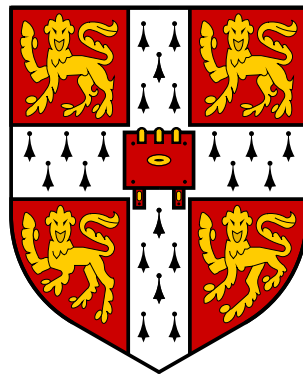


Quality Improvement in Volunteer Free and Open Source Software Projects

Exploring the Impact of Release Management



A dissertation submitted to the University of Cambridge
for the Degree of Doctor of Philosophy

Martin Michlmayr <tbm@cyrius.com>
King's College

March 2007

Centre for Technology Management
Institute for Manufacturing
University of Cambridge

Quality Improvement in Volunteer Free and Open Source Software Projects

Exploring the Impact of Release Management

Martin Michlmayr

Free and open source software has had a major impact on the computer industry since the late 1990s and has changed the way software is perceived, developed and deployed in many areas. Free and open source software, or FOSS, is typically developed in a collaborative fashion and the majority of contributors are volunteers. Even though this collaborative form of development has produced a significant body of software, the development process is often described as unstructured and unorganized. This dissertation studies the FOSS phenomenon from a quality perspective and investigates where improvements to the development process are possible. In particular, the focus is on release management since this is concerned with the delivery of a high quality product to end-users.

This research has identified considerable interest amongst the FOSS community in a novel release management strategy, time based releases. In contrast to traditional development which is feature-driven, time based releases use time rather than features as the criterion for the creation of a new release. Releases are made after a specific interval, and new features that have been completed and sufficiently tested since the last release are included in the new version.

This dissertation explores why, and under which circumstances, the time based release strategy is a viable alternative to feature-driven development and discusses factors that influence a successful implementation of this release strategy. It is argued that this release strategy acts as a coordination mechanism in large volunteer projects that are geographically dispersed. The time based release strategy allows a more controlled development and release process in projects which have little control of their contributors and therefore contributes to the quality of the output.

Preface

Except for commonly understood and accepted ideas, or where specific reference is made, the work reported in this dissertation is my own and includes nothing that is the outcome of work done in collaboration. No part of the dissertation has been previously submitted to any university for any degree, diploma or other qualification.

This dissertation consists of approximately 60,000 words, and therefore does not exceed the 65,000 word limit put forth by the Degree Committee of the Department of Engineering.

Martin Michlmayr

March 2007

Contents

Abstract	ii
Preface	iii
1. Introduction	1
1.1. A Definition of FOSS	1
1.2. Problems with the FOSS Process	5
1.3. Contribution of this Dissertation	7
1.4. Structure of the Dissertation	7
2. Quality Management	9
2.1. Quality Literature	9
2.1.1. Definitions of Quality	9
2.1.2. Process Improvement	12
2.2. FOSS Literature	13
2.2.1. Quality Considerations	13
2.2.2. Development Style	15
2.2.3. Challenges and Problems	19
2.2.4. Summary	21
2.3. Exploratory Study	22
2.3.1. Methodology	22
2.3.2. Comparison of Proprietary Software and FOSS	23
2.3.3. Development and Quality Practices	24
2.3.4. Quality Problems	27
2.3.5. Discussion	30
2.4. Refining the Scope	31
2.5. Chapter Summary	34

3. Release Management	35
3.1. Literature	35
3.1.1. Software Maintenance	36
3.1.2. Release Management in FOSS	38
3.2. Exploratory Study	40
3.2.1. Methodology	40
3.2.2. Types of Release Management	41
3.2.3. Preparation of Stable Releases	42
3.2.4. Skills	47
3.2.5. Tools and Practices	48
3.2.6. Problems	50
3.2.7. Discussion	52
3.3. Theory	53
3.3.1. Modularity and Complexity	54
3.3.2. Coordination in FOSS	55
3.3.3. Coordination Theory	56
3.4. Chapter Summary	59
4. Research Question and Methodology	61
4.1. Research Question	61
4.2. The Case Study Approach	62
4.3. Selection Criteria	64
4.4. Case Study Projects	68
4.5. Sources of Evidence	71
4.5.1. Interviews	73
4.5.2. Mailing Lists	73
4.5.3. Documents	74
4.5.4. Observation	75
4.6. Methodological Considerations	75
4.6.1. Construct Validity	75
4.6.2. Internal Validity	76
4.6.3. External Validity	76
4.6.4. Reliability	77
4.6.5. Personal Involvement	77

4.6.6. Ethical Issues	78
4.7. Chapter Summary	79
5. Time Based Releases: Learning from 7 Case Studies	80
5.1. Debian	80
5.2. GCC	85
5.3. GNOME	88
5.4. Linux	92
5.5. OpenOffice.org	97
5.6. Plone	100
5.7. X.org	102
5.8. Chapter Summary	105
6. Release Strategy	106
6.1. Problems Prompting a Change	106
6.1.1. Lack of Planning	107
6.1.2. Problems Caused by Lack of Planning	109
6.1.3. Long-term Issues	112
6.1.4. Summary	113
6.2. Time Based Strategy as an Alternative	113
6.2.1. Conditions That Have to be Met	113
6.2.2. Time Based Releases as a Coordination Mechanism	120
6.2.3. Advantages of Time Based Releases	129
6.2.4. Open Questions	133
6.3. Chapter Summary	136
7. Schedules	138
7.1. Choice of the Release Interval	138
7.1.1. Regularity and Predictability	138
7.1.2. User Requirements	140
7.1.3. Commercial Interests	141
7.1.4. Cost Factors Related to Releasing	143
7.1.5. Network Effects	146
7.1.6. Summary	148
7.2. Creation of a Schedule	148

7.2.1. Identification of Dependencies	149
7.2.2. Planning the Schedule	150
7.2.3. Avoiding Certain Periods	152
7.3. Chapter Summary	153
8. Implementation and Coordination Mechanisms	154
8.1. Implementing Change	154
8.1.1. Incentives for Change	154
8.1.2. Issues of Control and Trust	155
8.1.3. Implementation of Control Structures	157
8.1.4. Summary	160
8.2. Policies	160
8.2.1. Stability of the Development Tree	160
8.2.2. Feature Proposals	162
8.2.3. Milestones and Deadlines	164
8.2.4. Summary	165
8.3. Coordination and Collaboration Mechanisms	165
8.3.1. Bug Tracking	166
8.3.2. Communication Channels	169
8.3.3. Meetings in Real Life	172
8.3.4. Summary	173
8.4. Chapter Summary	173
9. Discussion and Conclusions	175
9.1. Discussion	175
9.2. Key Learning Points	178
9.3. Limitations and Future Research	179
9.4. Conclusions	184
Acknowledgements	185
Glossary	186
A. Exploratory Studies	188
A.1. Quality	188
A.1.1. Questions	188

A.1.2. Projects	188
A.1.3. Consent	189
A.2. Release Management	189
A.2.1. Questions	189
A.2.2. Projects	191
A.2.3. Consent	192
B. Case Studies	194
B.1. Interviews	194
B.1.1. Questions	194
B.1.2. Projects and Participants	195
B.1.3. Consent	197
B.2. Mailing Lists	197
B.3. Conference Presentations	198
Bibliography	200

1. Introduction

Free and open source software has had a major impact on the computer industry since the late 1990s and has changed the way software is perceived, developed and deployed in many areas (Vixie 1999). Free and open source software, or FOSS, can be seen as a disruptive technology that has changed the rules according to which the industry works (Christensen 1997). The sudden success and major adoption of this new and innovative software development strategy has raised many questions, attracted the interest of academics in a variety of disciplines and prompted interdisciplinary research (Feller et al. 2005; von Krogh and von Hippel 2003).

1.1. A Definition of FOSS

Even though the major breakthrough of FOSS came very rapidly in the late 1990s, the underlying ideas of communities which cooperatively develop and share software and its source code can be traced back several decades to the origin of modern computers (Levy 1984). The origins of the present FOSS movement are most commonly attributed to Richard M. Stallman, who in 1984 started the GNU Project and later a supporting organization, the Free Software Foundation (Williams 2002).

Every definition of FOSS faces several terminological and conceptual ambiguities which are connected to free and open source software. The term FOSS and similar terms such as free software, open source and libre software are used to refer to a number of different concepts and characteristics (Gacek and Arief 2004). One perspective focuses on FOSS from a legal point of view and investigates FOSS as a license model for software distribution. Unlike traditional software, which is inherently associated with the protection of proprietary rights, FOSS encompasses a class of licenses which give the user a

number of rights and permissions (O’Sullivan 2002).

The movement Stallman initiated led to the creation of the Free Software Definition which postulates that a license has to grant users certain rights in order for the software to classify as free software (Stallman 1999). Most importantly, such software guarantees unrestricted use, access to the source code, and the right to modify and to distribute this source code (and software generated from the original or modified source code). It is important to note that free software is “a matter of liberty, not price” (Stallman 1999; Stallman, Lessig, and Gay 2002).

Citing the ambiguity of the word ‘free’ in the English language, a group of people founded an organization in the 1990s and coined the alternative term open source (Perens 1999). Their Open Source Definition is largely compatible in spirit with Stallman’s Free Software Definition but their emphasis is more pragmatic. While free software is associated with a strong philosophical focus on freedom, adherents of the open source movement stress features of the software, such as high quality. Despite this philosophical difference in the two movements, the licenses, software and most importantly the development process are largely the same. Because of this, the all encompassing term free and open source software, or short, FOSS, will be used throughout this dissertation.

A second perspective on FOSS focuses not so much on the philosophical foundation or license terms of the software but on the software development process employed to create the software (Raymond 1999). While there is now considerable evidence against the idea that all FOSS projects follow the same process, given that there are differences in leadership styles (Mockus, Fielding, and Herbsleb 2002), quality assurance (Aberdour 2007), and other areas, there are common patterns among FOSS projects. Fundamentally, FOSS is a collaborative innovation process in which members from all over the world can participate. Traditionally, the majority of participants were unpaid volunteers working together over the Internet. Lerner and Tirole (2002) have identified the Internet as an enabler of this form of collaborative software development and it is therefore not surprising that large portions of the Internet infrastructure itself rely on FOSS (Mockus, Fielding, and Herbsleb 2002).

While FOSS software, defined by the license covering the software, can be

	Distributed	Co-located
Volunteers	Prototypical FOSS e.g. Perl	'sprints' and 'hackathons' e.g. Zope and Apache
Non-volunteers	Virtual work teams e.g. Ximian	Traditional workplace e.g. MySQL

Table 1.1.: Different types of FOSS organizations (Howison 2005).

— and sometimes is — developed in a traditional setting in which paid developers are co-located (Howison 2005), the classical model in which volunteers collaboratively develop and share software has attracted most interest from researchers (see table 1.1 for an overview of organizational models). This software development paradigm is associated with a user innovation process in which users actively become involved and drive the development (von Hippel 2005; Lakhani and von Hippel 2003). While not all users become actively involved (Daniel and Stewart 2005), there is a significant number of users who contribute to a range of areas, such as documentation and support (Crowston and Howison 2005; Sowe et al. 2007).

Mockus, Fielding, and Herbsleb (2002) argue that the collaborative innovation process employed by FOSS development groups has led to the production of software of significant quality and functionality. A cost estimation performed on a large collection of FOSS led to an estimated value of almost \$1.9 billion USD (González-Barahona et al. 2001) and the system has more than doubled in size since the study was carried out (González-Barahona et al. 2004). In February 2007, IDC predicted that the Linux ecosystem would be worth 40 billion dollars by 2010.¹ FOSS is now used in many areas, such as the Internet where, as an example, Apache holds the highest market share of web servers (Mockus, Fielding, and Herbsleb 2002), and operating systems based on Linux have become an attractive alternative to other systems (Moody 1997; Torvalds and Diamond 2001). There is also growing adoption in developing countries and other areas (Kshetri 2004).

Given the major economic impact that FOSS has had, it is not surprising that there is growing interest in understanding and utilizing the process underlying FOSS development. Lerner and Tirole (2002) have identified the

¹<http://www.internetnews.com/dev-news/article.php/3659961>

following three factors which have spurred the interest in the FOSS development process:

- The rapid diffusion of FOSS: within a very short time frame, various FOSS applications, such as Apache and Linux, have attained major market share and often dominate their product categories. According to Lerner and Tirole (2002), it is estimated that “Linux has between seven to twenty-one million users worldwide, with a 200% annual growth rate”.
- The significant capital investments in FOSS projects: major players in the IT sector, such as IBM and HP, have invested billions of dollars in FOSS development. Furthermore, companies like Red Hat which commercialize Linux have attained considerable profit (Lerner and Tirole 2002). MySQL, the provider of a FOSS database, has been very successful on the market (Garzarelli and Galoppini 2003).
- The new organizational structure: the collaborative innovation process employed by FOSS projects is often seen as a major organizational innovation (Lerner and Tirole 2002; von Hippel 2005; Mockus, Fielding, and Herbsleb 2002).

These factors show that the FOSS phenomenon deserves rigorous exploration and investigation. In the context of this dissertation, the focus is on the last of these three points, the new organizational structure employed by FOSS projects.

In recent years, increasing attention has been given in the literature to the identification of new processes in software production and other areas of technology that allow organizations to cope with rapid change (MacCormack, Verganti, and Iansiti 2001; Redmill 1997). Unlike past decades in which requirements were fairly stable, software companies are now facing a dynamic environment in which the pace of change is significantly higher than in the past. While many new strategies have been tested, including collaborative development using virtual teams, they often faced problems because of distance (MacCormack, Verganti, and Iansiti 2001; Rasters 2004).

Despite these failures of traditional companies to employ the Internet for rapid development and collaboration, the FOSS community has produced a significant body of software and has thereby proved that some forms of collaborative development are viable. In fact, some argue that FOSS poses a real challenge to traditional software companies (Vixie 1999). Because of the sheer success of the FOSS methodology, there is significant interest in applying the FOSS process to software sectors where it has not been used before, such as Air Traffic Management (Hardy and Bourgois 2006), other areas of technology and science (Kelty 2005) and even to other creative areas of production, such as the collaborative encyclopedia Wikipedia (Benkler 2002; Stalder and Hirsh 2002; Coffin 2006).

1.2. Problems with the FOSS Process

In the course of the debate whether the FOSS process can be applied to other areas, and given that FOSS has succeeded where many companies before have failed (MacCormack, Verganti, and Iansiti 2001), the question has been posed whether — or in which ways — the FOSS development methodology is different from other processes. While some researchers criticize FOSS practitioners for regarding themselves as too special and thereby ignoring useful insights from traditional software engineering (Massey 2003), there is considerable evidence that some parts of the FOSS process and organization are unique (Lerner and Tirole 2002; Aberdour 2007).

Mockus, Fielding, and Herbsleb (2002) have identified the following four differences from traditional software development which are most commonly cited:

- FOSS systems are built by potentially large numbers of volunteers.
- Work is not assigned; people undertake the work they choose to undertake.
- There is no explicit system-level design, or even detailed design.
- There is no project plan, schedule, or list of deliverables.

Rather than being the panacea that some have claimed FOSS to be, these factors appear to be a plan for failure. This seemingly unstructured and unorganized nature of FOSS development, depicted by Mockus, Fielding, and Herbsleb (2002) and supported by Zhao and Elbaum (2003) who found that most projects operate in an ad-hoc fashion, goes against traditional views that you need well planned and organized processes (Zahran 1997), in particular when development is geographically distributed (Herbsleb and Grinter 1999).

While the FOSS methodology has been used to develop a significant body of software (Mockus et al. 2002; González-Barahona et al. 2001), some with high quality (Halloran and Scherlis 2002; Schmidt and Porter 2001; Aberdour 2007), there have been major changes in the expectations of FOSS in recent years. FOSS is no longer seen as hobby projects done for fun (Torvalds and Diamond 2001), but significant economic factors are involved now. This increased economic interest in FOSS with millions of users and thousands of organizations relying on this collaboratively developed software has led to new requirements, such as the need for sustainability and reliance (Weber 2005; Fitzgerald 2006).

These new requirements may prompt a change of the FOSS process from an unplanned, ad-hoc style to a more organized development methodology. Such a change may be necessary to address certain problems and challenges related to the volunteer nature of FOSS which have been raised in recent years, such as the lack of deadlines (Garzarelli and Galoppini 2003) or problems with reliance on volunteer participants (Michlmayr and Hill 2003). Furthermore, geographically distributed development is associated with a set of problems related to distance, such as communication (MacCormack, Verganti, and Iansiti 2001; Rasters 2004). Finally, there is some evidence that the combination of the volunteer nature of FOSS with globally distributed development leads to unique problems that need to be addressed (Michlmayr 2004). For example, there may be problems due to the reliance on individual participants in volunteer projects. This issue is further complicated by the distributed nature because it makes it difficult to identify volunteers who are neglecting their duties, and to decide where more resources are needed.

1.3. Contribution of this Dissertation

This dissertation will explore how — and, more fundamentally, whether — FOSS teams consisting primarily of volunteers and collaborating over the Internet can produce not only software of high quality but also sustain their development so other organizations can rely on their products. Can volunteer teams, with their high volatility regarding project collaborators (Robles, Gonzalez-Barahona, and Michlmayr 2005), ensure consistent levels of quality in their output?

These are questions of coordination and management, an area of study that has so far received little attention in FOSS research (Iannacci 2005). This dissertation will therefore focus on organizational structures in FOSS projects in order to identify how projects can be managed and coordinated so consistent levels of quality can be ensured. In particular, the focus will be on release management, an important part of quality management (Levin and Yadid 1990) since it is concerned with the delivery of high quality software to end-users. Release management requires significant coordination efforts, and given the shortcomings in this area in FOSS projects that have been pointed out in the literature (Mockus, Fielding, and Herbsleb 2002; Iannacci 2005), the question how distributed volunteer teams can coordinate their work to perform high quality releases effectively is highly relevant.

1.4. Structure of the Dissertation

Following this introduction, chapter 2 will focus on quality aspects of FOSS. This issue will be investigated from a quality management perspective according to which management and coordination structures employed by FOSS projects to ensure and maintain high levels of quality will be taken into consideration. In addition to an investigation of processes used to ensure quality, an emphasis will be put on quality problems that remain unresolved.

Since the subject of investigation of this dissertation is fairly new and unexplored, chapters 2 and 3 include both a discussion of the literature as well as an exploratory study. The aim of these two explorations is to identify issues that are of high relevance and they will guide the focus of this dissertation.

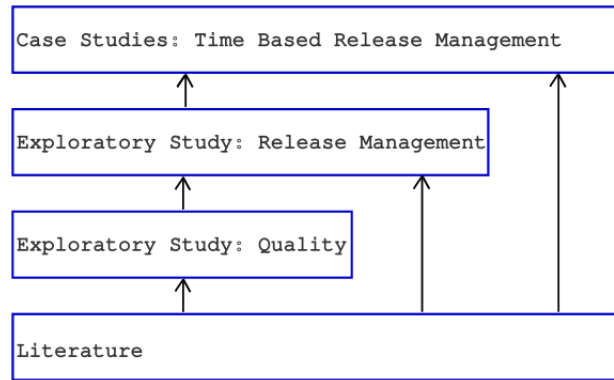


Figure 1.1.: Literature and two exploratory studies have led to in-depth case study research from which conclusions are drawn.

The study from chapter 2 explores quality practices and problems and concludes with the identification of three problematic areas that are interesting for further research. One of them, release management, will be chosen as the specific area under investigation in this dissertation about which conclusions will be drawn, both specifically about release management and more generally about quality management and coordination in FOSS.

Chapter 3 first explores the requirements of release management in a dynamic environment. This is followed by an exploratory study with members of FOSS projects about practices and problems related to release management. The findings from this study are subsequently related to literature about organizational complexity and modularity. The chapter concludes with the presentation of a theory on coordination developed by Malone et al. (1993) which will be used as a framework to discuss findings from the case studies of this dissertation.

Chapter 4 presents the research question, methodology and case studies of this research. It is followed by an individual presentation of each of the seven case studies. The following three chapters present findings from a cross-case analysis of these case studies: chapter 6 focuses on release strategy, chapter 7 discusses factors influencing the creation of a release schedule, and chapter 8 looks at the actual implementation of release management in the case study projects. The final chapter summarizes the major findings of this work and the contribution to knowledge made in this dissertation. Moreover, it will raise new questions which would benefit from being investigated in further research.

2. Quality Management

This chapter discusses important literature related to quality management and presents the findings from an exploratory study on quality in FOSS projects. The results from this study are used to guide the present research. The chapter is divided into the following four sections:

1. Quality literature: traditional views of quality are discussed and definitions of quality are investigated. The view of quality followed in this dissertation, process improvement, is presented.
2. FOSS literature: papers related to development practices and quality in FOSS projects are discussed as a basis for further investigation and to show gaps in the literature.
3. Exploratory study: findings from an exploratory study related to quality practices and problems in FOSS projects are presented.
4. Refining the scope: the findings from the exploratory study are used to focus on a particular topic for further investigation.

2.1. Quality Literature

2.1.1. Definitions of Quality

Quality is a very interesting concept to study — many people have an intuitive feeling of what quality entails, but when it comes to defining quality in a precise manner it turns out that the concept is very hard to pin down. In a way, quality is similar to the ability of riding a bike. People do it all the time but when they are asked to give a thorough explanation of this ability they realize how many different steps are involved and that they are not actually consciously aware

of them. Similarly, many people will claim that they know quality when they see it, but they would not be able to give a precise definition or describe the decision making process employed to make their assessment (Dale and Bunney 1999). Furthermore, people will often disagree about the quality of a specific object or service, which shows that quality is not the same for everyone.

This multifaceted and subjective nature of quality makes it hard to define. Quality originates from the Latin word ‘qualis’ which roughly means ‘such as the thing really is’ and one definition describes it as the “totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs” (Dale and Bunney 1999). There are a number of different definitions of, and approaches to, quality, stemming from different perspectives and taking different aspects of this concept into account. The two most common definitions of quality are:

- **Fitness for purpose:** one popular approach to solving the problem of defining quality is to assume that a product is of high quality if it is fit for its purpose (Coleman and Manns 1996). If a piece of software does what it is supposed to do, it fulfils fitness for purpose and can therefore be seen as high quality software. There are different approaches to measuring fitness for purpose but one common way of exploring this concept is by taking the specification as a starting point. Before a piece of software is created, the requirements can be gathered and written down in a specification. Hence, in an ideal world, such a document can be used to measure quality in a very easy way: if the software adheres to the specification it is of high quality. Unfortunately, this approach, known as conformance to specification, raises several issues. In particular, it assumes that the specification itself is of high quality but does not describe how this can be achieved or ensured. In many cases, the specification will lack important parts which later turn out to be fundamental to quality or it will simply be wrong.
- **Quality attributes:** another way to approach quality is to start from a high-level, user-oriented perspective and define different attributes of quality. For example, Boehm is known for this approach (Boehm et al. 1978). A possible list of attributes might include efficiency, reliability, us-

Attribute	Definition
Functionality	Functions and their specified properties
Reliability	The capability to maintain a level of performance under stated conditions over time
Usability	The effort needed for use
Efficiency	The relationship between the level of performance and the amount of resources used
Maintainability	The effort needed to make specified modifications
Portability	The ability of software to be transferred from one environment

Table 2.1.: Quality attributes defined by ISO 9126.

ability, extendability, portability, testability, understandability, reusability, maintainability, interoperability, and integrity. Related to Boehm's definition of quality is ISO 9126 which provides a list of characteristics of quality as a starting point for a quality evaluation process (ISO 1991). ISO 9126 defines the following six characteristics: functionality, reliability, usability, efficiency, maintainability and portability (see table 2.1 for a definition of each characteristic).

These two definitions of quality raise interesting questions when applied to FOSS. The first approach, in which conformance to specification is taken as a measure of quality, cannot directly be applied to FOSS projects which usually do not have an explicit design or a written specification (Mockus, Fielding, and Herbsleb 2002). Nevertheless, the idea of fitness for purpose applies to any product, including FOSS. The second definition shows that many different factors have to be taken into consideration for quality. This holistic approach may be in conflict with the priorities of many FOSS developers, who, for example, put more emphasis on maintainability than on usability, as will be seen later.

As of this writing, there are few metrics and tools which allow the measurement of the quality of a FOSS product. The EU acts as sponsor for a number of projects, including QualOSS and SQO-OSS, whose aim it is to create such quality assessment tools. Due to the lack of quantitative assessment tools, researchers often use success factors, such as the number of downloads, as indicators that a piece of software may be of high quality (Howison and Crowston 2004; Michlmayr 2005). After all, users have to download and in-

stall FOSS themselves and they will usually choose software which fulfils their needs and quality expectations. While success and quality are certainly not the same (Crowston, Howison, and Annabi 2006), it is important to take into consideration in the following sections that many researchers use success as an indicator for quality given the lack of good metrics.

2.1.2. Process Improvement

As discussed above, quality is multifaceted and therefore there are many approaches to quality improvement. The now predominant view of quality improvement is process thinking. It stresses the importance of taking into account the whole process which is employed to build and maintain a product and to find ways to improve this process. According to this view, a high quality product can only be created through a high quality process.

This view of quality improvement became popular because of the support of several ‘quality gurus’. Philip Crosby is usually associated with the concepts of ‘do it right the first time’ and ‘zero defects’, a paradigm which promotes process improvement over defect removal (Crosby 1979). W. Edwards Deming is a major advocate of quality control. His philosophy is that “quality and productivity increase as ‘process variability’ (the unpredictability of the process) decreases” (Slack et al. 1995, p. 812). By improving the process, variability in the output, such as low levels of quality, are reduced. Finally, Armand Feigenbaum stresses quality management and improvement is the responsibility of every single member of an organization. According to Feigenbaum, major “quality improvement can only be achieved in a company through the participation of everyone in the workforce” (Dale and Bunney 1999, p. 47). This view is compatible with process improvement which takes all aspects of the process into account, including developers and other people participating in the development process.

Process thinking stresses the importance of having complete processes and following the procedures and processes in a disciplined fashion. This will decrease the variability in observed quality. Zahran (1997) compares non-disciplined and disciplined processes and suggests that non-disciplined are chaotic while disciplined processes are mature. When processes are chaotic,

activities are not aligned towards a common goal, the results of individual activities could nullify each other, and the total capability of the group is minimized. On the other hand, when a mature process is reached, activities are aligned, the result of each individual strengthens the other, and the total capability of the group is maximized.

Even though process thinking and improvement is the dominant approach to quality improvement, it is not the only view of quality improvement. Sommerville (2001) points out that people, tools and budgets can have a significant impact on software quality too. In this dissertation, the emphasis will be put on process improvement. While people certainly play an important role it is hard to regulate who becomes involved in a volunteer project. Tools can be seen as part of the development process, and budget considerations are usually not a matter of concern in volunteer projects which often operate without any financial budget. Hence, the dissertation will follow the process improvement view with the aim to find problems with processes employed by FOSS projects, to identify ways to improve these processes, and to find best practices.

2.2. FOSS Literature

2.2.1. Quality Considerations

The success of FOSS has attracted the interest of many researchers who have sought to understand how collaborative development is effectively carried out according to this model and how the process can lead to high quality output. Even though there is little explicit documentation about the development process (Michlmayr 2005), the whole decision making process itself can be reconstructed because most discussions leading to decisions are archived online (Lerner and Tirole 2002; den Besten, Dalle, and Galia 2006). In contrast to research about traditional software which has often suffered from a lack of access to empirical data (Gittens et al. 2002), researchers investigating the FOSS phenomenon have access to the majority of online infrastructure used by FOSS projects. Studies investigating version control systems, in which every change to the source code is recorded (Koch and Schneider 2002; German 2005), mailing lists (Sowe, Stamelos, and Angelis 2006), and defect tracking

systems (Fischer, Pinzger, and Gall 2003; Fischer and Gall 2004; Michlmayr and Senyard 2006) have been conducted, slowly leading to empirical insights backing up or refuting anecdotal claims about FOSS.

One argument commonly put forward as a reason why quality may be high is that, unlike in traditional software development, FOSS developers are actually users of the software themselves and therefore have a clear incentive to make it as good as possible (Sowe et al. 2007; Raymond 1999). An important counter argument is that there are many types of users and that the development community is in fact a specific type of user that has vastly different requirements and interests than most users (Daniel and Stewart 2005). As an example, ISO 9126, which was mentioned in the previous section, defines six attributes contributing to software quality, including usability, maintainability and portability. A certain tension has been observed because most end-users have an interest in factors such as usability while the development community often focuses on areas such as maintainability and portability. Tawileh et al. (2006) criticize the FOSS community for being too technology-centric.

Another criticism put forward by Tawileh et al. (2006) is that the apparent lack of mechanisms to ensure quality has stopped users from adopting FOSS. This seemingly unorganized structure is summarized by Moon and Sproull (2000) when they say that “no architecture group developed the design; no management team approved the plan, budget, and schedule; no HR group hired the programmers”. For them, as researchers, this is the “real fascination with Linux”, but this chaotic nature of FOSS poses a problem for industry which increasingly depends on software produced by FOSS projects. In order to address their concerns, they have developed models, such as the Open Source Maturity Model (OSMM) (Golden 2005) and the Business Readiness Rating (BRR),¹ to assess whether a FOSS project is sustainable.

Aberdour (2007) observes notable differences in the quality management approach of traditional, closed source and FOSS development (see table 2.2 for an excerpt). While he finds that some best practices from software development, such as risk assessment and the use of quality metrics, are ignored in FOSS, their lack does not appear to have an impact on quality. He also notes that many best practice principles are followed and concludes that “all these

¹<http://www.openbrr.org/>

Traditional, closed source	FOSS
Well defined development methodology	Development often not defined or documented
Requirements defined by analysis	Requirements defined by programmers
Measurable goals used throughout project	Few measurable goals are used
Work is assigned to team members	Work is chosen by team members

Table 2.2.: Quality management practices compared (Aberdour 2007).

things do occur in [FOSS], they just occur differently”. In the following, the FOSS development model as well as quality practices will be discussed in more detail.

2.2.2. Development Style

The original account which observed and aimed to explain the FOSS process and methodology was given by Raymond (1999). In light of the success of the Linux kernel, which acted as a case study, the paper introduced the metaphors of the ‘cathedral’ and the ‘bazaar’. Traditional software development was compared to building a cathedral. The work is done by an architect (Crowston and Howison 2005), either an individual or a small team, working in isolation from the community (Bergquist and Ljungberg 2001). This approach has also been termed ‘closed prototyping’ (Johnson 2001).

In contrast, FOSS development is akin to a bazaar which is characterized by an unorganized and very open nature in which everybody can participate. Raymond (1999) argued that this open approach to software development actively encouraged users to participate and contribute in various ways, such as by submitting defect reports, doing code review and adding new functionality. This model relies on rapid prototyping, in which development is done iteratively and the development is driven by the active development community and their requirements. Furthermore, work is done in parallel, both development and particularly defect removal, yielding high levels of quality.

Raymond (1999) cites a number of factors which contribute to high levels of quality in FOSS projects, most notably the first one:

- Peer review: the open nature of the bazaar allows developers and capable users to review and modify the source code. Thereby, a large community of users and developers is established that engages in a thorough peer review process. This is in line with findings from software engineering research, which has established that peer review leads to high quality software (Fagan 1976).
- Best people: since FOSS projects are commonly performed by collaboration over the Internet, people with the best skills from all around the world can participate.
- Motivation: because people use their own software, they are highly motivated to improve it (Mockus, Fielding, and Herbsleb 2002). The rapid development cycle with its availability of frequent snapshots leads to feedback which can be used to improve quality and which motivates developers to work on the project (Hertel, Niedner, and Herrmann 2003).
- Fewer constraints: The lack of deadlines and constraints commonly found in commercial environments may also contribute to quality because developers have the opportunity to find the best solution.

While Raymond's account is commonly cited as an explanation for the FOSS phenomenon, it has also been criticized, mainly because it heavily relies on anecdotal evidence, and it can be argued that the model is too simplistic (Bezroukov 1999). Senyard and Michlmayr (2004) argue that the cathedral and bazaar are not conflicting models, as Raymond (1999) suggests, but rather complementary phases of the life cycle of a project. According to this theory, all projects start in the cathedral phase, working on a prototype in isolation, and potentially move to the bazaar once their prototype shows sufficient promise, thereby establishing a large community around the project. However, this transition can fail for a number of reasons, for example because people who start a project have to be good programmers but are not necessarily good project managers, a skill needed to coordinate larger projects.

A more recent model which focuses on the social structure of FOSS projects is the onion model which observes successive layers of member types (Crowston and Howison 2005). According to this model, a sustainable FOSS community

consists of a small group of core members, an increasing number of contributing developers and an even higher number of active users who report defects (Aberdour 2007). The outer layer is defined by those users who are not actively involved in the development process (see figure 2.1).

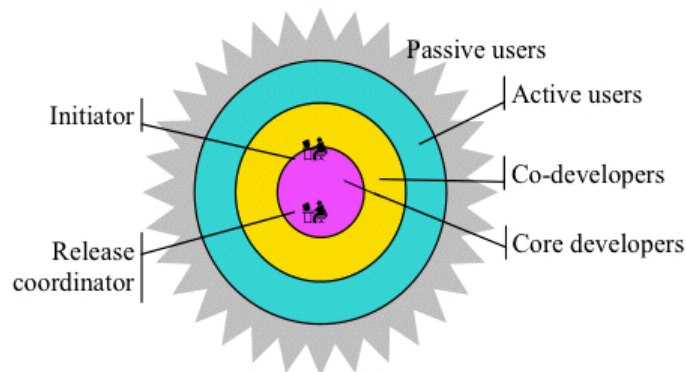


Figure 2.1.: The onion model (Source: Crowston & Howison 2005).

In the onion model, active participants become involved in the next closest layer through a process of meritocracy (Crowston and Howison 2005; Aberdour 2007). Users can participate in the development by sending in defects reports and other feedback, thereby becoming active members of the community. From this layer, participants can advance and become contributing developers by making a substantial contribution. The core team is reserved for the most experienced and active members of the community.

Each role in this model is associated with a number of responsibilities and tasks:

- Core team: studies have found that many FOSS projects have a relatively small number of experienced developers who act as the core team (Mockus, Fielding, and Herbsleb 2002). The main responsibility of the core team is to provide the project with leadership, such as vision (Lerner and Tirole 2002), and to control contributions. They have the say whether a new contribution is outside the scope of the project (Senyard and Michlmayr 2004) or does not meet the project's quality criteria (Godfrey and Tu 2000; Sharma, Sugumaran, and Rajagopalan 2002). It is interesting to note that FOSS leaders often do not have any formal authority over other participants (Lerner and Tirole 2002). Instead, leadership relies on trust which developers obtain by making

significant contributions.

- **Contributing developers:** most of the new functionality is contributed by a loose group of developers with an interest in the project. These earn respect by making contributions, often in the form of source code, for example by adding new functionality or working on abandoned parts of the software (Mockus, Fielding, and Herbsleb 2002).
- **Bug reporters:** one reason commonly cited for high quality levels in FOSS is that there is a large number of active users who supply good feedback (Raymond 1999; Villa 2003). This feedback can be in the form of defect reports but it is also possible to propose new functionality and requirements, thereby shaping the direction of the project. This is a user innovation process of a nature which has few parallels in other areas (Lakhani and von Hippel 2003; Weber 2005).

The organizational structure depicted by the onion model can be observed in FOSS projects of different sizes (Crowston and Howison 2005). While large projects have more hierarchies, such as the ‘lieutenants’ in the Linux kernel which are responsible for specific areas (Iannacci 2003; Iannacci 2005), the different groups within large projects essentially follow the onion model. Crowston and Howison (2005) therefore conclude that large projects are often aggregates of smaller ones.

An underlying assumption which can be found in the literature is that a large community (or bazaar) has to exist for a project to be sustainable, successful and of high quality (Raymond 1999; Mockus, Fielding, and Herbsleb 2002). Various reasons are given for this argument, including the following:

- **Defects:** Mockus, Fielding, and Herbsleb (2002) observe that the core group does not have the resources to remove all defects that are reported. Therefore, quality cannot exceed certain levels unless a project can attract a large community which focuses on defects.
- **Redundancy:** Michlmayr and Hill (2003) note problems with the reliance on volunteer contributors. From this it can be concluded that small FOSS projects are inherently risky (de Groot et al. 2006).

- Scope: it has been argued that FOSS project significantly benefit from ideas and feedback supplied by a large community (Raymond 1999). Without this healthy feedback, the scope of the project may stay limited (Senyard and Michlmayr 2004).

In contrast to these arguments for a large developer community, Krishnamurthy (2005) finds that there are good reasons for FOSS projects to maintain a closed-door approach and remain a small team. He argues that there are coordination problems with increasing size, that a closed-door approach will ensure that only the best and most committed developers are accepted, and that working in a small, intimate team is more rewarding.

Dalle and David (2005) introduce a new distinction regarding project size. They criticize the terminology proposed in Raymond (1999) because the term ‘bazaar’ traditionally refers to distribution rather than production. Instead, they introduce the notion of projects in independent and community mode, I-mode and C-mode respectively. In a related paper, Dalle, Daudet, and den Besten (2006) argue that many puzzles and interesting phenomena are related to large, C-mode, projects and that most literature published to date has implicitly focused on such type of projects. Indeed, most studies have focused on prominent examples of FOSS projects, such as Apache (Mockus, Fielding, and Herbsleb 2002), GNOME (Koch and Schneider 2002) and Linux (Schach et al. 2002). Traditionally, FOSS research has paid only little attention to smaller and less successful projects, and to problems with the FOSS process.

2.2.3. Challenges and Problems

In recent years, there has been a more holistic focus on the FOSS phenomenon and researchers have put increased emphasis on smaller and less successful projects. They found that the majority of FOSS projects, many of which are located on popular hosting sites such as SourceForge, only have one or a few members (Zhao and Elbaum 2003; Howison and Crowston 2004). In fact, many of these can be considered a failure, not having produced any source code at all after several years (Howison and Crowston 2004; Michlmayr 2004). To some extent, it can be argued that this is a selection process and that only the most promising projects attain sizable communities of volunteers. This can be seen

as a healthy process, as it has been argued that choice is a problem when there are too many projects (de Groot et al. 2006).

Several problems in the FOSS methodology have been identified in recent years. There are problems such as:

- Burnout: while FOSS development can be very rewarding, a study about the motivation of developers has found that a high percentage have also experienced feelings of burnout (Hertel, Niedner, and Herrmann 2003). Burnout can be associated with decreased participation and lower quality levels in a contributor's output.
- Resource limitations: many projects do not have enough people and there are tasks few volunteers are interested in. In a study of the GNOME project, German (2004) found that paid employees were responsible for a number of less attractive tasks, such as testing and documentation. However, the majority of projects do not have access to paid resources.

