Understanding Open Source Software Peer Review: Review Processes, Parameters and Statistical Models, and Underlying Behaviours and Mechanisms

by

Peter C. Rigby
BASc. Software Engineering, University of Ottawa, 2004

A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

University of Victoria

Understanding Open Source Software Peer Review:
Review Processes, Parameters and Statistical Models, and
Underlying Behaviours and Mechanisms

by

Peter C. Rigby
BASc. Software Engineering, University of Ottawa, 2004

Supervisory Committee

Dr. Daniel M. German, Co-supervisor
(Department of Computer Science)

Dr. Margaret-Anne Storey, Co-supervisor
(Department of Computer Science)

Dr. Laura Cowen, Outside Member
(Department of Statistics)

Supervisory Committee

Dr. Daniel M. German, Co-supervisor
(Department of Computer Science)

Dr. Margaret-Anne Storey, Co-supervisor
(Department of Computer Science)

Dr. Laura Cowen, Outside Member
(Department of Statistics)

ABSTRACT

Peer review is seen as an important quality assurance mechanism in both industrial development and the open source software (OSS) community. The techniques for performing inspections have been well studied in industry; in OSS development, peer review practices are less well understood. In contrast to industry, where reviews are typically assigned to specific individuals, in OSS, changes are broadcast to hundreds of potentially interested stakeholders. What is surprising is that this approach works very well, despite concerns that reviews may be ignored, or that discussions will deadlock because too many uninformed stakeholders are involved.

In this work we use a multi-case study methodology to develop a theory of OSS peer review. There are three research stages. In the first stage, we examine the policies of 25 OSS projects to understand the review processes used on successful OSS projects. We also select six projects for further analysis: Apache, Subversion, Linux, FreeBSD, KDE, and Gnome. In the second stage, using archival records from the six projects, we construct a series of metrics that produces measures similar to those used in traditional inspection experiments. We measure the frequency of review, the size and complexity of the contribution under review, the level of participation during review, the experience and expertise of the individuals involved in the review, the review interval, and number of issues discussed during review. We create statistical models of the review efficiency, review interval, and effectiveness, the issues discussed during review, to determine which measures have the largest impact on review efficacy. In the third stage, we use grounded theory to analyze 500 instances of peer review and interview ten core developers across the six projects. This approach allows us to understand why developers decide to perform reviews, what happens when reviews are ignored, how developers interact during a review, what happens when too many stakeholders are involved during review, and the effect of project size on the review techniques. Our findings provide insights into the simple, community-wide mechanisms and behaviours that developers use to effectively manage large quantities of reviews and other development discussions.

The primary contribution of this work is a theory of OSS peer review. We find that OSS reviews can be described as (1) early, frequent reviews (2) of small, independent, complete contributions (3) that, despite being asynchronously broadcast to a large group of stakeholders, are reviewed by a small group of self-selected experts (4) resulting in an efficient and effective peer review technique.

Table of Contents

List of Tables

List of Figures

Acknowledgement

I would first and foremost like to thank my advisers Peggy and Daniel for the wisdom, effort, and patience that they invested in transforming me from a forcefully independent undergraduate student into a balanced member of the software engineering research community. As an apprentice, I cannot imagine better mentors. I also am deeply indebted to the members of their research labs and greatly enjoyed being surrounded by such lively and intellectual people.

I am very grateful to Laura. Not only did she provide statistical advice, but she took a serious interest in my work and provided many insightful comments regarding the software engineering processes I describe.

I felt very privileged to have Tom as an external examiner, as I very much admire his work.

I appreciate the time I spent at UC Davis. Prem was a funny and insightful mentor and Chris was a great friend and collaborator. Also Chris' name aliasing tool was very helpful in my work.

I would like to acknowledge the Canadian government for providing me with an NSERC CGSD scholarship.

I would like to thank the open source software community for opening my eyes to original ways of sharing and working. The openness of this community made it possible for me to mine archival records on each project. The meritocratic nature of this community is an example that could be of great benefit to other communities around the world. I am also indebted to Justin and the developers who kindly took the time to answer my interview questions.

My family and friends provided a community in which I felt truly supported. I also very much appreciated the debates I had with them; Vinay is still convinced that I have actually done a Ph.D. in sociology.

This Ph.D. would not have been possible without the love and support of my mother and my father's infinite capacity to listen, provide wise council, reassure me, and keep me going.

Finally, I would like to thank Gargi whose understanding and support allowed me to overcome the hardest part of a Ph.D. – finishing.

Dedication

To my parents and my love.

Chapter 1

Introduction

For 35 years, software inspections (*i.e.* formal peer reviews) have been perceived as a valuable method to improve the quality of a software project. Inspections typically involve periodic group reviews where developers are expected to study the artifact under review before gathering to discuss it [38].

In practice, industrial adoption of software inspection remains low, as developers and organizations complain about the time commitment and corresponding cost required for inspection, as well as the difficulty involved in scheduling inspection meetings [71]. These problems are compounded by tight schedules in which it is easy to ignore peer review.

Given the difficulties with adoption of inspection techniques in industry, it is surprising that most large, successful projects within the Open Source Software (OSS) community have embraced peer review as one of their most important quality control techniques. Despite this adoption of peer review, there are very few empirical studies examining the peer review techniques used by OSS projects. There are experience reports [86, 114], descriptions at the process and policy level [34, 95], and empirical studies that assess the level of participation in peer reviews [3, 83]. There has also been some recent work that examines OSS review when conducted on bugtracking tools, such as Bugzilla [21, 69, 104]. Tracker based review is becoming increasingly popular on projects with a large number of non-technical users (*e.g.,* the KDE and Gnome projects). While the projects we examine all use bug trackers, they continue to conduct large numbers of reviews over broadcast email.

This dissertation is focused on email based OSS peer review, which we will refer to as OSS review in the remainder of this work. The following gives a simplified overview of this style of peer review. A review begins with an author creating a patch (a software change). The author can be anyone from an experienced core developer to novice programmer who has fixed a trivial bug. The author's patch, which is broadcast on the project's mailing list, reaches a large community of potentially interested individuals. The patch can be ignored, or it can be reviewed with feedback sent to the author and also broadcast to the project's community. The author, reviewers, and potentially other stakeholders (*e.g.,* non-technical users) discuss and revise the patch until it is ultimately accepted or rejected.

The objectives of this work are fourfold (See Figure 1.1). First, we want to understand the different review processes used in OSS. Second, many of the questions traditionally asked about inspection techniques, such as the length of the review interval or the number of defects found during review [110], have not been answered for OSS. We want to quantify and model these and other parameters of peer review in OSS. Third, we want to develop a broad understanding of the mechanisms and behaviours that underlie broadcast based peer review. Fourth, we want to develop a theory of OSS peer review that encapsulates our findings.

1.1 Research Statement and Overall Methodology

The purpose of this research is to better understand OSS peer review and to encapsulate this understanding in a theory. We use a multi-case study methodology and multiple data sets (*i.e.* review process documents, archival data, and interviews) and methods (*i.e.* measures of review, statistical analyzes, and grounded theory). By using a diverse set of cases, data, and methodologies we are able to triangulate our findings and improve the generality and reliability of our theory [142, 106]. Figure 1.1 shows the three stages of our work: review processes, parameters and statistical models, and underlying mechanism and behaviours of OSS peer review. First, there are thousands of OSS projects that we could study, so we collect "*limited information on a large number of cases* as well as intensive information on a smaller number" [143]. By examining the project structure and review processes of 25 successful OSS projects, we are able to determine the different types of OSS review and to select six cases for more detailed study. Second, on the six selected cases – Apache, Subversion, Linux, FreeBSD, KDE, and Gnome – we extract measures from software development archives and use *statistical models* to assess the impact of each measured parameter on the efficiency and effectiveness of OSS peer review. While these quantified parameters provide insight into the efficacy of the review process, we do not gain an understanding of the mechanisms and behaviours that facilitate review in OSS projects. Third, on the same six projects, we use *grounded theory* to manually analyze 500 instances of peer review and interview ten core developers. This approach allows us to understand how and why developers perform reviews, how core developers and other stakeholders interact, and scaling issues associated with this style of review. The outcome of this work is a theory of OSS review. We use the theory to compare OSS peer review to existing literature on peer review, software development methods, and review tools.

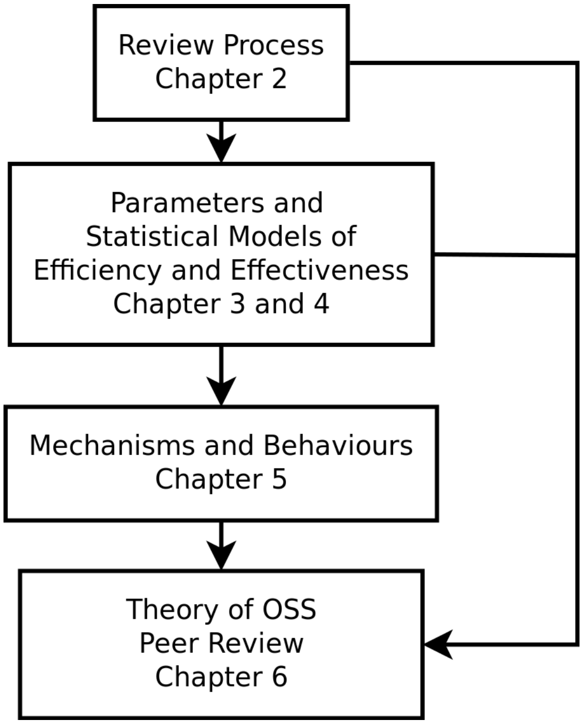


Figure 1.1: Stages in the Research Process

1.2 Outline of Thesis

We have organized the thesis so that each research stage has a distinct section for the literature, research questions, methodology and data, and outcomes. We feel that this division is appropriate given the distinct goals and different research methodologies of each stage: review processes, parameters and statistical models, and underlying behaviours and mechanisms (See Figure 1.1). The final chapter ties together the three stages and our findings into a theory of OSS peer review. This thesis is organized as follows.

Review Processes – Chapter 2

Research question: What are the review processes used by OSS projects?

Literature: We examine the processes used in formal inspection, walkthroughs, and pair programming to give a reference point for comparison with OSS peer review.

Methodology: Since it is not clear which cases to analyze in detail, we collect "limited information on a large number of cases" [143] (25 in total), and select six for further analysis.

Outcome: In OSS, the formality of the process is minimized and developers simply examine each others' code. There are two main types of review: review-then-commit and commit-then-review. We use theoretical sampling to choose six high-profile projects for more detailed analysis: Apache, Subversion, Linux, FreeBSD, KDE, and Gnome.

Parameters of OSS Review and Statistical Models of Efficiency and Effectiveness – Chapters 3 and 4

Research questions: What is the review frequency (Q1), level of participation (Q2), experience and expertise of participants (Q3), size (Q4) and complexity of the change (Q5), efficiency or review interval (Q6), and effectiveness or number of issues discussed(Q7)?

Literature: We reviewed the empirical literature on inspection including the measures used to evaluate the efficiency and effectiveness of inspection. Some of these measures have been answer by previous investigations of OSS peer review.

Methodology: We create measures on the archival data and statistical models.

Outcome: The parameters indicate that reviews are done by experts on small patches. The models reveal that while the complexity of the contribution and expertise of the reviewers have an impact, the number of participants or interest in a contribution by the community has the largest impact on review efficiency and effectiveness.

Mechanisms and Behaviours that Underlie Broadcast Based Peer Review – Chapter 5

Research questions: What are the techniques used to find reviews (Q1), the impact of ignored reviews (Q2), review outcomes, stakeholders, interactions (Q3), the effect of too many opinions during a review (Q4), and scalability issues involved in broadcasting reviews to large projects (Q5).

Literature: The literature is integrated with our grounded findings. Related literature includes email usage, inspection roles, Parkinson's Law of Triviality, and general OSS literature.

Methodology: Grounded theory [52] is used on 500 instances of review and ten core developers are interviewed, some measures are also used (*e.g.,* how many non-core developers comment on reviews?).

Outcome: Developers use simple techniques to find contributions they are interested in or obliged to review. Ignored contributions are those that fail to interest the core development team. Competent and objective outsiders interact with core developers to reach a working solution. While core developers have more power than outsiders, there are no explicit roles during review. Large numbers of uninformed stakeholders can deadlock the decision making processes, however, this rarely occurs during review. On large, diverse projects multiple mailing lists as well as explicit review requests allow broadcast based review to scale.

Discussion and Conclusion – Chapter 6

The main contribution of this work is our theory of OSS peer review:

(1) Early, frequent reviews (2) of small, independent, complete contributions (3) that, despite being asynchronously broadcast to a large group of stakeholders, are conducted by a small group of self-selected experts, (4) resulting in an efficient and effective peer review technique

The implications of this theory are discussed in the context of formal inspection techniques as well as Agile development methods. While there are differences, many of our findings resonate with research findings on other software development methodologies, and future work is necessary to determine if the ability of OSS projects to scale to large distributed teams can be transferred to Agile and other development environments. Our findings also illustrate how simple, flexible tools that rely heavily on the human can be used to efficiently accomplish the complex task of peer review.

Chapter 2

OSS Review Processes

The goal of this chapter is to provide a broad understanding of the review processes used by OSS projects. Since it is impractical to analyze the review processes of the many thousands of existing OSS projects, we need to select a sample of projects. We are only interested in successful, large projects that have well defined peer review processes. While future work may focus on, for example, the review processes of small or unsuccessful projects, we feel that it is important to first understand those of successful projects.

In this chapter, we first provide background on common review processes used in software development. This provides the reader with points of comparison when we later describe OSS processes. In Section 2.2, we describe the method we used to examine a wide swath of successful OSS projects, and summarize the types of review used by these projects (See Table 2.1). The details for each project can be found in Appendix A. The majority of projects provide little documentation regarding their review processes. However, our research uncovered two main types of review in OSS development: review-then-commit (RTC) and commit-then-review (CTR). In section 2.3, we introduce the six projects that we selected to analyze in detail for the remainder of this work. We describe how replication is used within a multiple case study methodology and why each project was chosen.

2.1 Peer Review Process Literature

> "The human eye has an almost infinite capacity for not seeing what it does not want to see ... Programmers, if left to their own devices will ignore the most glaring errors in the output – errors that anyone else can see in an instant."
>
> Weinberg, [134]

Software inspection, informal or lightweight peer review, pair programming, and OSS peer review are all based on the ability of peers to discover defects and other problems in software. Below we briefly introduce, these common types of peer review. Later (in Chapter 6), we compare and discuss possible transfers between our findings regarding OSS peer review and the general software inspection and peer review literature.

2.1.1 Software Inspection

Software inspections are the most formal type of review. They are conducted after a software artifact meets predefined exit criteria (*e.g.,* a particular requirement is implemented). The process, originally defined by Fagan [39], involves some variation of the following steps: planning, overview, preparation, inspection, rework, and follow-up. In the first three steps, the author creates an inspection package (*i.e.* determines what is to be inspected), roles are assigned (*e.g.,* reviewers, reader, recorder, moderator), meetings are scheduled, and the inspectors prepare for the meeting by examining the artifacts that will be inspected. The inspection is conducted, and defects are recorded but not fixed. In the rework and follow-up steps, the author fixes the defects and the mediator ensures that the fixes are appropriate. Although there are many variations on formal inspections [77, 79], "their similarities outweigh their differences"[138]. A notable variant that facilitates *asynchronous inspection* was introduced by Votta [82]. In contrast to Fagan-style inspections, where defect detection is performed only during the meeting phase, Votta suggested that the meeting could be used simply for defect collation. A series of results have shown that indeed a synchronous meeting is unnecessary [36, 109, 72].

2.1.2 Walkthroughs and Informal Reviews

Software inspection has a single objective: to find defects. Less formal reviews may have many objectives including finding defects, resolving varying viewpoints, finding solutions to defects, and integrating new developers. Fagan [38] notes that the use of inspection and informal reviews are not mutually exclusive, but complementary. Inspections are used only on final work products to find defects, while informal reviews can be used at any stage of development with a variety of objectives. Since the reviews are less formal, the development team can maximize a particular review benefit based on participant and managerial goals. For example, Yourdon's structured walkthroughs [144] follow the steps of an inspection, but reduce the formality of the steps. Wiegers [138] describes two informal review techniques: the peer desk check and the pass around. In the peer desk check the author asks a fellow developer to review a development artifact, while the pass around involves multiple developers (reviewers) checking an artifact and requires the author to collate the results.

2.1.3 Pair-Programming

Many Agile development methods employ pair programming, which involves two developers sharing a single workstation [9, 140]. One developer is responsible for typing ("the driver") while the other developer ("the partner") does more abstract thinking and provides concrete review of what the driver is doing. The pair alternates between the two roles. Pair-programming is not limited to coding; all aspects of software development are done in pairs [138]. Although pair-programming is not exclusively a review technique, "One of pair programming's biggest benefits is the continuous, objective review of design and code" [139, 204].

The premise behind pair-programming is that software developed by a pair of developers will be superior to the software developed by each of these developers working independently. While early evidence indicated that pairs produced higher quality with no loss of efficiency [26, 103, 140, 139], other researchers, including a recent meta-analysis by Hannay *et al.* [56], found high variability in the literature and concluded that pair programming is not always beneficial [67]. For example, Hannay *et al.* found agreement in the literature that pair programming does increase the quality of highly complex, but at a cost of higher effort. There are also a number of secondary, moderating factors that deserve further study. For example, Balijepally *et al.* [4] found that the stronger of the two pairs was held back by the weaker. These secondary factors can lead to subtle reductions in productivity. However, pair-programming can be an effective method of performing peer review in certain settings.

2.2 Review Processes: 25 Open Source Projects

As we discussed above, this dissertation is only interested in examining the review processes of successful, mature projects. Even with this limitation there are thousands of potential OSS case studies [44, 128, 66], and the accurate determination of success in OSS is difficult to quantify [48]. When it is not obvious which cases must be examined, Yin [143] recommends collecting "limited information on a large number of cases as well as intensive information on a smaller number." This limited information allows us to select appropriate case studies for more detailed analysis. An additional output of this activity is a summary of the review processes of many projects.

We use two approaches to identify potential case studies. First, we examine iconic, high profile projects which have obvious measures of success (*e.g.,* dominance of a particular software domain). Second, we use an online "catalogue" of thousands of OSS applications and sample the 17 highest ranked projects [44]. We manually classify the review processes of 25 OSS projects. For each we provide a description of the project, the type of project, the review types and policies used on the project, and any other observations, such as the size of the development team and the governance structure. We visited each project's main development page and searched for links relating to peer review or patch submission. Many projects had no formal policy (See Table 2.1). Of those that did, they were often geared towards new developers, as experienced developers already understood the processes. Policy relating to the review of code that had already been committed was rarely discussed but could be inferred by the existence and examination of a "commits" mailing list.

Since our unit of analysis is the peer review, the pertinent "limited information" is the review process. We leave the details of each project's context to Appendix A and summarize only those points relevant to peer review.

2.2.1 Summary of Review Types

In this section, we describe the types of review that we identified across our sample. There are two main types of review: review-then-commit (RTC) and commit-then-review (CTR). Figure 2.1 visually describes these types of review. Table 2.1 provides a summary of our findings for each projects. The full details of review on each project can be found in Appendix A.

The unit of review in an OSS project is a patch, or contribution (a modification request – MR – in industrial development terminology). A contribution is a development artifact, usually code, that the contributor feels will add value to the project. Although the level of formality of the review processes varies among OSS projects, the general steps involved in review are as follows: 1) the author submits a contribution by emailing it to the developer mailing list, 2) one or more people review the contribution, 3) it is modified until it reaches the standards of the community, and 4) it is committed to the code base. Many contributions are rejected and never make it into the code base [13]. This style of review is called review-then-commit (RTC). In contrast to RTC, some projects allow trusted developers

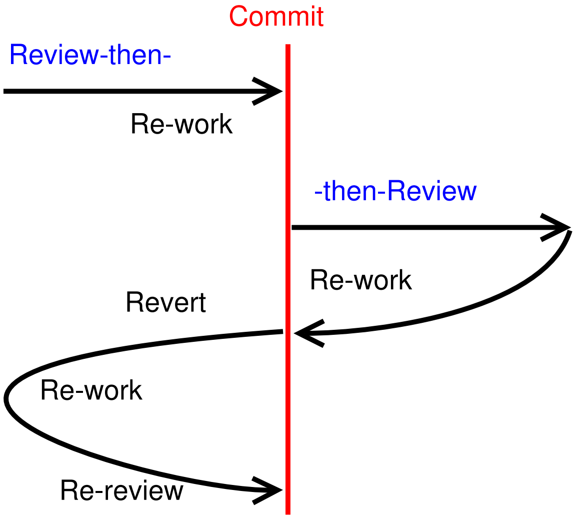


Figure 2.1: Review Processes: RTC and CTR

to commit contributions (*i.e.* add their contributions to the shared code repository) before they are reviewed. The main or core developers for the project are expected to review all commits. This style of review is called commit-then-review (CTR). Most projects use either RTC or CTR, but some (*e.g.,* Apache) employ both methods depending on the status of the committer and the nature of the patch.

There are a five variants within the RTC style of review: informal, “strong”, maintainer RTC, “lazy”, and tracker-based.

- **Informal RTC** exists when there is no explicit policy for review, but contributions are sent to the mailing lists where they are discussed. This is the most common type of review in OSS.
- In contrast, **“strong”** RTC occurs when all developers must have their code reviewed before committing it regardless of their status within the community. For example, on the MySQL and Mozilla projects, all developers, including core-developers, must have two reviewers examine a change before it can be committed. When a project uses “strong” RTC, CTR is not used.
- **Maintainer RTC** occurs on large projects that have explicit code ownership. For example, GCC, Linux, and FreeBSD use this style of review. In this case, developers must get the approval of the code’s maintainer before committing any changes in that part of the system.

Table 2.1: The review types used by the 25 projects examined

Project	Review Types	Appendix
Apache	RTC, Lazy RTC, CTR	A.1.1
Subversion	RTC, CTR	A.1.2
Linux	Maintainer RTC	A.1.3
FreeBSD	Maintainer RTC, CTR, Pre-release	A.1.4
KDE	RTC, CTR, Tracker (ReviewBoard)	A.1.5
GNOME	RTC, CTR, Bugzilla	A.1.6
Mozilla	"Strong" RTC in Tracker (Bugzilla)	A.1.7
Eclipse	RTC in Tracker (Bugzilla)	A.1.8
GCC	Maintainer RTC	A.2.1
cdrtools	Small and stable with no formal review	A.2.3
Postgresql	RTC and Tracker (Commitfest)	A.2.5
VLC	RTC	A.2.6
MPlayer	RTC, CTR	A.2.7
Clam AntiVirus	No explicit policy, commercially run	A.2.8
MySQL	"Strong" RTC	A.2.9
PHP	Informal RTC and CTR and Tracker (Bugzilla)	A.2.10
PHPMyAdmin	Informal RTC and CTR	A.2.11
NTop	Informal RTC and Tracker (Trac)	A.2.12
TightVNC	Tracker (Sourceforge tools)	A.2.13
GTK+	Bugzilla	A.2.14
LibJPEG	Small and stable with no formal review	A.2.15
WebGUI	No explicit policy, commercially run	A.2.16
NMap	RTC	A.2.17
DokuWiki	Informal RTC and CTR	A.2.18
Samba	RTC and CTR	A.2.19

- **"Lazy" RTC**, as used on Apache, occurs when a core developer posts a change to the mailing lists, asking for feedback within a certain time period. If nobody responds, it is assumed that other developers have reviewed the code and implicitly approved it.

