image on a cd, booted the computer from the cd, and selected what packages we wanted. We chose to install a graphical interface and the web packages.

Configuration

Setting up everything is done through a graphical interface. All our hardware were automatically detected and configured. We only discovered one problem: We did not have enough space in the main partition to install the selected packages. To fix this, we reinstalled on a larger partition.

Maintenance

The version of Debian that we downloaded was the currently stable one, Sarge. This was released in June 2005. To be a stable installation, means that the Debian team has frozen the packages they deliver with the system. This was a problem as Apache, PHP and MySQL were not up to date. We tried to fix this by updating to the newer, but currently unstable, version Etch. This did not solve our problems with Apache and PHP, but updated MySQL to the current release.

The update from Sarge to Etch was done by updating the file /etc/apt/sources.list to point to Etch instead of stable. We then had to run the package manager Apt to download the list of packages and install them.

Manual install

To get working versions of Apache and PHP, we had to install them manually. We got the tarball of Apache 2.2.3 from their webpage\(^4\) and installed it using the following commands:

```
./configure --enable-module=so --prefix=/opt/apache
make
make install
```

To install PHP, we got the tarball of PHP 4.4.4 from their webpage\(^5\) and installed it using the following commands:

```
./configure --with-mysql --with-apxs=/opt/apache/bin/apxs --with-zlib \
--enable-mbstring
make
make install
```

\(^1\)http://www.linuxguiden.no/index.php/Riktig_distribusjon - Norwegian (accessed 2006-11-15)
\(^2\)http://debian.org
\(^3\)http://www.debian.org/CD/netinst/
\(^4\)http://httpd.apache.org/download.cgi
\(^5\)http://www.php.net/downloads.php
Time

<table>
<thead>
<tr>
<th>Task</th>
<th>Time (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Installation</strong></td>
<td></td>
</tr>
<tr>
<td>Download</td>
<td>0.5</td>
</tr>
<tr>
<td>Initial install</td>
<td>0.5</td>
</tr>
<tr>
<td>Reinstall(^6)</td>
<td>0.5</td>
</tr>
<tr>
<td>Gnome + KDE</td>
<td>1</td>
</tr>
<tr>
<td>Updating to new version (Etch)</td>
<td>1</td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td></td>
</tr>
<tr>
<td>User setup</td>
<td>0</td>
</tr>
<tr>
<td>Hardware setup</td>
<td>0</td>
</tr>
<tr>
<td><strong>Maintenance</strong></td>
<td></td>
</tr>
<tr>
<td>Manual installs(^7)</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>5.5</td>
</tr>
</tbody>
</table>

Table D.2: Time requirements to install and configure Debian

Conclusion

Debian is a system that is quite easy to set up. A minimum of manual configuration is necessary to get it running. Debian is not a dedicated server system and is therefore probably not optimal as a production server. It was, however, more than adequate to test the packages we were interested in; and since all the packages can be used with any Linux distribution, there will not be any problems with compatibility.

D.3 Concurrent Versioning Systems

When writing a large, collaborative piece, no matter whether it is a program or a document, it is important that everybody involved has an updated version of every component. Revision control systems can be used for such purposes.

D.3.1 CVS

CVS is the tool for version control that have the most widespread use in the open source community. It is easy to use, easy to set up, and has proven capable of solve the version control problem. It is, however, growing old.

The success of CVS comes from a combination of its ease of use and the many projects that actually use it. We have used it while writing this report to keep track of our LaTeX-documents source files. CVS has performed this task well; giving us an opportunity to edit the same files without having to do the bookkeeping ourselves. We had a small problem setting up CVS, but that was due to problems with access permissions that were easily solved by creating the repository on another user account.

As CVS was first created in 1986, some of its original design choices are considered to be old fashioned and cumbersome. Problems that are often mentioned are: version numbering, versioning of files; moving or renaming of files, losing filehistory; limited support for Unicode, uses UTF-8 as this is normal on UNIX; and non-atomic commits, as versioning is on file, not tree.