There is growing evidence that many projects fail, for a number of reasons, many of which have not been identified yet. Nevertheless, the findings from a study suggest that it may be possible for many FOSS projects to become more successful and attain higher levels of quality with the help of a targeted process improvement programme. The study evaluated 40 successful and 40 unsuccessful projects (measured by downloads, an arguable measure but sufficient for the study) according to their process maturity (Michlmayr 2005). The study found that in 5 out of 9 tests the successful projects showed significantly more mature processes. While this paper did not study causality, it is clear that there is some link between process maturity, success and potentially quality. Furthermore, the paper argued that some areas, such as automated testing and documentation, are generally neglected. Similar findings have been reported in two surveys on FOSS quality assurance that have been performed. They found that few projects have testing plans, that many projects perform little peer review (Zhao and Elbaum 2000) and that most projects operate in an ad-hoc fashion (Zhao and Elbaum 2003). According to traditional insights from software engineering, process maturity is associated with quality, so the absence of these processes in FOSS would predict low levels of quality.

2.2.4. Summary

Despite these findings, there are many FOSS projects which have attained significant popularity and quality. There is also growing literature that empirically backs up the anecdotal evidence that FOSS is of high quality. Stamelos et al. (2002) found that structural code quality was higher than expected, and there are other empirical studies which show that FOSS is comparable or better than proprietary, closed source software with regards to various quality factors (Paulson, Succi, and Eberlein 2004; Godfrey and Tu 2000). However, these studies have to be regarded with some caution. First, it is not clear what software should act as a basis for comparison (McConnell 1999). Second, it is easy to misinterpret a given indicator. For example, a high number of reported defects can either indicate buggy and low quality software, but it could also mean that the software is rigorously tested (de Groot et al. 2006). Finally, code quality is only one measure of quality and there are other areas in which FOSS projects do not outperform closed source applications, such as usability (Nichols and Twidale 2003).

In summary, the FOSS methodology incorporates a number of elements, such as potentially high levels of peer review and user innovation, that may contribute to quality. In recent times, a body of academic research on FOSS has been established which gives supports to anecdotal claims that FOSS can attain high levels of quality, in particular regarding code quality. However, there is also increasing awareness that the FOSS methodology is facing a number of challenges, some of which are unique to the collaborative development effort performed by volunteer participants. Given that this area of research is still very young and that much focus has been on technical matters, such as language use (Zhao and Elbaum 2003), while this research is concerned with management aspects, further investigation is necessary. In the following section, an exploratory study will therefore be presented which has been performed in order to identify common quality practices and problems.² The insights gained from this study were subsequently used to refine the focus of this dissertation.

²This study has been published as Michlmayr, Hunt, and Probert (2005).

2.3. Exploratory Study

2.3.1. Methodology

For this study, exploratory interviews with seven FOSS developers have been conducted (see table 2.3). The projects were of a very diverse nature and it is believed that despite the relatively small sample size a good representation of various kinds of FOSS projects is given. One project started at a university and was aimed at giving undergraduate students the opportunity to get experience of a software development project. This project was later transformed into a community project in which students of this university participated with external contributors. Another project was part of research activities carried out by a large corporation, and one resulted from a university research project. The remaining projects originated as community projects. The size of the projects varied quite widely, from large projects such as GNOME and Debian, to smaller efforts such as Dasher.

Project	Origin	Size of Community
Apache	community	medium
Dasher	university research	small
Debian	community	large
GNOME	community	large
MirBSD	community	small
Postfix	corporate research	small
VideoLAN	university project	medium

Table 2.3.: Projects of this study: their origin and size of development community.

The interviews were semi-structured in order to allow a thorough exploration of quality in the projects in which the interviewees were involved. Questions that were developed based on the quality and FOSS literature were asked directly but the interviews also remained open for the exploration of other topics. The set of questions include the following:

- Are there any quality issues in your project? If so, what are they and how do you deal with them?
- What techniques can be applied in FOSS projects to ensure quality?

- What kind of facilities does your project have to ensure quality?
- How would you compare the quality of FOSS and proprietary software?
When might the quality in one be higher than in the other?

The complete list of questions can be found in appendix A.1.

2.3.2. Comparison of Proprietary Software and FOSS

The results from the interviews will be presented in two sections; current development and quality practices identified in the interviews will be discussed, followed by a summary of quality related problems facing FOSS projects. Before these two areas are covered, the findings from the question concerning quality in FOSS versus proprietary software will be discussed since it provides an overview of how quality in FOSS projects is perceived by their participants. While it is not expected that interviewees necessarily have an unbiased view regarding this question, it can shed light on the subjective perception of quality by FOSS developers.

Few interviewees said that quality in either FOSS or closed proprietary software is necessarily higher. However, many participants felt that FOSS had a higher potential to achieve greater quality and can react faster to critical issues such as security bugs. There are various reasons for this:

- First, its open nature promotes more feedback, which can be used to improve the software. Feedback can either be given in the form of bug reports or feature requests.
- Second, many participants felt that motivation was higher in FOSS projects because volunteers could work on whatever they wanted. Open collaboration with other developers and input from others also increase the motivation to work on a piece of software. Some interviewees thought that this increased motivation had positive effects on the quality of the software.
- Third, an interviewee suggested that FOSS could attract better human resources because of its distributed nature. He argued that FOSS “can benefit from a wider range of expertise and knowledge than a traditional

software company can usually bring to bear on a problem”. A downside of community projects compared to commercial development was the lack of resources and infrastructure.

- Finally, one interviewee responded that it is very hard to compare open and closed development models because of the opposing philosophy of these models. Since closed source companies often hide their defects and source code, it is hard to make a comparison. This is certainly an area where academic research can fill a gap.

2.3.3. Development and Quality Practices

One of the surprising insights gained was how greatly development practices and processes differed across projects. In this section, a summary of practices found in the interviews will be given. While most projects employ some of these processes, few projects follow all of them. It is not clear why this is the case or whether projects would benefit from the introduction of more of these practices. An interesting observation is that everyone who was interviewed stressed the importance of processes and, in particular, of good infrastructure.

The identified practices can be categorized broadly into the following three areas:

1. Infrastructure
2. Processes
3. Documentation

Infrastructure

FOSS projects rely heavily on infrastructure that allows distributed development and collaboration. Important parts of infrastructure are:

- Bug tracking systems (e.g. Bugzilla): these are used to capture feedback from users. They are often used to store both actual bug reports as well as feature requests.

- Version control systems (CVS, SVN, Arch): these allow multiple people to work on the same code base concurrently and keep track of who makes which changes.
- Automatic builds (e.g. tinderbox): these make sure that the newest code in the version control system still builds. The test builds can be done on a number of different hardware or software environments.
- Mailing lists: used for communication, both between developers and users.

While many projects make use of such infrastructure, the way they are used differs significantly from project to project. The different use of this infrastructure (i.e. the practices) may have important implications for project quality. For example, some projects employ a very rigorous policy through its automatic build system, halting further development until the defect has been removed. The practices related to the use of version control systems are also very diverse. In some projects, commit access is given immediately after the first contribution. In others, contributors have to work for months to gain respect from others, and there are also projects where only the lead developer has the right to commit code. The implications of these different practices on project quality need to be studied in further detail.

Processes

FOSS development follows many processes but a large number of them are not documented anywhere — developers adhere to them implicitly.

- Joining: projects require prospective members to follow specific, mostly undocumented, procedures in order to join a project. These procedures vary considerably across projects. Some explicitly make sure to admit only contributors from whom high quality submissions can be expected while other projects are more liberal. One specific joining script has been described in the literature (von Krogh, Spaeth, and Lakhani 2003).
- Release: different release policies are employed by projects but many follow freeze stages. A feature freeze is the point when no new features

are to be incorporated into the code base but there is sufficient time to fix bugs. More recently, string freezes have also gained popularity. They give translators of the software an opportunity to update their translations before a release is made.

- **Branches:** these are used to differentiate between versions of a program, for example by having a stable and a development branch. New development tools which make branches easier to deal with, such as Arch, have had a significant impact on the development process.
- **Peer review:** typically, changes made to the version control system are reviewed by members of the projects, though in most cases, this form of peer review is not very well formalized. Developers hope that other project participants will look at their changes but there is often no assurance that this is the case in reality. On the other hand, some projects have introduced very rigid procedures and expect code to be reviewed before it is committed.
- **Testing:** in order to ensure that a new release fulfils the standards of a project and that it has no major regressions (i.e. bugs that affect functionality that previously worked), some projects have testing check lists. These check lists contain the most important functions and briefly describe how they can be tested. A release is only made when testers on different platforms have gone through the check list and have confirmed that the new version does not display major show-stoppers or regressions. Since few projects have automatic test suites, having a check list is a good solution to guarantee that at least major components and functions operate.
- **Quality assurance:** some projects organize bug days or bug squashing parties to triage their outstanding bugs. During this work, duplicate bug reports are marked as such, old bugs are reproduced, and bugs are also fixed.

Documentation

Many interviewees criticized their projects for a lack of documentation regarding development practices. Few projects have explicit documentation describing ways of contributing to and joining a project. In some cases contributed source code was checked by the lead developer of a project and then rejected because it did not conform to the coding style, a style that was not clearly documented anywhere. However, some projects, mainly those that have a large number of contributors, have good documentation.

- Coding styles: documentation aimed at developers which describes the style which should be used for the source code.
- Code commit: documentation which describes when and who can make changes in a project's version control system.

2.3.4. Quality Problems

The quality problems that have been identified in the interviews are of particular interest since they are largely issues which have yet to be solved in the FOSS community. While many of the practices described in the previous section have simply not been adopted by certain projects, good solutions for some of the following quality problems have yet to be developed.

- Unsupported code: one of the unsolved problems is how to handle code that has previously been contributed but which is now unmaintained. A contributor might submit source code to implement a specific feature or a port to an obscure hardware architecture. As changes are made by other developers, this particular feature or port has to be updated so that it will continue to work. Unfortunately, some of the original contributors may disappear and the code is left unmaintained and unsupported. Lead developers face the difficult decision of how to handle this situation.
- Configuration management: many FOSS projects offer a high level of customization. While this gives users much flexibility, it also creates problems with testing. It is very difficult or impossible for the lead developer to test all combinations so only the most popular configurations

tend to be tested. It is quite common that, when a new release is made, users report that the new version broke their configuration.

- Security updates: in many cases they are made in a timely manner but sometimes fixes are not available for weeks or months.
- Users do not know how to report bugs: as more users with few technical skills use FOSS, developers see an increase in useless bug reports. In many cases, users do not include enough information in a bug report or they file duplicate bug reports. Such reports take unnecessary time away from actual development work. Some projects have tried to write better documentation about reporting bugs but they found that users often do not read the instructions before reporting a bug.
- Attracting volunteers: a problem some projects face, especially those that are not very popular, is attracting volunteers. There are usually many ways of contributing to a project, such as coding, testing or triaging bugs. However, many projects only find prospective members who are interested in developing new features. Few contributors are interested in helping with testing or triaging bugs. As a result, developers have to use a large portion of their time for tasks other people could easily handle.
- Lack of documentation: it is possible that the previous problem is related to the lack of documentation. Volunteers may want to contribute in an area but they might not know how to start. Little help is given to prospective contributors and almost no documentation exists. The lack of developer documentation also implies that there is no assurance that everyone follows the same techniques and procedures.
- Problems with coordination and communication: in some projects, there are problems with coordination and communication which can have a negative impact on project quality. Sometimes it is not clear who is responsible for a particular area and therefore bugs cannot be communicated properly. There may also be duplication of effort and a lack of coordination related to the removal of critical bugs.

In addition to these specific quality issues, three underlying areas which are sometimes problematic have been identified:

1. Leadership: many different leadership and management styles are employed in FOSS projects, and links between the leadership style and quality remain untackled. Some FOSS projects have leadership teams while other projects have a ‘benevolent dictator’ (someone who uses their authority to make decisions for the good of the whole project). One important question is which factors are linked to the choice of leadership style, such as, for example, the size of the project. Since management seems central to maintaining quality, evidence and insights to improve leadership and management in a project will contribute to quality.
2. Release cycle and management: different ways and strategies of performing a release have been observed, most notably the time based and feature based strategies. In the time based release strategy, a release is made according to a firm schedule and time frame. On the other hand, with the feature based strategy, releases are made when a certain set of features has been implemented. Research to compare these two release strategies is needed in order to find out when it is worthwhile to follow one or the other strategy.

Furthermore, the exploratory interviews have shown some problems with stable releases for users. While development versions are released very regularly, stable user releases are often forgotten because the development version works sufficiently well for the developer and they do not see the need for a new stable version for users. There is also often a problem with the coordination of a release, especially in large projects in which many participants have to align their work.

3. Company involvement: while most FOSS projects are carried out by volunteers, companies are increasingly getting involved in the development of FOSS projects, especially in large and successful projects. An interesting question is how FOSS projects can best benefit from company involvement. There are some constraints under which FOSS projects work that are not applicable to companies. For example, not many FOSS

projects have good user documentation, and few projects have automatic regression test suites. These are areas where companies might be able to contribute in a way most volunteers could not.

2.3.5. Discussion

The interviews have given rise to a number of interesting insights about quality in FOSS and have identified a number of outstanding issues that have potential implication for project quality. One striking observation of section 2.3.3 is the diversity of practices that appear to be employed by FOSS projects. Based on the interviews, it would be hard to make the claim that there is one FOSS model that all projects follow. While projects share common attributes, there are clear differences in their practices. Since practices might have a different impact on project quality, the relationship between these practices and the quality of the resulting products needs further investigation. This may allow the identification of best practices or show factors influencing the applicability of practices to particular projects.

The interviews have also shown that projects clearly face a number of problems. While all of the projects in this study can to some degree be considered successful FOSS projects, they also experience problems and therefore there is room for quality improvement. The problem of unsupported code is of high significance since the removal of existing functionality may impact users. On the other hand, it is not clear whether a volunteer project can guarantee that a particular functionality — or even the project itself — will continue to be maintained. This raises the more general question whether projects based on unpaid volunteers can not only deliver a high quality product but also ensure that high levels of quality can be maintained.

ISO 8402 defines quality assurance as all “planned and systematic activities” directed towards fulfilling the requirements for quality. Given the volatility of volunteer contributions and the unplanned nature of many FOSS activities, it is not clear whether FOSS projects can have a planned and systematic approach to quality, and as a consequence, whether they can ensure high quality. This question is of increasing interest as FOSS is deployed in more corporate and mission-critical settings. In fact, some problems due to the lack of

a systematic approach to quality have already been observed. While many important defects, especially security related issues, are fixed very quickly, often within hours, it was pointed out during the interviews that some remain unresolved after weeks or months. There is a great variance in defect removal time and more systematic quality assurance activities can contribute to timely fixes.

2.4. Refining the Scope

The exploratory study has identified three areas which are of specific interest for further investigation: leadership, release management and company involvement. In the following, one of these areas, release management, will be selected as the scope of further investigation. This particular area will be investigated thoroughly in the light of quality management and improvement in FOSS projects.

Company involvement is an area of increasing importance because a growing number of companies is deploying FOSS and may want to actively participate in development. There is evidence that companies play a very important role in the FOSS community since they focus on neglected areas (German 2004). There is also some evidence that the involvement of companies can lead to certain conflicts, for example related to the control of a project (Jensen and Scacchi 2004). While this is an interesting area of research, the majority of FOSS projects are still largely volunteer based. A report published by the European Commission shows that over two thirds of FOSS contributors are individuals (Ghosh 2006). As such, questions regarding the coordination of distributed volunteer teams are of higher significance than those related to paid contributors.

Leadership is an interesting area to which some attention has been drawn already. Lerner and Tirole (2002) report that FOSS leaders often do not have any formal authority and that leadership heavily relies on respect and trust. There is also evidence that there are a number of different leadership styles in use (Mockus, Fielding, and Herbsleb 2002; Iannacci 2005; Iannacci 2003). Finally, several practical guides to FOSS project management have recently been published, most notably one by an experienced FOSS developer working

on the Subversion project (Fogel 2005).

While leadership in FOSS projects is an interesting topic for research, the interviews have not revealed any pressing issues that need to be addressed. On the other hand, questions related to release management seem not only of a more fundamental but also of a highly practical nature. Fundamentally, it is not clear how a team of loosely connected volunteers spread all around the globe can work together in a way to release software, some of which consists of millions line of code written by thousands of people (González-Barahona et al. 2001), in a timely fashion and with high quality. This topic raises questions related to quality management and coordination, especially in the light of the unstructured organization in FOSS that has been described earlier. There are several open questions about release management in volunteer teams, such as the problem of following deadlines (Garzarelli and Galoppini 2003) and the reliance on project participants (Michlmayr and Hill 2003).

The importance of these questions is underlined by practical problems observed with release management in FOSS projects. The following three examples from real FOSS projects show that release management is in fact a matter of concern:

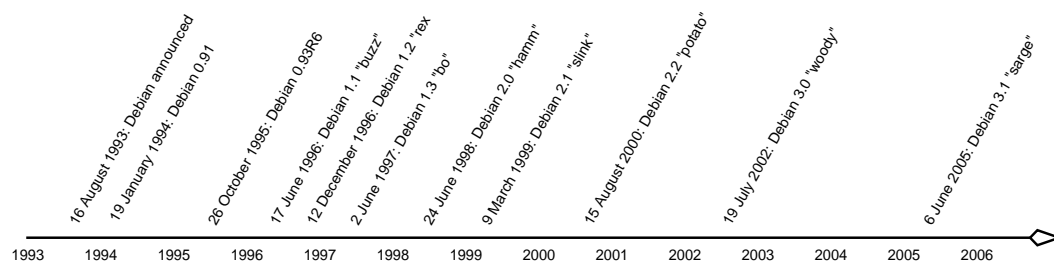


Figure 2.2.: Debian has experienced increasingly delayed and unpredictable releases in recent years (Source: Martin F. Krafft).

- Debian: this project combines other FOSS projects to a system in which the individual components are well integrated and easy to install and use. In recent years, the project has faced increasingly delayed and unpredictable releases (see figure 2.2). Most notably, the release process of Debian 3.1 was characterized by major delays. Initially announced for December 1, 2003, the software was finally released in June 2005 — a delay of one and a half years. By the time the new version was released, the previous stable release was largely considered out of date and did not

run on modern hardware.

- GNU tools: various utilities from the GNU Project form an integral part of virtually every Linux system and are also very popular in other systems. Despite their popularity and importance, development has been slow in recent years and few of these projects have attracted a large development community. Even though development snapshots were published, there were often huge delays between stable releases. For example, `tar` 1.13 came out in August 1999, followed by version 1.14 at the end of 2004. The compression utility `gzip` saw a new version in December 2006, more than a decade after the last stable release in 1993. As a consequence of these long delays between stable releases, several vendors started shipping pre-releases. While these versions contained important bug fixes and new functionality, they were less tested and their deployment led to certain problems, such as incompatible command line options in `tar`'s support for the `bzip2` compression program.

Version	Release Date
1.0	October 22, 1999
1.2	May 9, 2000
1.4	May 29, 2002
1.6	Still not released as of March 2007

Table 2.4.: Stable releases of the Mutt e-mail client.

- Mutt: the last stable version of this popular e-mail client was released in 2002. After a major discussion on the development mailing list about the status of the project at the beginning of 2005, the maintainer acknowledged neglecting the project, invited a second developer to play a more active role and promised to work towards a new stable release. Even though several development releases have been published since this discussion, a new stable release is still not in sight as of this writing (see table 2.4 for a list of past stable releases).

These examples from existing, popular FOSS projects show that release management is a real matter of concern. It is also important to note that the problems faced in the first example are of a quite different nature than those in

the second two projects. While the latter problems appear to be due the lack of resources, the former project may face challenges related to the coordination of an increasingly complex system — after all, Debian has doubled in size with each release (González-Barahona et al. 2001; González-Barahona et al. 2004). These issues will be studied in more detail in the following chapter.

2.5. Chapter Summary

This chapter began with exploring models of FOSS development presented in the literature and how they relate to quality of the output produced. This presentation was followed by a discussion of quality problems that have been observed in the literature. Given the novelty of this research area, an exploratory study was subsequently performed to identify quality practices employed by FOSS projects and actual quality problems they are facing. Based on these interviews three areas of interest were identified: leadership, release management, and company involvement. Because of its highly problematic nature, release management was subsequently chosen as the specific scope of this research, from which insights about quality management and coordination in FOSS projects will be drawn. The next chapter will identify practices and problems of FOSS release management and will discuss related literature, such as theories about organization and coordination.

3. Release Management

This chapter investigates release management in traditional, proprietary software development and in FOSS. It also presents theories that are relevant in a discussion about release management and coordination. This chapter is divided into the following three sections:

1. Literature: topics such as software maintenance and iterative development are discussed to show traditional views of software engineering; this is followed by an investigation of release management in FOSS projects.
2. Exploratory study: findings from an exploratory study investigating practices and problems in FOSS release management are presented.
3. Theory: theories about organizational complexity, modularity and coordination are presented. These theories will subsequently be used as a framework for analysis.

3.1. Literature

In the following section, an introduction to software maintenance will be given, followed by an overview of literature about release management in FOSS projects. Traditionally, development and maintenance have been considered different phases. In FOSS, it is harder to separate out maintenance and development since they tend to occur together. An emphasis will therefore be put on a new style of development, iterative development, which is increasingly employed by software development projects, including FOSS projects. According to this model, development is carried out in incremental steps and releases are made fairly frequently to deliver new functionality and fixes.

3.1.1. Software Maintenance

Software maintenance is the field which is concerned with the evolution and maintenance of a software system after its initial release. Even though software maintenance constitutes more than half of the total cost of a software system (Tan and Mookerjee 2005), the prime focus in software engineering has traditionally been on the development phase (Sommerville 2001). This is now changing as there is a move away from the rigid and monolithic waterfall model towards a more dynamic picture of software engineering in which development is done iteratively in incremental steps. There are claims that incremental software development addresses the problem of time-to-delivery which is of growing concern in today's fast paced and dynamic environment (Greer and Ruhe 2004). Instead of delivering a monolithic system after a long time of development, smaller releases are prepared and distributed sequentially (Greer and Ruhe 2004). The functionality of the system is thereby split into increments which are then successively delivered. Each increment is a collection of features which forms a system of value to the user (Saliu and Ruhe 2005). Redmill (1997) goes a step further and argues that each delivery "is not simply a new increment to be added to the existing system, but a new version of the system which may include changes to what had previously been delivered as well as new features".

Compared to the traditional, 'big bang' delivery, in which a monolithic system is shipped after a long development phase, iterative development promises a number of advantages (Greer and Ruhe 2004):

- Requirements can be prioritized in a way that the most important ones are delivered first.
- Because customers receive the system early on, they are more likely to support it and provide feedback.
- Since each increment is smaller than the monolithic system, scheduling becomes easier.
- Feedback from users is obtained at each stage and plans can be adapted accordingly.

- This way of development allows developers to react better to changes and additions in requirements.

In short, by splitting the functionality of the whole system into increments, smaller releases can be delivered more quickly in an incremental manner and feedback obtained at each step. This allows organizations to deal better with a dynamic environment. It has been argued that this change in the software development methodology has been promoted in part by the Internet which makes the distribution of software easier (van der Hoek et al. 1997). The continuous process of making new releases is an important part of quality management since it delivers fixes and important functionality to users (Levin and Yadid 1990). Continuous releases are also a mechanism against obsolescence because they ensure that the value of a piece of software is maintained (Baetjer 1997).

One question of increasing importance with the iterative software development model is when to do releases. Levin and Yadid (1990) criticize traditional models which only deal with the initial release but neglect to consider subsequent ones. There is evidence that an early introduction of a new release can be harmful since it might not be different enough and it may pose “a barrier for growth in the market share of the existing release” (Krishnan 1994). Levin and Yadid (1990) consider four factors which need to be taken into consideration:

- Fixed costs: there are costs associated with each release, such as costs of distributing and installing the software.
- Cost of error correction: defects are identified during normal operation of the software and fixes incorporated into the software. This cost is dependent on the number of errors removed before a new version becomes available.
- Cost of improvements: in a similar way to error correction, the implementation of improvements and new functionality is associated with costs.
- Opportunity loss due to obsolescence: the longer a release takes, the more obsolete will the current version become, possibly leading to a loss in market share.

Lehman (1980), who sees no difference between development and maintenance, argues that structural maintenance needs to be performed over the whole lifetime of a system. As a system grows in size and age, complexity of the software increases, there is architectural deterioration (Fischer and Gall 2004), and maintenance also becomes harder because of a growth and diversification in the user base (Krishnan 1994). However, these ongoing maintenance efforts which are needed are not limited to the source code. It is well known that there is a strong link between the structure of an organization and the software developed by it (Conway 1968). Work performed on the maintainability of the software therefore has to go hand in hand with improvements to organizational structures. This topic will be discussed in more detail in section 3.3.

3.1.2. Release Management in FOSS

Release management in FOSS is an area which is largely unexplored. FOSS is characterized by a highly iterative development model in which new development releases are made available very frequently following the motto “release early, release often” (Raymond 1999). The aim of this approach is to gather feedback early and it allows the community to influence the direction of a project. Most projects differentiate between developer releases and releases aimed at end-users but because the requirements of developers and users are often vastly different some projects put little emphasis on the latter (Daniel and Stewart 2005).

Erenkrantz (2003) has published an initial taxonomy to identify common properties of release management in FOSS. He differentiates between the following six characteristics:

- Release authority: in small projects, the project founder or leader is often responsible for making new releases available. In larger projects, there is usually a dedicated release manager, or a whole team, who has the responsibility for conducting releases.
- Versioning: a release is always associated with a version number which “signifies some property of the release to the community” (Erenkrantz

Traditional, closed source	FOSS
Often follows a waterfall model	Typically follows iterative development practices
Delivery of a monolithic release after a long time of development	Small releases published in an incremental fashion
Uses dedicated planning tools, such as Gantt charts	Few dedicated planning tools but good integration of infrastructure (e.g. bug tracking) with release planning
Development releases are private	Development is open, and releases and repositories accessible
Few releases made for the purpose of user testing	Development releases published according to the motto “release early, release often”

Table 3.1.: Release management practices compared: traditional, proprietary and FOSS development.

2003). Many projects follow a model in which three integers are arranged in the form x.y.z: the major release number (x), minor release number (y) and the patch level (z). A number of FOSS projects follow the versioning scheme introduced by the Linux kernel in which even numbers (2.0, 2.2, 2.4) denote stable releases while odd numbers (2.1, 2.3, 2.5) indicate development releases (Moon and Sproull 2000).¹

- Pre-release testing: many projects have a set of criteria a release must fulfil before it can be published.
- Approval of releases: a policy may be in place to indicate whether the release manager can make releases independently or has to seek the approval from another group first.
- Distribution: this characteristic involves two aspects, visibility and accessibility. The first aspect is concerned with writing release announcements and similar tasks while the second is related to distribution of the software, for example by arranging for a number of mirror sites which copy the software and make it available.

¹Interestingly, the Linux kernel no longer uses this versioning scheme.

- Formats: many projects only ship the source code but some also make binaries available for a number of architectures. There are a number of formats that can be used to make the software available.

While Erenkrantz (2003) has captured characteristics related to release authority and the actual work involved during a release, there is very little coverage in terms of actually moving from the development phase to the preparation for a release. Most projects have the notation of a code freeze, during which the release manager may “reject all changes other than bug fixes” (Jørgensen 2001).

Even though the exploration of quality problems in section 2.3 has identified release management as a major area of concern in FOSS, little research has been performed about this subject. In the next section an exploratory study will therefore be carried out in order to get a better picture of actual practices and problems associated with release management in FOSS projects.²

3.2. Exploratory Study

3.2.1. Methodology

For this study interviews with twenty experienced developers from different FOSS projects were carried out. The interviews were conducted at a conference over the course of three days. All interviews were recorded with the consent of the interviewees and transcribed for later analysis. Each developer who was invited for an interview was either a core developer or the release manager of a FOSS project. The range of FOSS projects in which developers participated was very wide, ranging from small to very large and complex projects, and included projects of all kinds of different nature (see appendix A.2 for a complete list of projects and questions of this study). This great variety gives a good coverage of practices found in the FOSS community and allows the identification of specific factors which may influence the choice of a project’s practices.

²This study has been accepted for publication in the Proceedings of the Third International Conference on Open Source Systems.

3.2.2. Types of Release Management

The interviews have revealed that release management is an important aspect of the development phase and that the general term release management is used to refer to three different types of releases. These types differ quite significantly regarding the audience they address and the effort required to deliver the release. The three types which have been identified are:

- Development releases aimed at developers interested in working on the project or experienced users who need cutting edge technology.
- Major user releases based on a stabilized development tree. These releases deliver significant new features and functionality as well as bug fixes to end-users and are generally well tested.
- Minor releases as updates to existing user releases, for example to address security issues or critical defects.

Since developers are experts, development releases do not have to be polished and are therefore relatively easy to prepare. Minor updates to stable releases also require little work since they usually only consist of one or two fixes for security or critical bugs. On the other hand, a new major user release requires significant effort: the software needs to be thoroughly tested, various quality assurance tasks have to be performed, documentation has to be written and the software needs to be packaged up.

In terms of release authority, it can be observed that major new user releases are typically performed by the project leader or a dedicated release manager whereas development and minor user releases can also be prepared by a core member of the development team. This again shows the significance that user releases have.

Project	Version Control System
GCC	SVN
GNOME	CVS, SVN
Nano	CVS
Synaptic	bzr
X.org	git, SVN

Table 3.2.: Version control systems used by some projects.

The interviews also revealed an interesting trend regarding development releases. Traditionally, development releases have played a big role in FOSS projects and it is known that during early development some projects made new development releases available almost every day (Torvalds and Diamond 2001). Nowadays, actual development releases appear to be of less importance. This change has been prompted by a major improvement and hence deployment of version control systems in recent years (see table 3.2 for examples of version control systems used by projects from this study).

Version control systems are an important development tool in which every change to the source code is recorded. They allow multiple developers to work on the same code base without causing conflicts and provide a historical record of all changes that have been made. Instead of downloading a development release that may be several days or weeks old, developers can obtain the latest code directly from the version control system. The advantage is that it allows developers to track development in real time. A disadvantage is that it is possible that the latest code in the version control system may not work correctly but most projects have policies in place to ensure that such a situation does not persist for very long.

Even though version control systems have displaced development releases as the main exchange of source code during daily development, development releases are by no means obsolete. Several interviewees have stressed that while version control systems are a great mechanism to stay up to date regarding the latest code, actual development releases play an important role as a common synchronization point, for example to find out when a defect has been introduced.

While development releases and minor updates to existing stable releases play an important function in every project, the main challenge is associated with the preparation of major new user releases. In the following, the focus will therefore be on releases aimed at end-users.

3.2.3. Preparation of Stable Releases

The act of preparing a stable release for end-users is a complex set of tasks in which all developers of a project have to work together to deliver a high

quality product. While the specific release approach may differ from project to project, a common pattern has been identified: staged progress towards a release where each stage is associated with increasing levels of control over the changes that are permitted. These control mechanisms are usually known as freezes since the development is slowly halted and eventually brought to a standstill.

The following types of freezes are typically found in FOSS projects:

- Feature freeze: at this point, no new functionality or features may be added. The focus is now on the removal of defects.
- String freeze: when this freeze is in place no messages which are displayed by the program, such as error messages, may be changed. This allows translators to work on their languages in the remaining time before the release and translate as many messages as possible, ideally reaching 100% support for their native language.
- Code freeze: the source code may not be touched by individual developers anymore and permission needs to be sought before making a change, even in order to fix bugs.

Throughout this staged approach towards a final release, test versions are published to promote testing and users are encouraged to test these pre-releases and provide feedback to the developers. These test releases can include alpha and beta versions, often leading to the publication of release candidates right before the actual release.

Among the twenty projects of the interviews for this exploratory study, there is no common pattern as to how often new user releases are published. The release frequency ranges from one month to several years. Even though there is no common pattern, a number of factors have been identified which are related to the release frequency:

- Build time: some projects require massive processor power to compile the source code into a binary that can be executed and shipped to users. Given that each test release needs to be compiled, the long compilation step puts a natural limit to the release frequency.

- Complexity of the project: in a similar way to the point above, complex projects which consist of many different components and in which a large number of developers are involved exhibit a tendency towards a slower release cycle. This is because the coordination work in large projects is significantly higher than in small projects.
- Fixed time, such as testing and release work: releases are associated with some fixed costs which have to be performed for every release. This includes testing of the software, release work and the actual distribution of the software.
- Age of the project: the interviews have shown that there is a trend for young projects to perform releases more frequently. The following two reasons have been given for this trend. First, younger projects need more direct feedback from users than already established projects. Second, it is easier for a small project (something which young projects are likely to be) to prepare releases. For example, the Bazaar project has chosen a monthly release cycle because they are a very young project and want to get new code out as soon as possible to maximize distribution and feedback.
- Nature of the project: this is another important factor that has an influence on the release frequency. Projects which are aimed at the desktop or other fast paced environments have a much higher release frequency than software which is mainly used on servers where there is a tendency to avoid upgrades unless they are strictly necessary. The audience also plays an important role. Projects which are mainly oriented towards developers or experienced users may make frequent releases because such users are often interested in the latest technology.

There is one special factor which has a major impact not only on the release frequency but on the whole release strategy of a FOSS project: the inclusion of the project's software in a collection of FOSS software, such as a Linux distribution. FOSS projects publish their work independently but for a complete system several hundreds or thousands of applications are required. Therefore, a number of non-profit projects and companies exist whose purpose it is to

take individual FOSS applications and integrate them to a system which is easy to install and use. There are commercial companies which provide such integrated systems, such as Red Hat or Novell with their SUSE Linux, as well as non-profit organizations, such as Debian and Gentoo. The inclusion in such systems is seen as a very positive factor for a project because its software is thereby exposed to a much wider audience. At the same time, this increase of end-users often requires changes to the release strategy because the project is no longer solely used by developers who know what they are doing.

As with release frequency, there is a great variety in terms of release processes which are employed by FOSS projects. Specific practices will be covered in section 3.2.5 but first a general classification of release management strategies is developed. While there are many differences regarding the specific details of the implementation of a release management strategy, the following two fundamental strategies have been identified according to which projects can be classified:

- Feature based strategy: the basic premise of this strategy is to perform a new release when a specific set of criteria has been fulfilled and certain goals attained, most typically a number of features which developers perceive as important. This strategy is in line with traditional software development which is feature-driven.
- Time based strategy: in this strategy a specific date is set for the release well in advance and a schedule created so people can plan accordingly. Prior to the release, there is a cut-off date on which all features are evaluated regarding the stability and maturity. Subsequently, a decision is made as to whether they can be included in the release or whether they have to be taken out again and can be considered for the following release.

While there is a minority of projects which do not neatly fit into this classification, these are the two major approaches to release management in FOSS today. Traditionally, the former approach, feature based releases, has been employed but there is growing interest in time based release management.

A number of projects in this study are experiencing growing frustration with the feature based release strategy. They argue that this strategy is very

ad-hoc and that releases are announced out of the blue. This is because all functionality that is desired is never achieved and so the release manager has to call for a release at some point and this often happens quite unexpectedly. In many cases, the last release is quite dated at this stage and so there is a great rush to get a new release out of the door. This lack of planning can lead to incomplete features and insufficient testing. According to some, this strategy is also often associated with the motto “release when it’s ready”. Even though this is a laudable goal, in practice it is often problematic, in particular in large projects, because there are always new features that could be added or bugs that could be fixed. The result of this approach can be associated with great delays because the project is constantly “nearly there” at the point where it could make a release but there is always something that remains to be done. In the meantime, the last stable release becomes increasingly out of date.

In order to address these problems about a quarter of the projects investigated as part of this study are considering a migration to a time based strategy. In their view, time based releases constitute a more planned and systematic approach to release management and make release coordination easier. This is something that large projects would especially benefit from. A number of interviewees have named the GNOME project as the reason for their interest in time based releases. This project has managed to deliver high quality releases on time on a six month time based schedule even though the project is quite large and complex.

Regardless of the specific release strategy employed by a project, the interviews have shown that release management plays an important role in a project. One interviewee has compared the delivery of regular releases to the pulse of the project: in addition to supplying users with fixes and new features, they are an important indication that the project is still alive and doing well. It is therefore important to make regular releases and to employ a release scheme that is easy to understand for people who are not well acquainted with the inner workings of the project. Releases are also an interaction with a project’s user base because feedback can be obtained through the publication of a new version. This way, developers know whether they are moving into the right direction. Finally, release management is an important part of a project’s approach to quality assurance because during the preparation for a release

developers stop adding new features and instead focus on the identification and removal of defects. The feedback obtained after a release furthermore give them information as to which parts of the software might need more attention.

3.2.4. Skills

The role of the release manager is diverse and demanding because they have to interact with a large number of different people, understand technical issues but also know how to plan and coordinate. Based on the interviews, the following taxonomy of skills and characteristics which release managers need has been developed:

- Community building: showing people that their input is useful. Release managers also need respect in the community in order to perform their work.
- Strong vision: showing developers in which direction the project should be moving.
- Discipline: saying 'no'. Release manager have to focus on an overall goal and can not make everyone happy.
- Judgement: gauging the risk and possible impact of a particular change.
- Attention to detail: walking through every line of code that has changed.
- Good communication: writing release notes, asking for feedback, interacting with users.
- Management skills: talking to people, organizing, planning, making sure that all the little things happen.

It is interesting to note that release managers in small and large projects play a vastly different role even though they essentially have the same responsibility, namely getting a high quality release out. In a small project, the release manager usually has an administrative role which involves the preparation of the release in different formats that can be distributed, the creation of release notes and the actual distribution of the software. In large projects, on the other hand, there is a big emphasis on coordination that needs to be performed

by the release manager. They have to make sure that different parts of the software are ready at the same time and that all developers are aligned towards the common goal of creating a stable release. While administrative tasks are also needed in large projects, they are by no means the majority of work that is involved.

3.2.5. Tools and Practices

Despite the important role release management plays in the delivery of quality software to users, there is little knowledge as to how FOSS projects perform releases. This section covers tools and practices employed during release management.