- **Tracker-based RTC** occurs when the review is conducted on a web-based tracking tool (*e.g.,* Bugzilla) instead of on a mailing list. Although tracker-based review is outside the scope of this work, we contrast this style of review with email based review in Section 6.2.3. Bugtracker review is used by several projects, including Eclipse [21], Mozilla [69, 104], KDE, and GNOME. On some projects, such as Apache and Linux, a bugtracker is used, but all reviews are still performed on the developers mailing list; the bugtracker is simply for reporting bugs.

Aside from the actual processes of review, there are two policies that apply to all changes to OSS projects. First, a contribution must be small, independent, and complete. Reviewers do not want to review half-finished contributions (*i.e.* incomplete contributions) or contributions that involve solutions to multiple unrelated problems (*e.g.,* a change that involves fixing a bug and correcting the indentation of a large section of code). Large contributions can take longer to review, which can be problematic for volunteer developers. Second, on projects with a shared repository, if any core developer feels that a contribution is unacceptable, he or she can place a veto on it and the contribution will not be committed or, in the case of CTR, it will be removed from the shared repository.

2.3 Selecting Projects for Further Analysis

While it is possible to study a large number of projects at the review process level, to gain a greater depth of understanding of OSS peer review, we selected a subset of projects to analyze. Below we describe how we selected the following six high-profile, large, successful projects for further analysis: the Apache httpd server (which we will refer to as Apache in this work), the Subversion version control system, the FreeBSD and Linux operating systems, and the KDE and Gnome desktop environment projects. Each project has well established review processes and multiple years of archival data that we examined. We used governance style, type of review, and project size as dimensions to ensure that we had adequate breadth in our analysis. In this section, we describe our dimensions as well as the rationale behind our project selections.

2.3.1 Dimensions

Governance: Large OSS projects are usually governed by a foundation consisting of core developers (*i.e.* an oligarchy) or by a “benevolent” dictator [12]. In the case of a foundation, developers who have shown, through past contributions, to be competent and responsible are voted into the foundation. These core developers are given, among other things, voting rights and the privilege to directly modify the shared source code repository (*i.e.* they are given commit privileges). In contrast, although a “benevolent” dictator may delegate certain tasks and decisions to the individuals he or she trusts (*e.g.,* maintainers of a particular subsystem), the dictator is the final arbiter of all decisions on the project.

Type of review: As described above, there are two major types of review in OSS development, review-then-commit and commit-then-review. While there are variants, the main analyzes are only performed on these two types.

Size: Size is an important dimension because certain mechanisms that facilitate review on smaller projects may not scale up to larger projects [1]. For example, the mailing list broadcast of general development discussion, patches, and reviews used by Subversion could become overwhelming on larger and more diverse projects, such as GNOME, thus requiring different techniques for finding and commenting on reviews.

2.3.2 Projects and Replication

The objective of this work is to produce a theory of OSS peer review based on multiple case studies. In case study research, one does not use sampling logic to generalize one's results [142]. Instead, replication logic is used. With sampling logic, one obtains a representative sample of the entire population. Replication logic, in contrast, requires the researcher to select cases such that each case refines an aspect, usually the weakest aspect with the least amount of evidence, of the researcher's theory. This sampling technique is also known as theoretical sampling [29]. The cases are not selected in a random manner because it would take too long or be too expensive to obtain a sufficiently large sample. Replications fall into two categories: literal replications and contrasting replications. A *literal replication* of a case study is expected to produce similar results to the original case. A single case study may lead to a preliminary theory. The components of this theory must be upheld in future case studies that the theory would predict as being similar. If the evidence from literally replicated case studies do not support the theory, then it must be modified to include these cases. Once the theory is strong enough to hold literal replications, *contrasting replications* must be performed to determine if other factors, outside of the theory, can account for the evidence found in the literal replications. These contrasting replications should produce contrasting results, but for reasons predicted by the theory [142].

We began with analyzing the Apache project. We presented our findings and a preliminary theory of peer review in Rigby *et al.* [119]. Based on our dimensions we discuss the reasoning behind and the order of our five replications.

Apache (See Appendix A.1.1) We first examined the Apache httpd server. Apache is a successful, medium sized project that is run by a foundation. It has been the focus of many

[1] Table 3.1 and Figure 5.4 show the size of the projects based on a number of attributes

empirical investigations because early on it formalized and enforced its project policies [95, 55, 14, 119]. Some OSS projects state that they are doing “Apache Style” development [43].

Subversion (See Appendix A.1.2) Subversion (SVN) is a version control system that was designed to replace CVS. Subversion is a good first test of our evolving theory because it borrowed many of its policies from the Apache project and several of the original Subversion developers were also involved in the Apache project[43]. It is also run by a foundation and is similar in size to Apache.

FreeBSD (See Appendix A.1.4) FreeBSD is both a UNIX based kernel and a UNIX distribution. FreeBSD is a literal replication in that it is governed by a foundation and has similar styles of review to Apache and SVN. However, it is a much larger project than either of the previous cases and so also serves as a contrasting replication on the size dimension.

Linux Kernel (See Appendix A.1.3) Linux is a UNIX based kernel, a literal replication with FreeBSD, but also contrasts sharply with the first three projects on the governance dimension. It is governed by dictatorship instead of by a foundation. This change in governance means that Linux can only use RTC and that patches are passed up a “chain-of-trust” to the dictator. Furthermore, the Linux mailing list is substantially larger than any of the lists in the other projects examined (See Figure 5.4).

KDE (See Appendix A.1.5) KDE is a desktop environment and represents not a single project, as was the case with all the previous projects, but an entire ecosystem of projects. KDE also contrasts with the other projects in that end user software is developed as well as infrastructure software. By being a composite project the relationship between each individual subproject is less defined. We are interested in understanding how a diverse and large community like KDE conducts peer review.

GNOME (See Appendix A.1.6) GNOME, like KDE, is a desktop environment and an ecosystem of projects. Developers on this project write infrastructure and end user software. GNOME is a literal replication of the KDE case study. For both KDE and GNOME, reviews were at one point conducted exclusively over email. However, many reviews on GNOME are now being performed in Bugzilla, and on KDE, in Bugzilla or ReviewBoard.

2.4 Conclusion

We began this chapter by describing the common styles of review in software development. OSS review has more in common with flexible, lightweight peer review techniques than it does with formal inspection. However, as we discuss later (See Section 6.2.1.1), there are also similarities between inspection practices and our findings.

The two most common review policies used by the 25 OSS projects we examined were RTC and CTR (The full list of review processes is available in Table 2.1 and the descriptions of the projects are available in Appendix A). RTC is the most familiar and common style of review. With RTC, developers submit a contribution for review and it is committed only after it is deemed acceptable. RTC can slow down core developers by forcing them to wait for reviews before adding simple contributions to the source repository, and it is the only option for developers without commit privileges. There are a number of minor variants on RTC, such as tracker based RTC (not studied in this work) and maintainer RTC. Conversely, CTR allows core developers to have their code reviewed after it has already been committed.

In accordance with our multiple case study methodology, we chose six high-profile projects to examine in the remainder of this work: Apache, Subversion, Linux, FreeBSD, KDE, and GNOME.

Chapter 3

Quantifying the Parameters of OSS Peer Review Practices

In the previous chapters, we laid out our research objectives and examined the review policies and processes of 25 OSS projects. Examining the OSS peer review policies it is clear that peer review is seen as an important quality assurance mechanism in the open source software (OSS) community. While the techniques for performing inspections have been carefully studied in industry, in OSS development, the parameters of peer reviews are less well understood. In this chapter, we quantify the following parameters of peer review: the frequency of peer review, the level of participation during review, the expertise and experience of the authors and reviewers, the contribution size, the contribution complexity, review efficiency, measured by the review interval, and review effectiveness, measured by the number of issues found during review. Each parameter leads to a set of questions that are answered for the following six OSS projects – the abbreviation used follows in brackets: Apache httpd (ap), Subversion (svn), Linux Kernel (lx), FreeBSD (fb), KDE (kde), and Gnome (gn). The results are summarized in Table 3.3.

This chapter is organized as follows. In the next section, we introduce our research questions and the related literature. In Section 3.2 we introduce the methodology and the data mining approach used in this chapter. The remaining sections, present results for each research question. A table summarizing the results can be found in Section 3.10.

3.1 Research Questions

We base our research questions upon those that have been asked and answered in the past for inspection techniques (*e.g.,* [1, 28, 110, 112]), so that our findings can be compared with and expand upon the last 35 years of inspection research. Each set of research questions are operationalized as a set of measures. Since these measures are dependent on the type of data and are often proxies for the actual quantity we wish to measure, the measures are introduced, along with any limitations, in the section in which they are used.

Our ultimate goal is to compare the efficacy of the two review types (See Section 4.1) and to model the efficacy of each review type in each project. Each measure will result in one or more variables, and our models will allow us to determine the importance of each variable (See Section 4.2). Table 3.3 summarizes the findings for each research question, while Section 6.1 discusses and proposes a theory of peer review.

In this section, we provide the background and rationale for each research question.

Q1. Frequency and Activity: What is the frequency of review? Is there a correlation between review frequency and project activity?
OSS review policies enforce a review around the time of commit. For pair programming reviews are conducted continuously [25], while for inspections reviews are usually conducted on completed work products [37]. As development activity increases, so to does the number of contributions and commits. If the level of reviewing does not increase with development activity, this could mean that contributions could go unreviewed. This concern is especially relevant in the case of CTR where an ignored commit becomes part of the product without ever being reviewed (*i.e.* "commit-then-whatever"). To study this concern, we correlate review frequency to development activity.

Q2. Participation: How many reviewers respond to a review? How much discussion occurs during a review? What is the size of the review group?
In his experience-based analysis of the OSS project Linux, Raymond coined Linus's Law as "Given enough eyeballs, all bugs are shallow" [114]. It is important to gauge participation during peer reviews to assess the validity of this statement. RTC policy usually specifies the number of reviewers that must be involved in a review (*e.g.,* three in Apache), while CTR contributions are supposed to be reviewed by the core-group of developers. Research into the optimal number of inspectors has indicated that two reviewers perform as well as a larger group [110, 122]. Previous OSS research has found that there are on average 2.35 reviewers who respond per review for Linux [83]. Asundi and Jayat [3] found a similar result for five other OSS projects including Apache. The amount of discussion is also measured to gauge participation by counting the number of messages exchanged during a review. One problem with the previous measures is that reviewers who do not respond (*i.e.* they may

find no defects) will not be counted as having performed a review. We measure the size of the review group (*i.e.* the number of people participating in reviews) at monthly intervals. Our assumption is that if a developer is performing reviews, he or she will eventually find a defect and respond.

Q3. Expertise and Experience: For a given review, how long have the author and reviewers been with the project? How much work has a developer done on the project? How often has a developer modified or reviewed the current files under review?
Expertise has long been seen as the most important predictor of review efficacy [110, 122]. We measure how much experience and expertise authors and reviewers have based on how long they have been with the project and the amount of work and the areas in which they work. Based on the experiences of prominent OSS developers (*e.g.,* Fogel), developers self-select work that they find interesting and for which they have the relevant expertise[41, 43, 114]. Two papers [3, 118] indicate that a large percentage of review responses are from the core group (*i.e.* experts). We expect OSS reviews to be conducted by expert developers who have been with the project for extended periods of time.

Q4. Change Size: What is the relationship between artifact size and peer review?
Mockus *et al.* [96] found that the size of a change, or churn, for the Apache and Mozilla projects were smaller than for the proprietary projects they studied, but they did not understand or investigate why. We investigate the relationship between OSS review policy and practice to the size of the review and compare OSS projects with each other. We want to understand whether the small change size is a necessary condition for performing an OSS style of review.

Q5. Complexity: What is the complexity of an artifact under review and what is the simplest way of measuring this complexity?
Experienced developers working on complex code may produce more defects than inexperienced developers working on simple code [136]. We must measure the complexity of the changes made to the system to control for this potential confound that could make inexperienced developers look superior to experienced ones. We also explore the impact of complexity on peer review; for example, do more complex reviews have a longer review interval?

Furthermore, unlike inspection that are performed on completed artifacts, in OSS development, changes are sent to the mailing list as diffs that contain only the section of code that have changed and some small amount of context. Determining the complexity of these fragments of code is different from determining the complexity of the entire system. To explore this question, we use seven different measures of complexity. We find that certain measures are highly correlated, so, in the interest of parsimony, we use the simplest measures in our models.

Q6. Review Interval: What is the calendar time to perform a review?
The review interval, or the calendar time to perform a review, is an important measure of review effectiveness [110, 112]. The speed of feedback provided to the author of a contribution is dependent on the length of the review interval. Interval has also been found to be related to the timeliness of the project. For example, Votta [82] has shown that 20% of the interval in a traditional inspection is wasted due to scheduling. Interval is one of our response variables. In Chapter 4, we use statistical modelling to determine how the other variables influence the amount of time it takes to perform a review.

Q7. Issues: How many issues are discussed during a review?
The number of defects found during a review is a common but limited measure of review effectiveness [39, 71, 110]. Since OSS developers do not record the number of defects found during review, we develop a proxy measure: the number of issues found during review. An issue, unlike a true defect, includes false positives and questions. For example, an issue brought up by a reviewer may actually be a problem with the reviewer's understanding of the system instead of with the code. In previous work, we manually classified a random sample of reviews to determine how many reviews contained defects[119]. This manual process limited the number of reviews about which we could assess review effectiveness. We develop a technique for automatically counting the number of issues discovered during a review. In Chapter 4, we statistically model the number of issues found, to understand which of the variables discussed above have the greatest impact on review effectiveness.

In the following sections, we present results related to each of these research questions. The findings are presented as descriptive statistics, and we provide minimal explanation to allow the reader to form his or her own view of OSS peer review. We defer discussion of their usefulness in our models until Section 4.2. We discuss the model only if there are many possible measures of the same attribute (*e.g.,* we have seven possible complexity measures, see Section 3.7). Here we use the principle of parsimony and the level of correlation among variables to determine which ones to keep for further analysis. Since each question requires a different set of measures, we describe each measure and discuss its limitations in the section in which it is used. A discussion of the limitations of this study can be found in

Section 4.3. Although the measures may be unfamiliar to readers, they are designed to mirror the measures used in traditional inspection experiments.

3.2 Methodology and Data Sources

OSS developers rarely meet in a synchronous manner, so almost all project communication is recorded [41]. The OSS community fosters a public style of discussion, where anyone subscribed to the mailing list can comment. Discussions are usually conducted on a mailing list as an email *thread*. A thread begins with an email that includes, for example, a question, a new policy, or a contribution. As individuals reply, the thread becomes a discussion about a particular topic. If the original message is a contribution, then the discussion is a review of that contribution. We examine the threaded reviews on the project's mailing lists. One advantage of this archival data is that it is publicly available, so our results can be easily replicated.

The most important forum for development-related discussion is the developers' mailing list or lists. In most projects, contributions that require discussion must be sent to this list. Gnome and KDE are notable exceptions that allow sub-projects to choose whether bugs will be sent to the mailing list, the bugtracker, or as is the case with FreeBSD, both. We examine mailing list based peer review; a quantification of bugtracker based review is outside the scope of this work. Table 3.1 shows the time period we study, the number of threaded email discussions, the number of commits, and the number of review threads. The table demonstrates the scale differences among the selected projects and the level of use of each review type. Certain projects, KDE and Gnome in particular, have drastically more commits and threads than reviews, indicating that many reviews occur in the bug repository or reviewing tool.

RTC. For review-then-commit, we identify contributions on the mailing list by examining threads looking for diffs. A diff shows the lines that an author has changed and some context lines to help developers understand the change. We examine only email threads that contain at least one diff. In previous work, we considered an email thread to be a review if the email subject contained the keyword "[PATCH]". This technique works well on the Apache project as most developers include the keyword in the subject line; however, other projects do not use this convention. For consistent comparison with the other projects, we re-ran the analysis on Apache, this technique identified an additional 1236 contributions or

Table 3.1: Project background information: The time period we examined in years, the number of threaded email discussions, the number of commits made to the project, and the number of RTC and CTR style reviews.

Project	Period	Years	Threads	Commits	RTC	CTR
Apache (ap)	1996–2005	9.8	53K	32K	3.4K	2.5K
Subversion (svn)	2003–2008	5.6	38K	28K	2.8K	2.1K
Linux (lx)	2005–2008	3.5	77K	118K	28K	NA
FreeBSD (fb)	1995–2006	12.0	780K	385K	25K	22K
KDE (kde)	2002–2008	5.6	820K	565K	8K	15K
Gnome (gn)	2002–2007	5.8	583K	450K	8K	NA

an additional 60% of the original sample.

CTR. Every time a commit is made, the version control system automatically sends a message to the "version control" mailing list containing the change log and diff. Since the version control system automatically begins each commit email subject with "cvs [or svn] commit:", all replies that contain this subject are reviews of a commit. In this case, the original message in the review thread is a commit recorded in the version control mailing list. Occasionally a response will stay on the commit list, but typically responses are sent to the developer mailing list.

Limitations of the data. The data can be divided into two sets: contributions that receive a response and contributions that do not. In this work, we limit our examination to contributions that receive a response, because when a response occurs, we are sure that an individual took interest in the contribution. If there is no response, we cannot be certain whether the contribution was ignored or whether it received a positive review (*i.e.* no defects were found). We do not include these data because ignored contributions would skew our measurements. For example, since an ignored contribution has no reviewers, if we include these data, we drastically reduce the number of reviewers per contribution, even though these contributions do not actually constitute reviews. Furthermore, we assume that contributions that received a positive review will use the same or fewer resources as contributions that receive a negative review. For example, we expect the review interval to be shorter when no defect is found than when one is found. In summary, we are forced to use a sample of OSS reviews (the sample is not random). We believe that our sample is the important and interesting section of the data (*i.e.* it is the data that received a response). We also believe that using "reviews" that do not receive a response would significantly reduce the meaningfulness of our measures.

Within the set of reviews that received a response (*i.e.* the data we have sampled), we make an additional assumption that *a reply to a contribution is a review*. To validate this assumption, we manually analyze a random sample of 460 email threads containing contributions. While the analysis and interpretation of these data is deffer to Chapter 5, we found that few of these email threads did not constitute a review. The exceptional cases were, for example, policy discussions or a response from a developer indicating that the contribution was not interesting and would not be reviewed.

We recognize that at least two important questions cannot be answered using the given data. First, since we cannot differentiate between ignored and positively reviewed contributions, we cannot address the exact proportion of contributions that are reviewed. Second, although we do provide the calendar time to perform a review (interval), we cannot address the amount of time it takes an individual to perform the review. However, in Lussier's [86] experience with the OSS WINE project, he finds that it typically takes 15 minutes to perform a review, with a rare maximum of one hour.

Extraction Tools and Techniques. We created scripts to extract the mailing lists and version control data into a database. An email script extracted the mail headers including sender, in-reply-to, and date headers. The date header was normalized to Coordinated Universal Time (UTC). Once in the database, we threaded messages by following the references and in-reply-to headers. Unfortunately, the references and in-reply-to headers are not required in RFC standards, and many messages did not contain these headers [115]. When these headers are missing, the email thread is broken, resulting in an artificially large number of small threads. To address this, we use a heuristic based on the date and subject to join broken threads (See Appendix B for more details).

Plotting the Data. We use two types of plots: beanplots and boxplots. Beanplots are one-dimensional scatter plots and in this work contain a horizontal line that represents the median [74]. When we have count data that is highly concentrated we use a boxplot. For all the boxplots in this work, the bottom and top of the box represent the first and third quartiles, respectively. Each whisker extends 1.5 times the interquartile range. The median is represented by the bold line inside the box. Since our data are not normally distributed, regardless of the style of plot, we report and discuss median values.

In summary, the main disadvantage of these OSS data is that unlike traditional inspection experiments, the data was not created with the goal of evaluating review efficacy, so there are certain values, which we would like to measure, that were never recorded. The main advantage is that the data is collected from an archive where there is no experimenter or participant bias. Although our data and measures are not perfect, they provide an unbiased (at least by human intervention) automated technique for collecting information about mailing list-based review processes.

3.3 Frequency and Activity

Q1: What is the frequency of review? Is there a correlation between review frequency and project activity?

We measure the relationship between development activity and reviews. We examine the frequency as the number of reviews per month.

For **RTC**, review frequency is measured by counting the number of contributions submitted to the mailing list that receive at least one reply. For **CTR**, we count the number of commits that receive at least one reply on the lists. Development activity is measured as the number commits.

Figures 3.1 and 3.2 show the number of reviewers per month for RTC and CTR, respectively. We see that RTC Linux has far more reviews per month (median of 610) than any other project. KDE, FreeBSD, and Gnome all have slightly over 100 reviews, while the smaller projects, Apache and SVN have around 40 reviews in the median case. A similar divide occurs when we look at CTR. While there are large differences in terms of frequency and number of reviews that appear to be appropriate given the project sizes, measures at the individual review level, which in subsequent sections, show much greater consistency across projects regardless of size.

In order to determine the relationship between commit activity and the review type, we conduct Spearman correlations – a non-parametric test. We assume that commit activity is related to development activity. The correlation between the number of CTRs and commits is strong ($r = .75$, .61, and .84 for Apache, SVN, and FreeBSD respectively), with the exception of KDE ($r = .16$). This correlation indicates that the number of CTRs changes proportionally to the number of commits. Therefore, when there is more code to be reviewed, there are more CTR reviews. This finding suggests that as the number of commits increases, CTR continues to be effective and does not become, as one Apache developer feared,

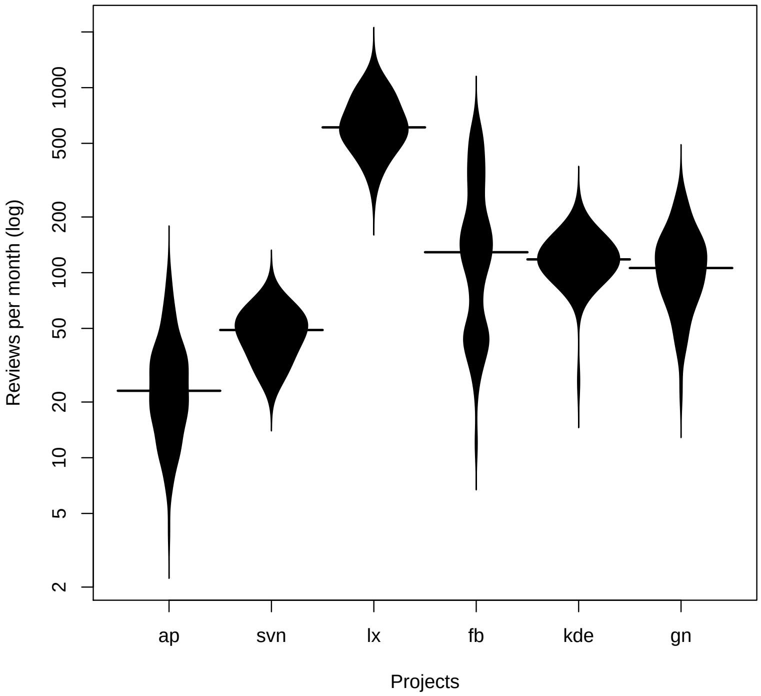


Figure 3.1: RTC – Number of reviews per month

"commit-then-whatever" (See Appendix A.1.1). Since KDE is a large set of related OSS projects, the lack of correlation between CTR and commits may be because not all projects use CTR.