\(^6\)Due to small partitions
\(^7\)Apache and PHP
D.3.2 Subversion

Subversion is meant as a replacement for CVS. Its commands are designed to resemble those of CVS so that the transition from CVS to Subversion should not be difficult. Subversion tries to fix some of the things that users considered problematic with CVS.

Setting up and running a project using Subversion, is done in a similar manner as CVS. Indeed, most of the principles used in CVS are kept in Subversion. The first major change can be seen in the version numbers. Where CVS works with separate version numbers for every file, Subversion will increase the version number of the whole tree at every commit. By doing this, it becomes easier to relate to a specific version of the entire source.

As Subversion is meant to target CVS users, there are tools for converting directly from a CVS repository to Subversion. This conversion will keep all the information from the CVS repository intact.

D.3.3 Other

There are other possibilities for version control than CVS and Subversion, both proprietary and open. Some that are worth mentioning are: Git, used for the Linux kernel; GNU Arch, used in the GNU project; and BitKeeper, proprietary, previously used for the Linux kernel. However, these are not commonly used on the larger portals, meaning that few developers are used to them.

D.3.4 Conclusion

When starting up a new project, Subversion will probably be the best choice. It has all the features of the well known CVS, but few of the bugs.

D.4 Documentation

D.4.1 LATEX

LATEX was originally designed by Leslie Lamport as a typographical system to ease the use of the TeX typographical system designed by Donald Knuth. Over the years, LATEX have ended up becoming more or less the de-facto standard for writing academic documents. The reasons for its widespread use are many; ease of use, the multitude of packages for typesetting of more or less anything, the stability of the design, the nice default setup, and the fact that it is free, have all helped in the distribution of the system.

The current version of LATEX, 2ε, has been stable since 1994. LATEX has some similarities with Linux considering how it is distributed; there exist many different distributions to handle package management and tools that come with LATEX. Most Linux variants ship with a version of LATEX, and it has also been ported to other operating systems.

All document development using LATEX, was originally done by writing standard text files in a normal texteditor. In the later years, editors with a graphical interface using LATEX as storage format have arrived. An example of this is LyX.

One of the major advantages of LATEX is that documents can be broken into pieces that are glued together by a main document. This results in a system where many developers can work simultaneously on different parts of the document and then glue them easily together. Changes to headers and other features are fixed by changing a definition at the start of the document. The actual formatting of documents is done by specifying how the document should look by manipulating the markup commands.
Most version control systems are able to run shell programs at commit. This can be taken advantage of by running \LaTeX\ when there is a new version of a file in the document. Considering the results from the compile, the version control system can accept or reject the changes. If the change is committed, can the version control system also put up the freshest version of the document as soon as it is made.

\LaTeX\ natively outputs to dvi, but there are also programs that make output in other formats. Some of the output formats that are normally included in a \LaTeX\ distribution are: pdf, rtf, HTML, png, ps, and gif, but not all packages and features are supported when exporting to these formats.

**Conclusion**

\LaTeX\ is a practical tool for creating technical documents. It makes it easy to cooperate with others, and is available free of charge.

**D.4.2 MediaWiki**

MediaWiki is probably the most commonly used wiki system. It is developed by WikiMedia Foundation for use in their wiki projects\footnote{Wikipedia, Wikinews, and WikiSource to mention some}. The reason for using MediaWiki for documentation is its easy editing and intuitive interface.

**Installation**

We downloaded the tarball for MediaWiki 1.8.2 from their webpage\footnote{http://www.mediawiki.org/wiki/MediaWiki}. This version of MediaWiki requires PHP5 which was not installed on the server. To fix this, we downloaded an older version of MediaWiki (v. 1.6.8) that supports PHP4.

The installation is web based and simple. The only problem we discovered came with setup of the database. After changing password of the root mysql user, everything worked. The reason for this problem was that MediaWiki does not allow an empty password for the root, even though the root account only was used to create the MediaWiki user.

**Configuration**

MediaWiki can be configured to a large extent. It is possible to change skins and images, and there are also many extensions that make it possible to change and add features to the wiki. It is possible to set editing rights at some or all of pages in the wiki.