Surprisingly, the interviews reveal that there are few dedicated tools used specifically for release management. This is in contrast to traditional software development where many planning tools and other mechanisms are employed to help with planning and coordination. While FOSS projects do not make use of specific tools for release management, they have solidly integrated their development tools with their whole development process, including release management. In particular, the following two tools serve an important function during release management:

- Version control systems: these systems keep track of every change that is made and are therefore a good means of coordination. Release managers can keep an overview of what is going on through them. Furthermore, modern version control systems which have been developed in recent years offer good support for branching: this allows developers to make changes in a private part of the system and when the changes are ready they can be merged into the main development tree without the loss of any historical information. This way, experimental features can be developed independently without causing disruption to the development version that is going to be released.
- Bug tracking systems: they perform a vital function for release management because they indicate whether the software is ready to be released. Most projects assign different severities to bugs, stating how big an impact they have on the functionality of the software. This gives a good

overview over defects that need to be addressed before a release can be made. Some projects have a specific classification of ‘release critical’ bugs which can be considered as blockers for the next release.

While these tools are tightly integrated with the FOSS development process, the interviews have shown that there is need for a tool which shows the big picture. While version control and bug tracking systems include vital information, there is other information that is essential for release planning and there are currently no tools which can easily visualize such huge amount of data in a meaningful way. This dissertation will not work on the development of such a tool because the main focus is on process improvement and because there are a number of EU-sponsored projects, such as QualOSS³ and SQO-OSS,⁴ that are working on tools on metrics to improve quality in FOSS projects.

In addition to the use of tools, there are specific practices which are related to release management. The following practices have been identified through the interviews:

- Freezing: as discussed previously, freezes are a means to lock down development and to increasingly focus on the removal of defects and publication of the release.
- Scheduling: relatively few projects make use of a schedule but it is a vital component in those projects which employ a time based release strategy.
- Establishing milestones: some projects have loosely defined milestones but given the volunteer nature of most projects there is no guarantee that certain milestones will actually be achieved.
- Setting deadlines: many projects set deadlines but they are not always effective because the release manager has no control over volunteer participants. There are two main elements of control. First, to institute freezes, thereby restricting code changes. Second, to use persuasion and communication to show other developers which areas are of high priority.
- Building on different architectures: as part of a project’s testing plan, it is beneficial to build the software on a number of different hardware

³Quality in Open Source Software

⁴Software Quality Observatory for Open Source Software

platforms because each of them may exhibit specific bugs which are not visible on other platforms.

- Unit testing: even though relatively few projects use automatic tests, there are some projects for which a complete test suite is very important. It is interesting to note that some FOSS communities, such as those related to the Python programming language, put a higher emphasis on automatic testing than others.
- User testing: one of the main benefits from preparing and subsequently publishing a release is the feedback potentially obtained from a wide range of users. Even those projects which heavily deploy automatic test suites think that the most significant insights usually come from actual users. It is therefore important to make snapshots easily available, not only in source form but ideally also in binary format for the most important operating systems. There are two factors which determine how much projects gain from user testing. First, the perceived risk of upgrades has to be low, otherwise users will not try the new version. Second, a project needs a large user base in order to get good testing coverage.
- Offering rewards: some projects offer rewards for finding or fixing bugs but this practice is relatively rare.
- Following a release check list: a number of projects use a check list to make sure that all steps that are necessary to make a new release are followed.
- Holding a post-release review: surprisingly few projects have a formal post-release review but there are often informal discussions on the mailing list of the project, in particular when there were problems with the release.

3.2.6. Problems

As discussed earlier, the process of preparing a new stable release for end-users is quite elaborate and complex since the software needs to be sufficiently

tested, documented and packaged for release. The release management process often faces certain problems, the most common of which are as follows:

- Major features not ready: planning in volunteer teams is very hard and it happens regularly that major features which are on the critical path are not ready. These blockers need to be resolved so the release process can continue.
- Interdependencies: with the growing complexity of software, there are increasing levels of interdependencies between software. For example, a piece of software may use libraries developed by another project or incorporate certain software components created elsewhere. This creates a dependency and can lead to problems with a project's release when those other components are not ready.
- Supporting old releases: there is generally a lack of resources in the FOSS community and in many projects old releases receive little support. Some vendors which distribute a given release may step in and offer basic support but it may still be problematic for other users of the software.
- Little interest in doing user releases: even though this study has emphasized the importance of user releases, many developers show little interest in making releases. Since developers by definition generally use the development version they often do not understand the need for a user release or do not see when a user release is massively out of date. Furthermore, some developers claim that preparing user releases only distracts them from the main task they are interested in: development of new functionality.
- Vendors shipping development releases: when projects do not publish new releases, some vendors start shipping development releases because they contain features or fixes which their customers need. This situation is problematic because it can lead to fragmentation and because development releases are generally not as well tested as user releases.
- Long periods without testing: in large projects, there are often long periods in which a large number of changes are made with relatively

little testing. At the time of the release, many issues are discovered and this leads to major delays. In the meantime, the last stable release becomes out of date and because of the delay developers may try to push new features into the system.

- Problem of coordination: development in FOSS projects is done in a massively parallel way in which individual developers independently work on features they are interested in. Towards the release, all of this development needs to be aligned and these parallel streams have to stabilize at the same time. Given the size and complexity of some projects, significant coordination efforts are needed. This coordination is difficult, not only because of the size of a project but because participants are volunteers who are geographically dispersed. As in the previous problem, delays can occur and because developers see that there is more time they make more changes, leading to further delays.

3.2.7. Discussion

This study has shed light on practices and problems related to release management in FOSS projects, an important area which has so far been relatively unexplored. A major observation regarding development releases is the strong impact that tools have had on the development practices in the last few years. While development releases played a very important role in the past, with some projects such as the Linux kernel making new releases available daily during certain periods (Torvalds and Diamond 2001), they have to a large extent been replaced by the ubiquity of version control systems through which developers can retrieve the latest source code and changes.

The second observation is that the preparation of user releases is associated with considerable problems, in particular in large projects. Small projects often face human resource problems but large projects deal with a more fundamental issue, namely that of coordinating a loosely defined team of virtual volunteers towards the common goal of making a release. These volunteers typically work independently in parallel development streams which require little coordination with other members of the project. However, during the preparation for the next release, these parallel streams need to be integrated

which requires coordination. Even more problematic is that each independent development stream has to be completed at the same time even though there is usually no schedule which provides guidance.

Ted Ts'o, a prominent Linux kernel developer, has eloquently described this problem as the “thundering herd of patches”.⁵ When a freeze is announced out of the blue, everyone wants to get their features and fixes (patches) in and the high number of changes pushes back the release date. Each delay is seen as a further opportunity to make more changes, thereby causing even more delays.

Project	Release Cycle
GCC	6 months
GNOME	6 months
OpenOffice.org	3 months
Ubuntu	6 months
X.org	6 months

Table 3.3.: Projects using time based release management.

The present study has identified these problems in a number of projects but it also found evidence that some projects have a release strategy which is working very well. According to some interviewees, time based release management, in which time rather than features is used as orientation for the release and a well planned schedule is published, is successfully used in a growing number of projects (see table 3.3 for a list of projects from this study which have started to employ a time based release strategy). This strategy promises solutions to cope with the complexity that occurs when a large team of distributed volunteers has to be coordinated toward a common goal. This release strategy will subsequently be investigated in more detail.

3.3. Theory

It has long been known that the coordination of engineering work is very difficult (Herbsleb and Mockus 2003) but the literature (Yamauchi et al. 2000) and this exploratory study show that there are additional challenges when volunteers are involved. The great complexity found in distributed teams

⁵<http://lwn.net/Articles/3467/>

with volunteers performing parallel development work can lead to management and coordination problems, in particular during times of release preparation when all work needs to become aligned. In the following, theories regarding organizational complexity and coordination will be discussed to establish a basis for further investigation of this matter.

3.3.1. Modularity and Complexity

Economic theory states that the problem of organization emerges when there is a “need to coordinate different human capabilities and comparative advantages to accomplish a variety of tasks in order to achieve an end” (Garzarelli and Galoppini 2003). Similarly, it is argued that the prime constraint on programming today is not physical resources but instead the “very complexity of what we are trying to do, and the limits of our ability to manage that complexity” (Baetjer 1997).

Complexity is characterized by two factors: the number of distinct parts in a system, and the interconnections and interdependencies between these parts (Simon 1962). Modularity is a set of principles for managing complexity (Langlois 2002) and it is believed to be increasingly important today because of growing complexity of modern technology (Baldwin and Clark 1997). While the idea of modularity has been applied to technological design for a long time, its application to organizational design is relatively recent (Langlois 2002).

The basic premise of modularity is to reduce the number of distinct elements by grouping elements into a smaller number of systems, thereby hiding the individual elements. The criterion is decomposability (Simon 1962) because through it a problem is decomposed into subsystems and subproblems that different people can then work on (Baetjer 1997). A modular system is not automatically a decomposable one because it is possible to break a system into modules whose “internal workings remain highly interdependent with the internal workings of other modules” (Langlois 2002). The goal is to remove such interdependencies so individual modules can be dealt with independently.

A decomposed system has many advantages because it allows division of labour with limited communication through well defined channels. However, if a project is “organized in a nondecomposable way, then interdependency will

be high, meaning that each development team will need constantly to receive and use information about what all the other development teams are doing” (Langlois 2002). This behaviour can be found in the famous analysis of the development of the OS/360 operating system (Brooks 2000). It came to the conclusion that at some point the communication costs outweigh the benefits of the division of labour and described the paradoxical effect that adding more people to a late project will make it even more late.

This example illustrates the benefits of a modular organization if the problem is decomposable. Further support for modularity comes from Parnas (1972) whose information hiding approach has successfully been implemented in the object oriented programming paradigm. Instead of sending all information to everyone, as done in the development effort described in Brooks (2000), the idea is to minimize interdependencies and keep communication to the minimum.

3.3.2. Coordination in FOSS

Modularity is a key part of the FOSS philosophy (Raymond 2003) and there is great evidence that FOSS is highly modular (Lerner and Tirole 2002; Iannacci 2005; Narduzzo and Rossi 2004). It is believed that this characteristic allows a large group of loosely connected volunteers to work on the same project in a highly parallel fashion with relatively little coordination. Concurrent development has been observed with many different streams of development going on at the same time and it has been suggested that the frequent release cycle of many FOSS projects acts as a mechanism to synchronize development (Johnson 2001). It has also been suggested that different people work on the same problem and that once solutions are available the best one is accepted and integrated (Aberdour 2007).

Yamauchi et al. (2000) performed a study about coordination mechanisms in FOSS projects and came to the conclusion that coordination is performed after the work is done. They argue that the limited communication channels used by FOSS developers make it difficult to discuss a plan in detail. Instead, it favours a plan of action, in which a FOSS developer implements a certain goal and after this is done proposes it for discussion. This is in contrast to

upfront coordination performed in most projects but Yamauchi et al. (2000) find that it is an effective means of coordination given the circumstances FOSS operates under.

This type of coordination after action is in line with a characteristic that has been described before: work is generally not assigned in FOSS projects but is independently chosen by project participants (Mockus et al. 2002; Aberdour 2007; Crowston et al. 2005). This mechanism works because volunteers are self-motivated and this form of self-selection is highly efficient because it “lets agents be aligned with the problems they are most inclined to solve anyway — in a way, one strives to let people be their own principals” (Garzarelli and Galoppini 2003).

The modularity found in FOSS projects allows this division of labour (Narduzzo and Rossi 2004). Modularity is known to decrease internal coordination costs which emerge when an individual tries to coordinate a number of their own activities simultaneously (Garzarelli 2003; Garzarelli and Galoppini 2003). At the same time, it increases external coordination costs, for example the coordination of different programmers simultaneously working on the same project. Essentially, external coordination costs are an overhead deriving from decreasing internal coordination costs (Houthakker 1956). Division of labour allows individuals to bring in their unique skills and capabilities, but it is in the nature of heterogeneous capabilities that they need frequent coordination (Garzarelli and Galoppini 2003). While an organization with similar capabilities is easier to manage, production rests on the coordination of activities that are complementary (Richardson 1972).

3.3.3. Coordination Theory

Coordination is very important and everyone has an intuitive sense of what it implies but finding a good definition can be quite hard. One broad definition says that coordination “consists of protocols, tasks and decision mechanisms designed to achieve concerted actions between and among interdependent units, be they organizations, departments or individual actors” (Iannacci 2005). A much simpler definition is given by Malone and Crowston (1994) who state that “coordination is managing dependencies between activities”.

Their definition relies on the simple fact that if there is “no interdependence, there is nothing to coordinate” and it is in line with organization theory that emphasizes the importance of interdependence (Malone and Crowston 1994).

Coordination theory provides an approach to studying processes that are employed by an organization to perform specific activities and dependencies that occur while these activities are carried out. In particular, it is an approach to analyzing and redesigning processes. Coordination theory can therefore be employed to identify alternative processes that would be more suitable for performing a desired task than the existing processes in use by an organization (Crowston 1997). While many organizations perform similar activities, such as bug tracking in organizations that produce software, the specific processes employed by different organizations can vary in many details. There can be differences as to “how abstract tasks are decomposed into activities, who performs particular activities, and how they are assigned. In other words, processes differ in how they are coordinated” (Crowston 1997).

Processes are coordinated in different ways, but organizations often face similar problems that are managed similarly among different organizations. When actors try to perform their activities, there are dependencies that influence and limit how particular tasks can be carried out. In order to overcome these challenges, also known as coordination problems, specific additional activities have to be performed (Crowston 1997). When analyzing activities, coordination theory therefore makes a distinction between the activities of a process that are needed to achieve the goal and those that are performed to manage dependencies. Coordination theory is mostly concerned with the latter, namely the coordination mechanisms. Some of these coordination mechanisms can be specific to a situation but many are general as they pertain to different situations and organizations. Malone et al. (1993) have proposed a framework of coordination theory which can be used to analyze general coordination mechanisms based on the dependencies they are aiming to address. This framework has identified the following kinds of dependencies (see table 3.4):

- Shared resources: a resource allocation process is needed when multiple activities require access to the same limited resource (such as time or storage space). An organization can use different mechanisms to deal

Dependency	Coordination processes
Shared resources	‘First come, first serve’, priority order, budgets, managerial decision, market-like bidding
Task assignments	
Producer/consumer relationships	
Prerequisite constraints	Notification, sequencing, tracking
Transfer	Inventory management
Usability	Standardization, ask users, participatory design
Simultaneity constraints	Scheduling, synchronization
Tasks and subtasks	Goal selection, task decomposition

Table 3.4.: Common dependencies between activities (Malone et al. 1993).

with this dependency. A simple strategy would be a ‘first come, first serve’ approach but it is unlikely that this mechanism is appropriate in complex situations because it might stall activities with high importance or urgency. Another way to address this dependency would be to perform bidding within the organization in a similar fashion as it would be done on a market.

- Producer/consumer relationships: these dependencies occur when an activity produces something that is used by another activity. This dependency has three different forms:
 1. Prerequisite constraints: the producer activity needs to be completed before the consumer activity can begin. A notification process needs to be put in place so the consumer activity can immediately start when the producer activity has been completed. Furthermore, organizations can perform active tracking to make sure prerequisite constraints are fulfilled, for example by identifying activities on the critical path.
 2. Transfer: when the producer activity creates something that must be used in the consumer activity, some kind of transfer has to happen. In some cases, the consumer activity is performed immediately after the producer activity is completed, so the output can be used directly without requiring storage. It is more common, however, that finished items need to be stored for some time before they are

being used by a consumer activity. Hence, an inventory of completed items is often maintained.

3. Usability: what the producer creates must be usable by the consumer. This can be done through standardization, but it is also possible to simply ask the users what characteristics they want. Another mechanism is participatory design, in which “users of a product actively participate in its design” (Malone et al. 1993).
- Simultaneity constraints: this dependency occurs when activities have to happen at the same time (or cannot occur simultaneously). One mechanism to ensure that dependencies will not interfere is the creation of a schedule. Synchronization can be used to make sure that events happen at the same time, for example by arranging for several people to attend a meeting at the same time.
 - Tasks and subtasks: this activity can occur when a goal is divided into subgoals (or activities) which are needed in order to attain the overall goal. It is possible to start with the overall goal and then decompose the task into different subgoals in a top-down fashion. A bottom-up approach would be goal selection where a number of individuals realize that they could combine their activities to achieve a higher goal.

While Malone et al. (1993) state that there are more forms of dependencies yet to be analyzed, the current model will act as a good framework according to which coordination issues related to release management in FOSS can be observed.

3.4. Chapter Summary

This chapter showed that iterative development, which is used in FOSS, is a development model of growing importance because it allows to deliver solutions faster and obtain solid feedback which can subsequently be used to refine the software. Exploratory interviews have identified a number of practices and problems with release management in FOSS projects. In particular, release management in complex projects has been identified as problematic.

These problems are related to challenges of coordination during release preparation to align all volunteer developers. Subsequently, theories of organizational complexity and coordination have been discussed which are related to the challenges that have been identified. The next chapter will formulate the research question, which is based on the findings of this exploratory study, and will discuss the research approach and methodology.

4. Research Question and Methodology

In this chapter the research question which will be the focus of this dissertation will be presented, followed by more detailed questions based on the overall research question. Subsequently, there will be a discussion of an appropriate research approach and methodology which will be employed to address the specific research questions that have been raised. Finally, a description of the selection criteria for FOSS projects to be studied as well as methodological issues regarding the rigour of this study will be discussed.

4.1. Research Question

The aim of this dissertation is to investigate quality improvement in FOSS projects and such an endeavour has to start with the identification of actual quality problems. Given the lack of studies concerning problems with the FOSS methodology, an exploratory study has been performed in section 2.3 about quality practices and problems. Based on these findings the focus has subsequently been on release management as one problematic area and a further exploratory study in section 3.2 has revealed several problems related to release management in FOSS projects.

While release management in small FOSS projects is mostly a matter of administrative tasks which can be associated with problems, such as a lack of human resources, large projects face a more fundamental problem. The FOSS methodology relies on parallel development streams independently conducted and work is often self-assigned (Mockus, Fielding, and Herbsleb 2002), requiring little coordination. This pattern breaks down during the preparation of a release because in this phase every participant suddenly has to align their work

with the whole project and coordination is required to achieve this task. As the exploratory study in section 3.2 has shown, this phase is often associated with delays and other problems.

Based on these findings, the underlying research question can be formulated as follows:

What are the factors involved in the coordination of a distributed volunteer software developer community that affect its ability to deliver timely releases of high quality software of value to a user community?

The exploratory study about release management has suggested that there are different release strategies and that the novel strategy of time based release management may be more effective than other strategies. This is an interesting observation that deserves more attention. Therefore, the following two more focused questions related to the broad one above can be formulated:

1. Why do FOSS projects which employ a time based release strategy apparently not face certain problems that other projects have?
2. In complex FOSS projects where a time based release strategy is adopted, what are the conditions that lead to effective implementation?

These are the questions that will be investigated in this dissertation using the methodology described in the following section. The research will be guided by theories about organizational complexity and coordination presented in section 3.3.

4.2. The Case Study Approach

One of the most important considerations to take into account in academic research is to use the research methodology which is deemed as the most appropriate to investigate the question raised by a researcher (Gill and Johnson 1997). Yin (1994) identifies the following three conditions in social research which influence the choice of the research strategy:

1. The type of research question posed.

2. The extent of control an investigator has over actual behavioural events.
3. The degree of focus on contemporary as opposed to historical events.

The research questions of this dissertation are of the types of “why” and “how” — essentially, the questions are: why does the time based release strategy work well for some projects and how have projects implemented this strategy? For this type of questions, which are of an explanatory nature, the use of case studies, histories and experiments is appropriate. Yin (1994) argues that this is because “such questions deal with operational links needing to be traced over time, rather than mere frequencies or incidence”.

The focus on contemporary events rules out histories and experiments are not possible because there is relatively little control over the object of study. FOSS projects are live organizations, often with several hundred or thousand members, which feature a very complex social structure and there are many factors which may have an impact on the question under investigation. The age, nature and size are just some examples of such factors and there are many more subtle factors that are not under the control of the researcher. It is also impossible to find a control group which only differs in one variable or to actively affect a project in a way that would only change the aspect of interest to the researcher.

Case studies, on the other hand, are a good approach to understand complex social phenomena and they allow “an investigation to retain the holistic and meaningful characteristics of real-life events” (Yin 1994). The unique strength of the case study approach is its “ability to deal with a full variety of evidence” (Yin 1994). This is exactly the approach needed to exploit the rich repertoire of information available about FOSS and to come to an understanding as to why and how some FOSS projects effectively employ a time based release strategy.

One question that is often raised is whether the findings from a case study are generalizable. While it is not possible to treat a case study as a sample from which conclusions are drawn about a bigger population in the manner of a statistical generalization, case studies allow analytic generalization (Yin 1994). In this approach, the findings are related to a theory and from this theory generalization is possible. In the following, the findings from the case study will therefore be linked to the theories about organizational complex-

ity and coordination presented in section 3.3. The generalizability of a case study thereby depends on the quality of the strategies selected in the research and the methodological rigour with which the research has been performed (Hamel 1993), matters that will be discussed in the following sections, in particular in section 4.6. The proposition of this research is that the time based release strategy acts as a coordination mechanism that allows large projects to cope better with the complexity arising through the collaboration of many heterogeneous capabilities.

4.3. Selection Criteria

One basic question regarding the selection of case studies is the unit of analysis. As mentioned above, the focus will be on large volunteer organizations producing FOSS. There is a terminological ambiguity that needs explanation. Instead of using the term ‘organization’, the word ‘project’ is usually used to refer to an entity producing FOSS. This is because many FOSS projects are not associated with an organization in the legal sense — only a fraction of projects have formed non-profit organizations, or similar entities, to support their endeavour (O’Mahony 2003). Unlike the traditional meaning of the term ‘project’, the term as applied to FOSS does therefore not describe a set of activities that an organization undertakes for a specific amount of time. Instead, the term refers to the whole organization itself whose aim it is to write software that fulfils a specific purpose. For example, the aim of the OpenOffice.org project is to create an office suite that can compete with and surpass Microsoft Office and similar products.

FOSS projects are established for a specific purpose and are therefore relatively independent. Nevertheless, they are part of a bigger community and ecosystem, with which they interact. For example, projects typically make use of software produced by other FOSS projects, such as important development libraries or tools. There are also a number of vendors which take software produced by FOSS projects, integrate them to one system and publish them on CDs. These vendors, often known as distributors, play an important role in the FOSS ecosystem since they provide a bridge between users and developers, and they often actively participate in the development process of some,

especially popular and important, FOSS projects. As such, it is important to take these vendors into account in the case studies.

Even though individual FOSS projects are the main unit of analysis since it is their task to develop and release software, the interaction with other projects and vendors will be studied as well when it pertains to release management. This is done by two means:

1. Interviews will not only be performed with members of the FOSS project under investigation but also with one developer from a vendor who is actively involved in this particular project.
2. One of the case studies will focus on Debian, which is a volunteer project with the aim of integrating the output from other FOSS projects into a system which is easy to install and use. As such, Debian plays a similar role to other commercial FOSS vendors, such as Red Hat and Novell, even though the project operates on a non-profit and volunteer basis.

This allows a richer analysis of release management because these vendors may provide important insights. After all, they directly interact with users, and they rely on the delivery of software by FOSS projects so it can be integrated into their systems. These systems, which are integrated collection of software developed by individual projects, are commonly based on Linux and are widely known as Linux distributions. Figure 4.1 depicts the relationship between users, FOSS projects, and Linux distributions or similar vendors.

After this clarification of the unit of analysis, criteria for the selection of the case studies for this research will be given. First of all, there are five criteria that projects must meet in order to be taken into consideration. A project must be:

1. Considerably complex: the exploratory study in section 3.2 showed that large projects face very different challenges regarding release management than smaller and less complex projects. The real challenge, that of coordination, is associated with large and complex projects, which are therefore chosen for this research. Instead of using specific figures for size, such as the number of contributors or lines of code, the theory proposed by Raymond (1999) will be used to judge whether a project

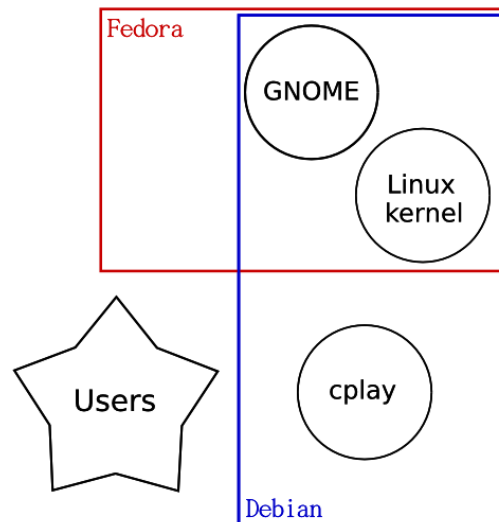


Figure 4.1.: The relationship between users, projects and Linux distributions: distributions, such as Fedora and Debian, incorporate FOSS projects, such as GNOME and the Linux kernel, and distribute the software to users.

has sufficient complexity. This theory describes different organizational structures in FOSS projects, as described in section 2.2.2. One of the characteristics of the bazaar structure is that it is based on division of labour. The selection criteria will therefore be whether there is a dedicated release manager, release team or similar role, as this is evidence that a project has reached considerable complexity (Crowston and Howison 2005; Iannacci 2003).

2. Voluntary: as argued before, projects based on volunteer participants often face unique challenges, particularly regarding the coordination of a project. The reason for this is that projects have little control over what and how these volunteer participants contribute. There are different approaches as to the meaning of a volunteer. In contrast to Robles, Gonzalez-Barahona, and Michlmayr (2005), which defines volunteers as developers participating “in their free time [and] not profiting economically in a direct way from their effort”, volunteer status will here be defined based on control. Companies increasingly contribute to FOSS projects, but their contributions may not be relevant in terms of release management. Unless the project leader or release manager can specify what someone works on, they are treated as volunteers because the project has no control over them, and they are free to do whatever they

(or their employers) wish. There is no guarantee that they will actively participate in the release process, and the lack of control over contributors in these non-coercive relationships introduces challenges related to the coordination of a project.

3. Distributed: the research interest of this dissertation is to study how distributed volunteer projects can effectively work together to perform releases. The projects therefore have to be distributed in a geographical sense.
4. Time based: since the time based release strategy is investigated in this dissertation, only projects that have switched or are currently moving to this release strategy will be considered.
5. Licensed as FOSS: this research focuses on collaborative development made possible by the nature of free and open source software and hence projects must follow such a license.

In addition to these criteria, projects have been chosen in a way that they give a good representation of different projects found in the FOSS community. This allows the identification of factors that might influence a project's release strategy or its implementation. In particular, projects have been considered that differ in their:

- Background: the nature and background of a project might be an important influence. The case studies represent projects that originated in a commercial environment as well as true community projects. There are projects that are mainly aimed at other developers as well as projects that write software for end-users.
- Release stage: projects at different stages regarding the implementation of a time based release strategy will be studied. Some projects are currently moving to a time based release strategy whereas others have implemented it successfully several years ago. No project could be found which is currently moving away from time based releases.

In summary, the focus of this research is on complex, volunteer based FOSS projects that have or are currently implementing a time based release strategy.

Case studies with a variety of projects will be performed in order to study different factors that might influence the choice or implementation of a project's release strategy.

4.4. Case Study Projects

Based on the criteria set out in the previous section, seven projects have been selected for in-depth case studies. In the following, these projects will be described briefly to give information about their background, show that they meet the five criteria from the previous section, and to illustrate unique aspects of each project. The case studies will be presented in more detail in chapter 5.

Project	Interval	Introduction
Debian	15-18 months	after their 3.1 release in June 2005
GCC	6 months	2001
GNOME	6 months	beginning of 2003
Linux kernel	2 week merge window, releases every 3-4 months	middle of 2005
OpenOffice.org	3 months	beginning of 2005
Plone	6 months	beginning of 2006
X.org	6 months	end of 2005

Table 4.1.: The release cycle of the case study projects and when they have implemented the current release strategy based on time based releases.

- Debian: the aim of this project is to integrate software produced by other projects and create a complete operating system based on FOSS. Debian originated as a community project and while a number of companies, such as HP, make important contributions, the majority of contributors are volunteers in the sense that they participate in their spare time. The project itself does not pay any developers, and contributors from all over the world participate in it.¹ The project has over a thousand developers, consists of over 10,000 individual software packages and can therefore be considered as one of the most complex FOSS projects in existence. This

¹A map of the world showing where Debian developers are located can be found at <http://www.us.debian.org/devel/developers.loc>

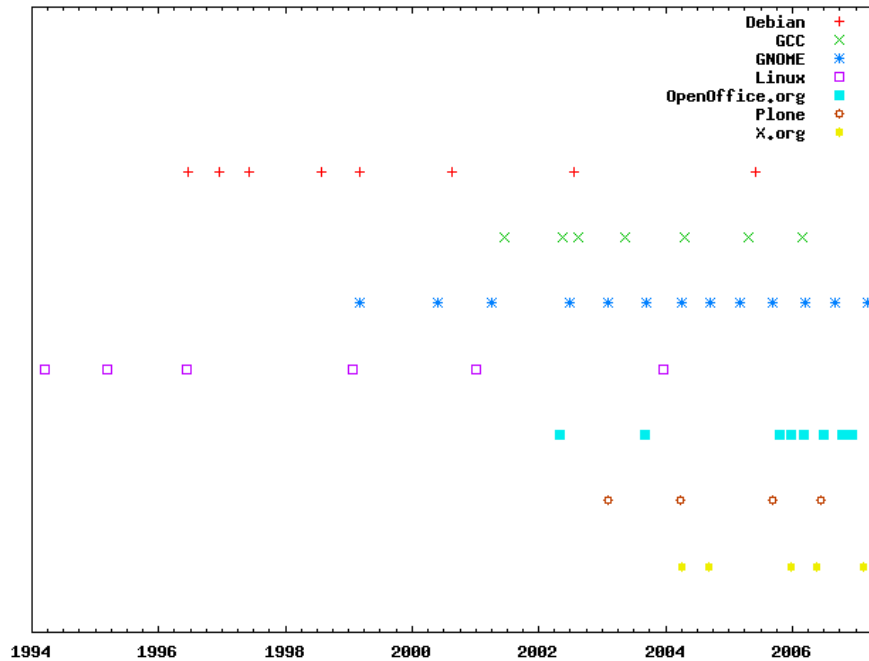


Figure 4.2.: Releases made by the seven projects selected for in-depth case studies.

complex nature of the project is reflected in its structure. The project has a dedicated project leader, a release team and many other specific positions. In the past, there used to be a single release manager but in 2003 the release manager asked for additional release assistants. Since then, the project has moved to a release team consisting of two release managers and several release assistants. The project has faced severe delays in recent years and has decided to move to time based releases after their 3.1 release in 2005.

- GCC: the GNU Compiler Collection is a compiler suite which supports a number of programming languages, such as C and C++. It is a very important development tool and is the standard compiler among FOSS projects. The project has a dedicated release manager and a steering committee, which among other tasks appoints new maintainers. Participants come from all over the world and there is high commercial interest in GCC, but the project itself has no paid staff. In theory, the project follows a time based release with an interval of six months. In practice, the project has released only one new version every year in recent times.²

²<http://lwn.net/Articles/192529/>

- GNOME: this project provides a complete desktop environment that is easy to use. The project started as a community project and is mainly led by volunteers, even though there is a fairly high level of participation of vendors that ship GNOME as part of their operating systems. There are several hundreds of contributors, a dedicated release team and a supporting organization, the GNOME Foundation. The project has published time based releases every six months for a number of years now and is considered as the reference model for a good implementation of time based release management.
- Linux kernel: this is one of the most prominent FOSS projects. It provides the part of an operating system that interacts with the hardware and provides important resources for other software applications. The project was started by Linus Torvalds while he was a student and has grown into a project with a large community. The Linux Foundation pays the project founder and a small number of other developers, but the majority of contributors are not paid by the project. The project has implemented a number of changes to their development and release process during the 2.6 series, most notably the introduction of time based releases based on a two week ‘merge window’ after which no new features are accepted.
- OpenOffice.org: the aim of this project is to provide a complete office suite, consisting of a word processor, spreadsheet and other applications. The project originated as a commercial product of a German software company, but was later bought by Sun Microsystems who subsequently made it available as FOSS. Sun still maintains high levels of control of the project, such as by paying the release manager, but there is significant community involvement. The project moved to a three month time based release cycle in the beginning of 2005. At the end of 2006, a proposal to move to a six month release interval was published.
- Plone: this project has created a content management system for the web. The project has a dedicated release manager and consists of volunteers. In 2006, the project decided to move to a time based release

strategy in order to synchronize their releases with that of Zope, an application server on which they build.

- X.org: this project was created in order to take over development from the XFree86 project which made a number of decisions which the community did not agree with. While the project is relatively young, the software itself is more than a decade old. With their 7.0 release in December 2005, X.org moved to six month time based releases. The project has a dedicated release team, is supported by the X.org Foundation and has no paid developers.

Each of these projects meet the selection criteria outlined before. They are all distributed FOSS projects that mainly rely on voluntary contributions. All of the selected projects have deployed, or are currently in the process of moving to, time based release management (see table 4.1). All of the projects are of considerable complexity and have dedicated release managers or teams. Even though complexity rather than lines of code is the selection criteria, it should be noted that four of the seven projects are in the list of the largest ten FOSS projects according to a study of the Debian distribution (Amor et al. 2005) — and Debian itself is one of the case study projects. This supports the claim that the projects selected for this research truly are some of the largest and most complex FOSS projects (see table 4.2).

Project	Lines of Code
Debian	229,495,824
OpenOffice.org	5,181,000
Linux kernel	4,043,000
GCC	2,422,000
XFree86/X.org	2,316,000

Table 4.2.: Size of projects in lines of code (Amor et al. 2005).

4.5. Sources of Evidence

Most of the evidence of this study is based on qualitative data. According to Hamel (1993) qualitative data is ideally suited for “describing, understand-

ing, and explaining”. Furthermore, Miles and Huberman (1994) argue that qualitative data have a number of positive attributes:

- They focus on naturally occurring, ordinary events in natural settings. This allows the researcher to find out what “real life” is like.
- Their richness and holism which allow a researcher to reveal complex matters in a vivid way.
- That they are usually collected over a sustained period which makes them a powerful tool to study a process. Indeed, the projects of this study were closely followed for well over two years, leading to a good understanding of their processes.

Given that the study is about projects that mainly collaborate via the Internet, a large body of evidence is in one form or another based on text. The majority of interviews were conducted in person or over the phone and direct observation took place at conferences but other evidence consists of text. While text is a lean communication medium, there are also advantages of text. For example, Thomsen, Straubhaar, and Bolyard (1998) argue that in “one sense, there is less for the ethnographer to miss in a text-based world of interaction. All speech, behavior, community rules, and community history is, in principle, likely to be available online for the researcher’s inspection”.

The case study projects outlined in the previous section were observed and investigated in-depth over the course of well over two years. Additionally, mailing list archives and other public documents going back several years were analyzed during this research, leading to a good understanding of the projects under investigation in this research. The qualitative data gathered during this research was interpreted through pattern matching which allowed the creation of concepts and assignment of patterns to these concepts. No scoring system, according to which the frequency of specific patterns are counted, was employed since the emphasis of this research was on holistic understanding of practices and problems rather than merely the frequency thereof.

4.5.1. Interviews

Interviews formed an integral part of this study. For every project, typically three people were invited for an interview. Most interviews were conducted in person at FOSS conferences or over the phone, recorded and transcribed verbatim, but two interviews were done via e-mail. In terms of audience, the interviews were clearly targeted at FOSS developers who played a key role in their project in release management, such as the release manager or a member of the release team. Additionally, for most projects, a developer working for a vendor, such as a Linux distributor, was interviewed. Their perspective was also very important because they can be seen as the connection between developers of the software on one hand and the actual users on the other hand. Finally, for some projects we found developers who were critical of or had a contrary view to the current release strategy employed by a project. This mix of interviewees ensures that the question of release management is investigated from different perspectives (see appendix B.1.2 for the list of interviewees and table 4.3 for a check list showing which type of people were interviewed for each project). There was a set of questions that were used during every interview (see appendix B.1.1). In addition to the questions everybody was asked to answer, focused questions specific to the project of the interviewee were asked based on previous observation of the project and analysis of the mailing list archives.

Project	Release Manager	Vendor
Debian	✓	N/A
GCC	✓	✓
GNOME	✓	✓
Linux	✓	—
OpenOffice.org	✓	✓
Plone	✓	—
X.org	✓	✓

Table 4.3.: A check list showing the coverage of the interviews.

4.5.2. Mailing Lists

Mailing lists are the main communication channel for FOSS projects and they are usually archived. The analysis of mailing lists allowed the development

of a good historical understanding of how a project had come to its current status regarding release management.

During the study, contemporary discussions were read completely and past discussions were searched for according to certain patterns, such as:

- release, release management, release coordination
- schedule
- time frame
- deadline
- time based
- delay

A list of mailing lists that were read and whose archives were consulted can be found in appendix B.2.

4.5.3. Documents

In addition to mailing list postings, several other documents were used for this study, largely documents written by the community and shared on the Internet. These came from a variety of sources. For example, the web sites of the projects under investigation were very informative as were personal journals and blogs of FOSS developers and users. Finally, several FOSS news sites were followed in order to find current trends and discussions regarding release management (see table 4.4).

News site	Web Address
Kernel Trap	http://www.kerneltrap.org
Linux Today	http://www.linuxtoday.com
Linux Weekly News	http://www.lwn.net
Newsforge	http://www.newsforge.com
OS News	http://www.osnews.com
Slashdot	http://www.slashdot.org

Table 4.4.: News sites followed during the research.

4.5.4. Observation

Direct observation was employed at developers' meetings and conferences. Furthermore, recordings of conference talks about release management and similar areas of interest published on the Internet were studied.

4.6. Methodological Considerations

There are many tests which are used to establish the quality of empirical social research. The most common four tests are construct validity, internal validity, external validity and reliability. In the following, several strategies suggested by Yin (1994) will be presented that have been employed throughout the design, implementation and analysis of the case study of this dissertation to ensure its methodological rigour and quality. In addition, some ethical issues that are important in social research will be discussed.

4.6.1. Construct Validity

Construct validity is given when a scale (that is, operational measures) actually measures the social construct or concept that it is supposed to measure. Construct validity can be improved through the following methods:

- The use of multiple sources of evidence: as section 4.5 shows, various sources of evidence have been taken into consideration. The strength of a case study is that multiple sources of information can be taken into account and this was done in this study. Multiple data sources allow data triangulation which addresses problems of construct validity because multiple measures of the same phenomenon are provided through the different sources of evidence.
- To establish a chain of evidence: the findings of the case study are used to build an explanation and different evidence contributes to the final conclusions. This chain of evidence improves construct validity.
- To have key informants review the draft case study report: an initial draft of this dissertation was sent to at least one person from each project who

participated in the interviews and feedback was subsequently incorporated.