In contrast, the correlation between the number of RTCs and commits is weak to moderate. Only two project (Linux with $r = .55$ and FreeBSD with $r = .64$, respectively) are correlated above $r = 0.50$ with the number of commits. This result may be in part related to the conservative nature of mature, stable projects. When OSS developers describe the iterative nature of patch submission, they report that a contribution is rarely accepted immediately and usually goes through some revisions [86, 43] (See Chapter 5). Researchers have provided quantitative evidence to support this intuition. Bird *et al.* [13] find that the acceptance rate in three OSS projects is between 25% and 50%. Also on the six projects examined by Asundi and Jayant [3] they found that 28% to 46% of non-core developers had their patches ignored. Estimates of Bugzilla patch rejection rates on Firefox and Mozilla range from 61% [69] and 76% [104]. The weak correlations between the number of commits

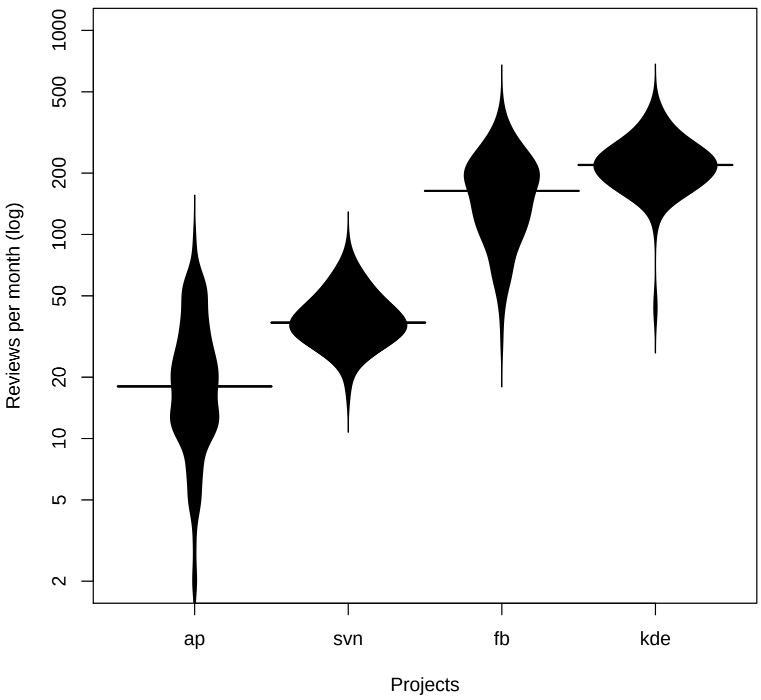


Figure 3.2: CTR – Number of reviews per month

and the number of submitted patches and a high rejection rate, may be explained by the conservativeness of mature OSS projects and because RTC is the only review method available to non-core developers. While this explanation has some supporting evidence, there may be other factors, such as development cycles, that could affect this relationship and deserve future study. The two operating systems appear to be exceptions and also warrant further examination; we provide preliminary explanations. Linux does not have a central repository and developers are forced to post patches to the mailing list regardless of status. FreeBSD has an extensive mentoring program that often requires developers to post contributions to the mailing list and get approval from their mentors before committing.

In summary, by reviewing a contribution around the time it is committed, OSS developers reduce the likelihood that a defect will become embedded in the software. The frequency of review is high on all projects and tied to the size of the project. The frequency of CTR has a reasonably strong correlation with the number of commits indicating that reviewers likely keep up with committers. The correlation between commits and RTC is less strong and may

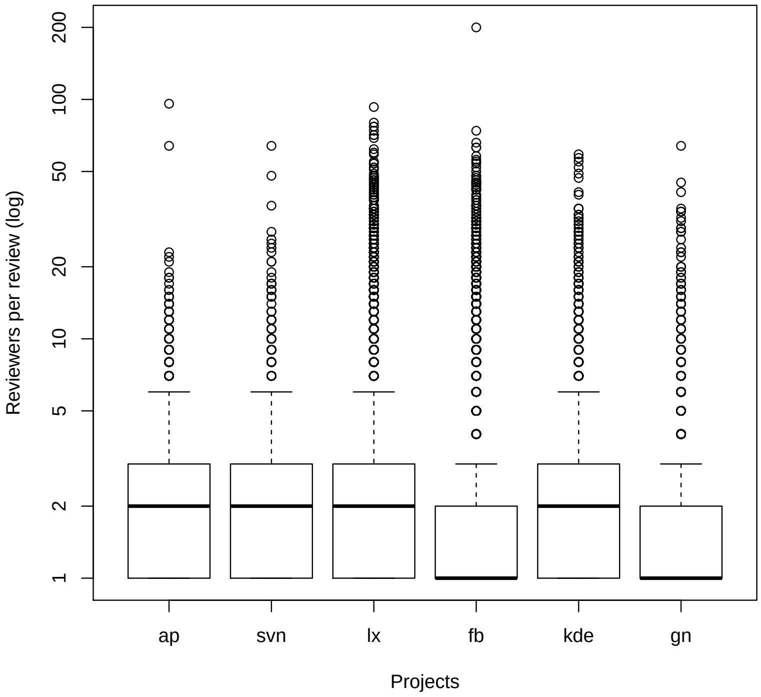


Figure 3.3: RTC – Number of reviewers per review

be related to the high levels of rejected RTC contributions and the conservative nature of the projects we examined.

3.4 Participation

Q2: How many reviewers respond to a review? How much discussion occurs during a review? What is the size of the review group?

It is simple to count the number of people that come to an inspection meeting. Ascertaining this measure from mailing-list-based reviews is significantly more difficult.

The first problem is that developers use multiple email addresses. These addresses must be resolved to a single individual. We use Bird *et al.*'s [14] name aliasing tool to perform this resolution. The remaining problems relate to the data available for each review type. In mailing list based review, it is only possible to count reviewers who respond to a contribution. So if an individual performed a review and did not find any issues or found the same issue

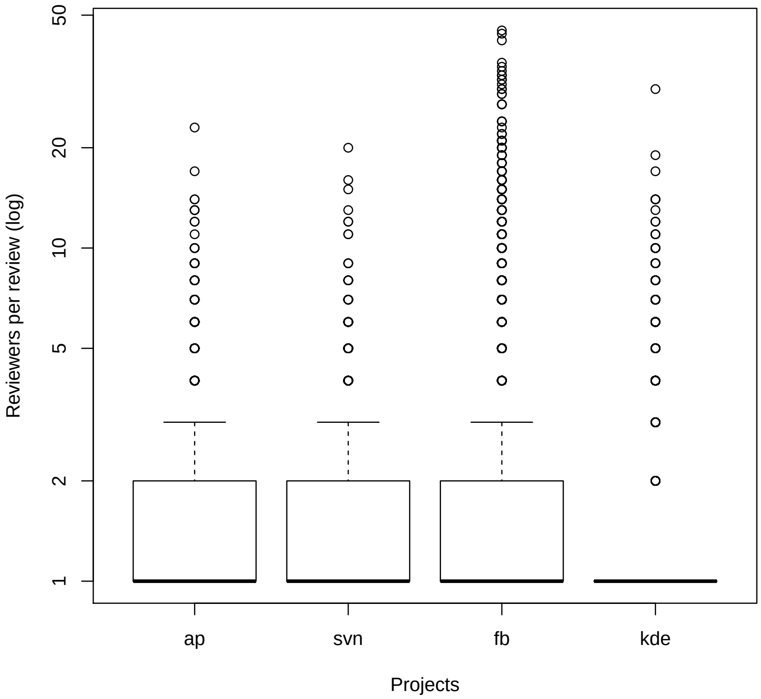


Figure 3.4: CTR – Number of reviewers per review

as other reviewers, this individual would not be recorded as having performing a review. To overcome the latter limitation, we assume that if an individual is performing reviews over a long enough period of time, he or she will eventually be the first person to find an issue and will respond to a contribution (if a reviewer never responds, the reviewer is not helping the software team). We define the **review group** as all individuals who have participated in a review over a given time period. For this work, we define it on a monthly basis (*i.e.* number of reviewers per month). In summary, we have three measures to gauge participation in reviews: the number of developers per review (roughly equivalent to the number of people who actively participate in an inspection – see Figures 3.3 and 3.4), the number of emails per review (the amount of discussion per review – see Figures 3.5 and 3.6), and the review group or the number of people who performed at least one review in a given month (roughly equivalent to the pool of reviewers who participate in inspections). For each measure, the author of the patch is not counted.

Figure 3.3 shows that, for RTC, all the projects have, with the exception of Gnome, a

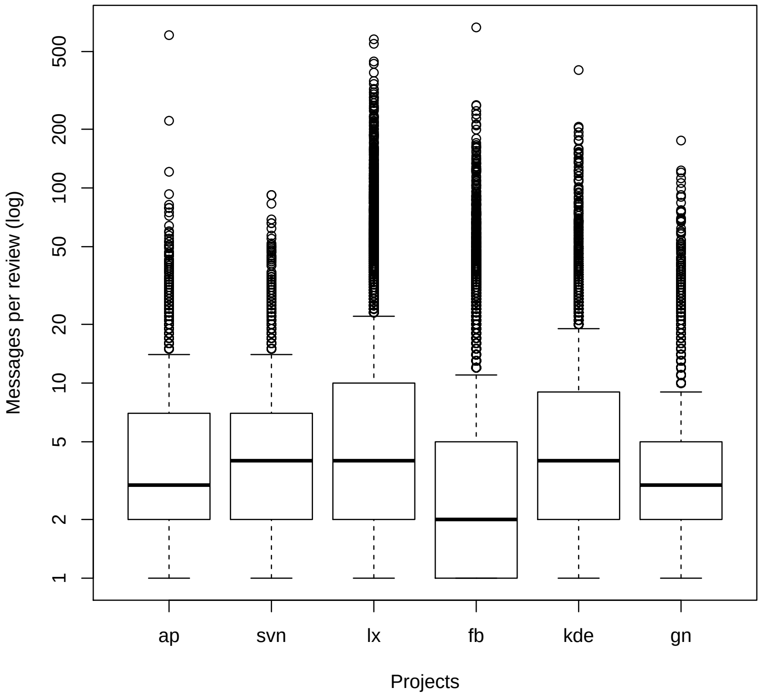


Figure 3.5: RTC – Number of messages per review

median of two reviewers per review. The number of reviewers for CTR is one in the median case (See Figure 3.4). The median number of messages ranges from three to five for RTC and from two to five for CTR, with RTC having more discussion during review. Despite the large differences in the frequency of review across projects, the size and amount of discussion surrounding a review appears to consistently involve few individuals and have a limited number of exchanges. This finding is borne out by our manual analysis and the OSS community's preference for short, terse, technical responses (See Chapter 5).

Surprisingly the number of reviews is very similar to the number of reviewers per month (the review group, which excludes authors). The size of the review group ranges from a median of 16 reviewers (Apache CTR) to 480 (Linux RTC), while the number of reviews ranges from 18 reviews (Apache CTR) to 610 (Linux RTC, see Figures 3.1 and 3.2). It is clear from these numbers and Figures 3.7 and 3.8 that most reviewers do not partake in very many reviews. In the median case, a reviewer participates in between two to three reviews per month. However, the top reviewers, *i.e.* the reviewers in the 95^{th} percentile, participate

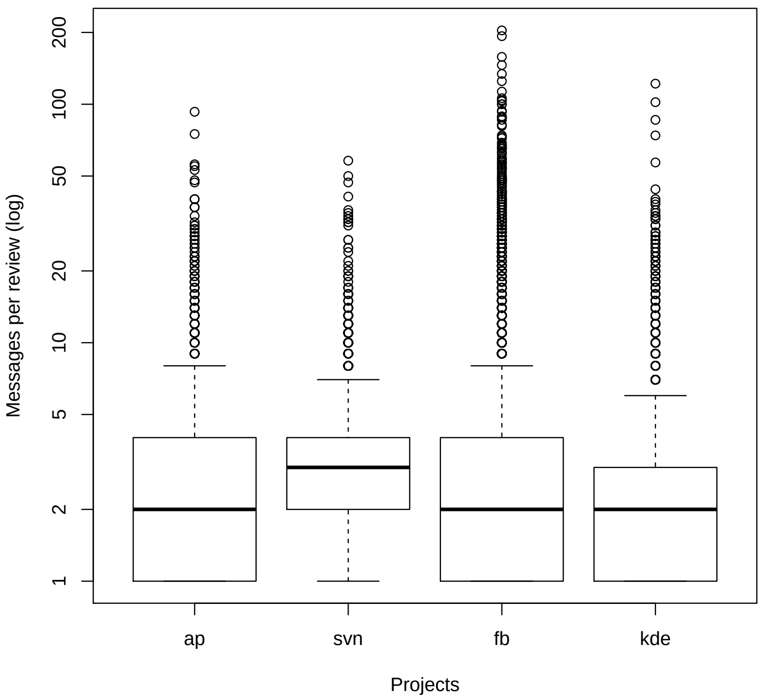


Figure 3.6: CTR – Number of messages per review

in between 13 (Apache CTR) and 36 (Linux RTC), reviews per month. Although 13 Apache CTR reviews may not appear to be many reviews, proportionally, the top reviewers are involved in 72% of all Apache CTR reviews.

This consistently small number of reviews a developer partakes in could relate to inherent human limits and developer expertise. Developer specialization may account for the review group being much larger than the number of reviewers per contribution. In Fogel's [43] experience, developers often defer to an individual who has worked in a given area and proven his or her competence. It is also very unlikely that a single individual will be able to partake in hundreds of reviews per month as is the case for the large projects and specialization is the likely solution.

Overall, we find that few individuals are involved and few messages are exchanged during each review. Surprisingly, there is a very large number of developers who participate in reviews in any month; however, most developers do few reviews per month, and only the top reviewers participate in a large number of reviews per month.

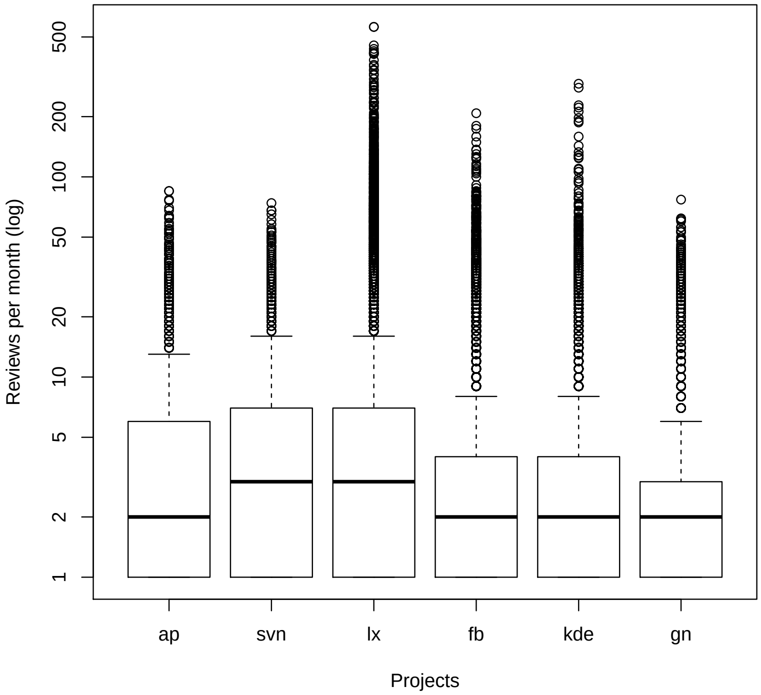


Figure 3.7: RTC – Number of reviews per month a developer is involved in

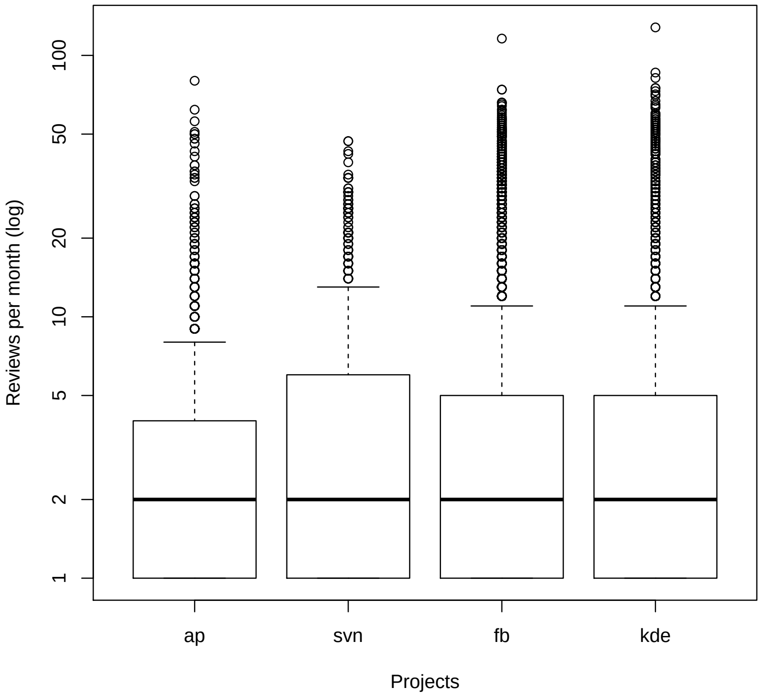


Figure 3.8: CTR – Number of reviews per month a developer is involved in

3.5 Experience and Expertise

Q3: For a given review, how long have the authors and reviewers been with the project? How much work has a developer done on the project? How often has a developer modified or reviewed the current files under review?

Expertise has long been seen as the most important predictor of review efficacy [110, 122]. We measure three aspects of expertise and experience, which we define and discuss below. For each review, the measures are calculated using reviews and commits that occurred before the current review. In the case of the reviewers, both the average value and maximum value is calculated. Since a small group of experts will outperform a larger group of inexperienced reviewers [122], it is important to record the level of experience of the most experienced reviewer. This maximal value will be unaffected by a large number of inexperienced reviewers supported by one experienced reviewer.

We have one measure of experience and two measures of expertise. For experience we measure how long the developer has been with the project. For expertise we measure the amount of work a developer has done and the files they have modified in the past.

3.5.1 Experience

The experience of an author or reviewer is calculated as the time between a developer's first message to the mailing list and the time of the current review.

Figures 3.9 and 3.10 show the distribution of author experience and the distribution of the experience of the top reviewer. For **RTC** the median author experience ranges from 275 to 564 days, and the median top reviewer experience ranges from 708 to 1110 days[1]. For **CTR** the median author experience ranges from 558 to 1269 days, and the median top reviewer experience ranges from 715 to 1573 days. Given that different time periods were examined for the projects (*i.e.* Linux = 3.5 years to FreeBSD = 12.0 years), it may not be appropriate to compare across projects. However, regardless of time period, both authors and reviewers appear to have been on the mailing lists for extended periods of time, and reviewers typically have more experience than authors. It is also clear that both authors and reviewers for RTC have been with a project for a shorter period of time, and they are likely less experienced than reviewers and authors involved in CTR. This result is not surprising, since CTR requires the author to have commit privileges and experienced core-developers are often required to monitor the commit mailing list making it more likely for experienced developers to be involved in CTR than RTC.

3.5.2 Expertise: Work and files modified

Files Modified: Mockus and Herbsleb [97] find evidence that links the amount of work a developer performs in a particular area to his or her level of expertise. They measure the number of commits a developer made to a particular file in the system. We extend this measure by summing the number of times per month a particular developer reviewed or committed to the area of the system that is under review.

Let P be the set of all file paths in the system, $f_r \subseteq P$ are the files that are modified in review r, and $c_{m,i} \subseteq P$ are all the files an individual i has changed in the version control system or reviewed in the calendar month m. The function $C(p, m, i)$ is the number of times an individual i has modified or reviewed the file p in month m.

[1] In all our measures, the correlation between the average and max reviewer is at or above $r = .89$. Due to this high correlation we use the maximal reviewer in the remainder of this work

Then the “file” experience the individual i has on a given review r in calendar month m is

$$file(r, m, i) = \sum_{p \in c_{m,i} \bigcap f_r} C(p, m, i)$$

We only calculate this “files modified” measure for Apache, Linux, and KDE. Appendix B.2 discusses the difficulties of matching file paths in email patches with those in the version control system on the Subversion, FreeBSD, and Gnome projects.

Work: The amount of “work” done by an author or reviewer is calculated on a monthly basis as the number of reviews and commits a developer has made before the current review regardless of area or files modified.

Then the total “work” experience the individual i has on a given review in calendar month m is

$$work(m, i) = \sum_{p \in c_{m,i}} C(p, m, i)$$

Unlike the “files modified” measure of expertise, the total “work” measure does not depend on which files are modified during a particular review.

Weighting: We use the following linear decay function to weight the measures. For both the measures we divide the life of the project into monthly intervals and divide the measure by the number of months since a given review. So, for example, the work measure becomes

$$work_w(m, i) = \sum_{w=1}^{m} \frac{work(w, i)}{m - w + 1}$$

where m is the calendar month in which the review occurred.

By using this weighting, we factor in the diminishing effect that inactivity and time has on expertise and status within the community. In the files modified measure, this weighting allows the expertise associated with files developed by one individual but then maintained by another to be appropriately divided. For the remainder of this work the “files modified” and “work” measures refer to the weighted versions.

Since these measures are weighted over time, their interpretation is more difficult. Instead of reporting raw numbers, we correlate each measure with our response variables: the number of issues discussed and the review interval. These correlations will help us in determining their usefulness in our model.

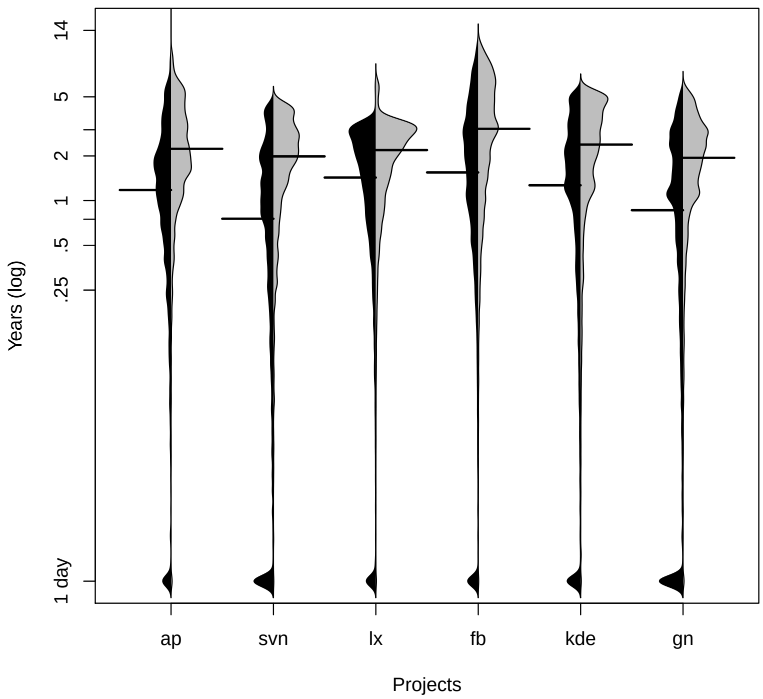


Figure 3.9: RTC – author (left) and reviewer (right) experience in years

Table 3.2 shows the resulting Spearman correlations. In almost every case, measures based upon the author are worse predictors than those based on the reviewer. Furthermore, for author based measures, there is no consistent effect of author expertise on our response variables. Although the correlations are very weak, they are significant and we keep them in our model. The correlation between author work and files modified is moderate in the mean case (min (lx) = .30, mean = .51, max (kde rtc) = .81). In our modelling section we keep only the work measure as it is simpler. Also the weak relationship between the author measures and the response variables does not warrant including extra variables that will overly complicate the model.