After some searching, we have found extensions to MediaWiki that allows users of forums to use a unified login. We have found extensions for phpBB (see D.5.1), Invision Power Board (see D.5.2), and vBulletin (see D.5.3). The use of tools like these ease the integration of a webpage as less new code is needed, but they might make integration with other tools harder.
Requirements

<table>
<thead>
<tr>
<th>Serverside requirements</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>or</td>
</tr>
<tr>
<td>IIS</td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td>4.0</td>
</tr>
<tr>
<td>MySQL</td>
<td>4.0 or later</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>8.1 or later</td>
</tr>
</tbody>
</table>

Table D.3: MediaWiki 1.6.8 software requirements

<table>
<thead>
<tr>
<th>Serverside requirements</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>or</td>
</tr>
<tr>
<td>IIS</td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td>5.0 or later (5.1 recommended)</td>
</tr>
<tr>
<td>MySQL</td>
<td>4.0 or later</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>8.1 or later</td>
</tr>
</tbody>
</table>

Table D.4: MediaWiki 1.8.2 software requirements

Time

<table>
<thead>
<tr>
<th>Task</th>
<th>Time (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Installation</strong></td>
<td></td>
</tr>
<tr>
<td>Download</td>
<td>0.1</td>
</tr>
<tr>
<td>Initial install</td>
<td>0.2</td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td></td>
</tr>
<tr>
<td>User setup</td>
<td>0.1</td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td></td>
</tr>
<tr>
<td>Reading documentation</td>
<td>0.6</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1</td>
</tr>
</tbody>
</table>

Table D.5: Time required to install and configure MediaWiki 1.6.8

Conclusion

A wiki is a very practical tool for the creation of documentation. It is easy to set up and customize, and really easy to use. One of its major advantages is the possibility of linking together documents, making it easy to navigate within the system.

D.4.3 Code documentation

There are many tools that can be used for code documentation. Making developers adhere to them can be essential to keep the source code readable. Most of the available tools have virtually the same tags as they
follow the same design principle. Some of the tools enable the user to design their own tags, thus other aspects of the code may be documented.

Most code documentation tools have the advantage that they are command line programs. This means that a version control system may run an automated check on commit to see if the necessary code documentation is done.

**Javadoc**

Javadoc is the documentation tool that automatically is included with the Java SDK. It only support Java input, and HTML output.

**Doxygen**

Doxygen$^{10}$ is a generic tool that is able to read and output many formats. Doxygen can read its own, native documentation tags in addition to Java, Perl, and Qt tags.

**Conclusion**

Enforcing some form of code documentation is a smart thing. If an external developer needs to go into the source code of a module, he will probably use much more time finding the documentation he needs than if he can just look it up in a practically structured document.

### D.5 Communication

**D.5.1 phpBB 2.0.21**

phpBB is one of the most commonly used forum packages according to big-boards.com$^{11}$, and is also the only forum package on their top 20 list that is open source.

**Installation**

We began the installation with the downloading and unpacking of the tarball$^{12}$. The setup is done through an easy web-interface. During the install, we manually added a database and database user in our mysql database.

**Configuration**

The administration of phpBB is quite simple. The administration panel, Figure D.1a, is divided into logical parts with an intuitive interface and easy use.

It is easy to group users and set permissions. It is possible to modify all aspects of the user interfaces through templates, and adding, managing, and removing new forums.

---


$^{11}$According to [http://rankings.big-boards.com/?filter=all,all](http://rankings.big-boards.com/?filter=all,all) (accessed 2006-11-21)

MediaWiki has an extension\(^{13}\) that enables login to MediaWiki and phpBB with only one login. This does not include the creation of new users.