4.6.2. Internal Validity

Internal validity is given when a causal relation between two variables can be properly demonstrated as opposed to being spurious relationships. This research has gathered rich qualitative data through which a good understanding of underlying relationships has been developed. This solid understanding of observed phenomena has been combined with theory to generate explanations of what is happening and why relationships between variables exist (Eisenhardt 1989). This approach ensures that the requirements for internal validity are met in this study.

4.6.3. External Validity

External validity is concerned with the applicability of a theory in a setting different from the one where it was developed. It is closely linked with the aim of a theory to be generalizable because a “theory that lacks such generalizability also lacks usefulness” (Lee and Baskerville 2003). This research followed an approach of analytic generalizability as defined by Yin (1994). Yin also refers to this approach as level 2 inference and recently this type of generalizability has been covered in a framework by Lee and Baskerville (2003). They classify analytic generalizability as type ET generalizability since the researcher generalizes from empirical statements to theoretical statements.

According to Walsham (1995), this form of research allows four types of generalizations. The present research has led to the generation of generalizations in each of those four categories:

1. Development of concepts: the two concepts regularity and the use of schedules have been developed in the light of release management and coordination in large software projects and operationalizations for them have been presented.
2. Generation of theory: the rich data gathered in this study about FOSS

projects has been used to generate a theory of time based release management.

3. Drawing of specific implications: a number of implications follow from the theory that have practical value to the FOSS community, such as insights as to factors influencing the choice of an appropriate release interval for a project.
4. Contribution of rich insight: since this research has gathered qualitative data from key personnel involved in release management, rich insights about the motives behind specific practices found in the FOSS community have been obtained. A good understanding of problems that can often be observed in FOSS projects and their causes has also been developed.

4.6.4. Reliability

Reliability is given when the steps of an investigation, such as data collection, can be repeated and lead to the same results. Reliability has been promoted through the maintenance of a case study database. Throughout this study, documents and other information, such as interviews, were recorded. Furthermore, a precise case study protocol was followed. The procedures of the protocol include the selection of appropriate projects for this study, the choice of key FOSS developers for the interviews of this research as well as specific interview questions that every interviewee was asked. The analysis of the data followed standard practices as described by Miles and Huberman (1994) and Yin (1994).

4.6.5. Personal Involvement

It must be noted that the researcher of this dissertation has been personally involved in FOSS projects for over ten years. This personal involvement leads to some methodological considerations that need to be taken into account.

On the one hand, this personal involvement is associated with a solid understanding of the FOSS community from which this research benefits. For example, the researcher has good access to the FOSS community and can draw

on many resources, both of which contribute to a comprehensive investigation of the research question.

At the same time, the personal involvement may introduce potential bias in this research. Two mechanisms have been implemented in order to ensure that no bias is introduced in this research. First, the project the researcher is involved in, Debian, is only one out of seven projects from which conclusions are drawn. Second, as mentioned above, several key FOSS participants were asked to review the findings of this research and their comments were incorporated into this dissertation to make sure a fair representation of release management of FOSS is given.

4.6.6. Ethical Issues

Since the objective of this study is to investigate a social phenomenon in which human beings are involved, ethical issues need to be taken into consideration. Everyone who participated in the interviews conducted as part of this research was informed about the nature of this research and had to give their consent to be part of this study. They were asked for permission to record the interview and transcribe it verbatim and were told that the recording would only be used as part of the analysis of this study. Where verbatim quotations are given in this dissertation, permission has been sought from each individual. Furthermore, to establish the credibility of this study interviewees were asked for permission to be included in a list of interviewees showing their name and role in their project (see appendix B.1.2). Everyone except one gave permission for this.

Various information sources from the Internet, such as web pages and mailing archives, were consulted throughout this study. It can be assumed that the publication of information on the web or on public and openly archived mailing lists implies automatic consent for this information to be used by others (taking copyright and other laws into account).

4.7. Chapter Summary

The aim of this dissertation is to study why and how a time based release strategy has been implemented in some FOSS projects. The findings can be used to guide release management in other FOSS projects which experience problems with their current release strategy. In order to address the questions of this research, a case study approach has been taken and seven in-depth case studies have been performed. The next chapter will present the findings from the individual case studies.

5. Time Based Releases: Learning from 7 Case Studies

In this chapter, seven case studies will be presented which investigate how the introduction of a time based release strategy has affected projects. Each case study will investigate problems that prompted the change to a new release strategy, solutions that were implemented as well as outstanding problems.

5.1. Debian

Debian is a volunteer project with the aim of producing a complete operating system based on FOSS. Even though the project writes some software on its own, the majority of software is obtained from other sources and integrated into a system which plays together very well (González-Barahona et al. 2004; Robles et al. 2005; Robles et al. 2006). The main objective of the project is to perform this integration work and to produce a system that is stable and easy to install and use. The project also acts as an important interface between users on one side and FOSS developers on the other side. Members of Debian forward bug reports from users to the respective developers of the software and integrate bug fixes made by those developers, thereby making them available to users.

Debian was founded in 1993 and has produced several releases of its software distribution since then. In recent years, the project has faced severe delays with its releases, yielding the project a reputation for being slow and never on time. While the project made regular releases in the past, there has been a significant increase in the time between releases in the last few years: Debian 2.2 was released 17 months after version 2.1, version 3.0 followed 23 months later and then users had to wait almost three years for version 3.1 (see table 5.1). As a

Version	Release Date	Months
1.1	1996-06-17	
1.2	1996-12-12	6
1.3	1997-06-02	6
2.0	1998-07-24	14
2.1	1999-03-09	7
2.2	2000-08-14	17
3.0	2002-07-19	23
3.1	2005-06-06	35

Table 5.1.: Stable releases of Debian.

result of these huge delays, many developers no longer believe that the project can release on time and this perception may contribute to further delays.

This study has identified several problems with the way release management was performed in Debian. For a long time, Debian has taken a very ad-hoc approach to release management: there were no clear goals or milestones and release updates were posted only infrequently. Because of this lack of planning, freezes that were instituted in order to start the preparations of a release were usually announced out of the blue. At this point, many developers tried to make changes which they felt were needed before the next version was released (see figure 5.1). However, this flurry of activity had the opposite effect intended by the freeze announcement. Instead of slowing down development in order to prepare for a release, many changes were made and the freeze was pushed back. Another problem caused by bad planning was that some crucial issues which had to be resolved before the release were discovered only after all software was frozen. As a consequence, much software was out of date by the time it was finally released.

In order to combat these problems, the Debian project has made some structural changes during the 3.1 release cycle and has significantly improved the release process. The project moved away from a single release manager to a release team which consists of several developers actively working on the release. After the release of Debian 3.1, they decided that a release cycle of 15-18 months was appropriate for the project and announced a target date for version 4.0: December 4, 2006, approximately 18 months after the release of version 3.1:

We showed people very early in the release cycle that we have goals,

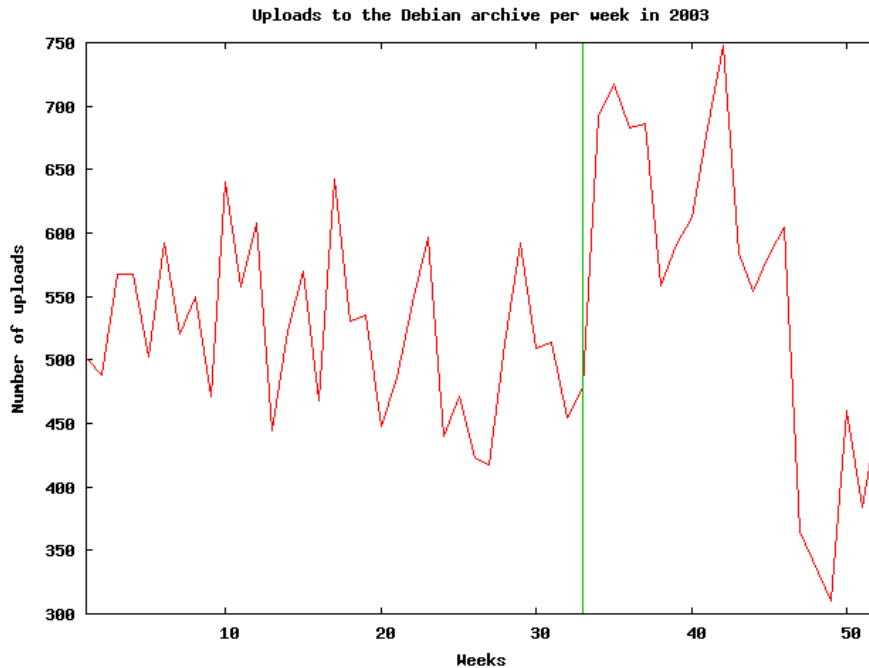


Figure 5.1.: Freeze and release announcements that are made out of the blue lead to increased changes instead of slowing down development: the vertical, green line indicates when a release plan was posted out of the blue. This is followed by a flurry of activity and it takes about two months before development becomes calm again.

that we have plans, that we're determined to really do it better, which is an important thing to make. (Andreas Barth, Debian)

Even though the project has not met this deadline, the release process of 4.0 has been a lot smoother compared to previous releases and many improvements were made. An important change is that the release team is more pro-active regarding release issues and keeps the whole project up to date regarding the release status. Updates are regularly sent out to the announcement list for developers and various information resources have been implemented which show the release status. This is possible because targets are now more clearly defined and kept track of. In particular, targets have been split into real blockers for the release and goals which are optional:

[Goals] still have a high visibility which allows people who would like them to happen to give the same attention to them as to release blockers but we're not going to hold our release for them. (Andreas Barth, Debian)

Another important move was to explicitly assign the responsibility for resolving blockers or implementing goals to specific teams or individuals. In the

past, many tasks fell into the hands of the release team simply because nobody felt responsible. A problematic example of this is the support for a wide range of different hardware architectures which is a unique feature of Debian. Sometimes there were issues with the support of a hardware architecture but nobody felt responsible. In a meeting in Vancouver, followed by a huge discussion on the mailing lists, specific criteria were defined which architectures had to fulfil in order to be considered for inclusion into a release. The release team also required at least five active developers for each architecture. Even though the decision to introduce these release criteria was initially quite controversial, it worked out very well in the end. Developers working on architecture support became more active and the explicit criteria which have been established show which work is necessary.

The release team also encouraged developers to help with bugs in software packages they are not responsible for. Most developers in Debian maintain software packages and there is a strong sense of ownership (Michlmayr 2004), but the release team has used its influence to make it easier for developers to fix packages maintained by someone else. This has not only led to many bugs being fixed but has also prompted developers to become more actively involved in the release process in general.

In summary, there have been a number of important improvements to the release process during the release cycle of Debian 3.1 and in particular during the preparation for version 4.0. The biggest challenge the Debian project still has to solve is to show its users and developers that the project can meet deadlines and release on time.

Past Problems

- Release management was not very organized and release updates were posted only infrequently. Because of this and the lack of a roadmap, freezes were often announced out of the blue.
- Due to the unorganized nature of the release, several new and unexpected blockers were found during the release process, leading to delays.
- The unexpected delays meant that software was frozen for a long time, in the case of Debian 3.1 for over a year. When this release was finally

published, many components were already out of date and would often not meet customer demands.

- The fact that Debian has experienced significant delays with several of its recent releases has led to problems with the image of the project. There is a perception that Debian is slow and cannot meet deadlines. This is also associated with frustration in the developer and user community.

Solutions

- The project has implemented more mature release management structures. In particular, Debian moved from a single release manager to a team during the 3.1 release cycle. The release is now handled by core release managers with the help of several release assistants.
- A release date for the next release has been set well in advance and there is more planning.
- Release announcements are sent more frequently and various information sources have been implemented through which developers can stay informed about the release status.
- Release targets have been defined better and there is a distinction between blockers and goals. Only blockers hold up the release whereas goals can be postponed for a future release.
- The release team has shifted the responsibility of achieving targets to specific developers and teams. There are better criteria now defining these responsibilities.
- The release team is increasingly giving encouragement to developers to help out with software packages and bugs which normally do not fall into their domain.
- A staged freeze has been implemented according to which software is frozen in stages according to its importance. The majority of software is now frozen when most blockers have been resolved.

- The use of the experimental repository has been promoted to make sure that the main development repository is in a good shape most of the time.

Outstanding Problems

- Developers still need to be convinced that targets can be met, that deadlines are real and that Debian can release on time.

5.2. GCC

GCC started as a compiler for the popular C programming language and used to be known as the GNU C Compiler. Over time, the scope of GCC changed as support for more languages, such as C++, was incorporated, and GCC is now known as the GNU Compiler Collection. GCC is a very important project because many FOSS projects rely on this compiler and there is also major deployment of GCC in closed source projects.

Even though the GCC project has a flourishing development community now, the project has a troubled past. In the middle of the 1990s, development of GCC was only open to a few people. The mailing list was by invitation only, development snapshots were not available to the public and it was hard to get features and fixes accepted:

Having a closed list, very long cycles between releases without allowing wide public testing of snapshots, with one person as bottleneck for all patches was a completely broken way to do something as large as GCC. (Joe Buck, GCC)

A group of people formed the EGCS team and produced a compiler based on GCC that incorporated many new features and bug fixes. The development environment was more open and releases were published regularly. In October 1998, Richard Stallman, the founder of the GNU Project, gave the EGCS team permission to take over development of GCC. The project instituted rigorous processes, such as high levels of peer review, and created a steering committee which has the power to appoint maintainers and make important decisions. The project split the development phase into three stages:¹

¹<http://gcc.gnu.org/develop.html>

1. In the first stage, major changes which have been proposed and accepted by the release manager can be incorporated in a controlled manner.
2. The second stage allows smaller improvements but major changes may no longer be merged.
3. The last stage is devoted to bug fixes and updates to the documentation.

This development method consisting of three stages has been introduced to ensure a controlled development process:

In a way it evolved from an earlier period when, like for many projects, there was a ‘develop like mad’ phase followed by a ‘freeze and stabilize’ phase, and the compiler would sometimes be broken badly by all of the last-minute code drops. (Joe Buck, GCC)

This development approach is also an important coordination mechanism because it shows developers which big projects are scheduled to be merged during stage one.

Over the years the GCC project has produced several releases. Since each of the three stages lasts two months, there should theoretically be a new release every six months. In January 2007, a discussion about the release cycle was started on the GCC development list. Someone argued that the release cycle has been getting longer over time (see table 5.2) and that this is a reflection of problems with the development process.

Version	Release Date	Months
3.0	2001-06-18	
3.1	2002-05-15	11
3.2	2002-08-14	3
3.3	2003-05-13	9
3.4.0	2004-04-18	11
4.0.0	2005-04-20	12
4.1.0	2006-02-28	10

Table 5.2.: Stable releases of GCC.

During this discussion, the release manager acknowledged that a lack of time on his part meant that he was not pushing other developers towards release preparations enough. Furthermore, it has been argued that there are problems with the branch criteria which say when the development tree is considered

for stabilization and subsequent release. The criteria say that there may be only 100 known regressions in the compiler but critics argue that this measure is wrong because it includes regressions which are also present in versions of GCC that have already been released. No firm conclusions have been reached at the time of writing, but the discussion shows that improvements to the release management process and the quality of GCC releases are possible.

Past Problems

- The GCC project suffered from a closed development style in the past: few people could make code changes and the mailing list was closed.
- There was a long time between releases and development versions, which contained bug fixes and features, were not available to the public.
- When development opened and picked up, significant code changes were often made which required a long stabilization phase.

Solutions

- The project moved to a more open development style and established a steering committee which has the power to appoint new maintainers and make important decisions.
- The development process was divided into three stages in order to coordinate code submissions and keep the development tree reasonable stable.
- All code submissions are peer reviewed on the development mailing list and need approval.
- Each development stage lasts two months, theoretically yielding a release every six months. A regular release cycle ensures that it is not the end of the world if a feature does not make it into the following release.

Outstanding Problems

- The release manager is busy and has not pushed the release forwards as much as would be possible.

- The branch criteria may need revision to make it easier to create a branch which leads to the next stable version.

5.3. GNOME

GNOME is a desktop environment for Linux and other Unix compatible systems. GNOME was started in 1997 because the major desktop environment for Linux at that time, KDE, relied on the Qt library from Trolltech that was not under a FOSS license. Even though Qt is now FOSS, GNOME has established itself and is installed as the default desktop environment on many Linux distributions.

Version	Release Date	Months
1.0	1999-03-03	
1.2	2000-05-25	15
1.4	2001-04-02	10
2.0	2002-06-27	15
2.2	2003-02-06	7
2.4	2003-09-11	7
2.6	2004-03-31	7
2.8	2004-09-15	6
2.10	2005-03-09	6
2.12	2005-09-07	6
2.14	2006-03-15	6
2.16	2006-09-06	6
2.18	2007-03-14	6

Table 5.3.: Stable releases of GNOME.

The exploratory interviews in chapter 3 have revealed that many FOSS developers consider GNOME with their six month schedule as the reference project for time based release management. Even though GNOME has a very smooth release process now, this was not always the case. The move towards a six month time based model in GNOME stemmed from major problems, in particular during the preparation for GNOME 2.0. The main goal of this release was to clean up internal interfaces, which was important for future development but changed little from the perspective of an end-user. During the 2.0 cycle several delays occurred, and, after more than a year of work, developers had the feeling that they had to add more value to make the release attractive to users:

There was this thought that because people had to wait so long they should really get a lot when they get to see something, which means they get to wait even more. (Murray Cumming, GNOME)

Another problem was that the release process was very disorganized and lacked planning. The release was always close, but for one reason or another, it never came. This led to a lot of frustration and stress for developers working on GNOME:

I think that freezes were sudden, and, like in Debian, we were promised a freeze and then it wouldn't happen for six months. This means six months of working incredibly hard for a deadline which is constantly moving away from you. (Murray Cumming, GNOME)

This unplanned nature of the release process had an impact not only on volunteer developers working on GNOME but also on vendors that wanted to improve GNOME and ship it as part of their systems. For them, the lack of a roadmap meant that they could not plan appropriately:

When you made your changes on the development branch you wouldn't know when you would be able to use those changes. But if you're making changes on the stable branch, you're changing very old code. (Havoc Pennington, Red Hat)

As a result, many vendors backported changes from the development version to the last stable version or implemented features based on that version. This practice led to a lot of fragmentation. Instead of one development tree to which everyone contributed, many vendors had their own version and little work happened on the official development tree.

When GNOME 2.0 was finally released in the middle of 2002, the project agreed to fundamentally change their release strategy. It was proposed to make releases according to a six monthly schedule. Because of the frustration developers experienced during the 2.0 release cycle, they were willing to try something new. The structure of GNOME at that time also meant that only a small number of around twenty core developers had to agree to the new release process.

The project designed a schedule, implemented policies to ensure that the development tree would remain reasonably stable and seven months later delivered their first time based release. GNOME 2.4 followed seven months later

and another seven months later the project released version 2.6 — this version got slightly delayed because of an intrusion on the project's servers. Starting with version 2.8, when they attained their goal of a six month release for the first time, the GNOME project has delivered every release after six months (see table 5.3). Twice a year, in March and September, users can expect a new version of GNOME. This accomplishment, made by one of the largest FOSS projects, is the reason why many projects view GNOME as the reference model regarding release management.

In the years that GNOME has made incremental improvements to their system available to end-users twice a year, some critics wondered whether the release cycle limits innovation and ambition. They call for radical changes that would lead to GNOME 3.0 and doubt that the current release model would allow such a major release. The argument is that the time based release strategy of GNOME, according to which releases are made every six months, prohibits radical changes which would take longer than one release interval to be implemented. The counter argument is that nobody is sure what exactly GNOME 3.0 should be and many features that have originally been proposed for a 3.0 release have actually been implemented in the meantime using the current development process and release strategy. Havoc Pennington, a core GNOME developer and visionary, thinks that something radically different would probably not be called GNOME 3.0:

There are plenty of people who just want what GNOME is now. I think if you made something radical like this the normal GNOME would just continue indefinitely. (Havoc Pennington, GNOME)

In fact, there are already a number of new projects that are built on GNOME technologies but which have structurally changed what the components of the interface are. Such projects include the interface called Sugar that was implemented for the One Laptop per Child (OLPC) project as well as the interface of the Nokia 770, known as Maemo. It is therefore expected that GNOME will continue to deliver incremental improvements twice a year.

Past Problems

- The main goal of version 2.0 was to change internal interfaces but after more than a year of work the project felt they had to deliver more user-

visible changes, therefore leading to delays.

- Developers were disappointed with delays and that their work was not available to users.
- It was not clear what was going on. Developers constantly had to ask about the release status.
- Freezes were announced and people worked towards them but then they were delayed. Freezes also often came unexpectedly.
- Vendors had deadlines but the GNOME schedule was unpredictable. Vendors did not know whether they should aim for the next version or focus on the previous version and backport fixes.
- Vendors had to backport many changes to a version that the GNOME project considered as old.

Solutions

- The project introduced a rigorous schedule promising a release every six months.
- There was a core team so few people had to agree to the introduction of a schedule.
- GNOME introduced policies to keep the development tree fairly stable.
- The project introduced the idea of reverting: if a feature was not ready on a certain cut-off date, it would be taken out again.
- The project gained credibility because releases were actually performed on time.
- The schedule allowed vendors better planning, and hence vendors could implement their features on the main development tree.

Outstanding Problems

- The six month schedule has been successful in the delivery of incremental updates. There are some concerns whether this release cycle makes the

project less innovative and ambitious regarding major changes that would lead to GNOME 3.0.

5.4. Linux

The Linux kernel is one of the largest and prominent FOSS projects. The project has seen major changes to its development and release strategy in the last few years, in particular since the first stable release of the 2.6 series in December 2003. This series was opened almost three years after the 2.4 series in January 2001 (see table 5.4). Linus Torvalds, the creator and maintainer of Linux, felt that this period was too long. He stated that the “problem with major development trees like 2.4.x vs 2.5.x was that the release cycles were too long, and that people hated the back- and forward-porting”.² Developers and vendors had to backport features from the main development tree to older releases and features developed for older, stable releases from vendors had to be forward-ported so they could be integrated into the official development tree.

Version	Release Date	Months
1.0	1994-03-14	
1.2	1995-03-07	12
2.0	1996-06-09	15
2.2	1999-01-25	31
2.4	2001-01-04	23
2.6	2003-12-17	35

Table 5.4.: Stable releases of Linux.

In the past, the Linux kernel was known for its clear development cycle, commonly known as the “Linux versioning scheme”: a release would receive a version number in the form of 2.x.y where x indicated whether the release was part of a stable or development series (Moon and Sproull 2000). Even numbers (2.4, 2.6) would denote a stable release while odd numbers (2.3, 2.5) indicated a development release. When a new stable series was released (e.g. 2.4), all development would focus on getting it even more stable. In that period, there was no development tree in order to encourage developers to focus on the stable

²<http://kerneltrap.org/node/4793>

release. After the new stable series had further stabilized for a few months, a new development tree (e.g. 2.5) would be opened and most attention of the developers would be drawn to it. Development on this tree would eventually cumulate into a new stable series (e.g. 2.6).

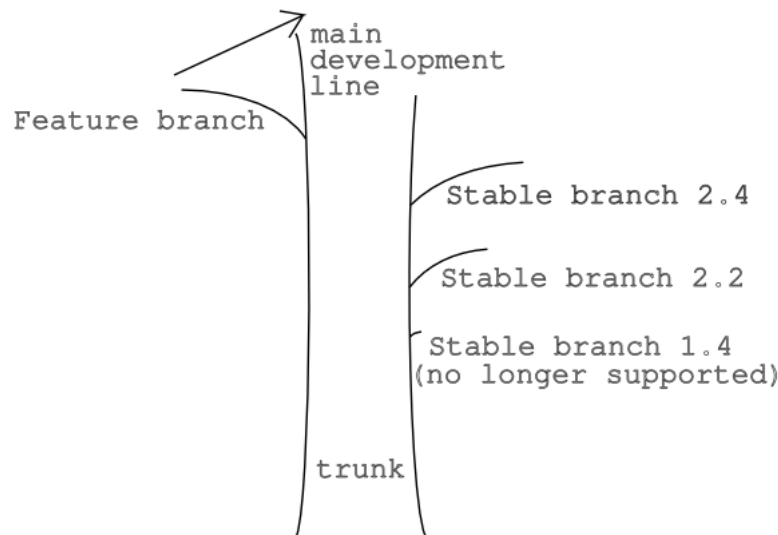


Figure 5.2.: Software development is often compared to a tree: the main development line takes place on the trunk of the tree whereas smaller development lines are seen as branches. They branch off the main development tree and are used to maintain old releases or to develop experimental features outside the main development tree with the hope that they can be integrated with (or ‘merged into’) the trunk again when the feature is complete.

With the release of the 2.6 series many developers wondered how long it would take for 2.7 to be opened. However, due to Linus’ and other developers’ frustration of the long major development trees, it was decided that 2.7 would not be opened at all, at least not in the foreseeable future, and that major development could still be done on the 2.6 series. A staging tree, the so called *-mm* tree which is maintained by Andrew Morton, various development trees of subsystem maintainers and Linus’ public version control tree would act as development platform, and new releases in the 2.6 series would be made from there (see figure 5.3 for the work flow in Linux kernel development). Compared to traditional stable releases, the 2.6 series sees much more development: not only are major features added, but interfaces are also changed and features removed — something quite unusual in a ‘stable’ series.

This new development model has faced much controversy. While some people, in particular developers of the kernel, claim that the new model is working

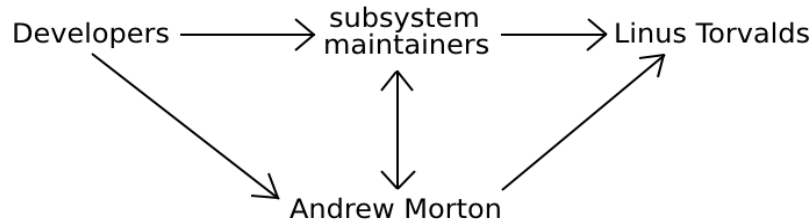


Figure 5.3.: Work flow in Linux kernel development: developers send contributions to subsystem maintainers or to Andrew Morton, who collects features and fixes in his *-mm* development tree. All contributions that are accepted get eventually sent to Linus Torvalds who maintains the official development line.

very well, some end-users are worried about the number of significant changes. At this point, it is important to note that the Linux kernel has always taken a special place among software applications. The kernel is the ‘heart’ of the operating system and is therefore not directly aimed at end-users. As such, non-technical end-users are not supposed to use Linux from kernel.org, the distribution site for official releases. Rather, they should use vendor kernels which come with their Linux system and which had thorough testing and have been polished for end-users. As the quote from Linus above indicates, a major problem in the past has been the ‘back- and forward-porting’ between releases: vendors would backport new features or bug fixes from a new development or stable kernel release from kernel.org to their vendor release, and they had to forward port their own changes to newer kernel.org releases:

Vendors had to ship a version of the kernel with hundreds or even thousands of backported fixes just so they could support contemporary hardware or fix bugs. (Andrew Morton, Linux Foundation)

The new 2.6 series, in which major development is allowed, makes the work for vendors easier since their changes can constantly be integrated into official kernel.org releases, thereby decreasing future maintenance efforts. Evidence for this can be found by investigating different kernel versions in Fedora Core (FC), a major Linux distribution. Table 5.5 shows how many modifications Fedora made to the official releases from kernel.org. The table lists how many lines of code were changed (that is, removed and added) by Fedora and how many files were changed in total. This table provides evidence that the 2.4 kernel series required substantially more changes by vendors than the current

2.6 series.

Fedora Version	Kernel Version	Lines Added	Lines Removed	Total Files
FC 1	2.4.22	418669	37870	2064
FC 2	2.6.5	27911	9651	631
FC 3	2.6.9	39323	4407	697
FC 4	2.6.11	124970	6677	846
FC 5	2.6.15	129310	1222	996

Table 5.5.: Changes made to kernels by Fedora: total number of lines that are added and removed, and the number of all modified files.

In an evaluation of the new development model, it is important to consider the ecology of the Linux kernel. The changes to the 2.6 cycle were not primarily made to ensure that end-users would get up-to-date releases, such as updates for new hardware: even in the days of 2.4 support for new hardware was backported from the development tree to 2.4 and certain other features added. The new development process has to be seen mainly as a means for coping with the increased pace and size of the Linux kernel. As more developers are getting involved and more development is being done, the old development tree was too slow. While changes could be integrated, it would later be impossible to test new releases because of the interaction between all the changes that had been made. With the 2.6 release, which currently aims to make a new release roughly every two or three months, the interaction of new changes can be tested constantly.

In recent months, the development and release process has increasingly been formalized. For example, with the release of 2.6.13 in August 2005, Linus Torvalds announced that he intends to merge new features only in a two-week window after a release. All the rest of the time between releases will be spent on fixing bugs. This gives developers a clear deadline, and contributes to a predictable development cycle. As such, one of the main problems of unpredictability, pointed out by developer Ted Ts'o in 2002, has been reduced. As initially mentioned in section 3.2.7 on page 53, Ts'o described the problem of the “thundering herd of patches”, the phenomenon when everyone tries to get their changes in when a freeze is announced out of the blue.

Developers working on the Linux kernel consider the new model a success,

but it is not without problems. There are a number of users who would like to use a release from kernel.org rather than from a vendor but who do not want to deal with a constantly changing kernel. For this reason, Adrian Bunk has created a long-term stable kernel based on 2.6.16. Another problem is that because of the increased pace of development more regressions are introduced. Adrian Bunk is trying to keep track of such regressions and reminds developers of outstanding problems. Another problem is that while newer hardware is tested fairly well by vendors, testing coverage is not complete:

I think what we're doing is to break old machines that not many people have and which are of little financial interest to anybody.
(Andrew Morton, Linux Foundation)

Morton believes that this is not a new problem, though, as there has always been more emphasis on new hardware and features. He stated in an interview in December 2006 that he would like to see a dedicated QA person who would track outstanding issues. At the end of January 2007, Google, who make significant use of the Linux kernel and employ Andrew Morton, posted a job opening for an engineer who would track and manage defects in the Linux kernel. This role includes working with the Bugzilla bug tracker, which is currently not well integrated into the development process.

Past Problems

- Because of the long release cycle, many changes accumulated. It was hard to get the development stable and there were few testers.
- Features got out very slowly because of the long release cycle.
- Hardware support and crucial features had to be backported to the latest stable kernel.
- Vendors backported many features to their own releases. The code base from different vendors diverged a lot from each other and from the official development version.

Solutions

- New versions are now released every two or three months.

- There is now a steady flow of code into production and many people get to test the new code.
- Features get out more quickly.
- Vendors can directly work with current releases and get their changes into official versions easily.

Outstanding Problems

- There is no long-term stable version based on the 2.6 kernel. This is being addressed with Adrian Bunk's 2.6.16 long-term maintenance tree but whether it will have a big impact remains to be seen.
- Regressions between versions are introduced more frequently. Better control and tracking of regressions are needed.
- The Bugzilla bug tracker needs to be integrated better into the development process and it would be helpful to have a QA person.

5.5. OpenOffice.org

OpenOffice.org is an office suite offering various integrated applications, such as a word processor and a spreadsheet. OpenOffice.org is based on StarOffice, an office suite originally developed by StarDivision and later acquired by Sun Microsystems. The source code of the suite was released in July 2000 and OpenOffice.org entered the FOSS world as the biggest freely available office suite. While Sun still maintains fairly tight control over the development of OpenOffice.org, many other vendors, in particular Novell, are important contributors to the project.

The initial release cycle of 18 months was chosen to accommodate StarOffice, the commercial product from Sun. Contributions are made to OpenOffice.org which forms the core of StarOffice and therefore having similar release cycles for both products was advantageous. Version 1.1 was released roughly 16 months after 1.0 and the 2.0 release was planned another 18 months later. However, there were many delays during the preparations for the 2.0 release. These delays led to several problems. In particular, vendors that wanted to

Version	Release Date	Months
1.0	2002-05-01	
1.1	2003-09-02	16
2.0	2005-10-20	26
2.0.1	2005-12-21	2
2.0.2	2006-03-08	3
2.0.3	2006-06-29	4
2.0.4	2006-10-13	3
2.1.0	2006-12-12	2
2.2.0	2007-03-29	4

Table 5.6.: Stable releases of OpenOffice.org.

include OpenOffice.org in their systems found it difficult to plan as nobody knew when 2.0 would actually be released. People were too optimistic, and as a result, some vendors shipped a pre-release of version 2.0:

Everyone wanted to ship 2.0 in their distributions but there were many false promises of the cycle being ended, and then it sort of slipped. It was very difficult to predict when it would be ready, and as a consequence, we shipped a product based on release snapshots made three months before 2.0 and trying to bug fix those in parallel. (Michael Meeks, Novell)

As a consequence, many vendors spent more time stabilizing their own release based on release snapshots rather than focusing on the official release.

Some developers believe that the delays were partly caused by the long release cycle:

You had a feature freeze nominally in the middle of your 18 month schedule. So 9 months after the previous release you freeze and then another 9 months it is bug fixed and it gets released. Unfortunately, if you do feature development for 9 months, and then come back 9 months later to bug fix it you have some problems. You've forgotten the code and what it was supposed to do. It's horrible. (Michael Meeks, Novell)

In other words, huge 'big bang' releases are hard to manage because there have been so many changes that it is hard to get the code base stable. A common problem of this development style is that widespread user-testing is often left to the end, which is much too late to have a real impact. Another problem was that several new features were added very late during the 2.0 phase when the project had already entered the beta phase.

As a way to tackle these problems and allow more rapid feedback, the project decided to move to a model according to which a new release would be published every three months. This change had several positive effects: different companies increasingly worked on the same code base, leading to more collaboration and peer review, and the new mechanism created a tight feedback loop with users. The OpenOffice.org release team has also taken several steps to make the release process more transparent. Minutes of release meetings are now published, blockers are identified more quickly in the project's issue tracker, and there is growing documentation about the release process. These changes have allowed voluntary contributors who are not working for Sun to participate more actively in the release process.

Several releases have been made according to the new model, with the interval of the actual releases ranging from two to four months (see table 5.6). The new model is widely accepted as a great improvement over the previous, long cycle. Nevertheless, changes to the release model were proposed by the OpenOffice.org release committee in November 2006. They argued that a six month release cycle would suit their users better while still retaining the benefits from a regular release cycle. According to the proposal, some users did not want new features every three months and the aggressive release schedule put a lot of pressure on the QA team. They suggested to make new major releases every six months, followed by minor updates with bug fixes three months after each major release.

Past Problems

- The long release cycle of 18 months, bound to the commercial StarOffice product, meant that little testing occurred for a long time because developers believed the release was far away.
- Many changes accumulated during the long development phase, making testing towards the end very difficult, and leading to a 'big bang' release.
- Features were put in very late, even during the beta cycle, because of the perceived 18 month delay to the next release.
- There was very little code review. The QA team only tested the program.

- Vendors shipped unreleased versions because the significant delays during the 2.0 cycle made planning impossible.

Solutions

- After the 2.0 release, the project moved to a three month release interval. This model promises a tight feedback loop with users.
- Because planning is now possible, collaboration between vendors on the same code base is much easier.
- The faster release cycle and more collaboration among vendors has promoted code review.
- Motivation in the project has increased because people see their contributions getting out to users within a reasonable time.
- The release process has become more transparent, allowing voluntary contributors to take a more active part in release preparations.

Outstanding Problems

- There are discussions about changing the release interval to six months. There is some evidence that some users do not want new features every three months and that the aggressive release cycle of three months puts a lot of pressure on the QA team.

5.6. Plone

Plone is a content management system that is built on the powerful Zope application server. It provides a system for managing web content that is ideal for a wide range of users. Archetypes, which has also been considered in this case study of Plone, is a framework designed to ease the building of applications for Plone.

In 2005, the project decided to move to a six month time based release. The reasons for this decision are two-fold. First, version 2.1 experienced many delays and took almost one and a half years to be released (see table 5.7). Second, Zope, on which Plone is built and with which it has very strong links,

decided to move to such a release schedule. The Plone project believes that a synchronization with Zope's release schedule will benefit the project since it can more easily make use of new Zope features in their software.

Version	Release Date	Months
1.0	2003-02-06	
2.0	2004-03-23	13
2.1	2005-09-06	17
2.5	2006-06-16	9

Table 5.7.: Stable releases of Plone.

The delays experienced during the 2.1 release cycle led to several problems which Plone aims to target with its new release cycle based on six monthly releases. One problem was that because of the long development period of 2.1 there was a large number of changes, some of which caused migration problems for users. Another problem was that many Plone developers work as consultants building web sites. For them, the unpredictable schedule of Plone meant that it was difficult for them to decide whether to use the current version of Plone in a commercial project or wait for the next release.

In June 2006, the project released its first time based release, nine months after version 2.1. This release was more structured and allowed the project to perform better planning and more control. For example, Plone required developers to propose new features through Plone Improvement Proposals (PLIP) which were evaluated by a framework team. If they were accepted by the framework team, the release manager would judge whether their implementation was ready for inclusion for the release.

According to a six month release cycle, the next release after 2.5 in June 2006 would have been during the Christmas period. Since many volunteer contributors might be unavailable during that time, Plone decided to move the release target to March 2007, followed by a release in October. The Plone project has taken their first step towards a regular time based release but it remains to be seen whether the project can meet deadlines and release future releases according to their roadmap.

Past Problems

- Releases, in particular version 2.1, took a long time to get out.

- Releases had many changes and caused some migration problems.
- Many Plone developers work as consultants building web sites. Because of the unpredictability of Plone, it was difficult for them to decide which version to use for future projects.

Solutions

- Plone moved to a time based release, partly because the Zope framework on which it builds has done so.
- Time based releases allowed the project to implement more structure. For example, new features have to be proposed as a PLIP which the framework team reviews.
- Deadlines have motivated developers to get their features done within a certain time frame.
- Plone consultants can decide in advance which version of Plone to use for future commercial projects.

Outstanding Problems

- As the project has moved to time based releases only recently, they still need to show whether they can consistently release on time.

5.7. X.org

X.org is an implementation of the X Window System which provides an interface between display hardware and a desktop environment, such as GNOME. Even though X.org has a long history, the project is fairly new in terms of a FOSS community. In previous times, most Linux distributions and other FOSS projects relied on the XFree86 system. Over the years, the structures within the XFree86 project became rigid and the project failed to innovate and keep up with the pace of the wider FOSS community. In February 2004, with version 4.4.0, the XFree86 project changed its license in a way which many people considered very controversial. As a direct consequence of this license change and because of the long-lasting structural problems within the project,

the majority of developers moved to X.org and established a productive work environment. Most Linux distributions have swiftly moved from XFree86 to X.org and nowadays the X.org project is where the majority of development is taking place.