Measures based on the reviewer are weakly correlated with the response variables (See Table 3.2). While weak, this correlation is stronger than that between the response variables and the author. The greater the reviewer's experience, the more issues he or she finds and the longer the interval. The mean correlation between file and work based measures is moderate (min (kde ctr) = .39, mean = .59, max (kde rtc) = .80). Like before, we keep only the work

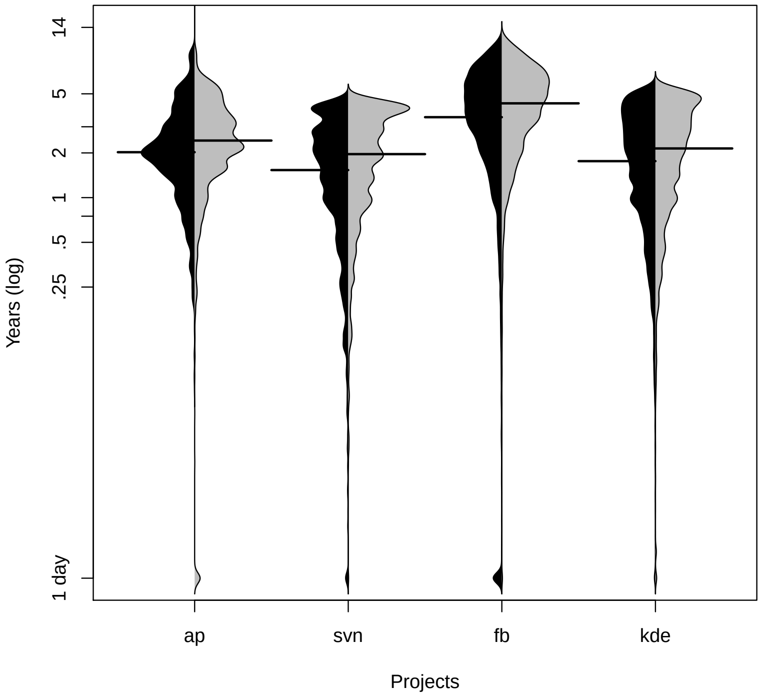


Figure 3.10: CTR – author (left) and reviewer (right) experience in years

measure to simplify our models.

It is not surprising that the author measures are inconclusive. Many issues and a potentially long interval may result from an *experienced* author working on difficult code or an *inexperienced* author modifying a section of the system for the first time. In contrast, the more experienced a reviewer is, the more likely he or she is to find defects and provide time consuming detailed responses, regardless of the author's experience.

In summary, we find that authors have less experience than reviewers and that RTC involves less experienced individuals than CTR. We also find correlations between the amount of expertise of reviewers and the number of issues and the length of the interval; no such correlation is found for authors and the response variables.

Table 3.2: The correlation between the number of issues discussed and the work and file measures. The correlation between the review interval and the work and file measures. $p < .01$, except * $p > .1$

Review type	RTC						CTR			
	ap	svn	lx	fb	kde	gn	ap	svn	fb	kde
Issues										
Auth Work	0.07	0.08	-0.10	-0.07	0.12	-0.22	0.03*	-0.02*	0.00*	-0.02*
Rev Work	0.28	0.32	0.35	0.15	0.20	0.06	0.12	0.16	0.18	0.09
Auth File	0.10	NA	0.12	NA	0.23	NA	0.09	NA	NA	0.04
Rev File	0.32	NA	0.43	NA	0.18	NA	0.15	NA	NA	0.10
Interval										
Auth Work	-0.09	-0.11	-0.10	-0.07	-0.23	-0.08	-0.07	-0.02*	0.02	-0.00*
Rev Work	0.21	0.22	0.28	0.12	0.29	0.14	0.15	0.24	0.21	0.10
Auth File	0.02*	NA	0.08	NA	-0.18	NA	0.01*	NA	NA	0.06
Rev File	0.23	NA	0.32	NA	0.41	NA	0.19	NA	NA	0.12

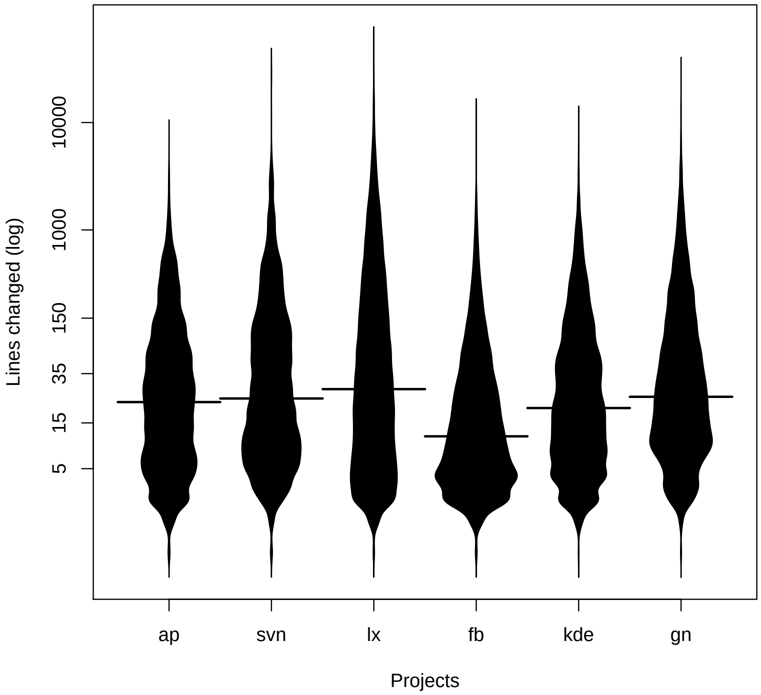


Figure 3.11: RTC churn – Number of lines changed

3.6 Change Size, Churn

Q4. What is the relationship between artifact size and peer review?

The change size of the artifact under review is a common measure in the inspection literature [110]. Mockus *et al.* [96] found that changes to Apache and Mozilla were smaller than changes to the industrial projects they examined, but they could not explain why this was the case. We examine the size of changes that are reviewed. The change size, or churn, is measured by summing the number of added and deleted lines in a software change or patch. Software changes are contributed in the form of diffs, which are fragments that indicate what has changed between two versions of a file [35].

Figures 3.11 and 3.12 show the churn size of reviewed contributions. For RTC the median churn ranges from 11 to 32 changed lines, while the 75^{th} percentile ranges between 37 and 174. For CTR the median churn ranges from 12 to 35 changed lines and the 75^{th}

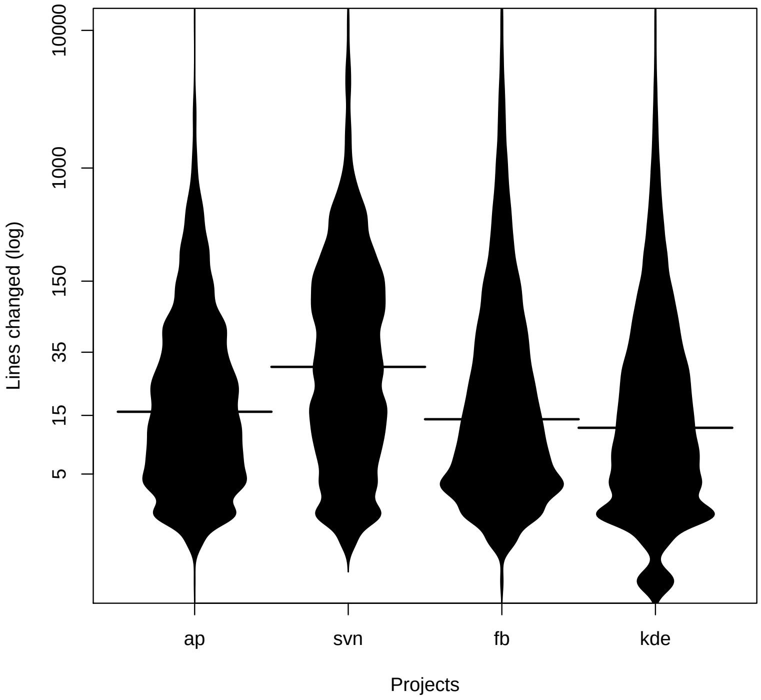


Figure 3.12: CTR churn – Number of lines changed

percentile ranges between 46 and 137. OSS projects have a policy that requires contributions to be small, independent and complete, so it is not surprising that the change sizes are smaller than those in industrial development. This policy, as illustrated by Lussier's experience [86], allows developers to periodically and quickly review a large number of contributions. In the next section, we introduce a series of complexity measures and show how they are related to the amount of churn.

3.7 Complexity

Q5. Complexity: What is the complexity of an artifact under review and what is the simplest way of measuring this complexity?

Contributions are sent to the mailing list in the form of diffs for review. Measuring the complexity of a change differs from measuring the complexity of a file or the entire system

[49]. For example, changing a large number of simple functions across multiple files may be as difficult as changing the code contained within a single complex function. Traditional complexity measures often require the whole file or system and were not designed to measure the complexity of code fragments [63]. Further, multiple studies have found that these measures correlate very strongly with the lines of code in a file [54, 85, 61] and may not provide much additional information about the complexity of the system.

In this section, we provide various possible measures of change complexity. Our measures are applied to patch contributions [49]. All of our measures are based upon the premise that changes that are farther from each other or involve multiple entities are more complex than those closer together involving fewer entities. In total, we measure change complexity in seven ways:

- the churn or change size (see previous section 3.6),
- the number of modified files in a diff,
- the number of diffs per review,
- the number of and distance between contiguous change blocks within a diff (*i.e.* hunks),
- the directory distance between these files, and
- depth of indentation of a change.

Since the goal of each measure is the same, to assess the complexity of a change, we expect some of our measures to be highly correlated. In the interest of parsimony, we will select the simplest set of measures that adequately captures change complexity. Future work could examine each of these measure outside of the context of peer review. Given the goals of this work, when appropriate, instead of providing a box plot, we show that a more complex measure correlates with a simpler measure[2]. In this case, the complex measure is dropped from further analysis.

Files and diffs: we count the number of files contained in a review and the number of diffs. The posting of a diff often represents a re-write of the original contribution or a sequence of diffs that implement a larger feature. As more files change and more diffs are produced, reviewers will likely need an understanding of a larger proportion of the system.

[2] We use non-parametric Spearman correlations, correlation are statistically significant at $p < .001$ unless otherwise specified

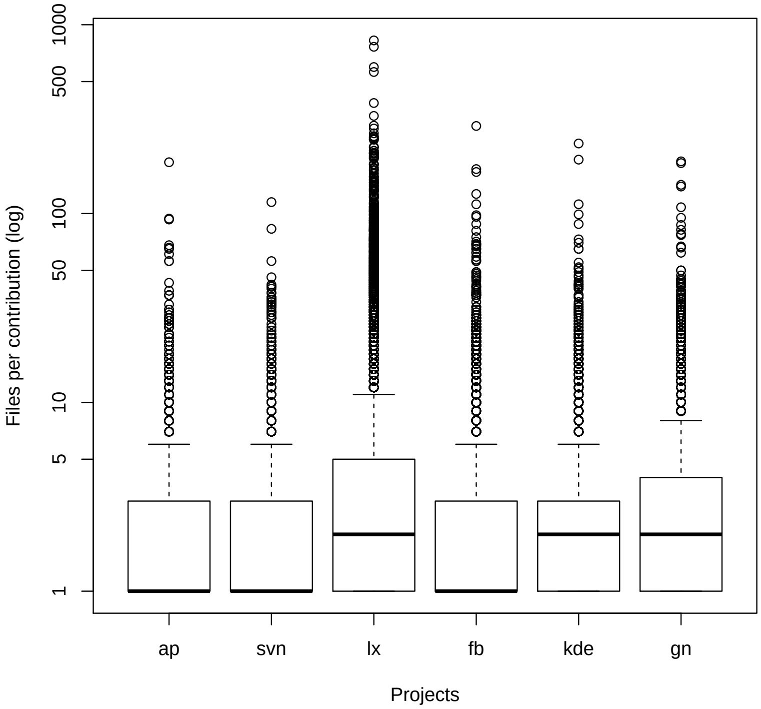


Figure 3.13: RTC – Number of modified files per contribution

Figures 3.13 and 3.14 show the number of files modified per review. For **RTC**, there are one to two files modified in the median case and between three and five at the 75^{th} percentile. For **CTR**, there is one and between three and four for the median and 75^{th} percentile, respectively. There is a moderate correlation between churn and the number of files changed (min (kde ctr) $= .56$, mean $= .64$, max (lx) $= .74$). Since Linux is the only project that shows a strong correlation with churn, we keep the number of changed files as a predictor, but also add an interaction term between churn and number of files (See models in Section 4.2).

The number of diffs per review is low. For **CTR** at the 90^{th} percentile it is one, indicating that few subsequent diffs are posted during CTR. For **RTC** it is one at the median and between one and two at the 75^{th} percentile, indicating that posting updated diffs is also rare with RTC. The correlation between the number of diffs and the number of files is weak (min (kde ctr) $= .11$, mean $= .26$, max (lx) $= .57$) as is the correlation with churn (min (kde ctr) $= .10$, mean $= .25$, max (lx) $= .54$). The number of diffs is kept as a predictor.

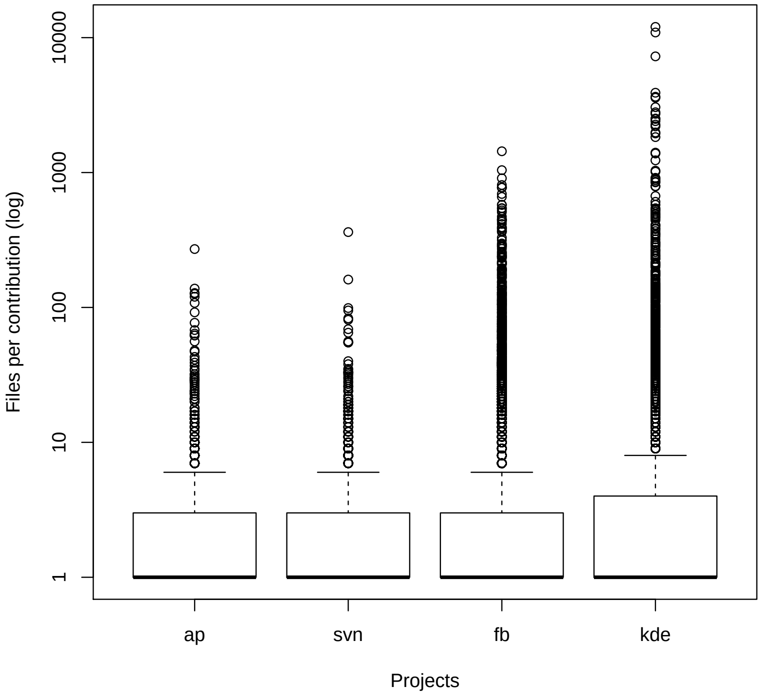


Figure 3.14: CTR – Number of modified files per contribution

Change blocks: we measure the number of contiguous change blocks, or hunks, and the distance between these blocks in a modified file and sum them across the entire review. Contiguous changes are likely easier to review than changes that are far apart and that are potentially in multiple functions within a file. We find that the number of change blocks correlates strongly with a simpler measure, churn (min (kde) $= .70$, mean $= .81$, max (gn) $= .86$). Intuitively and in practice, it appears that as the number of changed blocks of code increases, so to does the number of total changes, or churn. The distance between change blocks is dropped from further analysis, but could be investigated in future work with different objectives.

Directory distance: we measure the directory distance between the files in a review. The premise for this measure is that files that perform similar functions are typically closer in the directory hierarchy than files that perform dissimilar functions [20], thus the directory structure loosely mirrors the system architecture [117]. The distance of two files in the same directory have a distance of zero, while the distance for files in different directories is the number of directories between the two files in the directory hierarchy. We expect that changes that involve files that are far apart will crosscut the functionality of a system and be more complex to review. We find that the directory distance of the files in a change correlates strongly with a simpler measure, the total number of modified files in a review (min (fb rtc = .78, mean = .87 max (fb ctr) = .92). Intuitively we can see that as the number of files increases, the chance that two files will be further apart in the directory structure also increases.

Indentation: Hindle *et al.* [63] created a complexity measure that can be used to measure the complexity of the entire system or of a change by examining the level of indentation on source lines. We calculate their measure for reviews. They found that the strongest predictors of complexity were the sum and standard deviation of the indentation of source lines. Since our contribution sizes are so small, see Section 3.6, the standard deviation is not meaningful, so we just sum the amount of indentation. The sum of indentation correlates strongly with churn (min (fb) = .78, mean = .89, max (lx) = .94), so it is dropped from further analysis.

When one or more measures are highly correlated, parsimony requires us to drop the more complex measures in our model. Churn, see Section 3.6, is kept over indentation and number of change blocks, and the number of modified files in a change is kept over directory distance of files in a change.

In conclusion, it is intuitive that a large amount of churn and modified files would affect a large number and diverse sections of the system and thus represent a complex change. It is also reasonable that the more diffs generated during a review the more complex the change. In agreement with intuition and with previous literature comparing other complexity measures with the number of lines in a file or system, we find that the number of modified files and the churn correlate strongly with other proposed measures of complexity and seem to provide a reasonable proxy for the complexity of a review. Thus, we keep these two simple measures of complexity and the number of diffs per review, and we discard the other four more complex, *i.e.* more difficult to calculate, measures.

3.8 Review Interval

Q6. Review Interval: What is the calendar time to perform a review?

Porter *et al.* [112] defined review interval as the calendar time to perform a review. The full

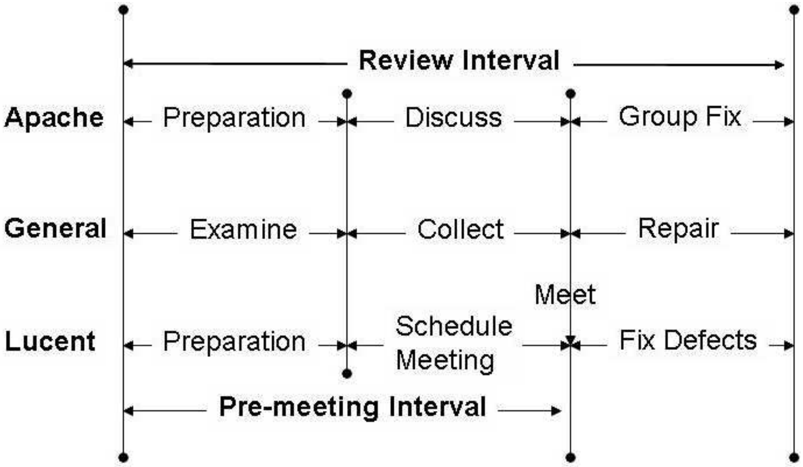


Figure 3.15: The typical stages involved in a peer review.

interval begins when the author prepares an artifact for review and ends when all defects associated with that artifact are repaired. The pre-meeting interval, or the time to prepare for the review (*i.e.* reviewers learning the material under review), is also often measured. Figure 3.15 provides a pictorial representation of review interval.

RTC. For every contribution submitted to the mailing list, we determine the difference in time from the first message (the author's diffed contribution) to the final response (the last review or comment).

The review interval is short on all projects. For RTC, on the right of Figure 3.16 we see that the median interval is between 23 and 46 hours, while 75% of contributions are reviewed in 3 to 7 days. We also measure the time to the first response. This time is roughly equivalent to what is traditionally called the pre-meeting time. The preparation time for the contributor is negligible, since it is a single command to create a diff. First-response time, on the left of Figure 3.16 is between 2.3 and 6.5 hours in the median case and between 12 and 25 hours at the 75^{th} percentile. Only 2 to 5% of RTC reviews begin one week after the initial post, indicating that initially ignored contributions are rarely reviewed.

CTR. Since the contribution is committed before it is reviewed, we need to know not only the time between the commit and the last response, the full CTR review interval (on the right of Figure 3.17), but also the time between the commit and the first response. This first response, shown on the left of Figure 3.17, indicates when the issue was discovered; the ensuing discussion may occur after the problematic contribution is removed. Ideally, the amount of time defective, unreviewed code is in the system should be short. The first review

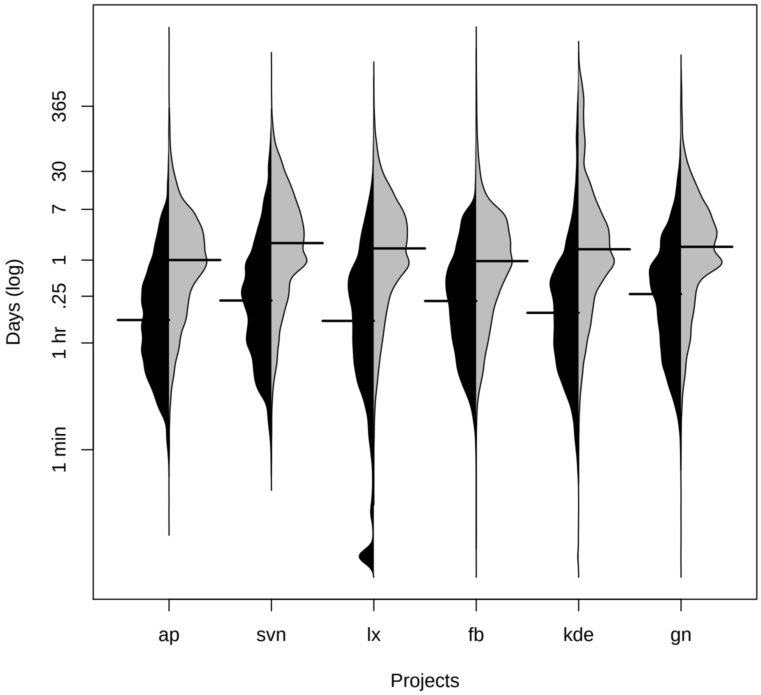


Figure 3.16: RTC – First response (left) and full review interval (right) in days

happens very soon after a commit: 50% of the time it occurs within 1.9 and 4.0 hours and 75% of the time it happens within 9.4 and 12.8 hours. The discussion or full interval of the review lasts longer, with a median value of between 5.6 and 19.4 hours. In 75% of cases, the reviews takes less than between 19.8 hours and 2.7 days. Only between 2 and 3% of issues are found after one week has passed. Like RTC, this final result indicates that if a change is not reviewed immediately, it will likely not be reviewed.

These data indicate that reviews, discussion, and feedback happen quickly. The following quotation from the Apache mailing list discussion in January 1998 supports these findings.