### Requirements

<table>
<thead>
<tr>
<th>Serverside requirements</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>or</td>
</tr>
<tr>
<td>IIS</td>
<td>or</td>
</tr>
<tr>
<td>PHP</td>
<td>4</td>
</tr>
<tr>
<td>MySQL</td>
<td>3.23 or higher</td>
</tr>
<tr>
<td>MSSQL Server</td>
<td>2000</td>
</tr>
<tr>
<td>MS Access</td>
<td>2000 &amp; XP</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>7</td>
</tr>
</tbody>
</table>

Table D.6: phpBB 2.0.21 software requirements

\(^{13}\)http://meta.wikimedia.org/wiki/PHPBB/Users_Integration (accessed 2006-12-05)
Time

<table>
<thead>
<tr>
<th>Task</th>
<th>Time (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installation</td>
<td></td>
</tr>
<tr>
<td>Download</td>
<td>0,1</td>
</tr>
<tr>
<td>Initial install</td>
<td>0,25</td>
</tr>
<tr>
<td>Set up</td>
<td>0,25</td>
</tr>
<tr>
<td>Configuration</td>
<td></td>
</tr>
<tr>
<td>User setup</td>
<td>0,1</td>
</tr>
<tr>
<td>Adding forums</td>
<td>0,25</td>
</tr>
<tr>
<td>Other</td>
<td></td>
</tr>
<tr>
<td>Documentation reading</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4</strong></td>
</tr>
</tbody>
</table>

Table D.7: Time required to install and configure phpBB 2.0.21

Conclusion

phpBB is one of the largest forum packages online. One of its main advantages is the fact that it is free. It is a feature-rich package that is definitely worth a try.

D.5.2 Invision Power Board

Invision Power Board (IPB)\(^{14}\) is one of the commercial alternatives to phpBB. IPB costs $69.95 for a yearly license, or $185 for a perpetual license.

We have not been able to test IPB as their online demo\(^{15}\) “will be up again in mid-october”, but from their official list of features\(^{16}\), they are offering more or less the same features as their competitors.

An advantage with IPB is that there already exist a tool to integrate the login between IPB and MediaWiki. IpWiki\(^{17}\) offers a modification to IPB and MediaWiki that allows users to log into both systems at the same time. This makes integration between IPB and MediaWiki easier, but may make it harder to integrate with other tools. IpWiki costs $25.

D.5.3 vBulletin

vBulletin\(^{18}\) is another of the commercial alternatives to phpBB. A yearly license cost $85, and an owned license costs $160.

vBulletin has a very rich panel for administration tasks. vBulletin gives you the opportunity to make forums subscribable.

There also exist tools for login integration between vBulletin and MediaWiki\(^{19}\).


\(^{15}\)http://community.ipslink.com/board/demo.html (accessed 2006-12-01)

\(^{16}\)http://community.ipslink.com/board/features.html (accessed 2006-12-01)

\(^{17}\)http://www.ipwiki.com/ (accessed 2006-12-05)

\(^{18}\)http://www.vbulletin.com/ (accessed 2006-12-05)

\(^{19}\)http://meta.wikimedia.org/wiki/VBulletin/Users_Integration (accessed 2006-12-05)
D.5.4 Forum of choice

Picking a forum depends on what kind of features you want for administration. All of the reviewed forums are capable of handling large communities and have a pretty easy administration that can do anything you need. To keep with the open source philosophy, we would recommend phpBB, but any of them will do fine.

D.6 Tracking

D.6.1 Bugzilla 2.22.1

Bugzilla is a tool developed by the Mozilla Foundation to keep track of the bugs in the Mozilla browser. It can also be used to track other things, such as features. The lifecycle of a bug in Bugzilla can be seen in Figure D.2. As you can see from this figure, it is enough flexibility in the system to allow for advanced bug handling.

Installation

Installation started by downloading Bugzilla from their webpage\(^{20}\). During the install, we had some problems with setting up privileges for running the server. To fix this, we had to modify `/etc/apache2/sites- enables/000-default` in the Apache configuration. We added a `<Directory>` setting for Bugzilla to set the

rights in order for Bugzilla to run as a cgi script. Information about how to do this came from SUSE Wiki\textsuperscript{21}. We added the following to the file:

\begin{verbatim}
Options +ExecCGI
AllowOverride Limit
DirectoryIndex index.cgi
\end{verbatim}

Configuration

The initial configuration of Bugzilla was easy and did not require much time. To administer everything was a more complex task. We have yet to find any description on how to integrate Bugzilla with the other tools we have considered. Considering that Bugzilla, as most other tools, use cookies for authentication, it should however be possible to emulate the login of Bugzilla using other tools.