Some problems that XFree86 and X.org had were related to the huge code base on which both projects build. The source code is one monolithic code base which uses an archaic build system many developers are not comfortable with. Because of this monolithic code base, it was hard to attract new contributors to the project, only few performed testing and it was difficult to prepare such a huge system for release. While X.org used this monolithic system initially, they soon made plans to move towards a more modular system. Using this opportunity of change, they also adopted a more modern build system. As of X.org 7.0, the project moved to the modular system in which components are developed and released separately. Effectively, the project moved to a development mechanism which features two release mechanisms: individual components can be released as needed and there is an overall release of X.org in which all stable components are put together. These roll-up releases take place every six months.

Version	Release Date	Months
7.0	2005-12-21	
7.1	2006-05-22	5
7.2	2007-02-15	9

Table 5.8.: Stable releases of X.org.

The first modular release of X.org took place in December 2005 and was followed by another roll-up release five months later (see table 5.8). The following release was planned for November 2006, six months after the release of 7.1, but the project failed to meet this target and released version 7.2 in the middle of February 2007. Nevertheless, the steps X.org has taken towards regular releases look promising.

Since the move to the modular system with time based releases, the project found that it is easier to find testers for individual components and to give contributors write access to specific parts of the system. The roll-up releases have taken a lot of pressure from the release manager because there is a fall

back: if an individual component is not ready at the time of the roll-up release, the last stable version of this component can usually be incorporated. This is possible because the interfaces between components are fairly stable.

The X.org community is very active and has implemented a number of changes which would have been difficult in the XFree86 project with its rigid structures, such as the move to the modular system and the adoption of a more modern build system. The modularization effort has been received very positively, both by users and in particular by vendors because it makes it easier for them to integrate new components in their system. The project needs to work out interfaces between the server and drivers in more detail so updates to hardware drivers can be made more frequently.

Past Problems

- XFree86 made only infrequent releases every few years, had no plan, and the project's structures were very rigid.
- The code base was huge and monolithic. It had an archaic build system that few new developers were comfortable with. This made it hard for new contributors to get involved and was bad for testing.

Solutions

- X.org moved from a monolithic to a modular system. This made it easier to perform testing and it made it possible to give contributors write access to specific components.
- The move to the modular system allowed them the introduction of two release mechanisms: individual components can make releases on an on-going basis and roll-up releases take place every six months.
- The roll-up releases have a fall back mechanism in case a specific component is not ready for release: the former version of this component can be incorporated.

Outstanding Problems

- The project needs to work the interface between the server and drivers, so updates to hardware drivers can be released more often than other components.

5.8. Chapter Summary

This chapter presented seven case studies investigating projects that have implemented a time based release strategy. Even though there are still outstanding problems and some case projects have not yet implemented time based releases with full success, the introduction of this release strategy has overall led to improvements in the release process. The next three chapters will present findings from a cross-case analysis in order to discuss why this release strategy may be associated with certain benefits and to investigate factors and practices that related to successful implementation of time based releases.

6. Release Strategy

This chapter performs a cross-case analysis of the projects presented in the previous chapter in order to locate common problems with release management and to discuss why the move to a time based release strategy has led to improvements in the projects under investigation. This chapter consists of the following two main sections:

1. Problems prompting a change: this section locates common problems related to release management found in the case studies and discusses the underlying factors leading to these problems.
2. Time based strategy as an alternative: this section investigates the time based release strategy in detail. Evidence from the case studies are used to argue why time based releases are an attractive alternative to other release strategies. This section concludes with open questions regarding the time based release strategy that require further investigation.

6.1. Problems Prompting a Change

The common problem that all projects in the case studies exhibited was that releases were delayed and that these delays were associated with a number of other problems, such as badly tested software, frustration and further delays. The delays in the delivery of new stable releases for end-users, however, are only the symptoms of a more fundamental problem these projects had. This problem, which led to delays and other issues, was lack of planning. In one way or another, every case study showed problems with, or even the complete absence of, planning:

- Some projects followed the “release when it’s ready” motto in which release decisions were made in an ad-hoc style. Debian is a project which

used to adhere to this motto and defined few goals for their releases in the past.

- Some projects defined goals loosely but had no clear plan how to attain them. For example, a number of projects aimed to include specific new functionality and features in their next release. However, their reliance on volunteers to implement these features meant that planning was virtually impossible.

Due to the lack of an established plan that the project could follow, the decision to start the preparation for a new release was typically based on the personal judgement of the release manager rather than on formalized criteria known to each member of the project.

In the following, implications of lack of planning will be studied, specific problems resulting from a lack of planning will be discussed and long-term issues will be investigated.

6.1.1. Lack of Planning

The lack of adequate planning and the ad-hoc development style observed in the case studies led to two conditions which had negative implications in terms of release management:

- First, the projects required high levels of active coordination, which is costly in distributed volunteer projects (Rasters 2004; MacCormack, Verganti, and Iansiti 2001). As described in section 3.3.2, FOSS contributions often happen through self-assignment of tasks with little coordination. Volunteers choose an area of interest and then independently work on it, effectively self-policing their work. However, this style of development is only possible when individuals have sufficient information to perform their tasks. Since there are dependencies between different work items, individuals need to know how their work fits into the global picture of the project.

For example, a documentation translator can only start their work when a substantial amount of documentation has been written and ideally the whole document has been finished. Due to the lack of an overall plan,

which outlines goals, dependencies and other information, developers had to explicitly coordinate with other contributors or the release manager:

My impression is that for almost two years people kept asking ‘is the API stable yet?’, or ‘have you finished with this user interface, can we start translating it now?’. (Murray Cumming, GNOME)

The lack of a release plan and schedule also put the main burden of coordination work on the release manager:

Getting all the parts in a good state at the same time requires a lot of management effort. (Stuart Anderson, X.org)

- Second, freezes were announced out of the blue. Due to the lack of a plan and well established release criteria, nobody could predict when preparations for the following release would actually start. The time to start the freeze (release preparations) relied on the personal judgement of the release manager and developers were often caught by surprise.

Essentially, the lack of a plan and a clear deadline made the development process chaotic:

There was no process. People just hacked on something, code was added and broken again, there was no agreement [about] when they’d stop development. People always wanted to put in more stuff. (Stefan H. Holek, Plone).

As a consequence of this chaotic and ad-hoc development, many unfinished features were put into the development version with the assumption there would be enough time to stabilize the code. Over time, many features accumulated but testing was often left until the end:

Another feature of doing a huge amount of feature development and then trying to do bug fixes means inevitably that the schedule slips. (Michael Meeks, Novell)

When a freeze was finally announced, there would often be features that were not ready and needed more time to stabilize. Many contributors were also surprised by the sudden announcement to slow down development in preparation for a release and tried to push their work in quickly. The disorganized development style followed previously made it difficult

for the release manager to suddenly stop developers from making more changes and to enforce deadlines.

In summary, the lack of planning leads to two unhealthy conditions: projects will require more coordination, which is costly in geographically distributed projects, and release preparations often come as a surprise to developers.

6.1.2. Problems Caused by Lack of Planning

The unplanned nature of development was often associated with delays and led to a number of problems related to the release:

- Many changes to test: the lack of an overall plan led developers to add features as they saw fit. Some projects, such as Debian and the Linux kernel, had long development phases, and many projects experienced additional delays, which meant that a huge number of changes had to be tested at the end of the development phase:

A major problem with the stable release model was that too many changes accumulated [over years of development]. It was hard to get the development stable while people were waiting for new features they didn't get because the series didn't stabilize. (Christoph Hellwig, Linux)

Another problem is that some projects added features which were not ready or well tested. This behaviour is promoted by long development cycles where the end is not in sight:

I think it's a problem if you start packing too many features into a release. The features are there but they are simply not stable. (André Schnabel, OpenOffice.org)

- Little testing of development releases: as the development version moves increasingly further away from the last stable release, fewer users are willing to test these releases. Most users do not start testing until a stable release is near, which requires a visible plan. The long interval between stable releases meant that little feedback was provided by the user community. Testing was therefore often left to the end:

Few people tested it and we didn't manage to squeeze out all the bugs. In a sudden time this whole, enormous amount of changes tried to move out to a stable tree that hadn't gotten enough attention in a long time. (Christoph Hellwig, Linux)

The long time spent in the development phase also prompted some projects to rush their releases out of the door without adequate testing.

Due to the chaotic style of some projects, it is also possible that they failed to make test releases for their development versions available:

Every time you [fail to publish a test] release, people have less faith that the next release will happen, and therefore won't be prepared to test it. (Murray Cumming, GNOME)

- Fragmentation of development: vendors which want to ship a piece of FOSS as part of their Linux distribution or other product need to know well in advance when a release will be stable. Due to the absence of release plans, many vendors avoided the official development version and worked on their own:

There was a GNOME stable release and everyone had huge sets of patches to the stable release. [Vendors] would ship the stable release plus their huge patches. All the developers were working on their different versions of the stable release. Then there was the development branch but nobody was really working on it. (Havoc Pennington, GNOME)

This behaviour leads to fragmentation between different vendor releases and important resources are taken away from the official development releases. This can have a substantial impact on projects in which companies make big contributions. It also means that much work is duplicated:

The lack of coordination between distributors means that [stabilization and release work] is repeated three, four, five, n times, and every time probably done badly. (Christoph Hellwig, Linux)

- Out of date software: because of the long interval between new stable releases for end-users, the last stable release was often considerably out of date. Some projects, like the Linux kernel, spent substantial effort to make updates to their stable releases:

Urgent features that users can't live without, especially support for recent hardware, had to be backported from the development series to the stable series. (Adrian Bunk, Linux)

While such efforts benefit users, they also take away resources which could be used to stabilize the development version and prepare a new stable release.

- Bug reports of little value: because the releases end-users were running were often quite old and therefore different to the latest development version, many bug reports were of little value:

All users were using a version that was a year old or so and they were reporting bugs against that version. They often heard that it had been fixed in the development version [but] nobody knew when the new version would come out. (Havoc Pennington, GNOME)

Even if a bug was important enough to warrant an update to the stable release it was often hard for developers to find the incentive to invest time in it:

It's just a really, really old code base and it's hard to make programmers care so much about it. (Michael Meeks, Novell)

- Users cannot plan: an unpredictable release schedule makes it impossible for users to plan. This can have a major impact on corporations with large FOSS deployment:

People had to wait a long time for the next release and didn't know what to do: whether to use the old release, to use pre-released code or whatever. (Jens Klein, Plone)

- Frustration among developers and users: delays and other problems with the release can lead to frustration, both among developers and users of the software:

I think on a personal level people were disappointed that they worked so hard for so long, and they had the feeling that they had to work hard because it was so close to a release, and that it was constantly being delayed, that their work was not available to the public, not in a usable, stable form. (Murray Cumming, GNOME)

- Further delays: a major problem of delays is that they may lead to further delays. This vicious circle is easy to enter when it becomes clear for developers that a target cannot be met. If this is the case, they might try to push in more changes, thereby destabilizing the code base and causing further delays. Every delay is perceived as an incentive to do more development. As the time between releases increases, developers may think that missing this release will cause their feature not to be available for years and they push harder to get it in.

6.1.3. Long-term Issues

If such delays and other problems occur during several release cycles, a project will face long-term issues, such as:

- Loss of credibility: when a project fails to meet release targets several times and their releases are delayed, the project as a whole and the release manager in particular lose their credibility. This makes release management work even more difficult because developers do not believe that targets and deadlines can be met. It is also a bad sign for users who start to realize that they cannot rely on this project and may evaluate software that is delivered on a more predictable schedule instead.
- Fewer contributors: as mentioned above, delays can be very frustrating for developers because their contributions do not get delivered to actual users within a reasonable time:

It's very frustrating for developers to write a patch and then have to wait a year and a half before that feature will get into a release. (Louis Suarez-Potts, OpenOffice.org)

As a consequence of repeated delays, volunteers may choose to spend their time on a project where they have a bigger impact. Hence, the project loses important resources which may be crucial, in particular during the preparation for a release.

6.1.4. Summary

The lack of adequate planning in large FOSS projects leads to two unhealthy conditions: higher levels of active coordination are required in a project and releases are often announced without giving enough advance warning. The effects can be disastrous on a project and the previous sections have given examples of actual problems that typically occur in projects without sufficient release planning.

6.2. Time Based Strategy as an Alternative

The exploratory study presented in section 3.2 has shown significant interest in the time based release strategy within the FOSS community. Chapter 5 presented an overview of seven FOSS projects that have moved, or are currently moving, to a time based strategy, and even though the projects still face certain problems, the move to a time based strategy can be considered an improvement over their previous release mechanism. In the following, conditions will be discussed that have to be met so that a project can consider implementing a time based release strategy. This is followed by the presentation of an argument based on theoretical propositions as to why the move to this release strategy has led to improvements to the projects under investigation and what the advantages of this release strategy are in general. This section will end with a discussion of open issues related to this relatively new release strategy that need further exploration.

6.2.1. Conditions That Have to be Met

Even though time based release management is very promising, this release strategy is not suitable for every project. Based on an analysis of the case studies that were performed as part of this research, the following four conditions have been identified that appear to be necessary so that a project can implement a time based release strategy:

1. Enough work gets done: with time based release management, new releases are made according to a certain interval. This strategy cannot be

used in projects where there is no guarantee that enough work is done during an interval to warrant a new release.¹

2. Distribution is cheap: since time based release management relies on the publication of regular releases, the distribution of new releases must be relatively inexpensive and easy.
3. The next release does not require specific new functionality: the main focus of time based release management is time rather than specific functionality. It is therefore not suitable for projects in which specific functionality needs to be delivered with the next release, which tends to be the case in traditional software development.
4. The code base and project is modular: it is important for a project to be modular because this allows individual components to be developed, fixed, released, and also taken out again independently.

In the following, the reasons for these conditions will be discussed in more detail along with an investigation into what kind of FOSS projects meet these four conditions.

Enough Works Gets Done

It is important to emphasize the first condition because it rules out time based releases as a good strategy for the majority of FOSS projects. It has been shown in an empirical analysis of 100 mature projects found on the popular FOSS hosting site SourceForge that the majority of projects are very small (Krishnamurthy 2002). The study showed that only 29% of projects had more than five developers and that, on the other hand, 22% of projects had only one single developer. With only one or a few volunteer developers working on a project, there is no guarantee that enough changes will accumulate during a release interval to warrant the publication of a new release. If no changes are made in several months, as is the case in many projects (Howison and Crowston 2004), there is no reason to make a new release. On the other hand, in large projects with hundreds or thousands of developers — which are the

¹Factors which influence what exactly warrants a new release will be discussed in more detail later in this section.

focus of this dissertation — there is always a steady flow of changes (Crowston, Howison, and Annabi 2006). This can be seen in the examples from Debian in figures 6.1 and 6.2. Time based releases are therefore particularly appropriate for large projects.

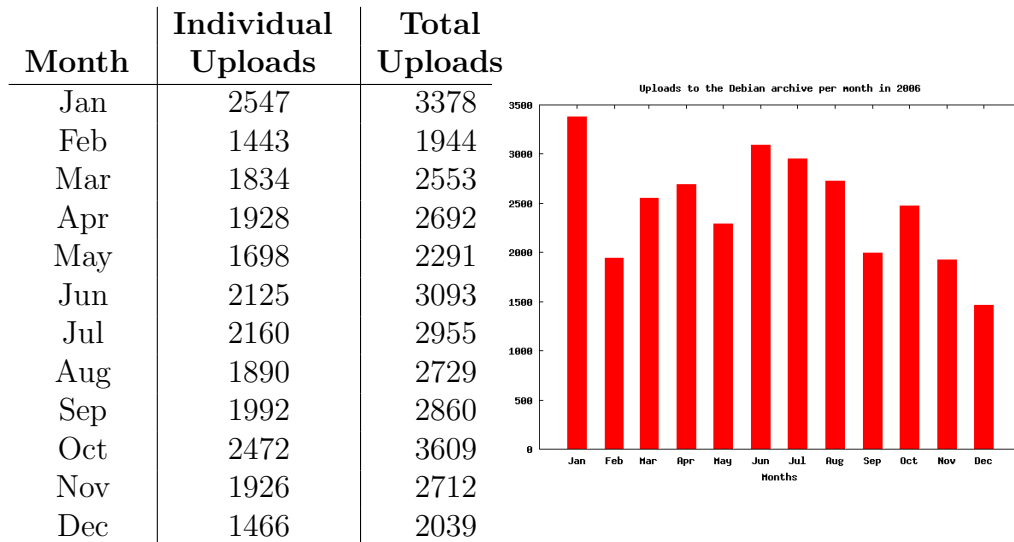


Figure 6.1.: Uploads to the Debian archive per month in 2006: the table on the left shows how many individual software packages were uploaded each month and how many uploads were performed in total. The figure on the right illustrates the total number of uploads.

Distribution is Cheap

Since time based release management relies on the publication and delivery of regular releases, distribution of the released software must be inexpensive and easy. Since FOSS projects operate over the Internet, distribution of such software meets these criteria. A project can simply put new releases on its web site and vendors will incorporate the new releases into their products and deliver them to customers. While FOSS vendors sell CDs, the majority of software updates are nowadays delivered over the Internet too. For example, the dominant Linux vendor, Red Hat, has established a platform known as the Red Hat Network.² This web site acts as their ‘enterprise systems management’ platform through which software updates can easily be installed on thousands of machines in an organization.

²<http://www.redhat.com/rhn/>

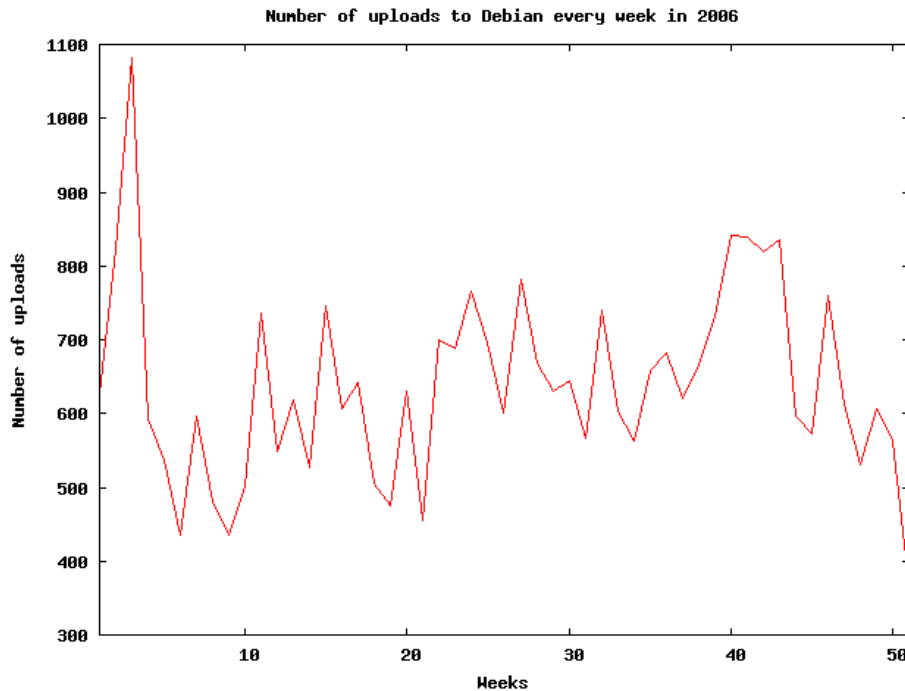


Figure 6.2.: Uploads to the Debian archive per week for the year 2006.

Specific Functionality is Not Necessary

In contrast to delivery of software over the Internet, shrink-wrapped products sold in shops are associated with higher distribution costs. There are two competing approaches to software production today: some view software as a product industry and sell shrink-wrapped boxes with their software in shops. On the other hand, there is a trend to viewing software production as a service industry. According to this view, revenue is not generated by selling shrink-wrapped boxes but by providing services, such as updates to software:

There's a move towards services and for that a shorter cycle makes more sense. As long as you have a shrink-wrapped product, having a longer release cycle makes sense because you get bigger PR and marketing and people purchase or obtain it. If you have services, you want people to subscribe and then users will get the latest updates and fixes. (Louis Suarez-Potts, OpenOffice.org)

These economic aspects of the software industry are related to the first and third conditions. If a company wants to sell shrink-wrapped boxes, there must be a large incentive for users to buy the new version. Hence, shrink-wrapped software is usually associated with long development cycles and 'big bang' releases which deliver major new functionality. In order to sell shrink-wrapped software, companies need to add certain new functionality in order to make

it attractive to buy the new version. On the other hand, the service industry puts more emphasis on continuous improvements and updates, even if they do not contain major new functionality. For this reason, FOSS projects, which often do not have a commercial interest or follow a service model, need less justification in order to make a new release:

If you work behind a closed door and want to make a big splash when everything is released, it really matters that certain features are in. But for an open source project it doesn't matter if they have features A, B and C. If they have A and B after six months, why would they need to wait for C? If the previous release was fine and they kept everything working and added more fixes and features, why would they not put that out? (Havoc Pennington, GNOME)

The implication of this is that large FOSS projects meet the first three conditions for time based release management: some changes always accumulate in a release interval given that there are hundreds developers or more that and there are fewer considerations to take into account to decide what exactly warrants a new release. This is because periodical updates are viewed positively, even if they only contain small fixes and little new functionality. Finally, distribution of the software can be done over the Internet without much effort or expense.

The Code Base and Project is Modular

The fourth and last condition a project needs to fulfil in order to be able to use a time based strategy is to have a modular structure. This requirement relates both to the code as well as the organization, but they can usually be considered as one factor since there is a strong link between the structure of an organization and the software developed by it (Conway 1968). The reason why a modular structure is so important is that it allows components to be developed, tested, fixed, and released individually. This is important for projects following a time based release management strategy because this strategy requires components to be in a releasable state on a specific date. If there is very strong coupling between a large number of components, defects in one component would cause the whole software system to be in a defective state that cannot be released. On the other hand, if the software and its components

are highly modular, one can use the current version of most components and either exclude the defective components or use previous versions of them which do not exhibit these defects.

As a matter of fact, there is strong evidence that large FOSS projects can be considered as an aggregation of many smaller projects. Crowston and Howison (2005) have found that,

As projects grow, they have to become more modular, with different people responsible for different modules. In other words, a large project might be an aggregate of smaller projects

This insight is very important in terms of time based release management because it allows the implementation of two complementary release mechanisms: individual components are developed independently and can make their own releases as they wish, and the overall release in which all components are combined and tested can be performed with a time based strategy. Such strategies can be observed in a number of projects, for example in Debian and GNOME. In Debian, individual software packages are uploaded on a daily basis. New software is uploaded, bugs are fixed and other modifications are performed. At the time of a major new release of Debian, all software packages which are in a releasable state will be included in the overall release of Debian. Software packages which have major defects will either be fixed or excluded, assuming the software package is not critical or that other packages have no dependencies on it. GNOME, which consists of hundreds of libraries and applications, follows a similar model. In fact, the insight that the project is an aggregate of smaller projects originally led them to the implementation of a time based release strategy after their 2.0 release in 2002:

As GNOME got increasingly bigger it was increasingly like a Linux distribution, meaning it was like a big collection of different software that had lots of different maintainers, different technology and maturity. There was always some stuff that was really stable and some that was unstable. You just had to declare a certain day to release and go with it. (Havoc Pennington, GNOME)

The X.org project presented in section 5.7 is another good example of a project exhibiting these characteristics. This project is interesting because it has just recently made the change to a time based release strategy. With their

7.0 release, the project made two fundamental changes. First, they moved from a monolithic code base to a modular one. Second, they decided to move to a time based release strategy. The second change would not have been easily possible without the former. By moving to a modular code base, they allowed individual components to be developed and released independently. Software is developed independently and “once the component is ready, the new version can be included in the next roll-up release” (Kevin E. Martin, X.org) that is done on a six month interval.

The modular nature means that the release manager has greater choice as to which components to include in the next release:

For the release manager, if you don't hear from somebody, you can always grab the latest stable version. It requires less coordination because there is now sort of a fall back. If I don't hear from that project, I will just take what I think is right and ship that even if it's the same we shipped last time. (Stuart Anderson, X.org)

Effectively, this fall back option provided by the modular nature of X.org supports the time based release strategy because it makes it easier to meet the target date.

Summary

In summary, four important conditions have been identified that appear to be necessary for a project to consider a time based release strategy. In this section, time based releases have been ruled out for projects that do not produce enough changes during a release interval to warrant a release. On the other hand, large projects, which often consist of several smaller projects, produce many changes and the inexpensive nature of software delivery over the Internet reduces the justification to ship a new release. Furthermore, frequent updates are often seen positively in FOSS projects. Finally, a modular structure is important to meet the target date since components can be developed, released and fixed individually, and there is evidence that large FOSS projects tend towards such a structure.

6.2.2. Time Based Releases as a Coordination Mechanism

In this section, it will be shown why the introduction of time based release management has led to improvements in the case studies presented in chapter 5. The explanation will draw on theories about organizational complexity, modularity and coordination presented in section 3.3 in order to conduct an analytic generalization according to Yin (1994). By relating the findings from the case studies to theoretical propositions, insights can be drawn not only about the specific cases of this research but more generally about projects which fulfil similar criteria. This includes the majority of large FOSS projects as well as other software projects which meet the four conditions presented in the previous section.

In section 3.3 it was shown that the major challenges faced in large projects today are not technological but organizational (Baetjer 1997). Large organizations rely on different human capabilities which work together to produce a product. While an organization with similar capabilities is easier to manage, production rests on the coordination of activities that are complementary (Richardson 1972). Division of labour, in which everyone brings in their own unique capabilities, leads to the availability of a set of heterogeneous capabilities. However, this mode of production is associated with external coordination costs (Garzarelli and Galoppini 2003). Modularity is seen as a way to manage organizational complexity because it reduces the number of systems that have to interact with each other. Nevertheless, even a highly modular system has dependencies which require coordination. The framework of coordination proposed by Malone and Crowston (1994) is particularly suited to study this problem as it defines coordination as the management of dependencies between activities (see section 3.3).

Coordination in FOSS projects is particularly challenging because the distributed nature of the projects under investigation is associated with high communication costs (Rasters 2004). Because of this constraint, the FOSS development method has implemented structures that reduce the amount of coordination that is needed. For example, instead of active assignment of tasks, FOSS projects generally rely on self-assignment of tasks (Mockus, Fielding, and Herbsleb 2002; Aberdour 2007). Another unique coordination mechanism

in FOSS projects has been identified by Yamauchi et al. (2000). They have shown that coordination often takes place ‘after the fact’. Instead of conducting long discussions as to how something should be done, many FOSS developers work on solutions in parallel and once solutions are available the best one is accepted and integrated (Aberdour 2007). In general, FOSS development is characterized by a highly parallel development fashion where multiple development streams are followed at the same time (Johnson 2001).

Problems and Solutions

The exploratory study in section 3.2 has shown that this mode of production breaks down during the preparation for a release. While FOSS participants can normally work highly independently with low levels of coordination, release preparation require an alignment of the work output of everyone involved in the project. In small projects, this alignment is fairly easy but in large projects with hundreds, or thousands, of contributors, such as developers, documenters, translators and testers, a huge amount of coordination is required. The problem, as has been shown before, is that release preparations are typically announced unexpectedly. Individual contributors do not have enough information about the release status to perform their work independently. This leads to a more centralized organization in which the release manager is constantly asked about the status of the release, as section 6.1.1 investigating the lack of planning has shown. The levels of coordination that are required explode because of the sudden requirement to align all work output and get it ready at the same time in preparation for the release.

This research has identified two factors associated with the introduction of a time based release strategy which both act as important coordination mechanisms:

- **Regularity:** releases are done according to a specific interval, giving the release process a regularity.
- **Schedule:** important dates are listed in the release schedule, giving developers enough information to perform their work independently.

Both of these factors are coordination mechanisms which together allow individual developers to remain self-policing even during times of release prepa-

ration. Essentially, they provide developers with all information needed so they can not only perform their work independently with low levels of coordination but also integrate their work in time for the release without the requirement of high levels of coordination from the release manager. It shifts the majority of coordination work from the release manager to individual developers and thereby maintains the decentralized structure commonly found during development. In other words, these coordination mechanisms give individual developers enough information to make it their responsibility to finish and integrate their work so that it can be included in the next release. In addition to shifting coordination work from the release manager to all members of the project, it minimizes overall coordination within the project because the information individual contributors require to be self-policing is provided in the schedule and emphasized by the regularity of the release process:

People need to know where they are. And that shared understanding helps people to adapt and work as a group. (Andrew Cowie, Operational Dynamics)

Regularity

In order to perform time based releases, a rule that says when releases are to be made needs to be established. When this established rule is followed by a project, its release process gains regularity. That is, there is a steady production of new releases according to a governing rule that developers and users can perceive and understand. In theory, it would be possible to employ a variable release interval, for example, an interval of three months, followed by one of twelve months and then one of six months. However, in practice, all of the projects under investigation in this dissertation and other popular time based projects, such as Fedora, have deployed a release strategy with a fixed release interval. The Linux kernel has a release interval of 3-4 months,³ Debian has opted for 15-18 months and all other case projects have chosen six months as their release interval.⁴ While variable and fixed intervals both lead to a sense of regularity assuming the underlying rule is easily understandable,

³In the Linux kernel, the target date is not the release itself but the two week ‘merge window’ after which no new functionality will be accepted. In practice, the project makes releases roughly every three or four months.

⁴At the end of 2006, OpenOffice.org announced their intention to move from a three month interval to one where major releases are made every six months.

there is an upper boundary as to how long a release interval can be for people to perceive it as regular. When a release interval becomes too long, people no longer have a feeling of regularity because there is no steady flow of releases that can be observed in an amount of time that is easily understandable. It is therefore important to make sure that releases are not too far apart.

This section will consider the possible benefits from the implementing of a regular release cycle. There are three benefits that a regular release cycle offers:

- Reference point: as discussed above, development in FOSS is done in a highly parallel fashion in which contributors perform work very independently. Even though this independent work is beneficial because it lowers the requirements for active coordination, it is important to regularly synchronize all development. This is required because developers need to be aware of changes made by other developers, some of which may conflict with their own changes, and some of which may be helpful in one's own work. The publication of regular releases provides this synchronization point which makes sure that every contributor is working on the same code base.

Jeff Waugh, who managed releases for the GNOME project for many years, has given a great analogy to underline the importance of regular releases. This analogy relies on video compression methods, such as MPEG. Such compression algorithms do not store each picture ('frame') individually. Instead, they store one frame and subsequent changes made to that frame. At some point, they include a full frame again and then record only changes made to this frame. This mechanism reduces the amount of storage space because not every frame is stored as an entire frame. At the same time, it protects from damage made to the disc and allows to fast-forward because it periodically stores a full frame from which to start again. This frame, which contains the entire screen, is known as the keyframe. Waugh argues that regular releases act as their keyframe:

For us, the stable release is the keyframe. A full complete picture of where we are. The development is the modification

to the keyframe. Then you have another keyframe — the full picture. There are only certain things changing, you're never unclear about what has changed, you know what needs to be tested. (Jeff Waugh, GNOME)

Essentially, regular releases act as a reference point which allows developers to synchronize their work. This is especially important in large projects which have many different components which are developed individually.

It is important to make releases and thereby establish a synchronization point regularly. As the time interval between releases increases, it becomes more difficult for developers to synchronize their work and make a new release on time.

- Discipline and self-restraint: regularity provides developers with a very orderly development phase and promotes self-restraint. One of the main downsides identified in section 6.1 regarding releases which are badly planned are that developers try to push their work in, even though the release manager has called for stabilization. Volunteer developers want to see their work included in the next release and they push harder the more uncertain it is when another release will happen. A regular release cycle removes this uncertainty by giving developers confidence that the next release will take place within a reasonable time:

For developers, regular releases are like trains; if you miss one, you know that there will be another one in the not-too-distant future. (Joe Buck, GCC)

This makes it easy to convince developers that new functionality which is not quite ready should not be included or taken out again. After all, the next release will take place in the foreseeable future:

All you ever have to say is that if you revert it you can put it into the next development phase. It will be immediately available. If it's one month before the release and we say 'revert this feature', it will appear again in the development code after one month. It will appear in stable code after seven months. It's really not so long to wait so it's not so bad to tell them to revert it. (Murray Cumming, GNOME)

In summary, a regular release cycle makes it easier for release managers to maintain control over the release process and to enforce deadlines. Developers are more willing to postpone unfinished work until the next release, and the predictability may give developers more motivation to work on new features, as will be discussed later.

- Familiarity: making regular releases gives developers a familiarity with the process. If a project only performs a new release every once in a while, developers are not used to it and problems may occur. However, if they implement a regular release cycle with not too infrequent releases, developers will get more experience and will get used to it:

With something like cooking, we're just used to doing it every day so you know how you do it and it doesn't fail. (Andreas Barth, Debian)

A fixed interval contributes more to familiarity than a variable interval because projects with fixed intervals can use very similar schedules for each release:

Because in the end, for every release we have a very similar schedule, just at different times. People got used to expecting what will happen. They became self-policing. (Murray Cumming, GNOME)

This familiarity with the process takes a lot of burden away from the release manager. As discussed previously, many projects without a well planned release process struggle a lot during release preparations. The lack of planning leads to fire-fighting, and release managers typically have to coordinate many activities. However, the introduction of a regular release cycle with a well planned schedule allows people to coordinate their own work and work together with their peers. The release managers in the GNOME project, which has perfected their release management and performed several releases after exactly six months, say that release management in their project works 'like a machine'. The release managers have to perform little coordination and their main roles are to carry out administrative functions, such as preparing test releases and writing press releases.

In summary, regularity is associated with three benefits which each contribute to an effective release process, namely:

- Regular reference points
- Discipline and self-restraint
- Familiarity with the process

Schedule

The second coordination mechanism employed by time based release management is the use of a schedule. One could argue that the implementation of a schedule does not rely on a switch to the time based release strategy. However, in volunteer projects it is very difficult to construct a schedule based on features or similar criteria. This is because the project has little or no control over what work will actually get performed given that the participants are all, or mostly, volunteers:

If you can have a feature based project in a finite time, that's great but it seems that in an open source environment the feature based strategy is just basically impossible unless you want to wait forever. . . which is what happens to a lot of projects. Some haven't had a release in five years. You cannot tell anybody to do anything. If you make a list of features, that's just a wish list, basically. (Havoc Pennington, GNOME)

By using time as the orientation, a schedule can be constructed. While planning is difficult or impossible with regards to features in a project consisting mostly of volunteers, having time as the criterion enables the project to plan and establish a schedule. In terms of the coordination framework proposed by Malone and Crowston (1994), the implementation of a schedule can therefore be regarded as an important coordination mechanism. In fact, the schedule in a FOSS project employing a time based release strategy plays a more important role than in the framework of coordination. In the framework, the schedule is used to handle simultaneity constraints. In the FOSS project, the schedule is the overall planning tool that allows a project to define dependencies between activities. Now that planning is possible, the different dependencies from section 3.3.3 can be analyzed by project managers and taken into consideration

while generating the schedule and other information for the release process. For example, the schedule can be used to share information among developers about prerequisite constraints: deadlines can be instituted for different activities and activities can be arranged in their natural order. This is crucial for many tasks and individuals, for example for translators who can only perform their work when the documentation they need to translate has been finished and is no longer in a state of flux. The schedule not only tells translators when they can start their work, but by specifying a ‘string freeze’ it will also tell developers when they must stop making changes to texts (‘strings’).

A schedule which identifies the most important phases for the development and release process provides contributors with important information:

By identifying that certain things will happen at certain times, you take away some mystery. You make it much easier for people to adapt to each other’s work. (Murray Cumming, GNOME)

Effectively, the schedule is a means which allows FOSS contributors to perform their work with as little active coordination as possible. As discussed before, the lack of a schedule and planning results in many questions being addressed to release managers as to what work has to be performed at what time. If there were no schedule, it would not be clear to developers when they had to stop changing texts and neither would translators know when to start their work. Both parties would have to constantly talk to each other as well as to the release manager who oversees the overall project. The introduction of a schedule makes this active coordination unnecessary because the schedule gives each individual the information needed so they can perform their work independently. Of course, they still need to coordinate with their peers, but the reliance on a central authority, the release manager, is greatly minimized. As a result, costly coordination is reduced and individuals can be self-policing to a much greater extent. This effect is supported by the regularity of the release process, which over time gives contributors experience with the process, thereby further decreasing the amount of active coordination that has to be carried out by the release manager.

Summary

The time based release strategy allows the introduction of two important mechanisms that reduce the amount of active coordination that is needed in a project: a regular release cycle and a schedule. The implementation of a schedule based on the completion of features is difficult, if not outright impossible, in a distributed project consisting mainly of volunteers over which the project has little control. However, by using time as an orientation, planning is possible. Dependency information between different activities and actors can be written down in the schedule and deadlines can be instituted.

The schedule provides contributors with important information they need to know in order to perform their work efficiently within the framework of the project in which hundreds of other contributors or more are actively involved. Instead of coordinating every contributor individually, the release manager publishes a schedule which includes all information developers need to know to perform their work independently. The schedule is therefore an important means of coordination that reduces the amount of active coordination which is costly in distributed projects. By having a regular release cycle, coordination is further reduced because developers have a reference point and gain familiarity with the process. It also leads to more discipline and makes it easier to postpone unfinished work because the next release is not too far away. This makes it more likely that individual deadlines set out in the schedule are kept, that the project meets its target release date and that over time a regular release cycle is followed.

This theoretical explanation of time based releases as an enabler of important coordination mechanisms allows the replication of the findings of this dissertation to projects which exhibit similar characteristics to those studied here. In particular, it can be argued that the simple use of time rather than features as a means for orientation for deadlines allows the institution of better planning in projects which have little control over their contributors. Such projects may be software projects, but they could also be volunteer projects in other disciplines.

6.2.3. Advantages of Time Based Releases

A project employing a time based release strategy with a well planned schedule and regular releases is associated with a number of advantages compared to projects which follow feature-driven development that is often associated with an ad-hoc release style. This section will describe the key advantages a project can gain by successfully moving to a time based release strategy. Before advantages gained by the development community will be explored in detail, a number of benefits will be discussed that time based releases offer to different stake-holders.

Four classes of stake-holders are affected by the release process and they gain the following benefits from a time based release strategy with predictable⁵ and regular releases:

- Organizations: the predictability of time based releases allow organizations to plan their software deployment better. For example, if an organization wants to upgrade a large number of machines, they need to plan well in advance. Therefore, it is beneficial if software projects have clear target dates and a good track record.
- Users: projects which follow a regular release cycle can provide users with fixes and new features periodically:

Generally, regular releases keep users happy since they get a steady stream of fixes and new features. (Christoph Hellwig, Linux)

Furthermore, each version is a gradual increase rather than a ‘big bang’ release which completely changes the software, which makes it easier for users to adapt to the new release.