> "I think the people doing the bulk of the committing appear very aware of what the others are committing. I've seen enough cases of hard to spot typos being pointed out within hours of a commit."
>
> Hartill, [121]

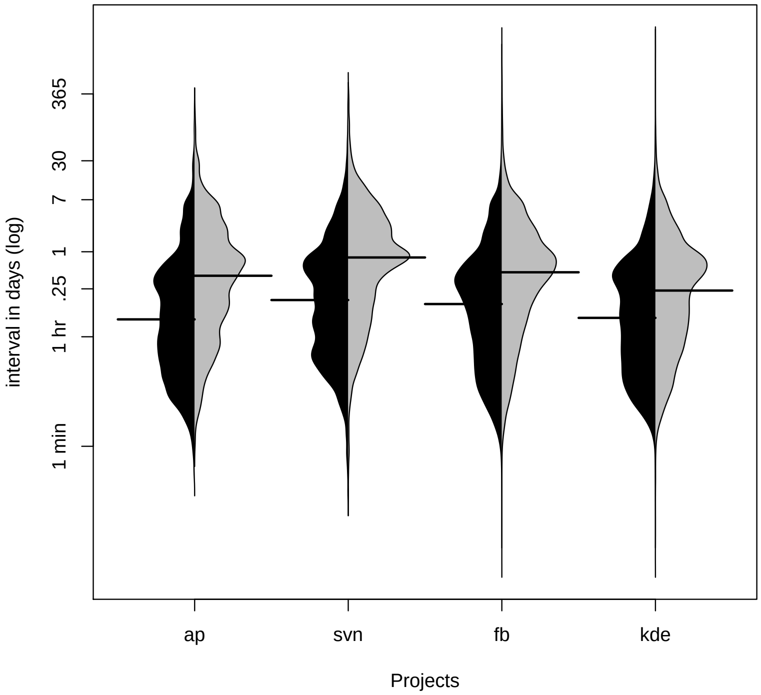


Figure 3.17: CTR – First response (left) and full review interval (right) in days

3.9 Issues and Defects

Q7. Issues: How many issues are discussed during a review?

Ideally, we would be able to use the traditional measure of the total number of defects found during review. Since this information is based on discussion, it is subjective and must be recorded manually. Unfortunately, OSS developers do not record the number of defects found in a review.

We have chosen to examine the review practices of projects that are successful and mature, with low defect rates. The widespread use of these projects indicates that the projects are useful and relatively defect free; some of the credit must be given to the peer review process. To further evaluate the effectiveness of the review process, we create a proxy measure of the number of defects: the number of issues discussed during a review.

In a previous study, we used random sampling to manually determine how many reviews

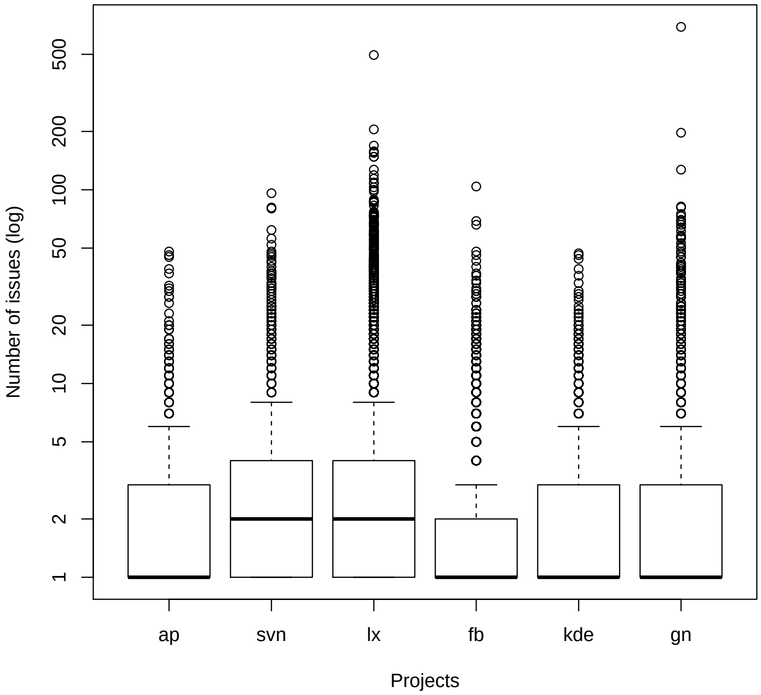


Figure 3.18: RTC – Number of Issues

contain at least one defect [119]. The language used in the reviews makes it apparent whether at least one defect was found; however, without being actively involved in the review, it is difficult to determine how many defects in total were found per review (the traditional measure of defects). Furthermore, since there are thousands of reviews on each of the six examined OSS projects (between svn = 5K and fb = 47K reviews, Section 3.2), this technique allowed us to only assesses a small sample of reviews.

While performing this manual categorization, we realized that reviewers usually responded by posting issues they had with the patch interleaved with the section of the patch they were referring to. This style of interleaved posting is an enforced norm on OSS projects (See Section 5.3.3 for further discussion). By looking for "breaks" in the replied-to text, we developed an automated method to count the number of "issues" discussed during a review.

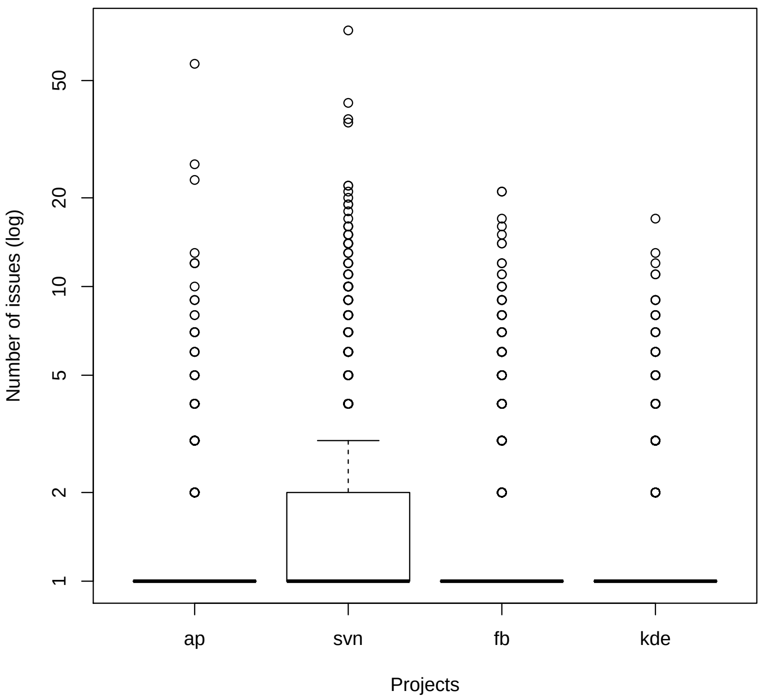


Figure 3.19: CTR – Number of Issues

The following example was taken from a Subversion review:

```
> +  if (err)
> +    return err;
> +
> +  /* svn_ra_svn_read_cmd_response() is unusable as it parses the params,
> +   * instead of returning a list over which to iterate. This means
> +   * failure status must be handled explicitly. */
> +  err = svn_ra_svn_read_tuple(conn, pool, "wl", &status, &list);
> +  if (strcmp(status, "failure") == 0)
> +    return svn_ra_svn__handle_failure_status(list, pool);
> +
What happens to err here. If an error was returned, status is garbage.

> +  if (err && !SVN_ERR_IS_LOCK_ERROR(err))
> +    return err;
> +
Parsing a tuple can never result in a locking error, so the above is
bogus.
```

From the above example, we can see that when the reviewer finds issues with the code, he or she writes the rationale under the section of code that is problematic, removing sections that are unrelated to the problem. This pattern may happen many times during a single email, indicating multiple issues with the code.

Our approach to measuring “issues discussed” is to count the number of times the replied-to text of a message containing a diff (*i.e.* lines starting with ‘$>$’) are broken by new text (*i.e.* lines that do not start with ‘$>$’). Each time this break occurs we consider the break to be an issue that the reviewer has found. Responses to reviewer comments constitute a discuss of an existing issue and not a new issue, so responses to responses are not counted as new issues.

For **RTC**, Figure 3.18 shows the number of issues discussed in a review is one or two in the median case and between two and four at the 75^{th} percentile. For **CTR**, Figure 3.19, the number of issues is one in the median case and between one and two at 75^{th} percentile. The small number of issues discussed per review is consistent with the small size of contributions, Section 3.6, and the policy of OSS projects to review small, independent, complete contributions.

In conclusion, we have created a measure of the number of issues discussed during a review. Traditionally, the only way to obtain this measure was to have developers or researchers manually record the number of defects found. OSS reviews appear to be small and find few defects per review. In the next section, we compare RTC to CTR on our two response variables: issues found during review and the review interval.

3.10 Summary of Quantitative Results

The goal of this chapter was to quantify and establish the parameters of peer review in OSS. Below and in Table 3.3 we summarize our findings.

Q1: Frequency: While all projects conducted frequent reviews, the number of reviews varies with size of project. Reviews increase with number of commits (CTR), but not necessarily with number of patch submissions (RTC).

Q2: Participation: There are hundreds of stakeholders subscribed to the mailing lists that contain contributions; however, there is a small number of reviewers and messages per review. The size of review group varies with size of project. With the exception of core developers, most reviewers participate in few reviews.

Q3: Experience and Expertise: Authors tend to have less experience than reviewers, and individuals involved in CTR have more experience than those in RTC. Reviewer experience tends to vary less than author experience, and the greater reviewer expertise the longer the review interval and more issues found.

Q4: Size and Q5: Complexity: We examined seven complexity measures; three simple measures correlate strongly with more complex measures (See Table 3.3). The changes to patches are clearly small, which likely makes providing feedback quicker..

Q6: Interval (efficiency): The review interval is very short on OSS projects. Facilitated by the small size of a change, reviews are conducted very frequently and quickly. For CTR, where code is already committed, the first response indicates the time it takes to find the initial defect. Very few reviews last more than one week.

Q7: Issues (effectiveness): There are few issues discussed per review. This finding is consistent with small change sizes.

In the next chapter we use these parameters to model the efficiency and effectiveness of OSS peer review. We also discuss the limitations of our parameters and models. In Chapter 6, we combine our findings in a theory.

Table 3.3: Summary of quantitative results. The median value for a project is represented in each case. For example, Apache CTR has a median of 18 reviews per month, while Linux has a median of 610 reviews per month.

Research Question	Measures	Min	Max
Q1: Frequency – Section 3.3			
	Number of reviews per month	18	610
Q2: Participation – Section 3.4			
	Number of reviewers	1	2
	Number of messages	3	5
	Monthly review group	16	480
Q3: Experience and Expertise – Section 3.5			
	Author experience RTC (days)	275	564
	Top review experience RTC (days)	708	1110
	Author experience CTR (days)	558	1269
	Top review experience CTR (days)	715	1573
	Units of expertise (amount and area of work) do not have easily interpretable units, see correlations in Section 3.5.2.		
Q4: Size and Q5: Complexity – Sections 3.6 and 3.7			
	Patch change size in lines (churn)	11	32
	Number of files modified	1	2
	Number of diffs per review thread	1	1
Q6: Interval (efficiency) – Section 3.8			
	First response RTC (hours)	2	3
	Full interval RTC (hours)	23	46
	First response CTR (hours)	2	4.0
	Full interval CTR (hours)	6	19
Q7: Issues (effectiveness) – Section 3.9			
	Number of issues discussed	1	2

Chapter 4

Comparing and Modelling the Efficiency and Effectiveness of OSS Peer Review Practices

The goal of a software process is to facilitate the production of high quality software in a timely manner. Review techniques have been compared and statistically modelled based on their effectiveness (*e.g.,* how many issues or defects they find) and efficiency (*e.g.,* how long it takes to find and remove those issues) [110, 112]. In this chapter, we compare the efficiency and effectiveness of Review-Then-Commit to Commit-Then-Review and then use statistical models to determine the impact that each of our variables, participation, complexity, experience, and expertise, has on the review interval and number of issues found in review. In Chapter 6, we use these findings to develop a theory of OSS peer review.

4.1 Comparing the Efficiency and Effectiveness of RTC to CTR

We want to compare RTC to CTR to give researchers and practitioners an idea of the differences in efficiency and effectiveness of the two techniques. While this comparison may be interesting, the context (See Chapter 2) in which the review types are used must be kept in mind. Although all the projects use RTC, only Apache, Subversion, FreeBSD, and KDE use CTR. For these projects, we compare RTC to CTR based on the number of issues found and the review interval.

The main reason for adopting a CTR policy is that RTC slows down development by increasing the review interval. For the Apache project, the frustration with RTC is apparent in the policy discussion regarding the acceptance of CTR in January of 1998 (See AppendixA.1.1).

> "This is a beta, and I'm tired of posting bloody one-line bug fix [contributions] and waiting for [reviews] ... Making a change to Apache, regardless of what the change is, is a very heavyweight affair."

Our first hypothesis relates to the efficiency of the review techniques. We hypothesize that *CTR has a shorter review interval than RTC.*

With CTR, the contribution is committed before it is reviewed, so if no issues are found in review, the author can commit immediately without waiting for review. In this case, it is obvious that CTR is more efficient than RTC – using the median value, RTC is between one and two days slower than CTR (See Section 3.8). However, if an issue is found, a discussion must occur and the contribution must be fixed or removed. In this case, the median interval for CTR is between six and 19 hours. To determine if there is a statistically significant difference between the two review types, we run a Wilcoxon test with the null hypothesis that RTC and CTR have the same interval (*i.e.* the two distributions are not statistically significantly different). Since we are testing multiple hypotheses, we use the Benjamini-Hochberg to adjust the p-values[11]. Even after this adjustment, we find $p << 0.001$, so we reject the null hypothesis and conclude that CTR has a shorter interval than RTC – using the median value, CTR is 2.1 to 6.5 times faster than RTC. In all cases, CTR has a shorter interval than RTC.

We have determined that CTR has a shorter review interval than RTC. However, if CTR finds fewer issues than RTC, it is a less effective review technique. We hypothesize that *CTR finds fewer issues than RTC.* Using a Wilcoxon test and correcting for multiple hypotheses, we find that although RTC does find more issues than CTR ($p << .001$), the magnitude of the difference is small (See Section 3.9). For SVN, in the median case RTC finds two issues and CTR finds one, while the other projects have no difference. At the 75^{th} percentile, for FreeBSD, RTC finds one more issue than CTR, while the other projects see a difference of two additional issues.

We have found that while CTR has a shorter review interval than RTC, CTR finds slightly fewer issues than RTC. Although these results indicate a trade-off between efficiency and effectiveness, we must remember the different contexts in which they are used by the projects (See Section 2.2). CTR is only used when a trusted, core developer feels confident in what they are committing, while RTC is used when the core developer is less certain or by developers without commit privileges. Since the artifact involved in CTR is likely more polished and is usually written by a more experienced developer (See Section 3.5), it follows that there will usually be less discussion than during an RTC review. These results do not suggest that one technique is superior to another, but rather that they are useful in different circumstances. For example, in an industrial environment, RTC could be applied to new hires or to particularly critical or difficult code, incurring a longer review interval, while CTR could apply to developers who have shown that they are competent and are working on "normal" tasks, thereby reducing the amount of time developers wait for their code to be reviewed.

4.2 Statistical Models of Efficiency and Effectiveness

We want to understand the impact that each variable we introduced in Chapter 3 (See Table 3.3 for a summary) has on review efficacy: efficiency and effectiveness. For efficiency (see Section 4.2.1), measured by the continuous variable review interval, we use a multiple linear regression model with constant variance and normal errors [30, pg. 120]. For effectiveness (see Section 4.2.2), measured by a count of the number of issues found, we use a generalized liner model (GLM) with Poisson errors and a logarithmic link function. A quasi-poisson model and a pseudo R-squared measure is used to correct for over- and under-dispersion [59] [89, pg. 219].

Explanatory variables that were shown in the previous chapter to be correlated with other variables in the model (*e.g.,* number of replies and reviewers) were eliminated, keeping only the stronger predictor (*e.g.,* number of replies). We group our variables into four categories: participation, complexity, expertise, and experience. The descriptive statistics in the previous sections have shown that each of our variables is right or positively skewed with a long tail; so we use a log transformation of the variables. A log transformation explains more variance than a linear transformation. To further illustrate why we use a log transformation, Figure 4.1 shows a quantile-quantile plot of Apache RTC review interval compared to a normal distribution. If review interval is normally distributed, then the points would lie on the $x = y$ line. Table 4.1 and 4.3 show the correlation between our explanatory variables and our response variables, interval and issues, respectively.

Since we examine six projects, four of which have two review types, and we have two response variables, we have a total of 20 models. Due to the large number of models and

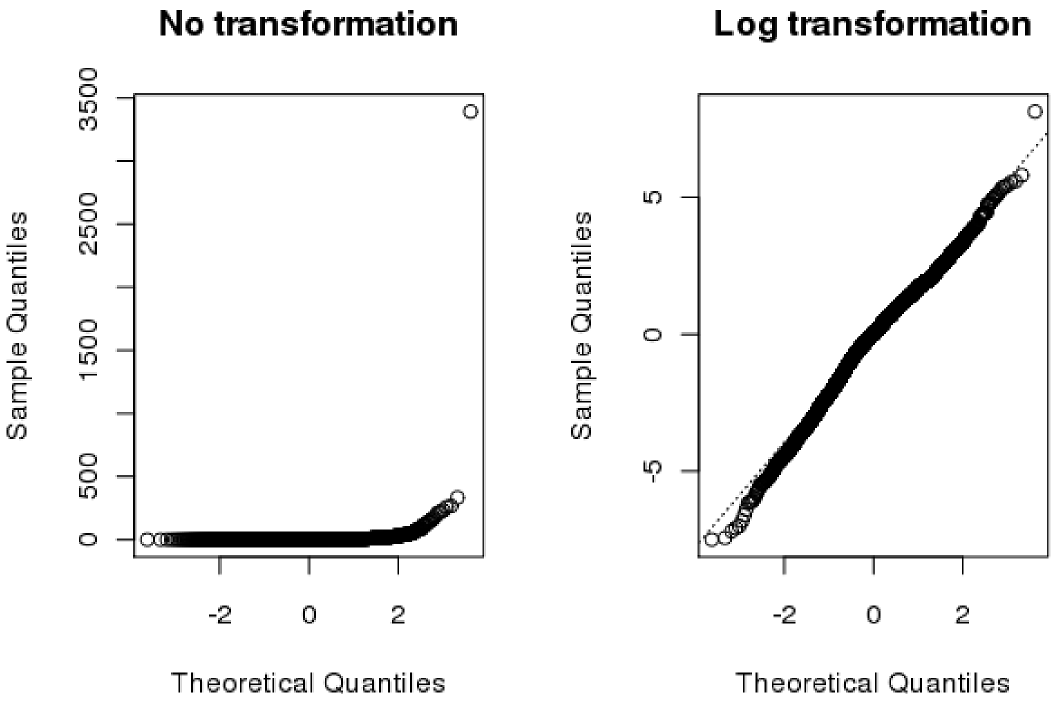


Figure 4.1: Q-Q norm plot for Apache RTC review interval before and after a log transformation. A normal distribution has points on the line $x = y$.

the inclusion of each variable for scientific reasons, we use the same set of explanatory variables in all of our models and we do not run a stepwise reduction on the models. Instead we provide the R-squared value and the analysis of variance or deviance table which shows whether the explanatory variable is statistically significant and how large its effect is on review efficacy. To represent our models we use the R language notation [113]. For example the formula $m \sim a + b * c$ means that “m is modelled by a, b, c and the interaction between b and c”.

4.2.1 Efficiency: Review Interval

We used a single model across all projects and review types. Our model of efficiency, review interval, has the following form [1]:

[1] To avoid taking the logarithm of zero we add one to some variables

$$\begin{aligned}\log(\text{interval}) \sim\ & \log(\text{auth experience} + 1) + \log(\text{auth work} + 1) + \\ & \log(\text{rev experience} + 1) + \log(\text{rev work} + 1) + \\ & \log(\text{files} + 1) * \log(\text{churn} + 1) + \log(\text{diffs}) + \\ & \log(\text{replies})\end{aligned}$$

Table 4.2 shows the analysis of variance for KDE RTC (Figure 4.2 shows the diagnostic plots). The remaining nine models and the graphs that indicate normal errors and constant variance in our statistical models are in Appendix C. The sum of squares indicates the amount of variance explained by each variable [2]. The RTC models account for between 25% and 43% of the variance, while the CTR models account for 19% to 27%. Porter *et al.* [110] are able to explain around 25% of the variance in the Lucent inspection interval. Ideally, we would be able to explain more than 50% of the variance.

Participation: In every model, the amount of participation from the community explained the most variance. In industry, the number of reviewers and length of review sessions is strictly controlled. While in OSS, contributions that fail to catch the interest of the community will be likely ignored and never reviewed. As a result, while these measures may be uninteresting in industry where reviews are assigned [110], they are important in OSS because the more interest a contribution receives the longer the interval.

Complexity: In every model, the complexity of the change, the churn, number of files modified, and number of diffs explained a large proportion of the variance. In KDE and FreeBSD the number of diffs was the most influential complexity measure on review interval. In most projects, the interaction between churn and files was not statistically significant, and when it was, it explained very little additional variance.

Expertise and Experience: In every model, we can see that the experience of the reviewer is more influential than the the experience of the author. While this same pattern holds for expertise, except in KDE. In the case of the author, expertise is more important than experience. For reviewers, there is no clear pattern indicating whether experience is more important than expertise. For Apache and FreeBSD experience is more important than expertise, while for SVN, Linux, KDE, and GNOME the opposite is true. The authors impact on interval is small, with more experience and expertise leading to shorter intervals. In contrast, the more expertise and experience a reviewer has the longer the interval.

[2] The amount of variance explained by each variable is dependent on ordering

Table 4.1: Spearman correlations of interval: * $p > .01$, otherwise $p <= .01$

	RTC						CTR			
	ap	svn	lx	fb	kde	gn	ap	svn	fb	kde
Participation										
replies	0.47	0.51	0.52	0.50	0.60	0.48	0.49	0.47	0.47	0.41
reviewers	0.41	0.39	0.47	0.46	0.45	0.34	0.46	0.35	0.44	0.35
Complexity										
churn	0.21	0.30	0.28	0.15	0.12	0.22	0.10	0.17	0.09	0.07
files	0.17	0.24	0.25	0.04	0.12	0.13	0.09	0.18	0.06	0.06
diff	0.20	0.32	0.34	0.25	0.44	0.26	0.14	0.17	0.17	0.09
Experience										
author	-0.00*	-0.04*	0.01*	0.08	-0.02*	0.01*	0.03*	-0.03*	0.02	0.05
reviewer	0.24	0.06	0.12	0.29	0.21	0.13	0.23	0.08	0.21	0.06
Expertise										
auth work	-0.09	-0.11	-0.10	-0.07	-0.23	-0.08	-0.07	-0.02*	0.02	-0.00*
rev work	0.21	0.22	0.28	0.12	0.29	0.14	0.15	0.24	0.21	0.10

Table 4.2: KDE RTC model of review interval: $R^2 = .43$

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth experience + 1)	1	182.39	182.39	52.43	0.0000
log(auth work + 1)	1	3116.03	3116.03	895.77	0.0000
log(rev experience + 1)	1	1568.05	1568.05	450.77	0.0000
log(rev work + 1)	1	2193.47	2193.47	630.57	0.0000
log(files + 1)	1	231.37	231.37	66.51	0.0000
log(churn + 1)	1	137.84	137.84	39.63	0.0000
log(diffs)	1	5203.06	5203.06	1495.74	0.0000
log(replies)	1	8191.15	8191.15	2354.74	0.0000
log(files + 1):log(churn + 1)	1	10.77	10.77	3.10	0.0785
Residuals	8146	28336.53	3.48		

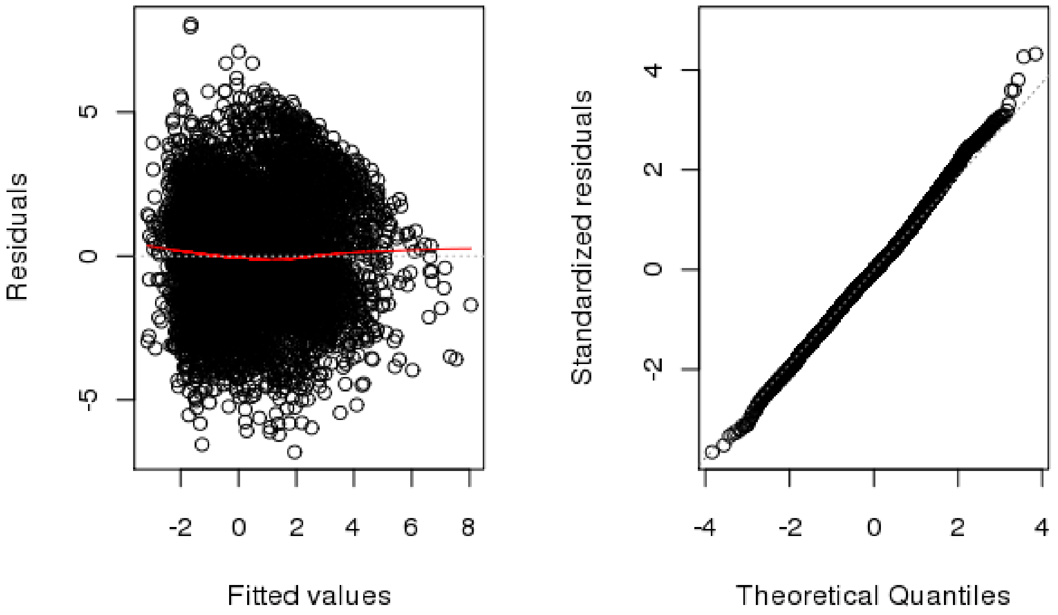


Figure 4.2: Diagnostic plots for KDE RTC interval model: On the left, the residuals vs the fitted values. A random distribution means that there is no systematic bias in our model. On the right, a Q-Q plot of the residuals showing a normal error distribution.