A nice thing about Bugzilla, is that its setup is so general that it can be used to track other types of items. Bugzilla is, e.g., often used to track new features in a project.

Requirements

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Serverside requirements} & \textbf{Version} \\
\hline
Apache or IIS & \\
Perl Assorted Perl modules\textsuperscript{22} & 5 \\
MySQL or PostgreSQL & 3.23.41 or higher \\
\hline
\end{tabular}
\caption{Bugzilla 2.22.1 software requirements}
\end{table}

Time

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Task} & \textbf{Time (h)} \\
\hline
\textbf{Installation} & \\
Download & 0,1 \\
Initial install & 0,1 \\
Set up & 0,5 \\
\hline
\textbf{Configuration} & \\
User setup & 1 \\
\hline
\textbf{Other} & \\
Documentation reading & 9\textsuperscript{23} \\
\hline
\textbf{Total} & 11 \\
\hline
\end{tabular}
\caption{Time required to install and configure Bugzilla 2.22.1}
\end{table}

\textsuperscript{21}\url{http://susewiki.org/index.php?title=Setting_up_Bugzilla} (accessed 2006-12-03)
\textsuperscript{22}\url{http://www.bugzilla.org/docs/2.18/html/installation.html} (accessed 2006-12-09)
\textsuperscript{23}Due to finding the configuration to get it up and running
Conclusion

Bugzilla is a nice tool. Setup may be a bit tricky, but the power and flexibility is very good. The configuration is a problem as it requires shell access at the server to actually run. Unless you are running on your own servers, the necessary Perl modules will probably be troublesome to install. Choosing a bugtracker implemented in PHP might be desirable because of this.

D.7 Portal solutions

D.7.1 SourceForge EE

SourceForge EE\textsuperscript{24} is a system developed by VA Software. It is developed as an option for commercial companies that want the opportunities of SourceForge, but at same time require more security.

Installation

We downloaded the installation file from VA Software’s webpage\textsuperscript{25}. The complete package is a 537MB zip-file. To unpack it, we had to download an extraction program that handled \textit{zip} as none of the normal Linux packages does this\textsuperscript{26}.

After unpacking SourceForge, we had to install it. SourceForge EE runs on VMware, which is included in the package, so we had to install this first. This was done by a perl-script. During the install, we had to recompile some parts of the system. As we did not have the source code for the kernel we were running, we could not finish the install.

Requirements

\begin{table}[h!]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Serverside requirements} & \textbf{Version} \\
\hline
Red Hat & Enterprise Server 3.0, Enterprise Linux 4, or Advanced Server 3.0 \\
\hline
Oracle or PostgreSQL & 9.2.0.5 \hspace{1.5cm} 7.4.7 \\
\hline
JBoss & 3.2.6 \\
Tomcat & 4.1.29 \\
BEA WebLogic & 8.1.4 \\
\hline
\end{tabular}
\caption{SourceForge EE software requirements}
\end{table}

\textsuperscript{24}Enterprise Edition
\textsuperscript{25}http://www.vasoftware.com/sourceforge/
\textsuperscript{26}We used 7-Zip, http://www.7-zip.org
### Serverside requirements

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk space</td>
<td>3GB</td>
</tr>
<tr>
<td>RAM</td>
<td>1GB</td>
</tr>
<tr>
<td>Operating system</td>
<td>Windows XP/2003 or Linux</td>
</tr>
<tr>
<td>Processor</td>
<td>P4 1.5GHz or equivalent</td>
</tr>
</tbody>
</table>

Table D.11: SourceForge EE hardware requirements

### Time

<table>
<thead>
<tr>
<th>Task</th>
<th>Time (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Installation</strong></td>
<td></td>
</tr>
<tr>
<td>Download</td>
<td>1</td>
</tr>
<tr>
<td>VMware install</td>
<td>3</td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td></td>
</tr>
<tr>
<td>Install zip-unpacker</td>
<td>2</td>
</tr>
<tr>
<td>Misc</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>8</td>
</tr>
</tbody>
</table>

Table D.12: Time required to install SourceForge EE

### Conclusion

We gave up the install of SourceForge EE due to the problems we encountered installing VMWare. Although SourceForge EE might be a good solution for your company, we cannot recommend it based on our tests.