Finally, regular releases are an indication that the project is doing well. This reassures users that the software they have deployed is still actively being developed. It may also create excitement among high-end users and give them an incentive to get involved in the project, for example by providing feedback (Crowston and Howison 2005):

⁵This assumes that the project actually meets their target date, which is not always the case as seen in some case projects in chapter 5. The following two chapters will discuss factors contributing to a successful implementation of the time based release strategy.

It's good to show people that something is actually happening. I think that the world is beginning to move so quickly that if you release something once only every 18 months then there's a big quiet gap in between. Whereas if you have more releases, with this and that new feature, people can see progress and they are much more excited about the project. (Michael Meeks, Novell)

- Developers: contributors know exactly when they have to finish their work in order to make it into the next release. Their contributions will get published and shipped to users relatively quickly, and this may be associated with increased motivation, as will be discussed below.
- Vendors: projects with clear schedules allow vendors that would like to distribute the software to make better decisions as to which releases to use:

Say you know that GNOME will be released in three months and you need a stable release in five months you know that you can count on getting the new release of GNOME with a margin of safety. (Havoc Pennington, Red Hat)

A clear schedule also allows a vendor to participate more closely in the development of the project. With the help of a schedule, vendors can decide whether new functionality they would like to ship should be developed as part of the official project or on their own development line. The predictability offered by time based releases promotes vendors to work on the official project and decreases fragmentation, which often occurred in projects that had little planning (see section 6.1.2 for a discussion of this problem and section 5.3 for a concrete example based on the GNOME project).

It should be noted that regular releases can also be associated with downsides, such as high upgrade costs and fragmentation of the user base. These issues will be discussed in more detail in the next chapter where factors associated with the choice of a release interval and the creation of a schedule are investigated.

In terms of the development community, time based releases may offer a number of advantages compared to ad-hoc releases. In particular, time based release management may:

- Create discipline: well planned releases are associated with a more organized development process and over time lead to discipline in project participants. This makes it easier for release managers to assert control and maintain firm deadlines. This may also contribute to quality of the output produced for two reasons. First, an organized release process can put a big emphasis on testing, which is often neglected in ad-hoc projects that face severe delays. Second, it allows everyone to work together in a more organized manner:

Part of the religion of GNOME's six month release cycle is the fact that everybody is beating at it at the same time which leads to the quality of the platform evolving very quickly. (Andrew Cowie, Operational Dynamics)

- Allow better planning: regular releases help planning because the time frame is foreseeable. Furthermore, they give developers experience and the schedule can be adjusted accordingly.
- Lead to better feedback: section 6.1.2 showed that a negative effect of lack of planning and constant delays was that users would provide little feedback or that comments from users would apply to very old software and therefore often no longer be relevant. Time based releases allow the publication of regular releases and therefore create a much tighter feedback loop with users:

If you can keep the development more tightly linked to actual usage you will get much better feedback, in terms of bugs and development direction. (Havoc Pennington, GNOME)

This can be related to the theory proposed by Raymond (1999) which argues that feedback from users significantly contributes to the quality of a FOSS project.

- Contribute to motivation: regular releases may increase motivation and the output produced by contributors for three reasons. First, developers who contribute features or bug fixes know that their contributions will be made available to users within a fairly short amount of time. This is different to projects with ad-hoc releases where it may take years for a contribution to actually reach end-users. When OpenOffice.org moved

from an 18 month cycle (that faced several delays) to a three month cycle, they observed less frustration and increased activity:

We've seen about the same level of involvement [in terms of number of developers] but less frustration. Activity, I think, has arguably picked up. (Louis Suarez-Potts, OpenOffice.org)

Second, positive feedback from users may also motivate developers who contribute to a FOSS project in their spare time.

Third, the more organized process which relies on clear deadlines may motivate developers to contribute more:

By having regular releases, we affected how the community works on their code. One is that right before a release we get a flurry of activity. There's also a constant amount of background activity, especially for larger projects. (Kevin E. Martin, X.org)

Each release and the deadlines associated with it may prompt a number of contributors to increase their involvement and finish work they have previously begun and that they wish to see included in the next release. Mechanisms that increase motivation are very important because volunteers contributing to FOSS projects may do so without any direct financial incentives but they typically require some rewards, such as positive feedback and wide deployment of their contributions. A release strategy which contributes to developer motivation is therefore important because it makes it more likely that volunteers will continue to contribute to the project.

In summary, time based releases may motivate developers and enhance their output by creating a tight feedback loop with users, guaranteeing that their contribution will be made available to users fairly soon and prompting them to finish their work in order to make the next release. The time based release strategy also promotes more discipline and allows better planning, both of which may contribute to the quality of the software. In addition to creating benefits for the development community, time based releases offer a number of advantages to all stake-holders, especially predictability, which is appreciated in particular by large organizations and vendors that rely on the software produced by volunteer FOSS projects.

6.2.4. Open Questions

In section 6.2.1, four conditions were identified that projects wishing to move to a time based release strategy have to meet. There are some open questions from the case studies that cannot be fully answered based on the evidence from the case studies as to the applicability of time based releases under certain circumstances. First, it remains an open question whether the maturity of the project is a factor that needs to be taken into account. Second, it is not fully clear whether a time based release strategy allows for major, radical changes. It is possible that the first case could lead to an additional condition for the move to the time based release strategy, and that the second issue may require a temporary change of a project's release strategy to accommodate the requirements of development. These issues will be explored in more detail in this section.

Time Based Releases and the Maturity of Projects

The first question is whether time based releases can be implemented in any project that meets the four conditions outlined in section 6.2.1 or whether projects also need a certain maturity. The time based release strategy requires that development will be completed and sufficiently tested within a limited time frame because the project would fail to meet its target date otherwise. In young projects, interfaces between different components are often still in flux, requiring changes to be made to the whole system. The stabilization of interfaces is an important step in the development phase towards the preparation of a release and it becomes easier in mature projects where interfaces remain fairly stable:

A large part of our releases depend on the API [application programming interface] freeze because you cannot build an application on a constantly moving API. Yet, eventually, an API doesn't change so much. Eventually it has the functionality it needs and it's easy to use. (Murray Cumming, GNOME)

Does this rule out time based releases for young projects which have interfaces that are still changing frequently? While changing interfaces are a potential risk that may delay a release, there is evidence that it is possible for a project to change interfaces during the release and still meet the target date.

An example is the Bazaar project which follows time based releases but is a young project with changing interfaces.⁶ As long as changes to the interfaces are done at the beginning of a development phase and stabilized in time for other development to occur and be tested, it appears to be possible to change interfaces when following a time based release strategy. In other words, the introduction of new interfaces or other large changes is a potential risk, but it does not completely rule out time based releases. The time based release strategy can therefore be applied to young projects that meet the conditions set out in section 6.2.1 as long as they are aware of the risks involved with changing interfaces and implement mechanisms to deal with potential problems.

A different question related to the maturity of projects is whether time based releases are appropriate for highly mature projects that do not change very much:

[Time based releases] may be overly restrictive in very mature projects in which there is not much change. It's not necessary to freeze your development, your API, for three months or your code for two weeks if it's not solving any real problems. (Murray Cumming, GNOME)

There are two counter arguments to this. First, even though it may not be necessary to freeze development for several months in a very mature project, time based release management does not have such a requirement. The only requirement is that a target date is set and that all development and testing is completed by that date so a new release can be published. How the interval between releases is divided, for example in development and testing phases, is up to individual projects. Mature projects can adjust their development phase and decrease the time spent in freezes, given that there may be few changes that need to be tested. Second, it is clear that mature projects would not be suited for a time based release strategy if there are not enough changes over time to warrant new releases regularly. However, this condition is already covered in section 6.2.1. In summary, it is likely that mature projects will benefit from regular releases, assuming they have enough changes to warrant a release. In most cases, they would however plan their development phases differently to younger and more volatile projects.

⁶<http://bazaar-vcs.org/>

Making Radical Changes with Time Based Releases

Another open question is whether the time based release strategy allows major and radical changes to be performed. Chapter 5 has shown that the introduction of time based releases has considerably improved the release process in a number of projects. The time based release strategy has allowed those projects to perform regular releases and deliver incremental improvements to users. However, it seems clear that a complete re-write of an application would not be doable with a time based release strategy because there is no guarantee that it will be completed on time. Nevertheless, few projects throw away all their code and start a complete re-write.

The issue of radical changes was first discussed in section 5.3 in light of the GNOME project. GNOME has delivered several releases according to a six month interval and each release can be considered an incremental improvement. However, some developers and users wonder whether the project will ever move from their 2.x line to GNOME 3.0. While this issue is debated from time to time, core developers of GNOME do not believe it to be a real issue:

And when we actually start to list what we have, what we know about (for a 3.0 release), you get some vague ideas about radically new concepts but nothing concrete... just a wish that something new and cool is there. Or you get specific things but when you start asking you realize that you actually don't need to break anything. It's actually quite easy to add it, and lots of things we've talked about in terms of 3.0 we've already added since they were mentioned. (Murray Cumming, GNOME)

Another example is the Linux kernel (section 5.4) which has made radical changes to their stable releases after moving to time based releases. It is important to note that even though time based releases have to meet their target dates, there is no requirement that a specific change has to be made during one release interval. If a developer wants to make a major change, they can work outside of the main development line (on a so called 'branch') and integrate their work when it is completed:

The other good thing is that when you're saying the feature has to be good enough in six months you're also saying 'you can go away and spend a year on it. We don't mind if you don't get it ready for this release... get it ready for the next one'. So you can work in a branch. (Murray Cumming, GNOME)

This effectively means that major changes that are fairly self-contained or can be coordinated easily with other developers can be developed over several release intervals outside of the main development line. Time based releases therefore only have to be ruled out if radical changes have to be made to the whole application, such as a complete re-write. However, such changes do not typically occur in most projects.

Summary

Two open questions regarding the applicability of the time based release strategy have been discussed. The first question is whether the strategy requires projects to have a certain maturity. Even though there are risks associated with young projects that have changing interfaces, they can use time based releases as long as they manage to stabilize their development in time. The second question is whether radical changes are possible with the time based strategy or whether this strategy is only suited for incremental updates. There is evidence from a number of projects, including GNOME and Linux, that major changes can be made, at least to some extent. Developers can simply skip one release interval, work outside of the main development line and integrate their changes once the work is completed. Even though there is not enough evidence to fully answer these questions, it seems that neither of them rule out time based releases. However, as projects gain more experience with this relatively new release strategy, the specific conditions needed for, and limitations of, this release strategy can be further refined. It is possible that some changes require a temporary change in release strategy, but more experience with this novel release strategy is needed to answer this question conclusively.

6.3. Chapter Summary

This chapter performed a cross-case analysis of the case studies presented in chapter 5. The chapter started with a discussion of problems experienced by the case projects that prompted the move to a time based release strategy. After conditions required to implement a time based release strategy were investigated, a theoretical argument was presented to answer why the introduction of this release strategy has led to improvements in the projects under

investigation. The argument showed that regularity and schedules, which are associated with the time based release strategy, act as important coordination mechanisms. They allow FOSS contributors to perform their work with little coordination because the schedule supplies them with dependency information between activities and regular releases act as a reference point and create familiarity with the process. The idea of using time rather than features as the main criterion for a release can be applied to other projects that have little control over their contributors, such as other volunteer projects outside the software sector.

Since the advantages of time based releases rely on a successful implementation of this release strategy, the following two chapters will focus on factors associated with the implementation. The next chapter will discuss factors affecting the choice of an appropriate release interval and the creation of a good schedule, whereas chapter 8 will look at specific coordination mechanisms used in successful implementations of the time based release strategy.

7. Schedules

As discussed in the previous chapter, the schedule plays a fundamental role in the successful implementation of time based release management in a project. This chapter will therefore focus on aspects related to the creation of schedules. This chapter is divided into the following two sections:

1. Choice of the release interval: factors that influence the choice of an appropriate release interval for a project are discussed.
2. Creation of a schedule: important factors that need to be considered for the creation of a schedule are discussed.

7.1. Choice of the Release Interval

There are numerous factors that influence the choice of an appropriate release interval. Based on a cross-case analysis of the case studies from chapter 5 the most important factors which have been identified will be discussed.

7.1.1. Regularity and Predictability

As argued previously, the publication of regular releases acts as an important coordination mechanism, not only because it creates discipline and familiarity regarding the release process, but particularly because it establishes a reference point. This reference point allows the synchronization of a large number of volunteers involved in a project who perform their work with high levels of autonomy.

In the creation of a reference point, it is important to keep the number of changes that have been made since the last release to a level that is easily manageable and understandable. Specifically, projects need to adopt a release

interval that keeps the balance with regards to the number of changes being made:

[Our six month cycle] gives you enough time to develop some new features without too much time to get too far away from the previous version. (Murray Cumming, GNOME)

If the release interval is too short, contributors will not have enough time to integrate their work and perform adequate integration testing to ensure high levels of quality. On the other hand, a very long release interval leads to a high number of changes and that may be associated with certain problems. As the number of changes increases, it becomes increasingly harder to keep an overview of all changes that have been made, to stabilize the code base so it can be released and generally to manage the project.

Furthermore, a project following long release cycles will find planning to be much more difficult than one with a shorter interval:

With a short time period you can foresee what will happen. But when you speak about release cycles of 1.5 years, it's much harder to make a forecast about what will happen, [and] how it will look like. (Andreas Barth, Debian)

One of the benefits of regular and fairly frequent releases is that release planning is always on the mind of contributors and releasing becomes part of the culture of a project. If the release interval is too long, developers may make changes that are not complete or not tested enough and this may lead to delays during the release preparations. For these reasons, it is important to find a balanced release interval that allows enough development while making planning and the creation of regular reference points possible.

Finally, regular releases are associated with predictability. As discussed previously, a regular release interval that offers predictability is valued by a number of stake-holders, including Linux distributions and vendors shipping software produced by FOSS projects:

You know when things are going to happen, and you know the schedule months in advance and you can decide which version to ship. That's a really good point from the distributor's point of view. (Michael Meeks, Novell)

7.1.2. User Requirements

The aim of this dissertation is to study and improve processes related to the creation and delivery of high quality software. While the focus is on time based release management as a mechanism to coordinate distributed volunteer teams, it is important to stress that the software being developed and released has to be of value to actual users. As such, a crucial step in the choice of an appropriate release interval is to take user requirements into account.

FOSS projects may target a number of different users. For example, GCC is a development tool and its main users are other developers. On the other hand, OpenOffice.org is an office suite. While OpenOffice.org is being used by technically savvy users, the majority of its users are end-users who do not necessarily know about the inner workings of a computer. It is obvious that users influence the requirements of a piece of software but their needs also have to be reflected in the release interval. This is because users can have very different expectations regarding the release cycle. Developers may often be interested in frequent releases delivering ‘cutting edge’ software. They can also cope with breakage while less technical users may prefer more software that has been rigorously tested. Such users may lean towards a slower release cycle that does not require them to upgrade too frequently.

The nature of the project may influence the release interval as well. For example, both the Linux kernel and X.org include drivers for hardware. Since new hardware appears on the market all the time, projects trying to support new hardware are dealing with a moving target. There are several ways to cope with these requirements. First, a project may choose a very short release interval and publish frequent releases. Second, a project may choose to split drivers from other components and perform frequent updates to drivers in addition to regular updates of the whole software package. The X.org project, presented in section 5.7, has implemented this solution in order to provide up-to-date hardware support to their users. With their 7.0 release, X.org moved to a modular code base that allows frequent updates to hardware drivers independently of the regular, less frequent releases published by the project.

Finally, a project may have users who exhibit conflicting requirements. It can be very difficult, if not impossible, to reconcile different needs and interests

of users. For example, Debian publishes a complete Linux system consisting of many different software applications. Their main target has traditionally been the server market but, as Linux has been gaining popularity on the desktop, additional user requirements have increased in importance. Unfortunately, there is a conflict between these interests because servers need to perform their work and can only be upgraded to new software infrequently whereas many desktop users prefer up-to-date software with frequent updates. As will be discussed later, as a non-profit volunteer project, Debian does not have the resources to make different releases for these two classes of users, as some commercial Linux vendors do. However, they found a compromise and have accommodated their new class of users by making up-to-date software available after it has gone through a partially automated testing system.

In summary, user requirements need to be reflected not only in the software itself but also in the release cycle. A project may face conflicting user requirements, and it may be difficult to accommodate different users given the resource constraints in volunteer FOSS projects. Nevertheless, it is important to consider different requirements during the choice of a project's release interval.

7.1.3. Commercial Interests

There may be commercial interests that have to be taken into consideration in the choice of the release interval for a project. Even if a FOSS project may not have any direct commercial incentives, it is typically part of a bigger ecosystem from which it may gain important benefits. For example, many FOSS projects rely on commercial Linux vendors for wide distribution of its software. While a FOSS project can publish its software on the Internet, commercial Linux vendors typically have access to additional distribution channels. Hence, FOSS projects receive an indirect benefit because its software is being distributed more widely, thereby leading to an increase in its user base. They may gain users who provide important feedback, such as bug reports, or who become directly involved in the project as contributors (Crowston and Howison 2005). It is therefore important to accommodate requirements of the ecosystem in which a FOSS project is embedded.

In order to illustrate the importance of this ecosystem, one concrete example is given. An important part of this ecosystem is a book author writing about FOSS. Books make software developed by volunteer FOSS projects more accessible to non-technical users and book authors often act as bridges between end-users and developers. The release interval of a project has a great impact on book authors. If the interval between releases is very short, it becomes impossible to keep books synchronized with the latest release of the software and books will become out of date:

We have a number of people in our community who are book authors, and they support and help us a lot. For them, as authors, it would be the death if you would really bring out a new release every three months with new features. Then they would not sell any books at all. (André Schnabel, OpenOffice.org)

On the other hand, too infrequent releases hurt book authors as well because they can not publish new editions of their books covering the new software release.

A related aspect that needs consideration is publicity. Good publicity and PR may attract more users to a project and may motivate more contributors to actively become involved. Some commercial software companies rely on ‘big bang’ releases to cause great publicity and PR. However, the case studies have shown that there are also great advantages to be gained from regular releases regarding PR. While a ‘big bang’ release may create one big splash of publicity, regular releases may lead to a constant press coverage of the progress of the project. Such coverage shows users and other interested parties that the project is still active. This is very important piece of information for users because there are many volunteer projects which cease to exist. In fact, there is evidence that the majority of FOSS projects are not actively being maintained (Howison and Crowston 2004). Frequent press coverage reassures actual or potential users that the project will continue to be developed and that they can rely on it.

Finally, the release interval can be used as a means for effective competition. There are many FOSS projects competing among each other, as well as with proprietary software, such as Mozilla Firefox with Internet Explorer, OpenOffice.org with Microsoft Office and Linux with Microsoft Windows. The

implementation of a short release interval may allow FOSS projects to compete more effectively with the proprietary software industry which often has much slower release cycles:

A fast release schedule gives substantial advantages over a competitor with a long cycle, for example, during their beta cycle, several new releases of your product will hit the market. (Michael Meeks, Novell)

This makes it possible for a FOSS project with a rapid release cycle to react more quickly to changes made and new functionality implemented by proprietary software vendors.

7.1.4. Cost Factors Related to Releasing

There are several cost factors related to releasing that have to be taken into consideration in terms of choosing the release interval. For example, old releases typically have to be supported, at least to some extent and for a certain amount of time. If new releases are published very quickly, projects will face an increased maintenance burden:

If you have to maintain lots of older releases, then it creates a huge burden. (Michael Meeks, Novell)

Since the majority of FOSS projects operate on a volunteer basis and have limited resources, it is impossible to support a large number of old releases. Support issues certainly have a big influence on the choice of the release interval as can be illustrated with an example from the Debian project. As mentioned earlier, Debian has two classes of users with conflicting interests: users who deploy Debian on servers usually do not want to upgrade too often whereas many desktop users expect frequent updates. The project considered introducing more frequent releases to address the needs of desktop users while supporting older releases for a longer amount of time so servers would not have to constantly upgrade to the latest version. This would effectively mean that the project would have to support two releases at the same time while performing development work on the forthcoming release. The security team of Debian, which was at that time already struggling with the support of one release, argued that they were not in a position to support two releases. In the

end, the project introduced a mechanism that fulfils the needs of some desktop users, but resource constraints stopped the project from adopting a solution that would meet the needs of everyone fully.

A project also needs to take into consideration that the creation of a new release is associated with a number of fixed costs, such as testing and release work:

There is lots of work associated with a release. I don't want to do a full release test every two months. (André Schnabel, OpenOffice.org)

It is important to note that this comment was made in an interview that took place several months before OpenOffice.org decided to implement a release interval of three months. After about a year with this release frequency, the project announced its intend to move to a slower release cycle, delivering releases with new features every six months. Two reasons were given as support for this new release interval. First, users expressed that they would prefer to see new features less frequently. Second, the QA team found it difficult to thoroughly test releases with the short release cycle of only three months. This supports the present argument that fixed costs, such as testing, are an important consideration in the choice of a project's release interval. Such work is crucial to ensure that the software shipped to users is of high quality and must therefore not be neglected.

Cost Factor	Description
Support for old releases	A project needs to support old releases, at least to some extent and for some time.
Fixed costs of releases	Each release is associated with some fixed costs, such as testing and release work.
Confusion among users	Frequent releases may confuse users as to which release they have to use.
Fragmentation of users	Frequent releases spread users over a large number of releases, making upgrades harder. It may also reduce the quality of feedback.
Upgrade costs for users	Each upgrade is associated with some costs, such as time spend downloading the software and getting acquainted with new features.

Table 7.1.: Cost factors related to the release interval.

Another problem of very frequent releases is that users become spread over a large number of different releases:

If the releases get too fast you obviously get the problem of spreading the user base over much more releases instead of having them on a few. (Christoph Hellwig, Linux)

This may make it harder to track outstanding issues, feedback may be less relevant because many users remain with old versions that are very different from the current development line and it may cause problems with upgrades.

In addition to creating more work for the project developing the software, there are also important upgrade costs that affect users. Every single update is associated with some costs, such as time spent downloading and upgrading the software or time needed to become comfortable with changes made in the new release. A balance has to be found between too frequent releases on one hand and infrequent, ‘big bang’ releases on the other hand, that are associated with a steep learning curve and with more coordination challenges for the development team.

Finally, frequent releases may also confuse users. Even though users will benefit from the delivery of incremental updates that deliver fixes and new features, a very fast release cycle may cause uncertainty among users as to which release is appropriate for them:

That’s one of the questions we often have to answer. Users say to us ‘I have just upgraded my system and now you give me another release’. We have to teach them that they’re not forced to use it. (Murray Cumming, GNOME)

There is usually a delay between a publication of a new release by a FOSS project and the availability of that version in a Linux distribution, which is typically the main source for end-users. In many cases, a FOSS project may publish a new version before the previous version is actually deployed in major Linux distributions and shipped to users. Even though this may create confusion among users, it is usually not a matter of concern because there is support from the vendor:

Distributions support it. Different distributions decide whether they want to be on the cutting edge. For instance, Red Hat Enterprise Linux and Debian update less often, and therefore have

less combinations of software to support. And that's understandable because they put such a high value on stability and lack of surprises. (Murray Cumming, GNOME)

In other words, vendors play an important role because they often support older releases that are no longer maintained by the FOSS project that produced them.

In summary, there are a number of cost factors that limit the frequency of a project's release cycle. While there are benefits with the delivery of regular updates, it is also important to consider that too frequent releases are associated with downsides. It is crucial to leave enough time in the release cycle for development, testing and other release work, and to consider upgrade costs faced by users. It is possible to work together with vendors of FOSS who may support older releases published by a project.

7.1.5. Network Effects

There are tremendous advantages to be gained if a project can synchronize its release schedule with those of other projects from which it may leverage benefits. For example, one of the key reasons why the Plone project has decided to move to a six month time based release strategy was to align its development closer with that of Zope. Plone is built on top of Zope and the implementation of a similar release strategy will allow the project to use the newest technologies developed by Zope:

In order to stay up to date, we need to tie ourselves to a specific Zope release and the way to do that is to have our release schedules synchronized. (Alec Mitchell, Plone)

While such a synchronization may offer mutual benefits to the projects, it also creates a dependency as one or both projects start to rely on the release cycle of the other. If Zope is delayed, there is a high probability that Plone, whose development line makes heavy use of new Zope features, may have to delay its own release until the next version of Zope is finally released.

Another source from which major benefits can be gained are vendors shipping software produced by FOSS projects, such as Linux distributors. They incorporate a large number of different software, and prominent and successful

FOSS projects typically attract contributors working for major FOSS vendors. Since they include the software in their own releases, they have a natural interest to participate in the development by implementing new features and getting the code stable. In some projects, the contributions made by FOSS vendors can be quite substantial, as for example in GCC, GNOME (German 2004) or Linux.¹ In February 2007, there was some debate on the GCC mailing list as to whether version 4.2 should be skipped since neither Red Hat nor Novell/SUSE, two main contributors to the project, had intentions to deploy that version.² This debate was preceded by problems with the stabilization of GCC 4.2, which some developers partly attributed to the lack of interest and participation of these vendors in the 4.2 release.

This is not to say that FOSS projects typically rely on FOSS vendors in order to produce releases on time and with high quality, or that only projects benefit from this cooperation. Vendors also gain important benefits if everyone is working on the same code base — volunteer developers from the FOSS project and competing vendors. An example of the downsides and challenges faced by vendors that do not work in collaboration with the community and other vendors is Sun's Java Desktop System, a desktop environment for Solaris and, formerly, Linux. This desktop environment incorporates GNOME, but in the past Sun did not deploy the latest version. This created increased work for Sun:

Sun uses a version that is about a year old and they have to do most of the bug fixing themselves. That's a disadvantage for them and that's one of the reasons why we keep telling them they should synchronize with our release schedule like other vendors do. (Murray Cumming, GNOME)

It should be noted that this situation has changed since the interview was performed and that Sun has made efforts to synchronize releases of the Java Desktop System with GNOME.

This example illustrates the importance for everyone to work together, even competing vendors. Many examples of this new form of cooperative competition can be found in the FOSS industry, which has even led to the creation of a new word to describe this phenomenon: co-opetition.³

¹<http://lwn.net/Articles/222773/>

²<http://gcc.gnu.org/ml/gcc/2007-02/msg00427.html>

³<http://en.wikipedia.org/wiki/Coopetition>

Important network effects can be created if many projects follow a similar release cycle and strategy. As table 3.3 on page 53 and chapter 5 show, a large number of time based FOSS projects have chosen a release interval of six months. Since major Linux distributions, such as Fedora, follow the same release interval this ensures that these distributions will be able to ship the latest releases produced by many FOSS projects. This increases the exposure of the software and may lead to better feedback. It may also provide further incentive for vendors to get involved in important projects and help them meet their release targets, as their own releases might otherwise become jeopardized.

In summary, projects may gain important benefits by synchronizing their own schedule with that of other projects, in particular with those by major vendors shipping software produced by FOSS projects. Nevertheless, it is also important to consider implications of possible dependencies that are thereby created that may complicate a project's release cycle.

7.1.6. Summary

There are many factors that influence the choice of a release interval that is appropriate for a project. Fundamentally, the release interval has to be chosen to be an effective coordination mechanism that allows the creation of regular reference points. However, there are many additional factors that need to be taken into consideration to create a schedule that can not only be successfully implemented, but that also meets the requirements of users and vendors. In this section, factors, such as user requirements, costs associated with releasing, and network effects gained through successful cooperation with other projects and vendors have been discussed to guide the choice of an appropriate release interval.

7.2. Creation of a Schedule

A well planned schedule can act as an important coordination mechanism in large projects. In addition to conveying information about dependencies between activities, a schedule contributes to a more controlled development process. This allows the implementation of deadlines and strict cut-off dates

after which no new code submissions are accepted. On these deadlines, progress of certain features and components is to be reviewed in order to evaluate whether they can be included for the next release or have to be postponed until later.

After the release interval has been chosen, a schedule covering the complete interval has to be created to guide development, testing and other activities. For the creation of a schedule, a number of tasks need to be carried out, such as dividing the interval into different development phases and identifying dependencies between activities.

7.2.1. Identification of Dependencies

This dissertation argues that the schedule plays an important role as a coordination mechanism because it allows contributors to work in high independence and stay self-policing. This is only possible if the schedule contains information regarding dependencies between activities. In large projects, which consist of many different components, there is much work that can only be started when other tasks have been completed:

We have dependencies. Applications depend on APIs, translations depend on strings, documentation depends on the UI [user interface], the UI depends on application writers and the API. (Murray Cumming, GNOME)

During the creation of the schedule, such dependencies need to be identified. Little practical advice can be given regarding the identification of dependencies since it relies on experience with a particular project. However, a good start appears to be the investigation of past problems that occurred in a project. If some participants had problems completing their work because tasks for which other people were responsible were not ready, a clear indication for a dependency has been found. Once dependencies have been identified, they have to be put down in their logical order.

A further step is to institute clear deadlines. Volunteers working on a specific activity must know well in advance, ideally long before the start of a new release cycle, when they are expected to complete their work. Even though the aim is to meet every single deadline, the schedule should also allow for

some slippage by adding buffers. This practice will not only allow smoother release preparations, but it also motivates contributors:

If you can adapt a bit, it's easier to get support from people because they feel that you're responsive to their needs. It's a bit like haggling. . . you give them a little bit and they feel more comfortable about it. (Murray Cumming, GNOME)

While there are risks in being overly strict, it is even more important to make it clear that deadlines are firm and are to be met unless an exception is made. If the release manager is not firm regarding deadlines, it is likely that volunteers will increasingly ignore them, which can lead to delays and other problems.

The release manager should also aim to reduce dependencies between activities as much as possible, in particular when there is one task on which many other activities depend:

If you get to the stage where you have multiple dependencies and all stuff has to be finished when this little thing is finished then it's difficult. (Murray Cumming, GNOME)

There are a number of ways to reduce dependencies, for example by arranging for fall back options. If the new interface provided by libraries is not ready by a certain time, it should be possible to go back to the previous version of these libraries which provide well-known interfaces. This is important because stable interfaces are a prerequisite for further development very early in the development cycle, as, for example, applications rely on them. A great way to decrease and identify dependencies is to study the structure of the code base, and to see whether modularity can be increased.

7.2.2. Planning the Schedule

The next step in the creation of the schedule is to divide the release interval into different phases, such as development and testing. This step takes both dependency information between activities into account as well as experience from previous releases. If a project decides to move to time based releases, they already have experience with their previous releases and know what worked and what did not. Largely, the structure of the release cycle should be maintained,

unless there are severe problems with it. This is because project participants already have experience with the structure.

Very broadly speaking, it is important to strike a good balance between leaving enough time to develop new features on the one hand, and to perform testing and release preparations on the other hand. One of the main criteria a schedule has to fulfil is to be realistic. While a majority of developers are mainly interested in adding as many new features as they can, the project as a whole, led by the release manager or core developer, has to be realistic as to how much new code can be added so that it can still be sufficiently tested within one release interval. As discussed previously, there is no requirement for a specific feature to be developed over the course of only one release interval. A developer can work on new features outside the main development line and integrate the work when it is ready. In fact, some projects develop the majority of their features outside the main development line and only integrate them when they have been completed and approved by the release manager. OpenOffice.org follows this model and use a system to manage the different development lines, as will be discussed in section 8.2.2. They argue that direct development on the main development line involves more complications because some features or changes may take longer than expected:

But if you insist on doing all development work on [the main development line], then yes, of course you have a problem because you need a certain amount of time to get anything done. (Michael Meeks, Novell)

This shows that the structure of the development process has an important influence on the schedule. Another example of this is GCC's development phase, which is divided into three stages (see section 5.2). Each stage makes an important step towards the release and permits fewer changes than the previous stage. Furthermore, major changes for stage one and two have to be proposed before stage one starts and will need approval from the release manager. This practice will be discussed in more detail in the next chapter.

While the emphasis in discussions about the production of software is usually put on the development phase, other phases are equally important and have to be considered in the creation of a schedule. A particular advantage of a well planned process with a schedule compared to more ad-hoc development

is that it can promote important phases which are otherwise often neglected. The schedule should include a dedicated period for testing, the preparation of translations and for the release itself.

Finally, it has been argued that there are advantages to be gained from following a similar schedule as it allows developers to gain familiarity with the process. While this is true, projects should also refine their schedule and process if they experience problems with it. After a release is out, it can be very beneficial to start a discussion about the release process to identify good and bad patterns. This research has found that surprisingly few projects perform such a post-release discussion but such a practice can substantially improve the development process and schedule.

7.2.3. Avoiding Certain Periods

Since most FOSS projects rely on volunteer contributions, there are a number of constraints and limitations that influence the release process and the schedule. As argued before, it is important to consider how much development can actually be completed during a release interval to allow for sufficient testing and release work. Another important constraint which some projects fail to take into consideration is the availability of volunteers. Even though proprietary software vendors have clear holidays, many FOSS projects operate throughout the whole year. While development may take place all the time, there are nevertheless restrictions as to when a release can be performed. In particular, there are some periods that should be avoided because important volunteers may not be available.

One period that the case studies actively avoid is the Christmas holidays, during which many volunteers may not be available:

For example, one shouldn't have the freeze of all packages while people are away on Christmas vacation. Also, one should make sure that on the weekend one wants to release some technical people like ftp-master will be available. So we cannot release exactly on Christmas or Easter where important people won't be available. (Andreas Barth, Debian)

There is also some debate whether the summer period should be avoided but evidence here is much less conclusive. While many volunteers will be on

holidays during summer, the absence of different contributors is spread over a period of several months, so it is not a problem for most projects:

People go on holidays at different times. In our project, people are doing work in their free time and even if I have a holiday I'm working on free software. It balances out. (Murray Cumming, GNOME)

In fact, some contributors, in particular students, might even increase their involvement during the summer period.

In summary, the schedule should take important events, such as Christmas, into account. While Christmas should most certainly be avoided, there are open questions as to which other periods are bad for release work, such as possibly summer. Empirical studies that evaluate the amount of volunteer participation over time may lead to further evidence to answer this question.

7.3. Chapter Summary

This chapter has discussed factors that influence the choice of an appropriate release interval and the creation of a schedule. The release interval is important because it allows the establishment of a regular reference point to synchronize developers. It must also take user requirements and other commercial interests into account as well as cost factors that restrict the release frequency. Finally, important network effects can be gained by synchronizing a project's schedule with that of other projects, in particular those by major FOSS vendors.

The creation of a schedule is a crucial step towards the successful implementation of a time based release strategy because it allows dependency information between activities to be put down. It also contributes to a more controlled and planned development and release process, which contributes to higher levels of quality. The next chapter will discuss specific implementation and coordination mechanisms that FOSS projects employ during release management.

8. Implementation and Coordination Mechanisms

The third part of the cross-case analysis of the case studies, presented in this chapter, focuses on aspects related to a successful implementation of time based releases. The chapter is divided into the following three sections:

1. Implementing change: this section will discuss aspects related to the implementation of change in volunteer projects, such as control.
2. Policies: control structures and policies implemented by projects to support release management will be presented.
3. Coordination and collaboration mechanisms: specific infrastructure and other mechanisms that allow coordination and collaboration among contributors will be discussed.

The chapter will close with a summary of different implementation and coordination mechanisms implemented by time based projects.

8.1. Implementing Change

The introduction of a new release strategy in a large and complex project is associated with several challenges. This section discusses aspects that need to be considered in order to implement significant changes, such as switching a project's release strategy.

8.1.1. Incentives for Change

The introduction of a new release strategy is associated with significant changes in a project which may be accustomed to certain policies and processes. As

discussed previously, the creation of a schedule is an initial step towards time based releases but many other changes appear to be necessary for a successful implementation. For example, the move to a time based release strategy may also require the implementation of new infrastructure or policies to make it possible for the project to follow their newly created schedule. As such, the change in release strategy may be associated with major disruption and there have to be good reasons for the change. As a consequence, projects whose release management strategy works should not risk changing it unless they may gain major benefits from the implementation of a new strategy.

On the other hand, projects which face severe problems with their current strategy have a clear incentive to evaluate and possibly adopt a new release strategy. All projects presented in chapter 5 faced severe problems with their releases, such as delays and problems with quality, as discussed in detail in section 6.1. GNOME, which is now commonly used as an example of a FOSS project with excellent release management practices, was associated with major problems during their 1.x series and had a good reason to move to a new release strategy:

Normally, change only happens when the project has clearly gone off the tracks. I guess GNOME was close to being a disaster. Everyone agreed that it had to change and they were looking for a [solution]. (Murray Cumming, GNOME)

It is important that all participants of a project must clearly see the problems their project is experiencing and must have incentives to find a solution to these problems. If this is not the case, it will be difficult or impossible to implement important changes and move to a new release strategy. The reasons for this will be discussed in the following section.

8.1.2. Issues of Control and Trust

Most FOSS projects depend on contributions from a large number of volunteers. Since their involvement is voluntary, the project has typically little control over them (Ljungberg 2000). As a consequence of this, it is very difficult, if not impossible, to implement changes unless the majority of project participants agree with them. Even if a project has a relatively high degree of control, for example through the project founder who is still actively involved

in development, it cannot introduce new changes that would stop volunteers from contributing their time and effort. This may make it difficult to adopt new techniques and policies and implement major changes since a consensus has to be found among contributors. On the other hand, it may also prompt a project to find good solutions that work for the majority of participants.

Instead of forcing contributors to a particular release strategy, release managers can give them the choice and describe the alternatives in great detail. Before GNOME introduced time based releases, they told contributors that they had the choice between staying with their current release strategy that was associated with severe problems or moving to time based releases which promised several advantages:

[We offered developers] a choice between 2 years, possibly more, with no outwardly visible progress and very painful development, losing support, losing developers, or having some constraints and delivering few features more often, with three or four time based releases in 2 years. (Murray Cumming, GNOME)

GNOME moved to time based releases and it did so very successfully. In fact, the success of GNOME may make it easier for other projects to introduce time based releases since GNOME can be used as an example of possible advantages that can be gained by a move to a time based release strategy.