4.2.2 Effectiveness: Issues Discovered

We use the following statistical model of effectiveness, number of issues found [3]:

$$\begin{aligned}\text{issues} \sim\ & \log(\text{auth experience} + 1) + \log(\text{auth work} + 1) + \\ & \log(\text{rev experience} + 1) + \log(\text{rev work} + 1) + \\ & \log(\text{files} + 1) * \log(\text{churn} + 1) + \log(\text{diffs}) + \\ & \log(\text{reviewers})\end{aligned}$$

We have both over- and under-dispersion in our models. Over-dispersion will make explanatory variables look more statistically significant than they actually are, while the opposite is true for under-dispersion [59]. To deal with this problem, we calculate a dispersion parameter, $\hat{\phi}$, that is used to adjust both standard errors and the t-statistic by a factor of $\sqrt{\hat{\phi}}$. This technique is suggested by McCullagh and Nelder [92, pg. 127] and the Pearson based dispersion parameter is χ^2 divided by the degrees of freedom:

$$\hat{\phi} = \frac{\chi^2}{(n-k-1)} = \sum_i \frac{\frac{(y_i - \hat{\mu}_i)^2}{\hat{\mu}}}{(n-k-1)}$$

where k is the number of covariates fitted (without intercept) in the full model[4].

In R, this technique is achieved by running a quasi-Poisson general linear model [89, pg. 213 – 216]. Although this techniques deals with the dispersion issue, it does not provide an R-squared measure (or measure of explained variation) that is adjusted for dispersion. Heinzl and Mittlböck [59] evaluate a series of deviance-based pseudo R-squared measures for Poisson models with over- and under-dispersion. We implemented their most robust adjusted R-squared measure in R. The formula is shown below:

$$R^2 = 1 - \frac{D(y : \hat{\mu}) + k\hat{\phi}}{D(y : \bar{\mu})}$$

where $D(y : \hat{\mu})$ is the deviance of the full model (not saturated model) and $D(y : \bar{\mu})$ is the deviance of the null or intercept-only model.

Table 4.3 shows the Spearman correlations between the number of issues and the independent variables. Table C.13 shows the analysis of deviance and adjusted R-squared

[3] To avoid taking the logarithm of zero we add one to some variables

[4] We use the notation from Heinzl and Mittlböck [59]

Table 4.3: Spearman correlations of issues: * $p > .01$, otherwise $p <= .01$

	RTC						CTR			
	ap	svn	lx	fb	kde	gn	ap	svn	fb	kde
Participation										
replies	0.50	0.62	0.68	0.69	0.40	0.57	0.26	0.30	0.36	0.25
reviewers	0.42	0.47	0.63	0.62	0.38	0.45	0.23	0.21	0.36	0.26
Complexity										
churn	0.20	0.39	0.41	0.10	0.24	0.21	0.19	0.36	0.12	0.06
files	0.18	0.34	0.37	-0.08	0.19	0.10	0.15	0.32	0.14	0.04
diff	0.31	0.48	0.51	0.36	0.09	0.37	0.28	0.31	0.36	0.19
Experience										
auth experience	0.09	0.10	0.01	0.17	0.04	-0.01*	0.01*	-0.04*	0.04	-0.03
rev experience	0.18	0.04*	0.09	0.41	-0.01*	0.17	0.09	0.01*	0.16	-0.01*
Expertise										
auth work	0.07	0.08	-0.10	-0.07	0.12	-0.22	0.03*	-0.02*	0.01*	-0.03
rev work	0.28	0.32	0.35	0.15	0.20	0.06	0.12	0.16	0.18	0.09

measure for Linux RTC. The remaining nine models can be found in Appendix C. The variance accounted for by the RTC models was Apache 29%, SVN 54%, Linux 58%, FreeBSD 46%, KDE 26%, and Gnome 37%. While the variance accounted for by the CTR models was Apache 18%, SVN 32%, FreeBSD 40%, and KDE 16%. Porter *et al.* [110] are able to explain around 50% of the variance in the number of defects found during inspections at Lucent. Three of our RTC models explain around 50% of the variance. For CTR, the Apache and KDE models explain less than 20% of the variance and are excluded from the discussion below. Like Porter *et al.*, on the whole, our models of the number of issues found during review have better explanatory power than those of review interval.

Table 4.4: Linux RTC model of issues found during review: $R^2 = .58$

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth experience + 1)	1	19.76	19.76	0.49	0.4819
log(auth work + 1)	1	896.91	896.91	22.44	0.0000
log(rev experience + 1)	1	8619.96	8619.96	215.71	0.0000
log(rev work + 1)	1	87476.99	87476.99	2189.09	0.0000
log(files + 1)	1	102287.71	102287.71	2559.72	0.0000
log(churn + 1)	1	50443.55	50443.55	1262.34	0.0000
log(diffs)	1	51474.60	51474.60	1288.14	0.0000
log(reviewers)	1	104882.92	104882.92	2624.67	0.0000
log(files + 1):log(churn + 1)	1	10583.63	10583.63	264.85	0.0000
Residuals	27657	1105187.25	39.96		

Participation: The level of participation was an important predictor of the number of issues found. Since developers are not assigned contributions for review, the level of participation indicates the level of interest in a contribution and likely impacts the number of issues found and discussed. However, unlike in the review interval models, the level of participation did not dominate all the models. This result is in contrast to Porter *et al.* [110] who found that the size of the review team did not lead to more defects being discovered. However, the number of reviewers only varied from one to four in the Porter *et al.* study. The level of participation varied more substantially in OSS reviews. Some exceptional OSS reviews have tens of reviewers, the number of reviewers varied between one, first percentile, and two to five, 90^{th} percentile (See Section 3.4).

Complexity: The greater the complexity of a contribution under review, the more issues will be found with it. While this relationship is clear, there is no single complexity measure that dominates the variance explained across all projects. Table 4.3 shows that the number of diffs in a single review thread is the complexity measure most strongly correlated with the number of issues found. Two factors likely affect this relationship: the number of times an author revises his or her contribution and large features that join a number of contributions into a single thread.

Complexity also increases as there are more lines changed (bigger contributions) and more files changed (likely more diverse contributions). OSS developers intuitively understand these realities and have project policies that encourage small, independent, complete contributions that make reviewing less complex (See Chapter 2).

Experience and Expertise of Authors and Reviewers: As was the case with review interval, the experience and expertise of the reviewer is more important than that of the author. Having knowledgeable reviewers clearly leads to more issues found and the best predictor is the amount of work done by a reviewer.

The impact of the author's experience and expertise is less clear. For CTR, where authors are all experienced, the amount of variance explained is small or not statistically significant. In the case of RTC, there is more variance in the skill of authors, and we see that greater author expertise leads to fewer issues found in Linux, FreeBSD, and Gnome while the opposite is true for the other projects.

For RTC, the longer an author has been on the project, the more issues will be found in his or her contributions. We expect that since authors involved in RTC are gaining experience, there is a point where many of their contributions are interesting to reviewers, but they still make a significant number of mistakes. Once they have a good understanding of the system and become core-developers, most of their contributions will go through CTR. Furthermore, on projects that do not use CTR extensively, the author's experience explains very little variance.

4.3 Limitations and Validity

In this section we discuss the construct, internal, and external validity and reliability of this study.

Construct validity: Our measures are based on those that have been used in the past to gauge the efficacy of inspection processes (*e.g.,* Porter *et al.* [110]) and should have high construct validity. However, the data set and creation of these measures are very different from past experiments. The data were not collected for the purpose of measurement. To alleviate this concern, we manually examined randomly chosen reviews to confirm that our measures were appropriate.

Reliability: Our study has high reliability because the data is publicly available and our measures are clearly described. Other researchers can replicate our results.

Internal Validity: It is possible that rival hypotheses and theories may explain the evidence we have collected. The descriptive statistics that we have collected clearly show that OSS review is drastically different from traditional inspection processes. Further, the stability of results across projects strengthens our findings. Our models explained between 16% and 58% of the total variance with an average of 33%. While many of our models are useful, some explain only a small amount of the total variance leaving open the possibility that other factors may be important in determining the efficacy of review. Another possibility is that outliers may be affecting our results. Due to the extremely large number of reviews, it is difficult to eliminate the large number of potential outliers. All attempts to eliminate outliers so far have resulted in a decrease in the variance explained by the models.

External Validity: By examining a diverse set of OSS projects, we increase the generalizability of our results. As discussed in Section 2.3.2, each project was theoretically sampled as a replication. The first case studies are of smaller infrastructure projects, Apache and Subversion, followed by larger infrastructure projects, Linux and FreeBSD. KDE and Gnome, which include many end-user applications, serve as a counterbalance to the infrastructure projects. There are three main limitations to the generality of this study. First, we purposefully choose to examine successful, large, mature OSS projects. While we now have a good understanding of how these projects conduct peer review, it is unclear whether these results transfer to smaller or less successful projects. Second, most of the projects we examined use either C or C++. Although it is possible that different languages may change our results, the review process is not coupled to language constructions and, as we discuss in Chapter 5, we saw that many of the review discussions quickly became an abstract discussion of the issue at hand. Finally, KDE and Gnome conduct a significant number of reviews in a bug tracking system and some discussions occur on IRC chat channels [125]. In Section 6.2.3.3, we compare, at a conceptual level, email based review to tracker based review.

4.4 Summary

In this section we have statistically modelled the review efficiency and effectiveness based on the participation, complexity, expertise, and expertise. The models are provided in Appendix C and the correlations between our explanatory variables and our response variables are shown in Table 4.1 and 4.3.

Efficiency was modelled by the review interval using a multiple linear regression model with constant variance and normal errors. The models accounted for between 19% and 43% of the variance. The most important predictor was the amount of participation (*i.e.* number of replies). The greater the participation or interest from the community the longer the review interval. The second most important predictor was the complexity of a contribution, with more complex contributions increasing the review interval. The level of expertise and experience of the author explained less of the variance than that of the reviewers. The greater the level of expertise and experience of the reviewer, the longer the interval; the opposite was true for authors.

Effectiveness was modelled by the number of issues discussed during review using a generalized liner model (GLM) with Poisson errors and a logarithmic link function. A quasi-possion model and a pseudo R-squared measure are used to correct for dispersion. The variance accounted for ranged from 16% to 58%. As was the case with review interval, the amount of participation (*i.e.* number of replies) was positively related to the number of issues discussed. Unsurprisingly, the greater the complexity of a contribution the more issues discussed. The experience and expertise of the reviewer is more important than that of the author, with greater reviewer and author knowledge generally leading to more issues found and discussed.

To optimize review efficiency and effectiveness, it is important to have knowledgeable reviewers. However, the amount of variance explained by the expertise and experience variables is generally smaller than that explained by the degree of participation or interest shown by the community and the complexity of the code under review. In the next chapter we add to these findings by investigating the mechanisms and behaviours that allow knowledgeable reviewers to find interesting contributions in broadcast of contributions and other development information. The conclusions and implications of these findings are discussed in Chapter 6 where we develop our theory of OSS peer review.

Chapter 5

Understanding Broadcast Based Peer Review in OSS

In the previous chapters, we described the review processes used on 25 OSS projects, quantified the parameters of open source software (OSS) peer review, and, using statistical models, concluded that it is a successful and efficient quality assurance technique. In this chapter we examine the behaviours and mechanisms that facilitate broadcast based, OSS peer review [1].

Broadcasting reviews and other development discussion to hundreds of potentially interested individuals has both significant advantages and potentially serious weaknesses. For example, we know that a diverse set of stakeholders will see the patch, but how can we be sure that any will review it and be competent enough to find defects? Additionally, there may be too many conflicting opinions during a review. How do stakeholders interact to avoid deadlock and reach an acceptable solution? Since we examine projects of different sizes, we also examine how scale affects broadcast based peer review. In this chapter, our goal is to inform researchers, developers, and managers about the advantages and weaknesses of this style of review.

The chapter is structured as follows. In the next section, we introduce our research questions. In section 5.2, we introduce our methodology and data sources. In each subsequent section, we provide evidence regarding the techniques used to find patches for review (Q1), the impact of ignored reviews (Q2), review outcomes, stakeholders, interactions (Q3), the effect of too many opinions during a review (Q4), and scalability issues involved in broadcasting reviews to large projects (Q5). In the final sections, we discuss the threats to credibility of this study, and summarize the findings.

[1] Portions of this chapter appear in Rigby and Storey [120]

5.1 Research Questions

The research questions we explore are the following:

Q1 Finding patches to review: How do experts decide which contributions to review in a broadcast of general development discussion?

Section 3.5.2 and previous research [3] has shown that there are expert reviewers involved in most reviews on OSS projects. This past research does not explain how experienced developers find patches they are competent to review. We aim to understand the techniques used to wade through a seemingly overwhelming broadcast of information as well as uncover what motivates a developer to review a particular patch.

Q2 Too few reviewers: Why are patches ignored? What is their impact on the project?

Many patches in OSS development never get reviewed [13]. Why does this occur and how significant an issue are unreviewed patches?

Q3 Stakeholders, Interactions, and Review Outcomes: Once a patch has been selected for review, which stakeholders become involved? How do these stakeholders interact to reach a decision?

One touted strength of OSS development is the high degree of community involvement [114]. How do the author, core developers, and other stakeholders (*e.g.,* external developers and users) collaborate to create a working solution?

Q4 Too many opinions: What happens to the decision making process when very large numbers of stakeholders are involved in a review?

Parkinson's law of triviality, or the "bike shed painting" effect as it is known in OSS communities [43, 73], states that "organizations give disproportionate weight to trivial issues" [107]. Intuitively, a broadcast to a large community of stakeholders should exacerbate this law and lead to unimportant changes being discussed to a deadlock. We measure the impact of Parkinson's law on OSS projects and discuss the implications for broadcast based review.

Q5 Scalability: Can large projects effectively use broadcast based review?

On a large mailing list with a wide variety of topics, we expect that developers will have difficulty finding the information and patches pertinent to their work. To investigate the influence of project size on peer review, we examine six projects. We want to understand how the techniques used to manage broadcast information on a moderately sized project like Apache differ from a much larger project like Linux or a much more diverse project like KDE.

5.2 Methodology

Reviews occur as email threads on a mailing list. An author emails a contribution of code or other software artifact to the mailing list, and reviewers respond to this email with defects and other issues. The reviews of patches are our unit of analysis they encapsulate which developers take notice of a contribution and how they interact during the review of the contribution.

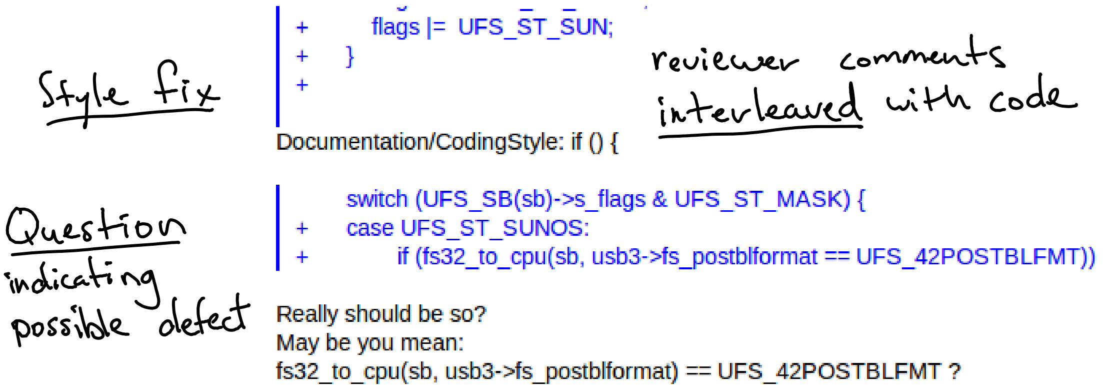
As in the previous chapter, we used a multiple case study methodology and examined the same six OSS projects. We began with an exploratory case study of the Apache httpd project, and developed a set of descriptive codes based on our manual examination of 200 Apache reviews. We abstracted these codes into themes which were validated by a core developer on the project. We used theoretical sampling to select projects as literal and contrasting case study replications [142, 29] (See Section 2.3.2 for more details) and examined 300 instances of peer review across the remaining projects. After we completed the coding of reviews from all six projects, we developed a semi-structured interview based on the themes that emerged and interviewed core developers from each project. These interviews allowed us to test, qualify, and enrich our findings. While the focus of this chapter is on descriptive themes, we were able to develop measures of certain themes. Each measure relies on an understanding of the associated theme, so we defer the discussion of these measures and their limitations to the section in which they are used.

Analysis Procedure

We randomly sampled and manually analyzed 200 reviews for Apache, 80 for Subversion, 70 for FreeBSD, 50 for the Linux kernel, 40 email reviews and 20 Bugzilla reviews for KDE, and 20 email reviews and 20 Bugzilla reviews for GNOME. Saturation of the main themes occurred relatively early on, making it unnecessary to code an equivalent number of reviews for each project [52].

The analysis of these reviews followed Glaser's [52] approach to grounded theory where manual analysis uncovers emergent abstract themes. These themes are developed from descriptive codes used by the researcher to note his or her observations. Examples of coded reviews and the themes that emerged can be found in Appendix E. Our analysis process for Apache proceeded as follows:

1. The messages in a review thread were analyzed chronologically. Since patches could



```
+       flags |=  UFS_ST_SUN;
+   }
+
Documentation/CodingStyle: if () {

        switch (UFS_SB(sb)->s_flags & UFS_ST_MASK) {
+       case UFS_ST_SUNOS:
+               if (fs32_to_cpu(sb, usb3->fs_postblformat == UFS_42POSTBLFMT))

Really should be so?
May be you mean:
fs32_to_cpu(sb, usb3->fs_postblformat) == UFS_42POSTBLFMT ?
```

Style fix
reviewer comments interleaved with code
Question indicating possible defect

Figure 5.1: Example fragment of review with three codes written in the margins: A type of fix, a question that indicates a possible defect, and interleaved comments.

often take up many screens with technical details, we first summarized each review thread. The summary uncovered high-level occurrences, such as how reviewers interacted and responded. The summaries were written without any interpretation of the events [52].

2. The reviews were coded by printing and reading the summaries and writing the codes in the margin. The codes represented the techniques used to perform a review and the types and styles of interactions among stakeholders. The example[2] shown in Figure 5.1 combines the first two steps with emergent codes being underlined.
3. These codes were abstracted into memos which were sorted into themes. These themes are represented throughout the chapter as paragraph and section headings.
4. A draft of the themes was given to the president of the Apache Foundation, Justin Erenkrantz, for conceptual validation.

This process was repeated for the other five projects, with the validation step replaced by interviews with experienced reviewers from each of the projects. While these interviews were originally intended as a validation of the coded review findings, new themes did emerge from the interviews and are incorporated into the chapter. These interviews were conducted after the coding and memoing of reviews had been completed for all projects.

The semi-structured interview questions, which can be found in Appendix D, were based on the themes that emerged from the manual coding of reviews on all projects. To analyze the interviews, we coded the email responses (or paraphrased interviews in the case of

[2] The full example review can be found at http://www.mail-archive.com/sparclinux@vger.kernel.org/msg01475.html

telephone or in-person interviews) according to grounded theory methodology – new themes and qualifications of old themes did emerge. See Appendix D for an example of a portion of a coded interview.

To select interviews for the Apache and Subversion projects, we had Erenkrantz send an email on our behalf to the respective developer mailing lists. Not knowing any developers on the remaining projects, we ranked developers based on the number of email based reviews they had performed and sent an interview request to each of the top five reviewers on each project. All these individuals were core developers with either committer rights or maintainers of a module. Initially, we asked for a phone interview; however, most developers refused to be interviewed by phone, though they were willing to answer questions by email. This was not particularly surprising since OSS developers are very experienced with and prefer communication via email. Most responses were detailed, and most developers quickly responded to follow-up questions.

Overall, we interviewed ten developers, not including Erenkrantz$_{AE}$[3] from Apache who validated a draft of our themes. Some developers were willing to be quoted and are referred to by project and name, while others wished to remain anonymous and are referred to by project only. We were able to interview one core developer from Apache in person, who preferred to remain anonymous$_{A1}$, and one Subversion developer (Wright$_{SW}$) over the phone. The remaining developers were interviewed via email: three Linux maintainers (Gleixner$_{LG}$, Iwai$_{LI}$, and Kroah-Hartman$_{LK}$) two core developers from FreeBSD who preferred to remain anonymous$_{F1,F2}$, two core developers from KDE (Faure$_{KF}$ and Seigo$_{KS}$), and one core developer from GNOME (Moya$_{GM}$). Interviewees were all long-serving core developers on their respective projects.

In summary, the analysis in this chapter focuses on the manual coding of 500 review discussions across six projects, as well as our interviews with ten core developers. We also use simple measures to assess the impact of certain themes. Throughout this chapter, named themes are presented as section and paragraph headings.

[3] To provide a quick reference to interviewees we use a subscript of the first letter of the project and first letter of the interviewee's last name or a number in the case of an anonymous interviewee *e.g.,* Erenkrantz of Apache becomes $_{AE}$

5.3 Finding patches to review

Q1: How do experts decide which contributions to review in a broadcast of general development discussion?

In contrast to an industrial setting, reviews are not assigned in OSS projects. In Section 3.4, we have shown that although there is a potentially large number of reviewers (*i.e.* everyone subscribed to the mailing list), only a small number of reviewers (usually two reviewers) are actually involved in any given review. We also showed, in Section 3.5, that the typical reviewer is a long-standing committer in the area of the project to which the patch under review belongs.

Interviewing core developers allowed us to understand why developers decide to review a particular patch. We asked, "How do you determine what to review?", all our interviewees said that they review patches that are within their area of interest and expertise that they see as important contributions.

> "Mostly [I review because of] interest or experience in the subject area [of the patch]."
>
> Anonymous, FreeBSD, Interview$_{F2}$

Additionally, developers who have contributed significantly to an area of the code base often become the defacto owner, or in a large project, the official maintainer, assuming a degree of responsibility and a sense of obligation to perform reviews in that area$_{A1,SW,LG,LI,LK,F1,F2,KS,KF,GM}$.