### D.7.2 eZ Publish

eZ Publish is an open source CMS system. It is mainly developed by eZ Systems\(^{27}\), but does also have external developers. The system is completely free to both download and use.

### Installation

We downloaded eZ Publish 3.8 from eZ Systems’ webpage\(^{28}\). The system came as a tarball that we unpacked. Installation and finetuning were done through a web interface where commands to set the owner of the folder and all other necessities were shown. Here too, we encountered a problem with setting up a database. This had to be manually created to support UTF-8. We found the information for fixing by going to their web pages and check the manual.

\(^{27}\)http://ez.no
\(^{28}\)http://ez.no/products/ez_publish
Requirements

<table>
<thead>
<tr>
<th>Serverside requirements</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>1.3</td>
</tr>
<tr>
<td>IIS</td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td>4</td>
</tr>
<tr>
<td>MySQL</td>
<td>3.23 or higher</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>7</td>
</tr>
</tbody>
</table>

Table D.13: eZ Publish software requirements

Time

<table>
<thead>
<tr>
<th>Task</th>
<th>Time (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Installation</strong></td>
<td></td>
</tr>
<tr>
<td>Download</td>
<td>0.1</td>
</tr>
<tr>
<td>Initial install</td>
<td>0.1</td>
</tr>
<tr>
<td>Set up</td>
<td>0.5</td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td></td>
</tr>
<tr>
<td>Add a package</td>
<td>0.3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1</td>
</tr>
</tbody>
</table>

Table D.14: Time required to install eZ Publish 3.8

Conclusion

The community connected to eZ is quite large. A large amount of components can be added to eZ Publish, providing good extensions to the original package. eZ Publish is a good framework to develop a portal around, but some of the tools needed to release code as open source are not available. This means that such tools will have to be developed from the ground, i.e., a bugtracker.
Appendix E

Portal overview

There are many tools that are used on open source portals out on the internet. We will here look into what tools is used in a few of these. We look into some portals that hosts many different projects, and some that are more specific.

<table>
<thead>
<tr>
<th>General portals</th>
<th>Forum</th>
<th>Bug tracker</th>
<th>Mailing list</th>
<th>CVS</th>
<th>SVN</th>
<th>License(s)</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceForge.net</td>
<td>Own</td>
<td>Own</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Any Open</td>
<td>Compile Farm, Project web, Shell, Databases, Statistics</td>
</tr>
<tr>
<td>Freshmeat.net</td>
<td>Own</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Any</td>
<td>Index of projects. Do not host any themselves</td>
</tr>
<tr>
<td>Apache.org</td>
<td>Mailing lists</td>
<td>Bugzilla</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Apache</td>
<td></td>
</tr>
<tr>
<td>Tigris.org</td>
<td>Own</td>
<td>IssueZilla</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Any Open</td>
<td>Hosts projects for collaborative software development</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Specialized portals</th>
<th>Forum</th>
<th>Bug tracker</th>
<th>Mailing list</th>
<th>CVS</th>
<th>SVN</th>
<th>License(s)</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mozilla.org</td>
<td>Newsgroup</td>
<td>Bugzilla</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>MPL</td>
<td></td>
</tr>
<tr>
<td>eZ.no</td>
<td>Own</td>
<td>wIT</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>eZFul, GPL</td>
<td></td>
</tr>
<tr>
<td>MySQL.com</td>
<td>phorum</td>
<td>bugs.php</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Dual: Proprietary, GPL¹</td>
<td></td>
</tr>
</tbody>
</table>

Table E.1: A overview of the tools available at different portals

¹with FLOSS License Exception