Since release managers have little or no power over contributors, their control mechanisms rely on trust (Ljungberg 2000; Iacono and Weisband 1997). If contributors have trust that the release managers are performing a good job, they are more likely to take their duties seriously and to acknowledge the authority of release managers and accept their decisions. As can be seen in the quote above, the move to a time based release strategy may be associated with certain constraints, such as restrictions as to which changes developers are allowed to make at certain times. Contributors will accept these constraints only if they have trust that they are beneficial for the project. In the end, a rigorous release process is only possible if everyone in the project believes that the project, with the help of its release managers, can meet its targets:

Basically, releasing only works if all involved developers just trust us [the release managers] to do it. If people think it won't happen anyway, they will just upload broken changes and then it won't happen. We need to have the trust of the people and we're working hard to get that. (Andreas Barth, Debian)

If this trust can be established and contributors are working towards a common goal, the release of high quality software on a particular date, they can meet their target, as can be seen in the GNOME project:

The GNOME community has proven that they can do a release any time they want. They say they'll release on the 15th of December and by God they will release on the 15th of December. (Andrew Cowie, Operational Dynamics)

Nevertheless, even though it is possible for volunteer projects to meet deadlines and release targets, there is substantial evidence to suggest that a successful implementation of time based releases takes considerable time. GNOME itself, which has performed releases every six months for the last few years, faced challenges during the introduction of time based releases, as can be seen by the frequency of their releases in table 5.3 on page 88. Even though the project is following a six month interval now, there were some minor delays after the introduction of time based releases. While these delays of less than a month are insignificant compared to delays experienced by other projects, such as Debian whose 3.1 release was delayed by one and a half years, it shows that projects cannot expect to meet all of their targets fully when they first implement time based releases. One reason for this is that it takes time to establish trust. If a project has failed to meet their targets for several years, the introduction of a time based release strategy will not convince them immediately that deadlines are now real and that targets can be met. Contributors have to gain positive experience with the new process gradually and this will reinforce their trust in the process and people and in the belief that targets can be met. Another reason is that the introduction of successful time based releases also requires new policies and infrastructure with which contributors need to get experience first, as discussed in the following section.

8.1.3. Implementation of Control Structures

In addition to establishing trust, a successful implementation of time based releases relies on the introduction of policies and infrastructure that support the new release strategy:

Simply declaring that you'd release every six months is not good enough. It's actually a whole set of rules that were implemented

at the same time that made it go again. (Havoc Pennington, GNOME)

This set of rules and policies can be considered as control structures because they contribute to a more disciplined development process. As discussed before, volunteer contributors will accept restrictions and other factors associated with a more disciplined process as long as they have trust that these mechanisms will be beneficial for the project as well their own involvement. Project participants may be willing to accept a more disciplined development process if it means that, for example, their contributions will be delivered to users as part of a release more quickly than with a more undisciplined process.

In terms of the introduction of policies and other control structures that will contribute to a smooth release process, two important considerations have to be taken into account:

- It takes time to establish policies and infrastructure: in complex projects that have relied on certain practices for a long time, the introduction of new rules, policies or other changes to the development process takes some time. Contributors need to get experience with the new process and, more fundamentally, they need to learn through experience that these new policies are beneficial for them and the project. In the long run, this allows them to establish trust in the process, and this is an important step towards an effective implementation of new policies. As mentioned before, a project that has failed to meet deadlines and targets for years will not be able to change their whole development process overnight in a way that all deadlines and targets will be met. However, as policies are implemented that improve the development process, contributors are more likely to believe in release targets and this will increasingly make it easier to actually meet them.

For example, while Debian (section 5.1) has failed to meet its target date for the 4.0 release, the project has considerably improved its development process, e.g. by clarifying release goals and responsibilities. Even though the project did not release on time, it implemented many changes that not only made it more realistic to meet more goals and targets but that also showed project participants that goals can be achieved. This will

make the next release process even smoother because contributors as a whole have more trust in the process and the people involved, such as the release team.

- Control mechanisms have to be enforced: when policies and deadlines are introduced, it is important to actually enforce them and to make sure that they are followed in the future. If this is not the case, contributors will ignore them and revert to a more undisciplined process, possibly leading to delays and other problems. For example, the GCC project (section 5.2) introduced a great control mechanism several years ago when they split their development phase into three stages. Each stage lasts for two months and permits less invasive changes, thereby allowing a natural progression towards the release. Even though the project has introduced these stages, they are not followed rigorously. As can be seen in figure 8.1, the stages often last longer than two months because they are not followed in a strict manner. The result of this is that releases are not performed every six months, as the project plan suggests, and that developers may not finish their work on time since they perceive that the development stages will last longer than initially planned.

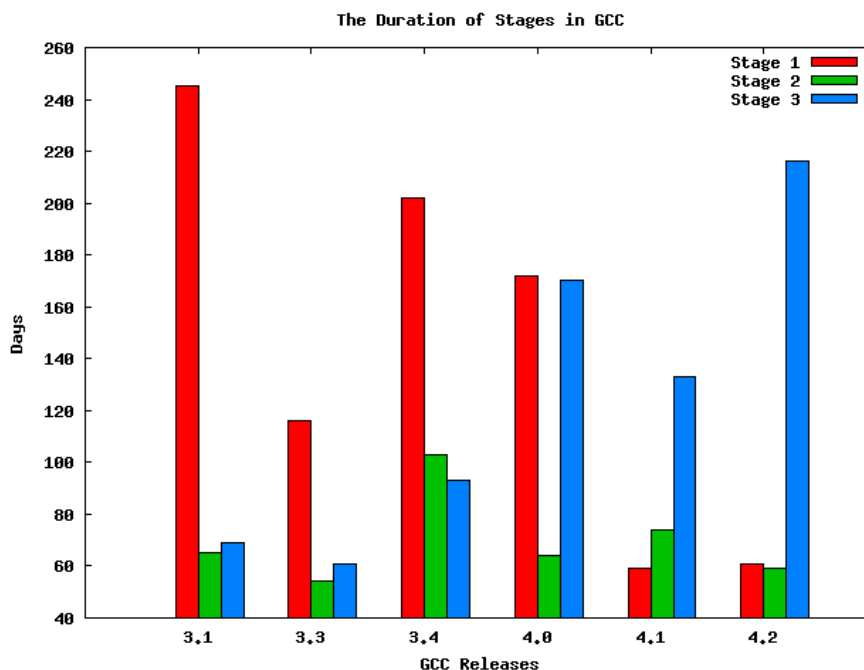


Figure 8.1.: GCC development stages: each stage should last for about 60 days.

In summary, the introduction of time based releases has to be accompanied with policies and infrastructure that support the release strategy. The implementation of these control mechanisms takes time but contributes to a disciplined development process.

8.1.4. Summary

The introduction of time based release management in a project is associated with considerable change. The successful introduction of change in volunteer projects relies on the motivation of participants and the trust they have in the new processes. As such, release managers are not able to introduce a new release strategy against the will of project participants. Instead, they have to offer them choice and describe what the alternatives are. Time based releases are associated with certain constraints and restrictions on one hand, but on the other hand they promise several advantages. A benefit is the regular delivery to users of the software one has contributed to, but this relies on the institution of deadlines and other control mechanisms. The way to meet deadlines in volunteer projects is to create incentives, to convince developers that they can actually meet them and to provide the infrastructure to make it possible. Unless contributors believe that deadlines and release targets are realistic and can be met, they will not work towards these goals.

8.2. Policies

A successful implementation of time based release management relies on policies that support a disciplined development process in which release targets are constantly taken into consideration during development.

8.2.1. Stability of the Development Tree

A regular release mechanism in which releases have a specific target date requires the development tree to be fairly stable all the time. If this is not the case, testing cannot be performed, problems may accumulate and the project may fail to meet its release target. Ideally, the development tree would be in a state that it could be released at any time. While this may not be the case

in many projects, a number of processes have been implemented in different projects to minimize breakage and ensure that the development tree remains fairly stable throughout the whole development phase leading up to the release. These processes include:

- **Branches:** features are not developed on the main development line but in branches. This makes sure that new development does not destabilize the existing development line. It also makes it possible to work on features that take longer than one release interval.
- **Peer review:** some projects promote peer review to make sure that changes that are being made are correct and will not cause any problems. In some projects, such as GCC, changes have to be reviewed and approved by a core maintainer before the change may be applied.
- **Testing:** a more disciplined process allows putting more emphasis on testing. If the development tree is in a broken state much of the time, the amount of testing that developers and experienced users can perform is limited. Periodic testing also promotes a disciplined development process because it allows an identification of errors and bugs that have been introduced, and this makes it possible to find problems with the development process.
- **Reverting and postponing:** if a particular feature is not ready in time for the release, it can be postponed or even taken out again from the development tree if the changes have already been applied. This ensures that only features that are of high quality will be included in a release.

While some of these processes can be found in projects that are not time based, the policy of having a working development tree is important particularly in time based projects and such projects therefore often give special attention to these processes. Furthermore, the introduction of time based releases gives developers an incentive to follow these processes. This can be illustrated with the act of reverting and postponing. One of the reasons why features put in by developers can easily be taken out again without causing too much dissent is the regular release cycle of time based projects. It guarantees

that the code can be put in again and released to users within the foreseeable future, so it is not a major issue if a feature does not make it into a release.

8.2.2. Feature Proposals

A number of projects require features to be formally proposed before they can be included in a release. This is another mechanism to ensure that the development tree remains stable, but it also allows better planning. It is important to emphasize that planning specific features is not incompatible with the time based release strategy. While time based releases use time rather than features as the main release criterion, planning for features is beneficial as long as it is clear that the release will not be postponed if new features are not ready. However, planning for features makes it more likely that those new features will be finished in time for the release.

In the following, several examples from projects will be given. Plone, for example, requires a PLIP (Plone Improvement Proposal) to be submitted for each feature or significant change. The introduction of PLIPs has allowed Plone not only to be more organized in terms of improvements, but it has also enabled release managers to delegate some of their work to others. Previously, it was the task of the release manager to review changes that had been made to ensure that they were appropriate for the release. By formalizing how features are proposed, it became possible to assign the task of reviewing PLIPs to a dedicated team, the framework team. In every release cycle, the framework team will review PLIPs that have been proposed and send feedback to its authors:

The framework team will go through each proposal and look at them and judge how much they are ready, what they need to be considered for the release, whether they're actually useful and necessary for Plone, and they send feedback to the developers who made those proposals about what needs to be improved before they can be considered. (Alec Mitchell, Plone)

Developers will receive feedback from the framework team about their PLIP and they can improve their work accordingly. The framework team will later review the PLIP again and decide whether it is ready to go. However, it remains the final authority of the release manager to decide whether a PLIP will be incorporated into the next release.

PLIPs have allowed the Plone project to be more organized but at the same time it has provided developers with motivation to finish their work:

It seems to have motivated people because they have deadlines to actually get features done within a certain time frame and have an organized plan by which we determine which of those features are ready and which aren't. (Alec Mitchell, Plone)

Even though PLIPs were introduced before Plone moved to time based releases, they only became effective when the project switched its release strategy. Before the project moved to time based releases, few developers saw the point of PLIPs because the whole development process was so unorganized and there was no promise when a new release would become available. PLIPs and time based releases can therefore be considered to be symbiotic: PLIPs promote an organized process that allows the project to meet its release targets, while time based releases give the project a regularity and gives developers a promise that their PLIPs can be incorporated into a release in the near future.

The GCC project has a very similar system, even if it is not as well structured and formalized as Plone's PLIPs. GCC requires major changes that developers wish to make during stage one and stage two to be proposed on the project's web site.¹ GCC does not have a framework team which reviews proposals and provides feedback. Instead, proposals are reviewed by the release manager and specific code changes have to be approved by GCC maintainers responsible for the part of GCC the change applies to. In addition to approving projects, the release manager will assign a sequence to them which says when they can be applied to the development tree. This ensures that changes are spread over the development stage and therefore minimizes destabilization of the development tree.

OpenOffice.org requires features to be developed on branches and maintains a system called Environment Information System (EIS) to keep track of all branches that are available. Through EIS, the status of a branch can easily be seen and developers can request QA activities and other tasks for their branches. EIS allows the release manager to plan future releases better as it gives a good overview of different features that are being worked on. Finally, Debian has taken steps during the release process of Debian 4.0 to separate

¹http://gcc.gnu.org/wiki/GCC_4.3_Release_Planning

release goals and blockers and to formalize them. As with Plone's PLIPs, this more formalized approach has allowed the Debian release team to clarify who is responsible for achieving specific features and goals and for resolving release blockers.

In summary, a number of time based projects have formalized how new features are incorporated into future releases. By requiring features to be proposed and reviewed, projects have created a more organized development process that allows better planning. The policy requiring feature proposals makes planning easier and therefore contributes to the adherence to release dates.

8.2.3. Milestones and Deadlines

Section 7.2 described the importance of having clear deadlines in the schedule. While this is a prerequisite for a successful implementation of time based releases, projects also need clear policies to ensure that deadlines are actually kept and milestones achieved. Feature proposals are one way to create more awareness that the responsibility of getting new features in by a certain time lies with the developers of these features. By actively taking out or postponing features that are not ready, the release manager can also assert control and show developers that deadlines have to be kept. Another important process that ensures a smooth progression towards the release is the use of freezes. They divide the development phase into different stages and set clear deadlines for certain changes. They also allow the management of dependencies between different work items.

As mentioned before, it can be quite hard to enforce deadlines in volunteer projects because there is relatively little control over participants. However, certain control structures, such as freezes, can and have to be introduced into the development process. Release managers gain authority through trust and that allows them to assert some degree of control. In the long run, developers have to see that it is in their own interest to follow deadlines. Even though developers will face some constraints in a time based project, they gain many benefits because their contributions are actually released to users on a regular basis.

Finally, it should be stressed that while deadlines and release targets are important, they are not the only factors that have to be taken into consideration for the actual release. The idea of the time based release strategy is to use time rather than features as the main orientation to plan releases. However, it would be bad for a project to be too rigid about the release date and to perform a release even if there are major problems. It is important to enforce deadlines, but it is equally important to know when more time is needed to achieve adequate quality levels and other characteristics:

It would be a mistake if we released on a given date independent of what [Debian] looked like. Quality is a top issue and Debian has been well known for its quality and we're not giving up that. We're just giving up how many new funky tools need to be included into Debian, which isn't so bad. My experience is that it would be wrong to say that something is the absolute top-most priority and nothing else has any influence on it. (Andreas Barth, Debian)

In summary, projects need policies to ensure that deadlines and milestones are reached on time. Different control structures, such as freezes, can be beneficial in this regard. At the same time, policies should take more factors than just deadlines and release dates into account. Releases have to be postponed if major problems occur or if a release would not meet inadequate quality levels.

8.2.4. Summary

A successful implementation of time based releases is not possible by simply setting a release date and introducing a schedule. A project moving to time based releases also has to introduce several policies that support the new release strategy. This section discussed important policies that were implemented by the case study projects as they moved to time based release management.

8.3. Coordination and Collaboration Mechanisms

Chapter 6 has argued that the time based release strategy acts as a mechanism to reduce the amount of active coordination needed in a project because it allows contributors to work with a higher degree of independence. Nevertheless, some levels of coordination are always required and therefore projects have to

create infrastructure and mechanisms that support coordination and collaboration. Additionally, these mechanisms are typically also a means of keeping project participants informed about the status of the release process. This is important because it increases trust in the process and motivates contributors to participate in the release process.

8.3.1. Bug Tracking

A crucial coordination mechanism during the development phase and release preparations is the use of bug tracking systems (Michlmayr 2005). Users and developers can submit bug reports and feature requests to these bug trackers where they are stored with a unique identifier. Since further information and updates regarding the bug can be added, these bug trackers give a good overview of the status of a bug. Projects typically assign a severity or priority to bugs to show how big an impact a bug has or how quickly it needs to be resolved. For example, bugs in Debian's bug tracker have one out of seven severities (see table 8.1). The project treats bugs with certain severities as 'release critical'. As the name implies, these bugs have to be addressed before a release can be made:

Bugs from a certain severity on (serious and higher) are considered release critical. This means that we cannot release if this bug is still present in the distribution. We have lot of information aggregators on such bugs, including information such as how long open a bug is, what its status is, etc. (Andreas Barth, Debian)

Severity	Description
critical	breaks unrelated software on the system, or causes serious data loss
grave	makes the package in question unusable or mostly so, or causes data loss
serious	a severe violation of Debian policy
important	a bug with a major effect on the usability of a package, without making it completely unusable
normal	the default value, applicable to most bugs
minor	a problem which does not affect the package's usefulness, and is presumably trivial to fix
wishlist	a feature request

Table 8.1.: Severity levels in Debian's bug tracking system.

In order to keep track of these release critical bugs, there are a number of mechanisms which allow developers to easily see the current status. A complete list of outstanding release critical bugs is sent to the announcement mailing list for Debian developers once a week. This acts as a reminder that these bugs need be worked on. There is also a web site² where all open release critical bugs can easily be reviewed and there is a graph showing how many release critical there have been over time to indicate whether the project is moving into the right direction (see figure 8.2).

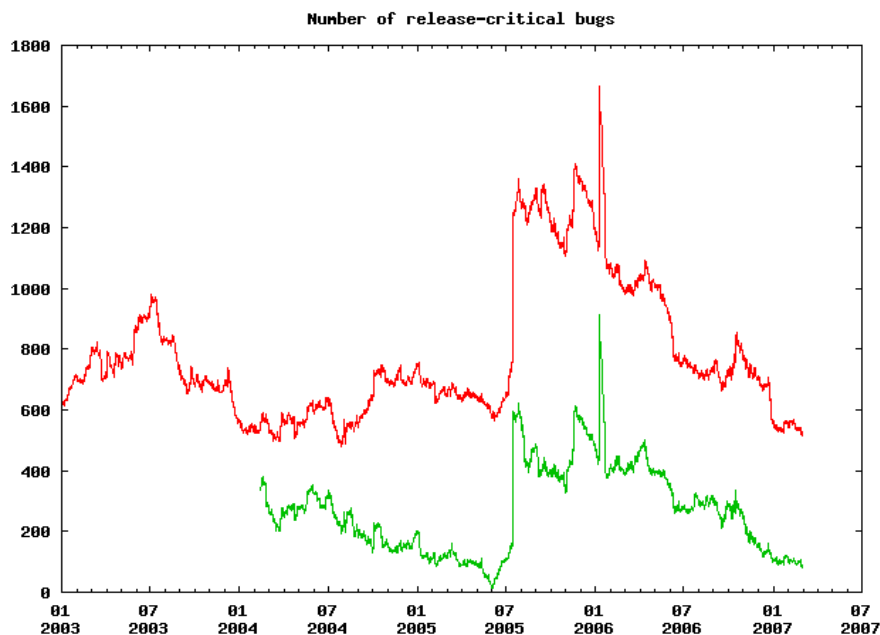


Figure 8.2.: Release critical bugs in Debian: the lower line shows those bugs which are relevant for the release (Source: Debian).

While bugs play a special role in the Debian project because release critical bugs are typically the biggest blocker for the release, bug tracking systems convey important information in other projects as well. For example, GNOME has a QA team known as the ‘bugsquad’³ whose task it is to review incoming bug reports (Villa 2003). They work together closely with the release managers because the number of incoming bug reports is an important indication of whether GNOME is ready to make another release. In particular, it is a clear sign that an issue has to be resolved before a release when it is reported several times by different users (i.e. when there are duplicate bug reports):

²<http://bugs.debian.org/release-critical/>

³<http://developer.gnome.org/projects/bugsquad/>

They report how often a bug has been duplicated. Either a bug is very rare or duplicated in the thousands, so you can see whether something is serious. It's quite a big deal to report a bug. Not everyone does it, but if thousands do it then you have quite a serious problem. (Murray Cumming, GNOME)

As such, bug tracking systems are an important part of a project's release process because they give a good overview of the status of a project.

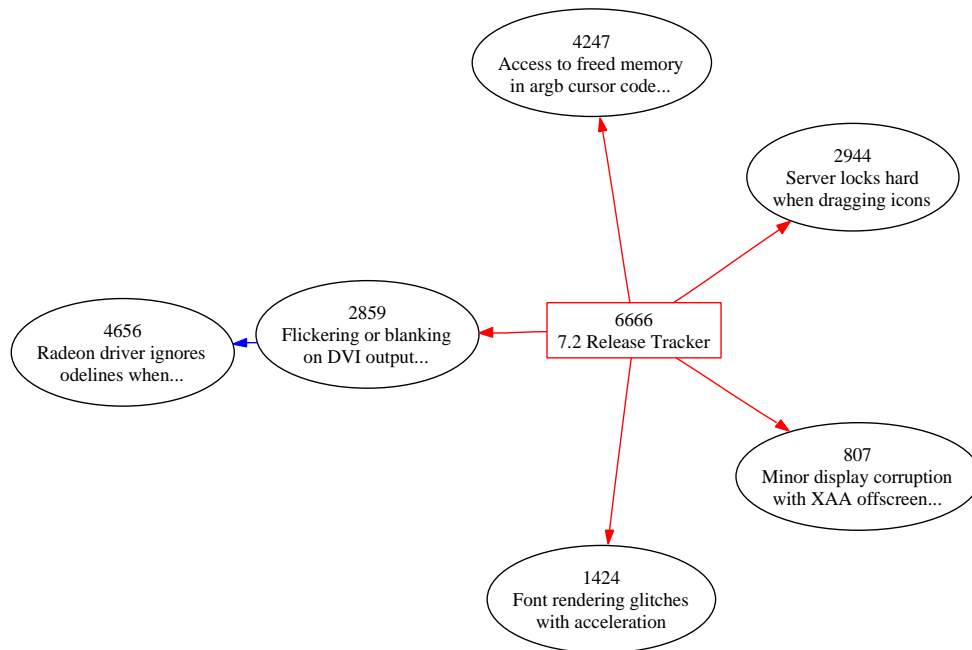


Figure 8.3.: An illustration of how a meta-bug is used to track release issues. The meta-bug, in the red box, depends on other bugs (red arrows) which need to be resolved first. Those bugs can themselves depend on other bugs (the blue arrow), creating additional, indirect dependencies for the meta-bug.

Some projects, such as X.org, have integrated their bug trackers with release planning even more tightly. Some bug tracking systems, such as Bugzilla, support dependencies between bugs. If one bug has to be resolved before another one can be tackled, this can be indicated through a dependency of the latter bug on the former. X.org uses this mechanism during release planning. The X.org release manager creates a meta-bug indicating a specific release (see table 8.2) and then creates dependencies to other bugs from this meta-bug (see figure 8.3). This mechanism easily allows the project to keep track of bugs that need to be resolved before a release. In fact, these bug reports do not necessarily have to be bugs in the software itself; they can also describe other blockers, such as problems with the infrastructure that need to be resolved

before a release can be made. In other words, bug tracking systems are often used instead of a dedicated planning system. The reason for this is two-fold. First, there are few good FOSS planning systems. Second, the bug tracking system is usually well integrated with a project's development process and hence using it for release planning is more natural than the deployment of other planning tools.

Release	Meta-Bug
6.7	213
6.8	351
6.9	1690
7.1	5041
7.2	6666

Table 8.2.: Meta-bugs used by the X.org project to track release issues.

8.3.2. Communication Channels

Since the majority of developers in FOSS projects are typically geographically dispersed, projects need solid communication infrastructure that allows participants to exchange information easily. This is important for development in general but particularly for release management when many different tasks and developers need to be coordinated.

Mailing Lists

Mailing lists are the main communication channel for FOSS projects. Messages are sent to a mailing list and then distributed to every single subscriber. Mailing lists are typically open to anyone interested in the project and they are usually archived. Mailing list archives allow developers and interested users to read past discussions to find out why specific decisions have been made and to learn how a particular project works. An advantage of mailing lists is that participants can read messages when they have time. Even if they cannot follow the project every day, which is often the case in volunteer projects, they can still stay informed about what is going on in the project. This is important during release preparations where everyone needs to stay informed about the current status of the release. Release managers use mailing lists,

and other communication channels, to stay in touch with other developers and to coordinate among the members of the release team:

We have a release mailing list, and a debian-release IRC channel. All members of the release team are supposed to read the full backlog of the release channel and are required to read all mails from the debian-release mailing list. (Andreas Barth, Debian)

The debian-release mailing list is the official contact address for Debian's release team. Everyone involved in Debian knows that they can reach the release team through this list, and the release team tries to respond to messages very quickly. This creates visibility and establishes trust (Iacono and Weisband 1997).

In addition to establishing the debian-release mailing list as a contact point for the release team, Debian, like other projects, use mailing lists to post updates regarding the release:

We send out monthly or bi-monthly mails to the debian-devel-announce mailing list which all developers are supposed to read, and actually just doing that gives the people trust in what you're doing. (Andreas Barth, Debian)

Sending information updates is an important part of release management because it shows developers where the project is and what needs to be resolved before a release can be made. This motivates developers and ensures that they actively take part in the release process. Another advantage of mailing lists is that a large number of people can easily be reached by sending a message to a mailing list. This allows effective communication during release coordination.

IRC

IRC stands for Internet Relay Chat and it is the most popular chat system based on written text used by FOSS projects. The big advantage of IRC in terms of release management is that it allows instant communication with members of the project who are currently online. Neither does every developer make use of IRC, nor are they online all the time given that developers live in different time zones. Nevertheless, IRC is a great tool for release managers and other developers to coordinate issues of high urgency with other project participants who are online at the same time. As the quote above mentioned,

the Debian project has a dedicated release IRC channel on which members of the release team can interact and where other members can ask questions regarding the release process (see table 8.3 for an example of IRC channels). This makes the release team more accessible to developers and ensures that release related issues can be handled quickly.

Channel	Network	Description
#debian-release	OFTC	Release discussions
#dev.openoffice.org	Freenode	Developer discussions
#qa.openoffice.org	Freenode	QA channel
#plone	Freenode	User and developer channel
#xorg-devel	Freenode	Developer discussions

Table 8.3.: An example of IRC channels used by the case study projects.

Conference Calls

A small number of projects, such as X.org, hold regular conference calls to discuss release planning. Only members of the release team and other important parties participate in these conference calls because it would be impossible to conduct a conference call with several hundred developers. The advantage of conference calls is that they allow higher bandwidth communication than written text (Rasters 2004). On the other hand, arranging conference calls is typically very difficult because members of the release team may be located in different time zones. Furthermore, since only few members of the project participate in conference calls, it is important to write summaries so developers can stay informed of the discussions that took place during a conference call.

Other Infrastructure

While mailing lists and IRC channels are the main coordination mechanisms employed by FOSS projects, a number of other infrastructures are gaining popularity. The following two will be discussed briefly: wikis and forums. Both of these technologies allow many people to exchange ideas quickly. Forums are similar to mailing lists, however they are not based on e-mail but on the world wide web. As such, they may allow more people to participate. A disadvantage is that users of forums typically need to manually check whether an update

has been posted whereas messages sent to a mailing list are distributed to each member automatically.

Wikis are a novel form of web sites in which readers can actively participate and make changes to the web pages they are reading. Wikipedia, a collaborative encyclopedia, is the most well known wiki. Thousands of people participate in the enhancement of Wikipedia by making improvements to pages they read. Wikis are also gaining popularity in FOSS projects, in particular during release planning, because it is very easy to make updates to existing information or to add new pages. As such, wikis are increasingly used as a coordination mechanism during release preparations.

8.3.3. Meetings in Real Life

While most FOSS development is carried out over the Internet, a number of projects are increasingly holding real life, face to face meetings, such as developer gatherings or conferences (Crowston et al. 2007). For example, Debian organizes an annual Debian Conference, GCC holds a GCC Summit, and GNOME organizes GUADEC, the GNOME Users' And Developers' European Conference. In addition to these key events, there are many smaller meetings and other conferences where developers meet, such as FOSDEM, the biggest developers' meeting in Europe which takes place in Brussels every February.

There are two reasons why such meetings can be beneficial in terms of release management. First, putting a number of developers into the same room can be highly effective in terms of communication. Meetings are a great venue to discuss long-term problems a project faces, but at the same time they are also particularly suited to perform release preparations. For example, putting core developers together in a room with the release team allows many issues to be resolved very quickly which would normally take considerable back and forth if the discussion would take place over the Internet, a much leaner communication media. Second, meetings make it possible for developers to get to know each other. This can have incredible long-term effects because familiarity among developers usually makes future communication and interaction easier:

You develop friendships. You can talk in a very different way to people once you know them IRL [in real life]. (Stefan H. Holek, Plone).

Meetings allow release managers to form friendships with other developers and establish trust. By meeting developers in real life, it is also possible to find out more about them, such as their preferences and skills. This information can be very useful during release management when specific tasks have to be performed and release managers are looking for volunteers. Another example where this information is useful is when a volunteer is busy. If a release manager is aware that a particular volunteer is, for example, a student currently preparing for finals, they can make sure that the duties and responsibilities of this person are temporarily performed by someone else.

In summary, real life meetings can be a very effective means of communication and they allow to establish trust and to get to know each other. While real life meetings can be very effective, it is important to take into consideration that not every member of the project can participate in such meetings, for example because they do not have the time to attend or live too far away. Hence, it is important to summarize discussions that have taken place at real life meetings, in particular when decisions have resulted from these discussions.

8.3.4. Summary

Even though time based release management decreases the amount of active coordination needed in a project, a successful implementation of time based releases relies on release management and coordination. It will never be possible to perform releases in which hundreds of developers are involved without any coordination whatsoever. Hence, it is important to create infrastructure that allows communication, coordination and collaboration. This section has shown the most important mechanisms that are used in FOSS projects to keep developers informed about the release process and to coordinate among developers.

8.4. Chapter Summary

This chapter discussed factors related to a successful implementation of the time based release strategy. Issues related to the implementation of change in volunteer projects, such as control, were investigated as well as specific

mechanisms to implement time based releases. This chapter argued that the introduction of schedules is not enough for a successful implementation of time based releases and that projects also need to institute policies and create coordination infrastructure.

9. Discussion and Conclusions

This concluding chapter of the dissertation is divided in the following four sections:

1. Discussion: the aim and findings of this research are discussed to show the main contribution to knowledge made in this dissertation.
2. Key learning points: this section summarizes the key learning points derived from this research.
3. Limitations and future research: the particular focus and scope of this work are discussed to show limitation of the present research. This is followed by a discussion of questions and areas of interest for future research that have been identified in this research.
4. Conclusions: the main findings of this dissertation and contributions to knowledge made in this research are summarized.

9.1. Discussion

This dissertation started with the observation that software produced by FOSS projects has seen substantial deployment in a number of areas. Since large corporations and others increasingly depend on FOSS, the question was raised as to how this reliance on FOSS can be reconciled with the variable, volatile and often chaotic nature of FOSS production. FOSS production has traditionally been perceived as unstructured and unorganized, and the majority of FOSS projects consist of voluntary contributors. As such, it is not clear how corporations can rely on the output of FOSS projects. The aim of this dissertation was to study aspects of quality improvement and find ways to ensure high

levels of quality in the output of FOSS projects. Based on exploratory studies, release management has been chosen as the specific focus of this research.

This research has subsequently identified the time based release strategy as a novel concept of release management worth investigating in-depth. In contrast to traditional software development which is feature-driven, the goal of time based release management is to produce high quality releases according to a specific release interval. This dissertation has shown that feature based release management in FOSS projects is often associated with lack of planning, which leads to problems, such as delays and low levels of quality. Since the FOSS projects under investigation in this dissertation mainly consist of volunteers, it is difficult to perform planning because there is no guarantee that features will be ready in time for the next release.

The key contribution to knowledge of this dissertation is a theoretical argument based on evidence from the case studies and theory of organizational complexity and coordination as to why the time based release strategy is more effective in complex projects than traditional, feature based releases. Time based releases are associated with two factors that act as important coordination mechanisms:

1. **Regularity:** the production of releases according to a specific interval allows projects to create regular reference points which show contributors what kind of changes other members of the project have made. Regularity also contributes to familiarity with the release process, and it leads to more disciplined processes.
2. **Schedules:** by using time rather than features as the orientation for a release, planning becomes possible in voluntary projects. Time based projects can create schedules which describes important deadlines and which contains dependency information between different work items and actors.

Together, these mechanism reduce the degree of active coordination required in a project. Developers can work on self-assigned work items independently and with the help of the schedule integrate them into the project in time for the release. As such, the time based release strategy is a means of dealing

with the complexity found in geographically distributed volunteer projects with hundreds of contributors.

In addition to explaining why time based releases are effective, this dissertation has identified a number of factors that are associated with a successful implementation of this novel release strategy. In particular, chapter 7 has identified factors related to the choice of an appropriate release interval and the creation of a schedule. Chapter 8 has discussed issues related to the implementation of change in volunteer projects and has presented important policies and infrastructure that need to be introduced for a successful implementation of time based releases.

Release management has been investigated in this dissertation in the light of quality improvement in FOSS projects. This dissertation has found considerable evidence that the introduction of time based releases leads to a more controlled and disciplined development and release process. There is further evidence that the publication of regular releases may be associated with increased levels of feedback provided by users which can be used to improve the quality of the software. Finally, the implementation of time based releases reduces problems found with other release strategies, such as significant delays and little testing before a release. As such, the time based release strategy can be considered as an important means of quality improvement in FOSS projects. In particular, the introduction of time based releases allows projects to reduce problems associated with the volatility of volunteer contributions. Time based releases can be seen as the ‘pulse’ of a project since they ensure that new features are not only developed but also delivered to users on a regular basis. This provides many advantages for organizations and other users which rely on the software produced by volunteer FOSS projects. Time based releases therefore contribute to a more sustainable development process. Nevertheless, it should be noted that a move to time based releases will not address all quality problems found in volunteer FOSS projects or eliminate the risk of failure. However, it is an important step towards a more planned and sustainable development process leading to high quality products.

9.2. Key Learning Points

This section summarizes the key learning points which can be derived from the research performed in this dissertation. The key learning points are as follows:

- The exploratory work in sections 2.3 and 3.2 about practices and problems related to quality and release management has contributed towards a more holistic view of the FOSS development process. While the majority of academic work published so far has focused on successful FOSS projects and positive aspects associated with this development paradigm, this dissertation has presented considerable evidence that there are major problems in FOSS projects, even in those which can be considered successes.
- Unlike previous research which has described FOSS development as unstructured and unorganized, the present work has identified a major shift in large and complex projects towards a more organized and planned approach using increasingly disciplined processes. This finding suggests that FOSS development is gaining maturity, possibly in part as a response to new requirements of users and large corporations which rely on the output of FOSS projects.
- The introduction of time based releases is an effective mechanism to establish better planning in projects with little control over voluntary contributors. The evidence found in this research suggests that time based releases are more appropriate in complex FOSS projects than feature-driven releases because the completion of features in volunteer projects is hard to predict and therefore makes planning difficult, if not entirely impossible.
- The time based release strategy is effective because it is associated with two factors that can be considered as coordination mechanisms: regularity and the use of schedules. Regularity establishes reference points, contributes to familiarity with the release process and leads to a more disciplined development process. Schedules are a means of describing dependency information between different work items and to set deadlines and milestones. As such, time based release management allows

contributors to perform their work with a high degree of independence, thereby reducing the amount of active coordination required.

- In addition to the introduction of a schedule, a successful implementation of time based releases requires policies and infrastructure that support this new release strategy. These policies establish more disciplined development processes whereas infrastructure allows contributors to stay informed about the status of release planning and to coordinate among each other. Both of these mechanisms contribute towards trust in the process, which is an important prerequisite in order to actually meet deadlines.
- Even though time based releases allow contributors to perform their work independently with little coordination, active coordination is a vital component in release management. Release managers have to actively ensure that work is completed and deadlines are followed, that contributors work with each other, and that members of the project have trust in the release process.

9.3. Limitations and Future Research

This dissertation has delivered interesting new insights into aspects of release management in FOSS projects, but there are limitations of the present research that stem from the particular research focus and scope that have been chosen. The factors that determine the scope of this work are presented before unanswered questions and other questions of interest for future research are discussed.

This dissertation has investigated:

- Complex projects: the exploratory work has shown that the key challenges of release management are found in large and complex projects rather than in small projects which mainly face resource limitations.
- Distributed projects: there is considerable evidence that projects in which participants are geographically dispersed are significantly harder

to manage than projects in which all members are co-located. This has an important influence on release management.

- Voluntary projects: there is further evidence that voluntary projects in which there is little control of participants are related to considerable challenges in terms of coordination and management.
- Release management: the overall topic of this dissertation is quality management and improvement in FOSS projects. As a particular aspect of this overall topic, release management has been chosen as the focus of this research in response to the findings of the exploratory work in section 2.3.
- Major new releases: projects produce a number of different releases, such as new development versions as well as minor updates and major releases for users. This study has found major releases for end-users to be associated with the most challenges and has therefore focused on this aspect of release management.
- Time based releases: based on the exploratory study about release management, the time based release strategy has been chosen as the focus of this research. While it is possible that there are other release strategies that deserve investigation, the exploratory study has shown considerable interest in time based releases.

In the following, interesting areas and questions for future research are presented based on the findings of this dissertation. They will be presented in three sections, one about release management, one about quality management, and finally one describing issues that go beyond the FOSS phenomenon.

Release Management

- This dissertation has taken a case study approach in order to answer why and how time based releases are effective. The research has found considerable evidence that time based releases act as a coordination mechanism that has positive effects on projects. These effects need further empirical investigation in order to answer questions that were raised during this research.

In particular, this dissertation found some evidence that time based releases lead to more motivation in, and involvement of, project participants. These effects need to be studied in more detail, for example through a survey of developer motivation and empirical studies of code contributions. The latter study could also shed light on the question raised in section 7.2.3 as to which periods projects should avoid when making releases. There is considerable evidence that Christmas should be avoided but it is presently not clear whether summer is associated with increased or decreased levels of contributions.

Such research could answer the following questions:

- Does the introduction of time based releases lead to more motivation among developers and users? What factors that are associated with release management have an impact on motivation?
 - Does the introduction of time based releases have an impact on the number of code contributions made by developers?
 - Are time based releases associated with higher levels of feedback and is this feedback of higher quality?
 - Which time periods should be avoided when making releases?
- This research has shown that large projects are often an aggregation of smaller ones and that those might produce their own releases which are then considered for the next roll-up release of the whole project. Since these individual releases contribute to the quality of the overall project release, more work needs to be conducted to study ways to improve release management in small projects.
 - Some projects, such as Debian and Linux, have raised the question of whether they should produce releases for end-users at all. The argument goes that volunteer projects should simply develop software and that commercial entities can then take this software, stabilize it and perform a release, and then provide support and other services. Debian was once compared to a supermarket from which other vendors can take components they are interested in and produce a system based on them.¹

¹<http://business.newsforge.com/business/06/05/22/1240231.shtml>

Findings from this dissertation suggest that the production of releases is associated with important benefits, such as feedback from users that can be used to improve the quality of the software. However, the question as to whether projects should make releases and whether there are conditions that influence this choice deserves to be investigated in more detail.