> "[I review] Because usually it [the patch] is for code I maintain and I have to review it. Or it uses an api/subsystem that I am responsible for."
>
> Kroah-Hartman, Linux, Interview$_{LK}$

For maintainers, if a patch is accepted, it will affect their work and in many cases they will have to maintain and evolve the code once it is added to the system. As a result, the maintainer's opinion can be worth more than the opinion of other developers$_{A1,LG,LI,LK,F1,GM}$.

> "I feel a sense of responsibility to ... take a look at it [the patch]. If that goes in then I'd have to do work on my side, so I'll try and make sure that I'm involved in the review of that [patch] ... the people who have been working on the project for a long time are generally accorded a sense of respect. If one of them weights in with a negative comment, then you're likely to take it pretty seriously"
>
> Anonymous, Apache, Interview$_{A1}$

While we now have an understanding of how developers decide which patches they will review, it is surprising to us that developers are able to find these contributions in a large and general broadcast of development discussion and code changes. Some patches are assigned to or sent to specific developers, a topic we quantify and discuss in detail in Section 5.7.2, but for the most part, developers self-select broadcast patches for review. In this section we present the techniques that core developers use to find and refind reviews: email filters, progressive detail within patches, an interleaved and full history of past discussions, and recipient building based on past interest.

5.3.1 Filtering

All interviewees agreed they receive an overwhelming quantity of broadcast information via email on a daily basis, the entirety of which they cannot process manually. Core developers reported they are subscribed to between 5 and 30 mailing lists, with most developers receiving more than 300 messages daily. This quantity of correspondence is drastically larger than the number of emails received in inboxes studied by Fisher *et al.* [42], Whittaker and Sidner [137], and Mackay [88] where individuals received between a median of 40 and 60 emails per day. However, these studies examined personal email, where one is usually expected to respond to most messages. In contrast, community based broadcasts rarely require a response. As illustrated by the following quotations, broadcasting emails allows developers to maintain a general awareness of the activity on the project[55, 141].

> "I tend to monitor the development list discussion and I participate fairly frequently in the ones I care about. And even the ones I don't care so much about, I at least try to [conceptually] know what's going on."
>
> Wright, Subversion, Interview$_{SW}$

> "If anything, there is almost too much awareness"
>
> Anonymous, Apache, Interview$_{A1}$

To avoid becoming overwhelmed, most interviewees separate "interesting and important" emails from general list traffic by using email filters. Three exceptions where A1, F2, GM who scanned all email subjects manually – "I generally use the bad approach of trying to read and understand everything."$_{F2}$ Since email archives do not record the filters used by developers, the following result is based exclusively on interviews. Interesting email is placed into folders that are checked on a regular basis. Some folders are kept simply as a location where "quick" searches on a particular topic can be performed$_{KF}$. There are varying levels of filter sophistication, listed from simple to complex below:

- Separate tags or folders for each mailing list
- Differentiation between mail sent directly to a developer (*e.g.,* developer is on Cc list) and mail sent generally to the list$_{LG,KF,KS}$
- Differentiation of emails that come from certain people or "important" individuals$_{LI,LK,KS}$
- Parsing the subject for interesting keywords, usually related to areas maintained by the developer$_{LG,LI,F1}$
- Parsing email for changes to files that belong to the developer's area of interest$_{LG}$

These filtering techniques reduce information overload and allow for mail to be sorted according to interest:

> "Because of those filters, no, I do not feel overwhelmed [by the Linux Kernel Mailing List (LKML)]. On subsystem-specific mailing lists (like the linux-usb or linux-pci mailing lists), I read all emails"
>
> Kroah-Hartman, Linux, Interview$_{LK}$

Within this filtered set of emails, there are still many emails that have potential relevance for a particular developer. The subject, format of the patch, and certain properties of email allow developers to identify which patches to review.

5.3.2 Progressive Detail

The author's email, which contains his or her patch, has three components that progressively provide increasingly detailed information to potential reviewers (*i.e.* progressive detail): the subject, the change log, and the file differences. Our evidence for the importance of patch structure comes from the project policies describing the structure (See Chapter 2), instances where we saw authors rebuked for not following the correct format, and the interviewees$_{LG,LI,LK,KF,KS}$ who said that a well structured patch made it easier to review. Figure 5.2 shows the structure of a patch through an example. The following quotes are taken from our interview with Gleixner$_{LG}$.

Subject: "A good and descriptive subject line draws immediate attention."

The email subject provides an overview of the problem. All interviewees remarked on the importance of subject scanning for finding interesting patches.

Change log: "The change log message is equally important as it helps one to understand why a particular change is necessary, which problem it fixes or how the new feature is supposed to work."

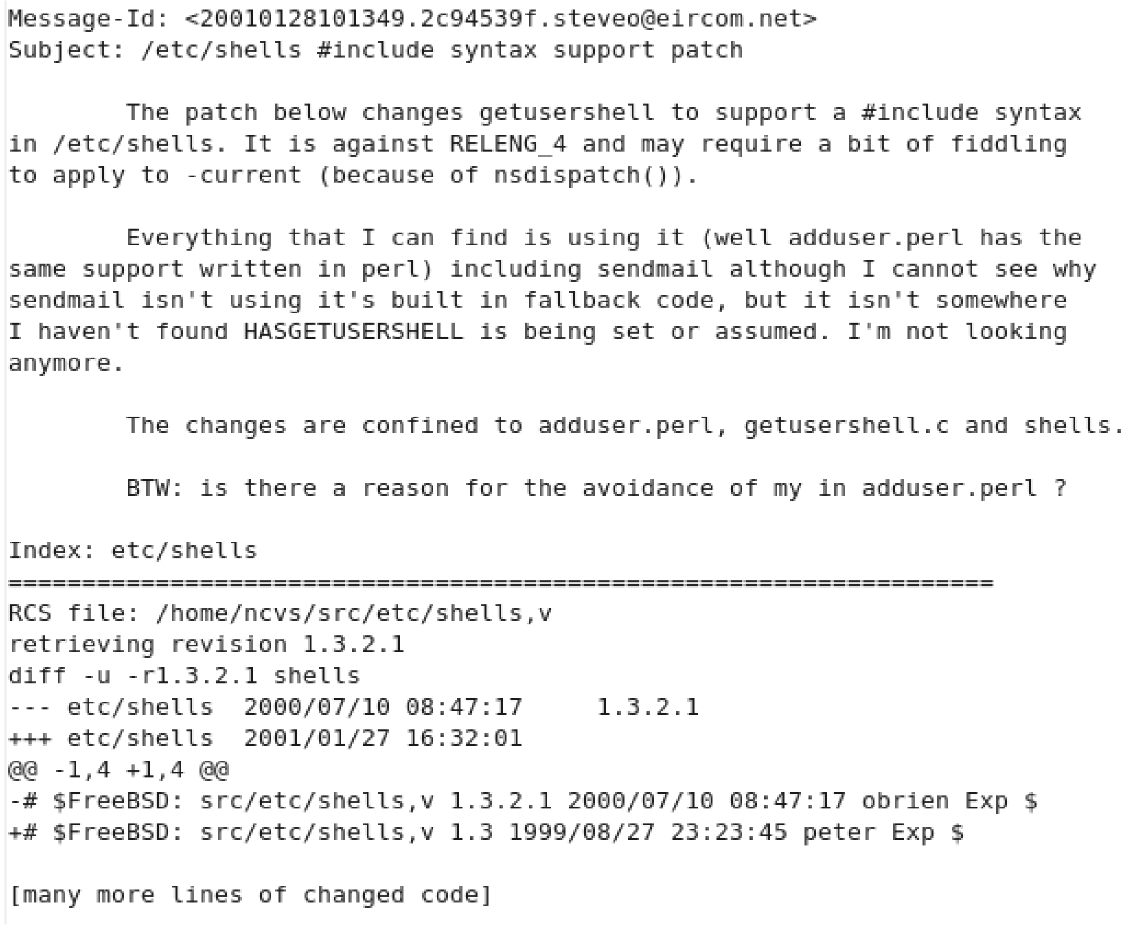
The change log introduces the developer to the problem and allows them to determine at an abstract level whether the change is worth reviewing. While some projects require detailed change logs (*e.g.,* Linux and Subversion), others do not. Our interviewees had differing opinions about the importance of the change log with some preferring to skip it and look at the changes immediately.

File changes: "The diff [the code change] itself should be short, *i.e.* large patches should be split into logical and functional steps. That makes reviewing easier."

The file differences, the essential part of the patch, are the actual changes to the files and are the only aspect some reviewers look at$_{F1,GM}$.

By progressively providing increasing levels of detail, reviewers can quickly determine (1) if they have the necessary interest, skill, and time to perform a detailed review (*e.g.,* if the reviewer does not understand or is uninterested in the subject, they need not look at the file changes) and (2) if the contribution is at a sufficient level of quality to warrant a detailed review.

This second aspect of progressive detail allows developers to quickly ignore patches that are not of sufficient quality. Instead of immediately examining the file differences and performing a full review, the reviewer can assess the commit log and determine at a high level if there are any issues. Once a reviewer finds a significant problem, there is no need to actually examine the code until the problem is addressed. We saw many patch reviews revolve around a critical comment relating only to the change log. As we will discuss in later sections, these "obvious" mistakes often indicated that the author is not competent. This assessment of competence is particularly important in OSS development where developers rarely meet face-to-face and must build trust exclusively through the quality of previous contributions.



```
Message-Id: <20010128101349.2c94539f.steveo@eircom.net>
Subject: /etc/shells #include syntax support patch

        The patch below changes getusershell to support a #include syntax
in /etc/shells. It is against RELENG_4 and may require a bit of fiddling
to apply to -current (because of nsdispatch()).

        Everything that I can find is using it (well adduser.perl has the
same support written in perl) including sendmail although I cannot see why
sendmail isn't using it's built in fallback code, but it isn't somewhere
I haven't found HASGETUSERSHELL is being set or assumed. I'm not looking
anymore.

        The changes are confined to adduser.perl, getusershell.c and shells.

        BTW: is there a reason for the avoidance of my in adduser.perl ?

Index: etc/shells
===================================================================
RCS file: /home/ncvs/src/etc/shells,v
retrieving revision 1.3.2.1
diff -u -r1.3.2.1 shells
--- etc/shells  2000/07/10 08:47:17     1.3.2.1
+++ etc/shells  2001/01/27 16:32:01
@@ -1,4 +1,4 @@
-# $FreeBSD: src/etc/shells,v 1.3.2.1 2000/07/10 08:47:17 obrien Exp $
+# $FreeBSD: src/etc/shells,v 1.3 1999/08/27 23:23:45 peter Exp $

[many more lines of changed code]
```

Figure 5.2: The structure of a patch on FreeBSD

5.3.3 Full and Interleaved History

When individuals perform reviews, they respond by extracting problematic sections of the review request email. Critiques or issues are written directly under the code section that has a potential defect and irrelevant text is removed (See Figure 5.1 and Appendix E for examples). This technique is known as "interleaved posting" and is widely used in email communities[4]. As author and reviewers discuss a problem, a stack of quoted text with responses creates a refined focus and a discussion around a particular problem. Indeed, there may be multiple related and unrelated discussions in a single email thread. Seigo noted that the inability to interleave comments within code was one of the significant weakness of performing reviews in Bugzilla$_{KS}$.

[4] It is often referred to as "bottom posting" http://en.wikipedia.org/wiki/Posting_style

By viewing the extracted sub-problems presented in chronological order, a reviewer can quickly form an opinion regarding an issue. The reviewer can also assess which issues have or have not been discussed, and whether the current participants are making progress towards a solution. This history is of great utility to developers on large projects where interviewees admitted they are unable to keep up with all discussions they are potentially interested in$_{LG,LI}$. In contrast, interviewees on smaller projects, such as Apache and SVN, stated they have little trouble keeping track of daily discussion and have limited need for email thread histories$_{A1,SW}$

5.3.4 Refinding: Recipient Building

As we have seen, developers respond to contributions when they are interested in or feel a sense of obligation to review a patch. However, as the author revises his or her contribution, it would be very inefficient to refind this discussion thread in the breadth of broadcast information each time a change is made. The recipient fields (*i.e.* the ‘To’ and ‘Cc’ fields) provide a simple and natural mechanism for remaining aware of the development of a thread once interest has been indicated$_{SW,LG,LI,LK,F1,KF,KS}$. Unlike private email for which the recipients of a message remain the same unless a new recipient is explicitly added, on mailing lists the number of recipients grows with the number of distinct people that respond. Once interest has been indicated by a response to the mailing list, subsequent responses are sent not only to the mailing list but also specifically to the growing number of developers in the recipient fields. As we discussed above, interviewees pay more attention to emails sent to them directly than to those sent to the general list. These direct emails are responded to in a manner similar to private email.

Recipient building also allows for a subdivision of discussion within a single email thread. An example of this is the case where there exists a long discussion concerning a particular contribution, but a reviewer finds an unrelated problem. The reviewer responds not to the most recent email, which is irrelevant to his or her problem, but to the original email. In this way, a new recipient group around the discovered sub-problem forms without distracting from the pre-existing discussion. This natural subdivision is especially important in large projects (see Section 5.7.2 for further discussion of recipient building).

In summary, we have learned that developers select contributions to review based on their interest in the area and in some cases, a developed obligation. We also discussed the techniques that allow reviewers to locate contributions they are competent to review and to remain aware of future discussion: email filters, progressive detail within contributions, a full and interleaved history of past discussions, and recipient building based on interest in a topic. These techniques were present in all six projects examined. In the next section, we describe what happens when a contribution is neglected.

5.4 Ignored patches: Too few reviewers

Q2: Why are patches ignored? What is their impact on the project?

Previous work has demonstrated that a large number of patches posted to mailing lists are ignored [13, 3]. In this section, we examine what effect ignored patches have on an OSS project.

Although the traditional pressures of finishing a meeting on time no longer exist in an asynchronous style of peer review, developers on OSS projects are still restricted by the amount of time they have to spend on a given project. When asked how time pressures affect the review process, interviewees stated that quality does not suffer, but that reviews of contributions are sometimes postponed or dropped$_{A1,LG,F1,KF,GM}$. The following quotation summarizes the finding:

> "Lack of time can postpone the actual reviewing, but not really have an effect on whether to review or the quality of it."
>
> Faure, KDE, Interview$_{KF}$

> " I'm not sure it [lack of time] affects the quality of review for me near as often as it affects whether or not I will review something."
>
> Anonymous, FreeBSD, Interview$_{F1}$

While the postponement of a review (and sometimes permanent "postponement", *i.e.* ignored low priority patches) may seem an obvious solution, it is in stark contrast to what normally happens in industry: the goal of simply reviewing everything is taken over the expense of review quality [38], or code is added to the system without review when a deadline approaches and more pressing activities, *e.g.,* creating a workaround so the system can ship, replace longterm activities, *e.g.,* peer review [122]. The lack of extreme deadlines in OSS development and the reduced importance of monetary work incentives appears to produce an environment where proper reviewing is seen as more important than, for example, ensuring the latest feature makes it into the product.

However, by postponing reviews, the burden of making sure a patch is reviewed and committed by a core developer lies with the author, not the reviewer. While this may be frustrating for the author, it means the reviewer, who is usually more experienced and pressed for time, is simply "pinged" about, or reminded of the contribution by the author periodically$_{LG}$ (See Policy Chapter 2). An author's patch will only be reviewed once it is considered by a core developer to be sufficiently interesting in relation to his or her other work$_{A1,SW,LG,LI,LK,F1,F2,KF,KS,GM}$. Inexperienced authors can be intolerant of this waiting period and may become frustrated or discouraged and provoke core developers. The following quotation is extracted from a response to a FreeBSD mailing list discussion that was started by Kamp[73]. It illustrates the sentiments of one annoyed developer:

> "There is no sense in wasting the time of one informed developer to help one uninformed developer; this is a bad tradeoff unless the uninformed developer is showing signs of promise . . . Are you [the inexperienced author] willing to accept that you may have been judged 'not worth the effort' on the content of your questions [and so are ignored]?"

Textual communication is known to remove social cues and may lead to individuals responding in what appears to be a negative or even confrontational manner [126]. However, the interviewees we asked about list manners said that they were important$_{LG,LI,LK}$ and that they make a concerted effort to be polite and to respond to authors$_{LG,F1}$. During our manual analysis, we found that authors who were overconfident, rude, or displayed a sense of entitlement were frequently ignored, "flamed", or made an example of. Developers who fixed authentic problems or conducted themselves modestly usually received some form of response, even if only to point out that their contribution was redundant or outside the scope of the project.

To reinforce the notion that reviewing patches from new developers was not worthwhile, we saw instances where a polite detailed review was ignored by the author[5]. This lack of follow-through and apparent lack of effort demonstrated by some newbie authors likely aggravated core developers and made them less likely to spend time reviewing patches from new developers in the future.

There are instances where interesting contributions can be inadvertently lost, *i.e.* ignored, in the 'noise' on a mailing list. As we have seen, it is common project policy to ask for

[5] For example see KDE review http://markmail.org/message/cie4h7pxxoookvck

ignored contributions to be reposted. Inadvertently ignored contributions are one of the stated reasons that projects start using bug tracking systems. However, when we examined 40 bugs on KDE's and GNOME's Bugzilla we found that bug tracking may not be the solution. Although external developers' bugs were recorded and often discussed in Bugzilla, many had the complaint, "This still hasn't been committed!" For some bugs, a core developer would respond "I'll look at this tomorrow" and then never respond again. Although tracking is certainly an aspect of the ignored review problem, our findings indicate that the time, priorities, and interest of the core development team are the more central issues.

To summarize, code contributions that do not generate sufficient interest within the core development team of an OSS project are usually ignored. It is up to the author to resend his or her patch to the mailing list until a response is received. Tracking mechanisms do not appear to solve the problem of ignored reviews since reviewer time and interest are the main obstacles.

5.5 Stakeholders, Interactions, and Review Outcomes

Q3:Once a patch has been selected for review, which stakeholders become involved? How do these stakeholders interact to reach a decision?

As Votta [82] and Sauer et al. [122] make clear through both empirical investigations of software inspections and comparisons to the literature on group behaviour, group synergy does not exist in traditional inspection meetings. Extreme discipline and well defined roles inhibit teams from adapting to individual member strengths. The single goal of a traditional inspection meeting is to find defects; no discussion of a fix is allowed during the meeting [38]. Furthermore, time pressures can have the unintended effect of changing the goal from finding all defects to finishing the meeting on time [82].

In the previous section, we discussed how OSS developers are allowed to naturally gravitate towards contributions they are interested in or responsible for, and are competent to review. In this section, we first discuss the types of patches are review outcomes. We then describe stakeholder characteristics and ways that stakeholders with different skill sets and individuals of varying status participate in a review.

5.5.1 Patch Types and Review Outcomes

Our interviewees provided a useful classification of patches$_{SW,LG,GM}$:

(1) those that lead to a purely technical discussion, and

(2) those that lead to a discussion of project scope or architecture and whether the patch is necessary given the project's objectives.

The different patch types lead to different styles of review and different interactions among stakeholders. Purely technical patches are comparatively simple to review. The reviewer asks, "does the patch do what it claims to do and are there any flaws in the implementation?"$_{SW}$ Both author and reviewer accept that the patch is a necessary change to the system. In contrast, feature and scope reviews question the value of the contribution and often require a great deal of diplomacy. Instead of focusing on technical aspects, discussions focus on whether the change is necessary and the effect it will have on other aspects of the system. Interviewees found these discussions often became unpleasant since they usually involve personal opinion and can lead to prolonged, unproductive discussions and conflicts (see Section 5.6 for more details on unproductive discussions).

In the context of the two types of patches, we list the reasons why our interviewees rejected a patch or required further modification before accepting it.

Purely technical issues:

- Obviously poor quality code (*e.g.,* novices that are clearly out of their depth)$_{LG,KS,GM}$
- Violation of style or coding guidelines (*e.g.,* not portable code)$_{LK}$
- Gratuitous changes mixed with "true" changes (*e.g.,* re-indenting a file in the same patch as a bug fix)$_{F1}$
- Code does not do or fix what it claims to$_{LK}$
- Fix introduces redundant code (*e.g.,* novice re-invention)$_{LG}$
- Fix introduces more bugs$_{LK,KF,KS}$
- Fix conflicts with existing code (*i.e.* integration defects)$_{A1,LG}$
- Use of incorrect API or library$_{LK}$

Feature, scope, or process issues:

- Timeline prevents addition of fix or feature (*e.g.,* new code during a code freeze)$_{LI}$

- Poor design or architecture$_{LG,LK,GM}$
- Fix is misplaced or does not fit design of the system (*e.g.,* optimization for a rare use case)$_{SW,KS}$
- Addition of a large codebase or feature that no individual is willing to maintain or review (*e.g.,* new subsystem that is outside of the core developers' interest)$_{A1,F1}$
- Project politics (*e.g.,* ideology takes precedence over objective technical discussions)$_{A1,F2}$
- Addition of code that provides no significant improvement over existing code[6]

While purely technical issues dominate the discussions on OSS mailing lists, project scope and other highly political issues, such as an outsider suggesting a feature or specialization that does not fit the project's mandate, are not technically difficult but can lead to tiring, vitriolic debates. The most difficult patches combine technical issues with scope and political issues. An example is the case where a fix required significant code cleanup and undergone many revisions, but only produced minor performance improvements. Here, the individuals, including reviewers, who invested in the patch may argue with developers who are not convinced of its value. According to our interviewees, this level of conflict is rare. In our manual analysis we saw that normally, when there is a disagreement, one reviewer will remove his or her comment and endorse the view of another reviewer or the author. However, on widely used, stable projects, there is a preference by core developers to keep what works over adding a new feature with questionable value and comparatively little testing performed on it. This inclination to maintain the status quo is exemplified in the following quotation taken from our manual analysis:

> "I'm not claiming this [proposal ...] is really any better/worse than the current behaviour from a theoretical standpoint, but at least the current behaviour is _tested_, which makes it better in practice. So if we want to change this, I think we want to change it to something that is _obviously_ better."
>
> Torvalds, [132]

5.5.2 Stakeholder Characteristics

When asked, "Which roles do you see during a review?", interviewees seemed confused by our question and described characteristics or manners in which stakeholders interacted instead of roles.

[6] This value assessment was found only during manual analysis

> "[The organizational structure during review is] fairly flat: author and reviewers, with a 1:many relationship there. Reviewers tend to be the maintainers of the project in question as well as other contributors to the codebase the patch touches."
>
> Seigo, KDE, Interview$_{KS}$

In our manual analysis of reviews, two main roles emerged: reviewer and outsider. Interviewees added subtle depth further describing the characteristics and interactions during a review.

5.5.2.1 Reviewer Characteristics

Our analysis of reviews revealed several reviewer characteristics. These were supported by discussion with our interviewees. The terms used by Gleixner$_{LG}$ of the Linux project were particularly descriptive and we have used these terms: objective analyzer, fatherly adviser, grumpy cynic, frustratedly resigned, and enthusiastic supporter. We also use a further subdivision of positive and negative personas.