Quality Management

- This dissertation has shown that quality is an important topic and that there are problems in FOSS projects. The majority of FOSS literature has focused on positive aspects of this development paradigm but there is increasing awareness that FOSS development is not automatically associated with success. As such, more work on quality problems would be beneficial.
- The exploratory study in section 2.3 has revealed three areas of interest. In addition to release management, which was subsequently chosen as the focus of this dissertation, there are questions as to how paid people can contribute to quality and how leadership is associated with high quality output and success. These topics can be studied on their own as well as in relation to release management. For example, it would be interesting to investigate what tasks paid people could perform as part of release preparations to make it more likely that release targets are met. Similarly, this dissertation has found that asserting control in voluntary projects is difficult and studies on leadership might offer insights as to how management is performed in FOSS projects.

Beyond FOSS

- As discussed in this dissertation, the Internet has influenced release management because it allows cheap and fast delivery of software. There are some discussions that the nature of releases might change fundamentally with Web 2.0, a term for the second-generation of web services that emphasize online collaboration and sharing among users. There is a trend towards making software applications available through web

services, such as Google's recent move towards an online office suite.² This could fundamentally change release management because the application is stored on the web site rather than on users' PCs. This makes it possible to roll out major changes immediately because only a single computer needs to be upgraded. It is impossible to predict the impact this might have on release management in general and in FOSS projects, but it is an area of research that will gain importance in the near future.

- Traditional, proprietary software development is usually feature-driven. While time based releases are particularly beneficial in projects with little control over their contributors, it would be interesting to study whether proprietary software would gain advantages from a time based release strategy as well and what else they can learn from release management in FOSS projects.
- This research has identified the open question as to whether radical changes are possible when a time based release strategy is employed (see section 6.2.4). This issue is connected with the more fundamental question in software engineering as to when radical changes, such as a complete re-write, are advantageous to incremental improvements and code refactoring. Similar questions are studied in other areas, such as in research on innovation where a distinction between incremental and radical innovation is made (Bessant and Tidd 2007). Future research could map the insights from this research to software engineering in general and specifically to the open question raised in this research.
- The argument presented in this dissertation as to why time based releases work suggests that the idea of using of time rather than features for deadlines is beneficial in projects with little control over their contributors. As such, it would be interesting to study whether the ideas of time based releases can be applied to other volunteer groups whose aim is not software production.

²<http://docs.google.com/>

9.4. Conclusions

Many complex FOSS projects face challenges related to the coordination of volunteers during release preparations. Traditionally, release management is guided by the completion of certain features. However, this research has shown that this approach is associated with significant problems in volunteer projects which have little control over which features actually get developed. This dissertation has investigated an alternative, and relatively novel, approach to release management: the time based release strategy in which the target for making a release is a specific release date rather than the completion of features.

This dissertation has found evidence and developed an argument based on theory of organizational complexity and coordination as to why and how time based releases are effective in complex volunteer projects. This research has found that the introduction of time based releases is associated with two coordination mechanism that allow FOSS projects to better cope with great organizational complexity: regularity and the use of schedules. By using time rather than features as the criterion for a release, the time based release strategy allows better planning in projects which have little control over their contributors. The introduction of a time based release strategy therefore leads to a more planned and controlled software development process. This in turn contributes to more consistent levels of quality. Finally, it is argued that the findings about coordination and quality from this dissertation can be replicated and used in other volunteer projects whose aim is not the creation of software.

In summary, this dissertation has made a significant contribution to the academic understanding of the FOSS phenomenon and has raised interesting questions for the study of other volunteer projects. Moreover, this research has many practical implications from which FOSS projects can benefit. Therefore, it is the hope that the publication of the findings of this research will contribute to better release management practices and increase the quality of the output produced by FOSS projects.

Acknowledgements

This dissertation could not have been completed without the support and guidance of a number of individuals and organizations. I would therefore like to express my sincere gratitude to them.

My supervisor, David Probert, and advisor, Francis Hunt, for their guidance, support and encouragement from the beginning to the end of this research.

I would like to thank the following individuals for their extensive comments on earlier drafts of this dissertation: Jason Bucata, Andrea Capiluppi (University of Lincoln, UK), Murray Cumming (Openismus; GNOME), Matthijs den Besten (University of Oxford, UK), Giampaolo Garzarelli (University of the Witwatersrand, South Africa), James Howison (Syracuse University, USA), Matthew Jones (University of Cambridge, UK), Gregorio Robles (Universidad Rey Juan Carlos, Spain), and Louis Suarez-Potts (CollabNet; OpenOffice.org).

I would also like to express my thanks to everyone who participated in the interviews that were conducted as part of this research, and, more generally, to everyone who has contributed to free and open source software.

The following organizations have provided financial support and thereby allowed me to pursue this research: NUUG Foundation, Fotango, Intel, and, in particular, Google. This research has also been supported by a Doctoral Training Account from the Engineering and Physical Sciences Research Council (EPSRC).

Finally, I would like to thank my colleagues at the Institute for Manufacturing, in particular Simon Ford, for their support and friendship. I would like to express my deepest thanks to my parents who have supported me all these years. Daniela Pajek for providing me with the motivation to complete this dissertation and for her patience and encouragement during the writing up process.

Glossary

API: Application Programming Interface. The standard interface defined by an application or library on which other programs can build and rely.

Bazaar: a development model described by Eric S. Raymond who argued that open source projects follow a ‘bazaar model’ in which anyone can contribute to a project. The development style is very open and allows for users to influence the development.

Branch: a line of development that has diverged from the main development line; see tree.

Cathedral: a development model described by Eric S. Raymond who observed that traditional software development projects follow a ‘cathedral model’ in which the developers have tight control over the software. Most of the software is created in “splendid isolation” before test versions are made available to users.

CVS: the Concurrent Versions System, a version control system which allows multiple people to work on the same source code at the same time. It records by whom and when a specific change was made and handles conflicts between changes made by different people.

Distribution: The term ‘distribution’ (or ‘distro’) is used to refer to a Linux distribution, which is a complete operating system based on individual software packages.

Feature based release: a release strategy in which a release is made when a specific set of features or goals have been achieved.

Free software: a philosophical movement which argues that software should give its users specific freedoms, such as the rights to modify and share

the program and source code.

GNU: the project which started the free software movement. The GNU tools are a core element of every Linux system.

IRC: Internet Relay Chat allows people to talk to each other in real time via text. IRC supports private conversations between two parties or chat rooms (known as channels) in which multiple people can talk to each other.

Open source: a pragmatic movement which argues that the open source development model leads to better and cheaper software.

Source code: describes what a human programmer writes when creating software. Source code is later transformed into something suitable for machines (the binary). The reverse step from binary to source code is not easily possible.

Time based release: a release strategy in which the release follows a very clear schedule and features are postponed if they do not meet deadlines.

Tree: different development lines are often compared to a tree: the main development takes place on the trunk whereas branches are used to create a new development line from the trunk (also known as the 'main line'). Branches can be used to maintain stable releases or to develop new features outside of the main development line until they can be integrated ('merged') into the trunk.

Trunk: the main development line; see tree.

Volunteer: project participants which are involved on a voluntary basis and over which the project therefore has relatively little control.

Wiki: a web site where users can easily make changes to the text they are reading. This allows the collaborative creation and maintenance of web sites. The most popular Wiki based web site is Wikipedia, a collaborative encyclopedia.

A. Exploratory Studies

A.1. Quality

The aim of this exploratory study was to identify practices and problems of release management in volunteer FOSS projects. Semi-structured interviews were carried out in person at conferences and summarized during the interviews.

A.1.1. Questions

The following questions were asked as a basis for further exploration about quality aspects of the interviewees' projects:

- Are there any quality issues in your project? If so, what are they and how do you deal with them?
- What techniques can be applied in FOSS projects to ensure quality?
- Are there any unresolved quality issues in your project? If so, what are they and why are they unresolved?
- Which development model and life cycle do you follow?
- What kind of facilities do you have to ensure quality?
- How would you compare the quality of FOSS and proprietary software? When might the quality in one be higher than in the other?

A.1.2. Projects

Only seven projects were taken into consideration for this study but they reflect a number of different dimensions, such as size and the nature of a project.

- Apache
- Dasher
- Debian
- GNOME
- MirBSD
- Postfix
- VideoLAN

A.1.3. Consent

Developers who were invited to take part in the interviews were informed of the background of this study, their voluntary participation and the anonymous nature of this study.

A.2. Release Management

The aim of this exploratory study was to find out more about quality practices, tools and problems in FOSS projects. Semi-structured interviews with core developers and release managers of twenty projects were carried out at a conference and recorded for later analysis.

A.2.1. Questions

The interviews were quite detailed and started with a long range of questions but there was also time to explore other issues that came up during the interviews.

The standard questions were:

- What kind of release cycle does your project follow, both in terms of user and development releases? When and how do you decide to make a new release?

-
- What was the rationale of adopting this release cycle? Have other models been proposed or suggested, and why have they not been adopted instead?
 - What kind of goals did you take into account when choosing your release strategy? Were requirements of users and developers taken into account, and if so, how have they been gathered?
 - Do you think there is a big disparity of users' and developers' requirements regarding the software your project produces?
 - How often do you make a release? Do you think the release cycle is right, or are releases made too rarely or too often?
 - What advantages and disadvantages are associated with the release strategy your project employs? Are there any problems you're currently facing with your release strategy? Do you think there is any room for further improvement?
 - Have you looked at how release management is done in other FOSS projects in order to incorporate some of their ideas into your project?
 - What is the relationship between your release strategy and the final quality of the software? What kind of mechanisms are in place as part of the release strategy to assure high quality? In which ways does your release strategy lead to high quality, and in areas is your strategy not sufficient to assure high quality?
 - Who is in charge for the release? How are they appointed?
 - In your opinion, what kind of skills do the people in charge of the release need?
 - What practices and tools are being used in release management in your project?
 - Are deadlines and milestones defined? If so, who sets them and how? How does the project make sure they are met? Are they met? How much control do release managers have over other developers?

- How do you determine release critical issues and how do you determine when you are ready to make a release?
- What are unforeseen problems which have come up in the past and how have they been mitigated? Have any measures been put in place so that such unforeseen problems would not occur again? What are major obstacles and problems for getting releases done?
- What kind of tools would help you during release management? Is there anything else you'd like to see?

Below are some follow-up questions which were asked in several interviews:

- What kind of version number scheme do you use?
- Upgrading (do people stay with old versions? Is that a problem?)
- Is there a post-release review?
- Is there a check list?
- Is there a roadmap?
- Which changes are made to a released stable version?

A.2.2. Projects

Lead developers or release managers from the following twenty projects for interviewed:

- Alexandria
- Apache
- Arch
- Bazaar
- Debian
- GCC
- GNOME

- Libtool
- Lua
- Nano
- OpenOffice.org
- Postfix
- Squid
- Synaptic
- Template Toolkit
- Twisted
- Ubuntu
- Vserver
- X.org
- XFree68

A.2.3. Consent

Participants in this study were informed of the background of this study and their rights using the following consent form:

My research is about quality in FOSS projects. Very broadly speaking, the objective is to identify quality problems and to find ways to further improve quality in FOSS projects. In this interview, I specifically focus on release management. I'd like to identify current release processes employed by a variety of projects to get a better picture of the current state of release management and release cycles in FOSS projects.

Do I have permission to record this interview? The recording will not be published anywhere and it will only be used for my records so I can go back to see what we discussed exactly.

You agree to participate in this interview on a voluntary basis. You may refuse to answer any question you do not want to answer for whatever reason, and you are also free to terminate this interview at any point if you no longer feel comfortable participating.

The interview will be anonymous. This means that your name will not show up anywhere apart from my private records. However, I would like to give the name of your project.

Again, thanks for your participation. It is my hope that the results of these interviews will be published in a journal or conference. Once results become available, would you like to receive a copy of the paper?

B. Case Studies

These in-depth case studies were conducted to answer the main questions posed in this dissertation and to come to an understanding as to why and how some FOSS projects effectively employ a time based release strategy. Seven case studies were conducted and they took various information sources into account.

B.1. Interviews

Interviews with developers actively involved in the release process were conducted as well as with vendors which distribute the software produced by these project. For each of the case projects, typically three people were interviewed. When possible, the views of developers with a critical view of the release process were taken into consideration to obtain a complete picture.

B.1.1. Questions

The questions took a number of topics into account, including problems of release management, time plans, regular releases and incentives and motivation of developers. In addition to the questions below, specific questions depending on the project were asked based on a study of mailing list archives and other documents.

- What problems have there been with your release management and release cycle (with respect to a particular release and in general)?
- Why do these problems occur?
- What mechanisms have been implemented to prevent such problems from happening in the future or to deal with them better should they arise again.

- What effects do these problems have on your project (both the development and user community)?
- What kind of coordination activities are there in your project and what kind of requirements of coordination?
- Do time plans have an impact on the requirements for coordination?
- What, if any, are negative effects of the use of time plans?
- What, if any, problems are there with generating time plans?
- How should a time plan be generated (for example, development versus testing and stabilization phase)?
- How do you make sure deadline are actually followed? How do assure that you stick to your time plans?
- What benefits are there of regular releases?
- What upgrade costs are there? What is a good balance between regular releases and upgrade costs?
- What negative effects and costs are related to regular releases? What ways are there to keep them as low as possible?
- Why would developers be interested in doing releases?
- What advantages and disadvantages do time based releases provide for the developer and user community?

B.1.2. Projects and Participants

- Debian
 - Andreas Barth (release manager)
 - Colin Watson (release manager)
- GCC
 - Joe Buck (member of the GCC steering committee)

-
- Matthias Klose (Debian)
 - GNOME
 - Andrew Cowie (release manager for java-gnome)
 - Havoc Pennington (Red Hat; core developer of GNOME)
 - Jeff Waugh (release manager)
 - Murray Cumming (release manager)
 - Linux kernel
 - Adrian Bunk (maintainer of 2.6.16-stable series)
 - Andrew Morton (maintainer of the *-mm* series)
 - Christoph Hellwig (core developer)
 - OpenOffice.org
 - André Schnabel (member of the QA team)
 - Louis Suarez-Potts (community manager)
 - Michael Meeks (Novell)
 - Plone and Archetypes
 - Alec Mitchell (release manager, Plone)
 - Jens Klein (release manager, Archetypes)
 - Stefan H. Holek (release manager)
 - X.org
 - anonymous (release manager)
 - Kevin E. Martin (Red Hat; X.org release manager)
 - Stuart Anderson (Board of Directors, X.org Foundation)

B.1.3. Consent

Participants in this study were informed of the background of this study and their rights using the following consent form. Unlike in the exploratory studies, permission was obtained to list the name of each person taking part in the interview in addition to their project. This should increase the validity of the study since the interviewees were experts in their fields.

My research focuses on quality management in FOSS projects with a focus on release management. I am interested how release management is being performed in large projects. In particular, I'm interested in time based release management.

Do I have permission to record this interview? The recording will not be published anywhere and it will only be used for my records so I can go back to see what we discussed exactly.

Do I have permission to list your name as one of the people I have spoken to and maybe I cite you in verbatim if you have a very good quote?

Where individuals are quoted in verbatim in this dissertation, they were asked to read the section in which the quotation appears and to confirm that their opinion was represented correctly.

B.2. Mailing Lists

Archives of the following mailing lists were studied to find discussions about release management:

Project	Mailing List
Debian	debian-announce debian-devel debian-release
GCC	gcc gcc-patches
GNOME	gnome-1.4-list gnome-2.0-list gnome-devel-list gnome-hackers release-team
Linux	linux-kernel
OpenOffice.org	releases
X.org	release-wrangers

B.3. Conference Presentations

A number of conference presentations about the case study projects or release management in FOSS in general were attended during this research. In addition, some recordings of past presentations were obtained, for example from the following sites:

- <http://meetings-archive.debian.net/pub/debian-meetings>
- <http://mirror.linux.org.au/linux.conf.au/2007/video/>
- <http://www.fosdem.org/2007/media/video>

Below is an abbreviated list of presentations that were considered in this research:

- Jeff Waugh, *To The Teeth: Arming GNOME for Desktop Success*, Linux Conference Australia, January 16, 2004.
- Andreas Barth, *Debian Release Processes*, Debian Conference 5, July 9, 2005.

-
- Andreas Barth, *Debian Release Management*, Debian Conference 5, July 11, 2005.
 - Andreas Barth, *Debian's release process — behind the scenes*, Libre Software Meeting, June 19, 2006.
 - Michael Bemmer, *OpenOffice.org 2.x and beyond*, OpenOffice.org Conference 2006, September 12, 2006.
 - Keith Packard, *What's up, X?*, Linux Conference Australia, January 15, 2007.
 - Andrew Morton, *Trends in Linux Kernel Development*, FOSDEM, February 24, 2007.

Bibliography

- Aberdour, M. (2007). Achieving quality in open source software. *IEEE Software* 24(1), 58–64.
- Amor, J. J., J. M. Gonzalez-Barahona, G. Robles, and I. Herraiz (2005). Measuring libre software using Debian 3.1 (Sarge) as a case study. *Upgrade Magazine*.
- Baetjer, H. (1997). *Software as Capital*. Los Alamitos, CA: IEEE Computer Society Press.
- Baldwin, C. Y. and K. B. Clark (1997). Managing in an age of modularity. *Harvard Business Review* 75(5), 84–93.
- Benkler, Y. (2002). Coase’s penguin, or, Linux and the nature of the firm. *Yale Law Journal* 112(3), 369–446.
- Bergquist, M. and J. Ljungberg (2001). The power of gifts: Organising social relationships in open source communities. *Information Systems Journal* 11(4), 305–320.
- Bessant, J. and J. Tidd (2007). *Innovation and Entrepreneurship*. West Sussex, UK: Wiley.
- Bezroukov, N. (1999, October). Open source software development as a special type of academic research (critique of vulgar Raymondism). *First Monday* 4(10).
- Boehm, B., J. Brown, J. Kaspar, M. Lipow, M. G, and M. M (1978). *Characteristics of Software Quality*. Amsterdam: North-Holland.
- Brooks, F. P. (2000). *The Mythical Man-Month: Essays on Software Engineering* (2nd ed.). Addison-Wesley Publishing Company.
- Christensen, C. M. (1997). *The Innovator’s Dilemma: When New Technologies Cause Great Firms to Fail*. Boston, Massachusetts: Harvard

- Business School Press.
- Coffin, J. (2006). Analysis of open source principles in diverse collaborative communities. *First Monday* 11(6).
- Coleman, M. and T. Manns (1996). *Software Quality Assurance*. London: Macmillan.
- Conway, M. E. (1968). How do committees invent? *Datamation* 14(4), 28–31.
- Crosby, P. B. (1979). *Quality Is Free: The Art of Making Quality Certain*. New York, NY: McGraw-Hill.
- Crowston, K. (1997). A coordination theory approach to organizational process design. *Organization Science* 8(2), 157–175.
- Crowston, K. and J. Howison (2005). The social structure of free and open source software development. *First Monday* 10(2).
- Crowston, K., J. Howison, and H. Annabi (2006). Information systems success in free and open source software development: Theory and measures. *Software Process: Improvement and Practice* 11(2), 123–148.
- Crowston, K., J. Howison, U. Y. Eseryel, and C. Masango (2007). The role of face-to-face meetings in technology-supported self-organizing distributed teams. *IEEE Transactions on Professional Communications*. In press.
- Crowston, K., K. Wei, Q. Li, U. Y. Eseryel, and J. Howison (2005). Coordination of free/libre open source software development. In *Proceedings of the International Conference on Information Systems (ICIS 2005)*, Las Vegas, NV, USA.
- Dale, B. and H. Bunney (1999). *Total Quality Management Blueprint*. Oxford, UK: Blackwell Business.
- Dalle, J.-M., L. Daudet, and M. den Besten (2006). Mining CVS signals. In J. Gonzalez-Barahona, M. Conklin, and G. Robles (Eds.), *Workshop on Public Data about Software Development*, Como, Italy, pp. 10–19.
- Dalle, J.-M. and P. A. David (2005). The allocation of software development resources in open source production mode. In J. Feller, B. Fitzgerald,

- S. A. Hissam, and K. R. Lakhani (Eds.), *Perspectives on Free and Open Source Software*, pp. 297–328. Cambridge, MA: MIT Press.
- Daniel, S. L. and K. J. Stewart (2005). Cohesion, structure and software quality. In *Proceedings of Eleventh Americas Conference on Information Systems*, Omaha, NE, pp. 973–979.
- de Groot, A., S. Kugler, P. Adams, and G. Gousios (2006). Call for quality: Open source software quality observation. In E. Damiani, B. Fitzgerald, W. Scacchi, M. Scotto, and G. Succi (Eds.), *Open Source Systems*, Volume 203, pp. 57–62. IFIP Working Group 2.13: Springer.
- den Besten, M., J.-M. Dalle, and F. Galia (2006). Collaborative maintenance in large open-source projects. In E. Damiani, B. Fitzgerald, W. Scacchi, M. Scotto, and G. Succi (Eds.), *Open Source Systems*, Volume 203, pp. 233–244. IFIP Working Group 2.13: Springer.
- Eisenhardt, K. M. (1989). Building theories from case study research. *Academy of Management Review* 14(4), 532–550.
- Erenkrantz, J. R. (2003). Release management within open source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, Portland, OR, USA, pp. 51–55. ICSE.
- Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15(3), 182–211.
- Feller, J., B. Fitzgerald, S. A. Hissam, and K. R. Lakhani (2005). *Perspectives on Free and Open Source Software*. Cambridge, MA: MIT Press.
- Fischer, M. and H. Gall (2004). Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution: Research and Practice* 16, 385–403.
- Fischer, M., M. Pinzger, and H. Gall (2003). Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, Amsterdam, The Netherlands, pp. 23–32.
- Fitzgerald, B. (2006). The transformation of open source software. *MIS Quarterly* 30(3), 587–598.

- Fogel, K. F. (2005). *Producing Open Source Software*. Sebastopol, CA: O'Reilly & Associates.
- Gacek, C. and B. Arief (2004). The many meanings of open source. *IEEE Software* 21(1), 34–40.
- Garzarelli, G. (2003). Open source software and the economics of organization. In J. Birner and P. Garrouste (Eds.), *Markets, Information and Communication: Austrian Perspectives on the Internet Economy*, pp. 47–62. London: Routledge.
- Garzarelli, G. and R. Galoppini (2003, November). Capability coordination in modular organization: Voluntary FS/OSS production and the case of Debian GNU/Linux.
- German, D. (2004). The GNOME project: a case study of open source, global software development. *Journal of Software Process: Improvement and Practice* 8(4), 201–215.
- German, D. M. (2005). Software engineering practices in the gnome project. In J. Feller, B. Fitzgerald, S. A. Hissam, and K. R. Lakhani (Eds.), *Perspectives on Free and Open Source Software*, pp. 211–225. Cambridge, MA: MIT Press.
- Ghosh, R. A. (2006). Economic impact of open source software on innovation and the competitiveness of the information and communication technologies (ICT) sector in the EU. Technical report, Maastricht Economic and Social Research and Training Centre on Innovation and Technology, United Nations University, The Netherlands.
- Gill, J. and P. Johnson (1997). *Research methods for managers*. London: Paul Chapman Publishing.
- Gittens, M., H. Lutfiyya, M. Bauer, D. Godwin, Y. W. Kim, and P. Gupta (2002). An empirical evaluation of system and regression testing. In *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, pp. 3. IBM Press.
- Godfrey, M. W. and Q. Tu (2000). Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, San Jose, CA, pp. 131–142.

- Golden, B. (2005). *Succeeding with Open Source*. Boston, MA: Addison-Wesley.
- González-Barahona, J. M., M. A. Ortuño Pérez, P. de las Heras Quirós, J. Centeno González, and V. Matellán Olivera (2001, December). Counting potatoes: the size of Debian 2.2. *Upgrade II*(6), 60–66.
- González-Barahona, J. M., G. Robles, M. Ortuño Pérez, L. Rodero-Merino, J. Centeno González, V. Matellan-Olivera, E. Castro-Barbero, and P. de las Heras-Quirós (2004). Analyzing the anatomy of GNU/Linux distributions: methodology and case studies (Red Hat and Debian). In S. Koch (Ed.), *Free/Open Source Software Development*, pp. 27–58. Hershey, PA, USA: Idea Group Publishing.
- Greer, D. and G. Ruhe (2004). Software release planning: An evolutionary and iterative approach. *Information and Software Technology* 46, 243–253.
- Halloran, T. J. and W. L. Scherlis (2002). High quality and open source software practices. In *Proceedings of the 2nd Workshop on Open Source Software Engineering*, Orlando, FL, USA. ICSE.
- Hamel, J. (1993). *Case Study Methods*. Newbury Park, CA, USA: Sage Publications.
- Hardy, J.-L. and M. Bourgois (2006). Exploring the potential of OSS in air traffic management. In E. Damiani, B. Fitzgerald, W. Scacchi, M. Scotto, and G. Succi (Eds.), *Open Source Systems*, Volume 203, pp. 173–179. IFIP Working Group 2.13: Springer.
- Herbsleb, J. D. and R. E. Grinter (1999). Splitting the organization and integrating the code: Conway’s law revisited. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, pp. 85–95. ICSE.
- Herbsleb, J. D. and A. Mockus (2003). Formulation and preliminary test of an empirical theory of coordination in software engineering. In *Proceedings of the 9th European software engineering conference*, Helsinki, Finland, pp. 138–147.
- Hertel, G., S. Niedner, and S. Herrmann (2003). Motivation of software

- developers in open source projects: an Internet-based survey of contributors to the Linux kernel. *Research Policy* 32(7), 1159–1177.
- Houthakker, H. S. (1956). Economics and biology: Specialization and speciation. *Kyklos* 9, 181–189.
- Howison, J. (2005). Unreliable collaborators: Coordination in distributed volunteer teams. In M. Scotto and G. Succi (Eds.), *Proceedings of the First International Conference on Open Source Systems*, Genova, Italy, pp. 305–306.
- Howison, J. and K. Crowston (2004). The perils and pitfalls of mining SourceForge. In *Proceedings of the International Workshop on Mining Software Repositories (MSR 2004)*, Edinburgh, UK, pp. 7–11.
- Iacono, C. S. and S. Weisband (1997). Developing trust in virtual teams. In *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, Volume 2, Las Vegas, NV, USA, pp. 412–420.
- Iannacci, F. (2003). The Linux managing model. *First Monday* 8(12).
- Iannacci, F. (2005). Coordination processes in open source software development: The Linux case study. *Emergence: Complexity and Organization* 7(2), 21–31.
- ISO (1991). *International Standard ISO/IEC 9126. Information technology – Software product evaluation – Quality characteristics and guidelines for their use*. Geneva: International Electrotechnical Commission.
- Jensen, C. and W. Scacchi (2004). Collaboration, leadership, control, and conflict negotiation in the netbeans.org community. In *Proceedings of the 4th Workshop on Open Source Software Engineering*, pp. 48–52. ICSE.
- Johnson, K. (2001). A descriptive process model for open-source software development. Master’s thesis, Department of Computer Science, University of Calgary.
- Jørgensen, N. (2001). Putting it all in the trunk: Incremental software engineering in the FreeBSD open source project. *Information Systems Journal* 11(4), 321–336.
- Kelty, C. (2005). Free science. In J. Feller, B. Fitzgerald, S. A. Hissam, and

- K. R. Lakhani (Eds.), *Perspectives on Free and Open Source Software*, pp. 415–430. Cambridge, MA: MIT Press.
- Koch, S. and G. Schneider (2002). Effort, cooperation and coordination in an open source software project: GNOME. *Information Systems Journal* 12(1), 27–42.
- Krishnamurthy, S. (2002). Cave or community?: An empirical examination of 100 mature open source projects. *First Monday* 7(6).
- Krishnamurthy, S. (2005). About closed-door free, libre and open source (FLOSS) projects: Lessons from the mozilla firefox developer recruitment approach. *Upgrade* 6(3), 28–32.
- Krishnan, M. S. (1994). Software release management: a business perspective. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, Toronto, Canada, pp. 36.
- Kshetri, N. (2004). Economics of Linux adoption in developing countries. *IEEE Software* 21(1), 74–81.
- Lakhani, K. R. and E. von Hippel (2003). How open source software works: “free” user-to-user assistance. *Research Policy* 32(6), 923–943.
- Langlois, R. N. (2002). Modularity in technology and organization. *Journal of Economic Behavior & Organization* 49, 19–37.
- Lee, A. S. and R. L. Baskerville (2003). Generalizing generalizability in information systems research. *Information Systems Research* 14(3), 221–243.
- Lehman, M. M. (1980). On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software* 1, 213–221.
- Lerner, J. and J. Tirole (2002). Some simple economics of open source. *Journal of Industrial Economics* 50(2), 197–234.
- Levin, K. D. and O. Yadid (1990). Optimal release time of improved versions of software packages. *Information and Software Technology* 32(1), 65–70.
- Levy, S. (1984). *Hackers: Heroes of the Computer Revolution*. London: Penguin.

- Ljungberg, J. (2000). Open source movements as a model for organizing. *European Journal of Information Systems* 9(4), 208–216.
- MacCormack, A., R. Verganti, and M. Iansiti (2001, January). Developing product on ‘internet time’: The anatomy of a flexible development process. *Management Science* 47(1), 133–150.
- Malone, T. W. and K. Crowston (1994). The interdisciplinary study of coordination. *ACM Computing Surveys* 26(1), 87–119.
- Malone, T. W., K. Crowston, J. Lee, and B. Pentland (1993). Tools for inventing organizations: Toward a handbook of organizational processes. In *Proceedings of the Second IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pp. 72–82.
- Massey, B. (2003). Why OSS folks think SE folks are clue-impaired. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, Portland, OR, USA, pp. 91–97. ICSE.
- McConnell, S. (1999). Open-source methodology: Ready for prime time? *IEEE Software* 16(4), 6–8.
- Michlmayr, M. (2004). Managing volunteer activity in free software projects. In *Proceedings of the 2004 USENIX Annual Technical Conference, FREENIX Track*, Boston, MA, USA, pp. 93–102.
- Michlmayr, M. (2005). Software process maturity and the success of free software projects. In K. Zieliński and T. Szmuc (Eds.), *Software Engineering: Evolution and Emerging Technologies*, Kraków, Poland, pp. 3–14. IOS Press.
- Michlmayr, M. and B. M. Hill (2003). Quality and the reliance on individuals in free software projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, Portland, OR, USA, pp. 105–109. ICSE.
- Michlmayr, M., F. Hunt, and D. Probert (2005). Quality practices and problems in free software projects. In M. Scotto and G. Succi (Eds.), *Proceedings of the First International Conference on Open Source Systems*, Genova, Italy, pp. 24–28.
- Michlmayr, M. and A. Senyard (2006). A statistical analysis of defects in Debian and strategies for improving quality in free software projects. In

- J. Bitzer and P. J. H. Schröder (Eds.), *The Economics of Open Source Software Development*, Amsterdam, The Netherlands, pp. 131–148. Elsevier.
- Miles, M. B. and A. M. Huberman (1994). *Qualitative Data Analysis*. Thousand Oaks, CA, USA: Sage Publications.
- Mockus, A., R. T. Fielding, and J. D. Herbsleb (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology* 11(3), 309–346.
- Moody, G. (1997, 4 August). The greatest OS that (n)ever was. *Wired*.
- Moon, Y. J. and L. Sproull (2000). Essence of distributed work: The case of the Linux kernel. *First Monday* 5(11).
- Narduzzo, A. and A. Rossi (2004). The role of modularity in free/open source software development. In S. Koch (Ed.), *Free/Open Source Software Development*, pp. 84–102. Hershey, PA, USA: Idea Group Publishing.
- Nichols, D. M. and M. B. Twidale (2003). The usability of open source software. *First Monday* 8(1).
- O’Mahony, S. (2003). Guarding the commons: how community managed software projects to protect their work. *Research Policy* (32).
- O’Sullivan, M. (2002). Making copyright ambidextrous: An expose of copy-left. *The Journal of Information, Law and Technology* (3).
- Parnas, D. L. (1972). On the criteria for decomposing systems into modules. *Communications of the ACM* 15(12), 1053–1058.
- Paulson, J. W., G. Succi, and A. Eberlein (2004). An empirical study of open-source and closed-source software products. *Transactions on Software Engineering* 30(4), 246–256.
- Perens, B. (1999). The open source definition. In C. DiBona, S. Ockman, and M. Stone (Eds.), *Open Sources: Voices from the Open Source Revolution*. Sebastopol, CA: O’Reilly & Associates.
- Rasters, G. (2004). *Communication and Collaboration in Virtual Teams*. The Netherlands: Ipskamp.

- Raymond, E. S. (1999). *The Cathedral and the Bazaar*. Sebastopol, CA: O'Reilly & Associates.
- Raymond, E. S. (2003). *The Art Of Unix Programming*. Addison-Wesley.
- Redmill, F. (1997). *Software Projects: Evolutionary vs. Big-Bang Delivery*. West Sussex, UK: Wiley.
- Richardson, G. B. (1972). The organization of industry. *Economic Journal* 82(327), 883–896.
- Robles, G., J. M. Gonzalez-Barahona, and M. Michlmayr (2005). Evolution of volunteer participation in libre software projects: Evidence from Debian. In M. Scotto and G. Succi (Eds.), *Proceedings of the First International Conference on Open Source Systems*, Genova, Italy, pp. 100–107.
- Robles, G., J. M. Gonzalez-Barahona, M. Michlmayr, and J. J. Amor (2006). Mining large software compilations over time: Another perspective of software evolution. In *Proceedings of the International Workshop on Mining Software Repositories (MSR 2006)*, Shanghai, China, pp. 3–9.
- Saliu, O. and G. Ruhe (2005). Software release planning for evolving systems. *Innovations in Systems and Software Engineering* 1(2), 189–204.
- Schach, S. R., B. Jin, D. R. Wright, G. Z. Heller, and A. J. Offutt (2002). Maintainability of the Linux kernel. *IEEE Proceedings - Software* 149(1), 18–23.
- Schmidt, D. C. and A. Porter (2001). Leveraging open-source communities to improve the quality & performance of open-source software. In *Proceedings of the 1st Workshop on Open Source Software Engineering*, Toronto, Canada. ICSE.
- Senyard, A. and M. Michlmayr (2004). How to have a successful free software project. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, Busan, Korea, pp. 84–91. IEEE Computer Society.
- Sharma, S., V. Sugumaran, and B. Rajagopalan (2002). A framework for creating hybrid-open source software communities. *Information Systems Journal* 12(1), 7–25.
- Simon, H. A. (1962). The architecture of complexity. In *Proceedings of the American Philosophical Society*, Volume 106, pp. 467–482.

- Slack, N., S. Chambers, C. Harland, A. Harrison, and R. Johnston (1995). *Operations Management*. London: Pitman Publishing.
- Sommerville, I. (2001). *Software Engineering*. Essex, England: Pearson Education Limited.
- Sowe, S., I. Stamelos, and L. Angelis (2006). Identifying knowledge brokers that yield software engineering knowledge in OSS projects. *Information and Software Technology* 48(11), 1025–1033.
- Sowe, S. K., I. Stamelos, L. Angelis, and I. Antoniadis (2007). Understanding knowledge sharing activities in free/open source software projects. *Journal of Systems and Software*. In press.
- Stalder, F. and J. Hirsh (2002). Open source intelligence. *First Monday* 7(6).
- Stallman, R. M. (1999). The GNU operating system and the free software movement. In C. DiBona, S. Ockman, and M. Stone (Eds.), *Open Sources: Voices from the Open Source Revolution*. Sebastopol, CA: O'Reilly & Associates.
- Stallman, R. M., L. Lessig, and J. Gay (2002). *Free Software, Free Society*. Cambridge, MA: Free Software Foundation.
- Stamelos, I., L. Angelis, A. Oikonomou, and G. L. Bleris (2002). Code quality analysis in open-source software development. *Information Systems Journal* 12(1), 43–60.
- Tan, Y. and V. S. Mookerjee (2005). Comparing uniform and flexible policies for software maintenance and replacement. *IEEE Transactions on Software Engineering* 31(3), 238–255.
- Tawileh, A., O. Rana, W. Ivins, and S. McIntosh (2006). Managing quality in the free and open source software community. In *Proceedings of the 12th Americas Conference on Information Systems*, Acapulco, Mexico.
- Thomsen, S. R., J. D. Straubhaar, and D. M. Bolyard (1998). Ethnomethodology and the study of online communities: exploring the cyber streets. *Information Research* 4(1).
- Torvalds, L. and D. Diamond (2001). *Just for Fun*. London, UK: Texere.

- van der Hoek, A., R. S. Hall, D. Heimbigner, and A. L. Wolf (1997). Software release management. In *Proceedings of the Sixth European Software Engineering Conference*, pp. 159–175.
- Villa, L. (2003). Large free software projects and Bugzilla. In *Proceedings of the Linux Symposium*, Ottawa, Canada.
- Vixie, P. (1999). Software engineering. In C. DiBona, S. Ockman, and M. Stone (Eds.), *Open Sources: Voices from the Open Source Revolution*, pp. 91–100. Sebastopol, CA: O'Reilly & Associates.
- von Hippel, E. (2005). Open source software projects as user innovation networks. In J. Feller, B. Fitzgerald, S. A. Hissam, and K. R. Lakhani (Eds.), *Perspectives on Free and Open Source Software*, pp. 267–278. Cambridge, MA: MIT Press.
- von Krogh, G., S. Spaeth, and K. R. Lakhani (2003). Community, joining, and specialization in open source software innovation: a case study. *Research Policy* 32(7), 1217–1241.
- von Krogh, G. and E. von Hippel (2003). Editorial: Special issue on open source software development. *Research Policy* 32(7), 1149–1157.
- Walsham, G. (1995). Interpretive case studies in is research: nature and method. *European Journal of Information Systems* 4(2), 74–81.
- Weber, S. (2005). *The Success of Open Source*. Cambridge, MA: Harvard University Press.
- Williams, S. (2002). *Free as in Freedom: Richard Stallman's Crusade for Free Software*. Sebastopol, CA: O'Reilly & Associates.
- Yamauchi, Y., M. Yokozawa, T. Shinohara, and T. Ishida (2000). Collaboration with lean media: how open-source software succeeds. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, Philadelphia, PA, pp. 329–338.
- Yin, R. K. (1994). *Case Study Research: Design and Methods*. Thousand Oaks, CA, USA: Sage Publications.
- Zahran, S. (1997). *Software Process Improvement*. London: Addison-Wesley.

Zhao, L. and S. Elbaum (2000). A survey on quality related activities in open source. *Software Engineering Notes* 25(3), 54–57.

Zhao, L. and S. Elbaum (2003). Quality assurance under the open source development model. *The Journal of Systems and Software* 66, 65–75.

\$Id: text.tex 6864 2007-06-17 19:33:29Z tbm \$