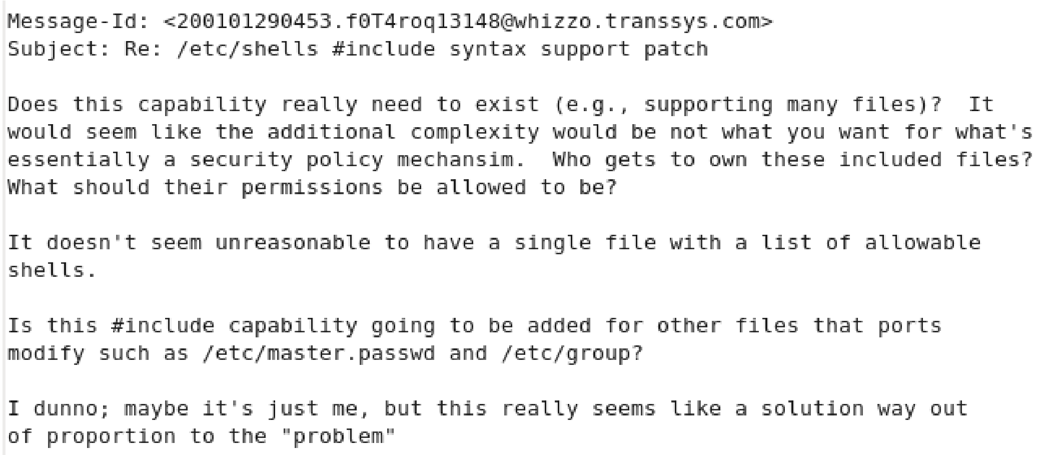
Positive personas:

Objective analyzer. All reviewers are expected to provide a disinterested analysis of the patch. This style of analysis is the goal and intended purpose of peer review. Interestingly, reviewers often provide their criticisms as questions (See Figure 5.1), which often avoids adversarial behaviour and encourages objective discussion, use cases, and "what if" scenarios (See Figure 5.3).

Expert or Fatherly adviser. New developers may not understand all the subtleties and quirks of both the system and the development community$_{KF}$. This experience is built up over time. Expert reviewers often provide *ad hoc* mentoring$_{SW,LG}$ to novices by, for example, pointing out dead ends, providing links to email discussions of related past issues, and passing on the culture of the community[7]. Also, if the author and current reviewers are making little headway on a solution, a third party, usually an experienced developer, can read over the existing discussion and provide, often at a conceptual level, an insight that leads to the solution.

However, in some cases, the author may fail to understand and implement the required changes. It is usually easier for the "fatherly adviser" to simply rewrite the code himself$_{LG,F2,KS}$. This often happens when the patch is trivial, requiring more effort to fix it than to just rewrite

[7] For an example of mentoring see the KDE review thread at http://markmail.org/message/bvttneosh7t63tnn



```
Message-Id: <200101290453.f0T4roq13148@whizzo.transsys.com>
Subject: Re: /etc/shells #include syntax support patch

Does this capability really need to exist (e.g., supporting many files)?  It
would seem like the additional complexity would be not what you want for what's
essentially a security policy mechansim.  Who gets to own these included files?
What should their permissions be allowed to be?

It doesn't seem unreasonable to have a single file with a list of allowable
shells.

Is this #include capability going to be added for other files that ports
modify such as /etc/master.passwd and /etc/group?

I dunno; maybe it's just me, but this really seems like a solution way out
of proportion to the "problem"
```

Figure 5.3: A FreeBSD example of a review that questions the necessity of a patch and the way it was implemented. Different scenarios and possibilities are also described.

the naive patch properly from scratch. This also provides the novice with an example of proper implementation.

Enthusiastic supporter or Champion. When the author of a patch is not a committer (*i.e.* not a core developer), an enthusiastic reviewer or champion must shepherd the patch and take responsibility for committing it. If no one champions an outsider's patch, it will not be committed (See policy Chapter 2.)

Champions are also present when it is unclear what the best solution is. In this case, different developers put their support behind different competing solutions, indicating the feeling of various segments of the community.

Negative personas:

Grumpy cynic. Experienced project members can become cynical when, for example, novices repeatedly suggest a "solution" that has been explored in the past with an outcome of failure. "I usually try to be calm and polite, but there are times where sarcasm or an outburst of fury becomes the last resort."$_{LG}$

Frustratedly resigned. When a review discussion has been carrying on for an extended period of time, a reviewer may simply resign from the discussion and refrain from commenting. Since "silence implies consent" in OSS communities [119], the developers who continue the discussion ultimately make the decisions that influence the system. When any reasonable solution will suffice, the most effective approach is to leave the discussion. These fruitless discussions are discussed in detail in Section 5.6.

5.5.2.2 Outsiders

Having a large community of knowledgeable individuals, outside of the core development team, who can provide feedback and help with testing is one of the strengths of OSS development [114].

However, unlike a core developer who has been vetted and voted into the inner circle or "appointed" as a maintainer of a subsystem, outsiders come from a wide swath of experience and expertise, which can vary drastically from that of the core development team. These differences can be both an asset and an obstacle. The interviewed core developers had mixed feelings about the contributions from outsiders. Outsiders made positive contributions by suggesting real world use cases and features$_{A1,SW,LG,GM}$, providing expert knowledge that the core team lacks$_{A1,F1}$, testing the system and the APIs$_{SW}$, providing user interface and user experience feedback$_{F1}$, and filing bug reports$_{LG,F2,GM}$ and preliminary reviews$_{KF,KS}$. According to interviewees, negative contributions came in the form of, often rude, request for features that had already been rejected for technical reasons$_{LG}$, as well as use cases and code optimizations that were too specific to be relevant to the general community$_{SW}$.

The themes of competence and objectivity emerged from the interviews and manual analysis as attributes that make an outsider an important contributor to OSS projects$_{SW,LG,F1,F2}$. Competence need not be purely technical when it comes from an outsider, but it is usually in an area that is lacking in the core development team. In general, non-objective opinions are shunned in this meritocratic community. The following quotation illustrates the feelings of core developers towards outsiders:

> "If the [outsider] input is objective, it's welcome. Unwelcome input are from uninvolved (*i.e.* incompetent) people who insist on requesting features which have been rejected for technical reasons."
>
> Gleixner, Linux, Interview$_{LG}$

While outsiders are involved in many reviews, any individual outsider will participate in few reviews (See quantification of outsider influence in Sections 5.6 and 5.7.1) and will be of limited help$_{LI,F1,F2,KF,GM}$. However, there are a small number of outsiders that are critical to the success of an OSS project$_{SW,LK,F1}$[8]. These outsiders, which were seen by some of our interviewees and during our manual analysis, are aspiring or "new" developers, external developers, and "power" users.

[8] For an example of an outsider see the GNOME discussion at http://markmail.org/message/gs6qz6id5yr42zky

Aspiring or "new" developers who are showing promise, but have not yet been given commit privileges or made a maintainer can make important contributions to the project.

> "As far as 'new' developers who are establishing themselves in the community, they can provide review comments across a wide spectrum. Some of them have experience with similar software in industry, so they may be able to offer quite advanced and detailed review on complex topics from the start. Other folks may start off with simpler feedback on less complex patches."
>
> Anonymous, FreeBSD, Interview$_{F1}$

External developers$_{SW,F1}$ typically write code for a subproject that depends on the software being developed by the main project. For example, in the Apache foundation, the httpd server is used by other modules that have a larger and faster growing code base than the httpd server itself. Developers working on these dependent modules find bugs and provide use cases that are based on objective, real world experience. They also often provide competent code and reviews during the resolution of a bug that affects their project.

"Power" users$_{SW}$ come from industrial environments where a particular system is used on a very large scale. As a result, these users are typically very competent, although they usually do not write code, and have a significantly different perspective from the core development team. They have useful input when new or significant features are being discussed. They also prevent the core team from only considering its own use cases. The "power" user provides an expanded viewpoint that is helpful when a project is large and mature with significant market share. While "normal" users can also be helpful, they are often lacking the required competence to participate in a meaningful and objective manner.

In summary, the shift away from explicit roles and time constrained meetings has allowed peer review to move from a purely defect finding activity to a group problem solving activity that recognizes and harmonizes the varying strengths of participants. Authors must, however, capture the attention of core developers in order to ensure their contribution is considered. This leads to a patch being assessed in either a purely technical manner or in context of a project's scope. Reviewers assume one of several personas, depending on the type of patch under review. Competent, objective outsiders may also contribute knowledge that the core development team is lacking.

5.6 Bike shed discussion: Too many opinions

Q4: What happens to the decision making processes when very large numbers of stakeholders are involved in a review?

A disadvantage of broadcasting to a large group is that too many people may respond to a patch, slowing down the decision process. In traditional organizations, this problem is known as Parkinson's Law of Triviality. It has been repopularized in the OSS community by Kamp as the "painting the bike shed" effect [73] – it does not matter what colour a bike shed is painted but everyone involved in its construction will have an opinion.

Our interviewees were aware of the bike shed effect and many referred to it explicitly$_{A1,SW,LG,F2}$. They indicated a dread for these types of discussions and made every effort to avoid them. Some projects have explicit policies on how to avoid the bike shed effect; for example, on the SVN project, when a core developer says "this discussion is bike shedding," core developers are supposed to ignore any further conversation on the thread[43]. Interviewees stated that this level of conflict was rare$_{A1,SW,F1}$ and usually involved uninformed outsiders$_{LG,LI,F2}$. However, we wished to quantify how influential outsider opinions are in OSS and how often bike shedding does occur.

To create a measure of the bike shed effect, we first note that in Fogel's [43] experience, "bike shed" problems are identifiable by a lack of core group members in a large discussion thread. We divide all stakeholders involved in all reviews into two categories: core developers, who have commit privileges, and outsiders, who do not[9].

If the bike shed effect is a significant problem on OSS projects, we would expect to see many reviews dominated by outsiders. Table 5.1 shows that the number of messages sent by outsiders only exceeded that of the core development group between 2% (in the case of FreeBSD) and 5% (in the case of Apache) of the time. However, outsiders were involved in between 13% (for FreeBSD), and 30% (for Apache) of all review discussion.

We also create an "outsiders' influence" metric by dividing the number of outsider messages by the total number of messages in the review thread. Examining threads that had at least one outside reviewer, we found that outsider influence was weakly correlated with threads that lasted for a *shorter* period of time ($-0.17 \leq r \leq -0.32$), moderately correlated with *fewer* total messages ($-0.38 \leq r \leq -0.60$) and strongly correlated with *fewer* mes-

[9] We were unable to use this measure for Linux, there is no central version control repository and commit privileges are not assigned, and for the GNOME, there was no clear linking between commits and email addresses

Table 5.1: The influence of outsiders during the review process. The number of outsiders involved in a review. The number of reviews were outsiders make up the majority of participants. The correlation between the "outsiders' influence" measure and the review interval, the total number of messages, and the number of messages sent by core deveopers.

	Apache	Subversion	FreeBSD	KDE
Percentage of reviews:				
Outsiders involved	30%	23%	13%	18%
Outsiders majority	5%	3%	2%	4%
"Outsiders' influence" measure correlated with:				
Time for review	$-.17$	$-.24$	$-.32$	$-.20$
Total messages	$-.53$	$-.59$	$-.60$	$-.38$
Core-dev messages	$-.76$	$-.77$	$-.77$	$-.63$

sages from core developers ($-0.63 \leq r \leq -0.77$) [10]. These negative correlations indicate that outsiders receive less attention than core developers.

While outsiders are clearly involved in a significant number of reviews, the "bike shed" effect, which Fogel defines as long discussion that involve few core developers, does not appear to be a significant problem during OSS review. Our findings appear to support Bird et al.'s [14] finding that individuals with lower status in an OSS community (*i.e.* outsiders) generally receive little attention from the community.

5.7 Scalability

Q5: Can large projects effectively use broadcast based review?

Intuitively, broadcasting information to a group of individuals should become less effective as the number and range of discussions increases. Our goal is to understand if and how large projects, *i.e.* Linux, FreeBSD, KDE, and GNOME (see Figure 5.4), have adapted the broadcast mechanism to maintain awareness and peripheral support of others, while not becoming overwhelmed by the number of messages and commits. The themes of multiple topic-specific mailing lists and explicit review request to developers emerged on the periphery of our previous analyzes as potential effective scaling techniques. In this section, we focus on these themes and develop research questions that allow us to understand the

[10] All Spearman correlations are statistically significant at $p < 0.01$

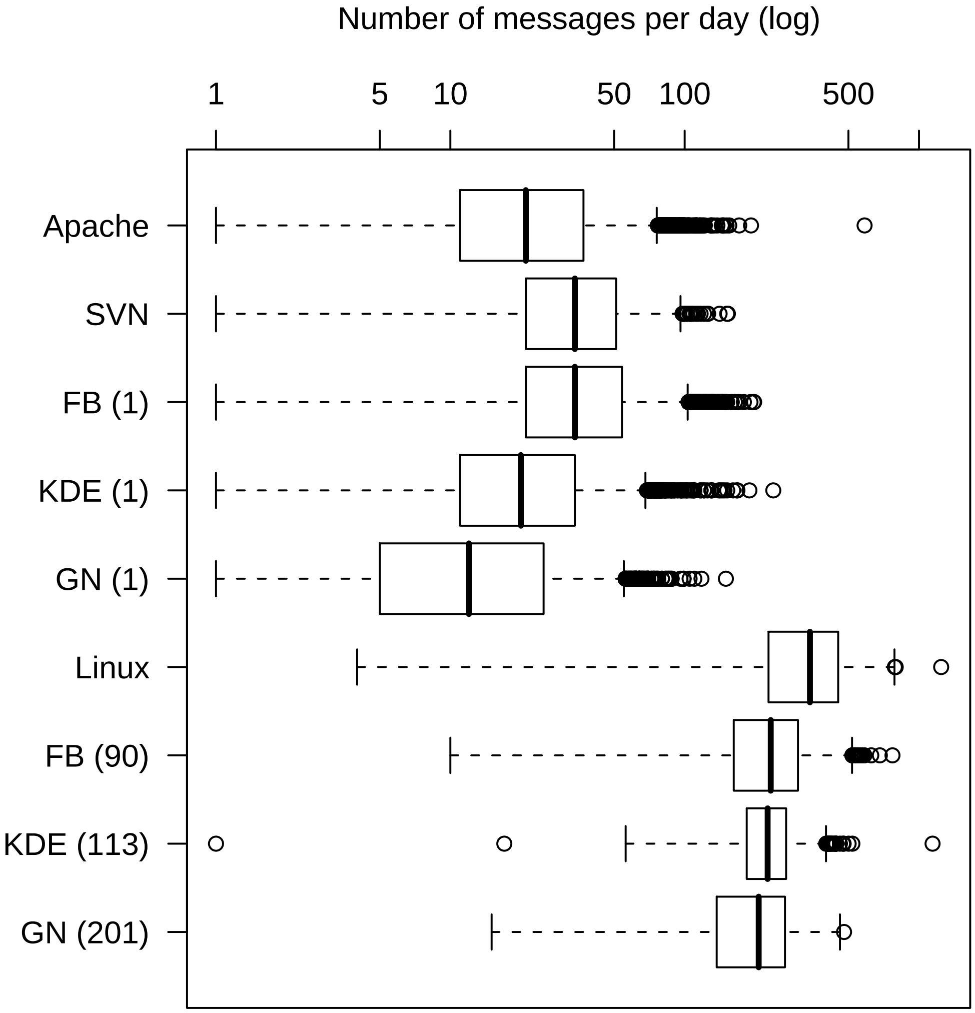


Figure 5.4: Scale of the project by emails sent daily. For FreeBSD, KDE, and GNOME we show the sum across all 90, 113, and 201 mailing lists, respectively, as well as the busiest mailing list on each project (*i.e.* FB (1), KDE (1), GNOME (1)). For example, the median number of messages per day for Apache is 21, for FreeBSD it is 233, and for Linux it is 343. The single busiest FreeBSD list (*i.e.* FB(1)) has a median of 34 messages per day.

advantages and disadvantages of using each technique to manage the review process on large projects.

5.7.1 Multiple Lists

Large projects have more than one mailing list that contains review discussions. Lists have specific charters that isolate developers working on a specific topic or set of related topics into a workgroup$_{A1,SW,LG,LI,LK,KF,KS,}$. These divisions also allow developers to create mail filters based on the mailing lists in which they are interested, as discussed in Section 5.3.1.

Do multiple mailing lists reduce the amount of traffic on individual lists?
Figure 5.4 shows that FreeBSD has 90 mailing lists with an overall median of 233 messages per day, while its busiest single mailing list has a median of 34. KDE has 113 mailing lists with an overall median of 226 messages per day, while its busiest single mailing list has a median of 20. GNOME has 201 mailing lists with an overall median of 207 messages per day, while its busiest single mailing list has a median of 12. The development lists for Apache and SVN have a median of 21 and 34 messages, respectively. Some of our interviewees discussed how they were able to keep up with all the messages on topic specific lists$_{A1,SW,LI,LK,F2}$ (See quotation in Section 5.3.1). With the exception of the Linux Kernel Mailing List [11], which has a median of 343 messages per day and is discussed in the next section, it appears that while the total amount of traffic on large projects is unmanageable for any individual, an individual should be able to follow all the traffic on a specific list.

Do topic specific lists become isolated and lose valuable outsider input?
One worry with list specialization is that developers who are peripherally related to a discussion will not be made aware of it since they are not subscribed to that particular list. In this way, valuable outsider input could be lost. Large projects dealt with this issue by having individuals who bridge multiple lists.

There are two types of bridging: discussion threads that span more than one list and individuals who are subscribed and discuss topics on more than one list. Our interviewees stated that they subscribed to between 5 and 30 mailing lists (See Section 5.3) and that while cross-posting does happen, it is generally rare and only occurs when a issue is felt to be related to an aspect of the system covered by another mailing list$_{SW,LG,LI,LK,F1,F2,KF,KS,GM}$.

[11] Although Linux does many separate mailing lists, in this work we examine only one list, the Linux Kernel Mailing List (LKML).

Discussions can also be moved off more general lists to specific lists, as illustrated by the following quote:

> "We do forward messages on from "general interest" mailing lists (such as kde-core-devel or kde-devel) to more specific ones (*e.g.,* project specific ones such as plasma-devel or kde-multimedia) as appropriate. Sometimes mail on one list may touch on topics relevant to another, though in that case personal email is used before forwarding to a full email list first."
>
> Seigo, KDE, Interview$_{KS}$

How often do review threads span multiple lists? How many lists do individuals comment on?

While our interviewees indicate that mailing lists are bridged, we use three measures to quantify the strength of relationships between lists. In the context of review threads, we examine how many messages are posted to more than one list, how many threads span more than one list, and how many lists an individual has commented on.

We find that few messages are posted to more than one list and few review discussions span more than one list: 5% and 8% respectively for FreeBSD, and 1% and 15% respectively for KDE, and 4% and 5% respectively for GNOME. That this only occurs between 5% and 15% of the time is not surprising as the lists are set up to refer to modular components of the system. High levels of cross-posting might indicate that lists need to be merged.

In contrast to the relatively small proportion of cross-posted threads, many individuals posted to multiple lists – 30%, 28%, and 21% of all individuals[12] have posted to at least two lists for FreeBSD, KDE, and GNOME, respectively. Only 8%, 5%, and 3% of individuals posted to more than three lists for FreeBSD, KDE, and GNOME respectively. However, on the FreeBSD project, one individual posted to 24 different lists.

What is the relationship between developer status and the number of lists a developer bridges?

We suspect that high status individuals who are invested in the project, *i.e.* core developers, would serve as a bridge between lists. We correlate the number of lists an individual posts on to the number of review messages they have sent and the number of commits they have made (high status developers have made more contributions than low status developers). There is a moderate correlation between the amount an individual participates in review and the number of lists a developer posts to: $r = .37$ for FreeBSD, $r = .61$ for KDE, and $r = .51$

[12] In this case an "individual" is represented by a distinct email addresses

for GNOME[13]. A moderate correlation exists between the number of lists and the number of commits an individual makes: $r = .45$ for FreeBSD and $r = .51$ for KDE[14]. While there is some relationship between developer status and the bridging of multiple mailing lists, further investigation is clearly required.

In summary, multiple topic-specific mailing lists appear to be one technique used by large projects to deal with what would otherwise be an overwhelming amount of information. These lists allow developers to discuss details that may not be relevant to the larger community and then to sometimes post final changes to more general lists for further feedback. The mailing lists for FreeBSD, KDE, and GNOME appear to be relatively well separated as there are few cross-posted messages and threads. Approximately $1/4$ of the community is involved in reviews on more than one list. These individuals bridge multiple lists, allowing for outsider input on otherwise isolated lists. There is also a moderate relationship between the number of lists a developer has sent messages to and the developer's status within the community.

5.7.2 Explicit Requests

In traditional inspection, the author or manager explicitly selects individuals to be involved in a review (*e.g.,* Fagan [39]). With broadcast based review, the author addresses a patch email to the mailing list instead of particular individuals; thus, information an author has about potential reviewers is unused. Explicit requests or messages sent directly to potential reviewers as well as the entire mailing list can combine an author's knowledge with the self-selection inherent in broadcasting. Since core developers filter messages and pay more attention to messages sent directly to them (see Section 5.3), if an author wants to increase the chance that a particular reviewer will comment on a patch, the author could explicitly include the reviewer in the To or Cc fields of an email. In this section, we measure the impact of the author explicitly addressing reviewers in the Linux project.

On the Linux project, 90% of patch review requests had more than one recipient (*i.e.* the mailing list and one or more individuals). Compared to the other projects, this is remarkably high: Apache 9%, SVN 12%, FreeBSD 27%, KDE 13%, GNOME 19%. Since there are relatively few explicitly addressed emails on all lists, with the exception of Linux, it

[13] All Spearman correlations are statistically significant at $p < 0.01$

[14] We were unable to calculate the correlation between commits and mailing lists for GNOME developers because there was no clear linking between commits and email addresses

appears that most of the projects examined rely on multiple, small mailing lists to ensure that interested reviewers find patches they are competent to review. Given that the Linux mailing list has a median of 343 messages per day, more than the combined multiple lists on FreeBSD, KDE or GNOME, and that there is a much higher percentage of explicit review requests we decided to investigate the following:

Does the LKML represent a breakdown in broadcast mechanism whereby self-selection is ineffective and review requests must be explicitly sent to expert reviewers?

To investigate this phenomenon, we measured the following for each review thread on Linux: (1) the number of people who were explicitly addressed – explicit request (or requested), (2) the number of people who responded that were explicitly addressed – explicit response, (3) the number of people who responded that were *not* explicitly addressed – implicit response, and (4) the total number of people who responded – total response. Since individuals may use multiple email addresses, we resolved all email addresses to single individuals as per Bird et al. [14]. Furthermore, a mailing list can only receive mail, so we used regular expressions and manual examination to determine which addresses were mailing lists and which were individuals. Explicit requests to other mailing lists were removed from this analysis, but could be examined in future work.

If there is indeed a breakdown in the broadcast mechanism we would expect to see few implicit responses. Figure 5.5 shows that this is not the case. The median number of people explicitly requested is 3. The median number of people who explicitly responded is 1 and the median number of people who responded without an explicit request is 1, giving a total of 2 reviewers. A Wilcoxon test failed to show a statistically significant difference between the number of people who explicitly responded compared to the number that implicitly responded. Also, a Spearman correlation test revealed that the number of explicitly requested individuals had almost no effect on the number of individuals that implicitly responded, $r = -.05$. However, a moderate correlation existed between the number of explicitly requested individuals and the total number of respondents, $r = .30$ [15].

These results show that, for the Linux project, although the original message usually has explicit recipients, this does not stop other developers from responding, and implicit responses occur at least as often as explicit responses. Despite high levels of explicit requests to potentially expert reviewers, Linux sees a nearly equal number of implicit responses (*i.e.* self-selection).

[15] All Spearman correlations are statistically significant at $p < 0.01$

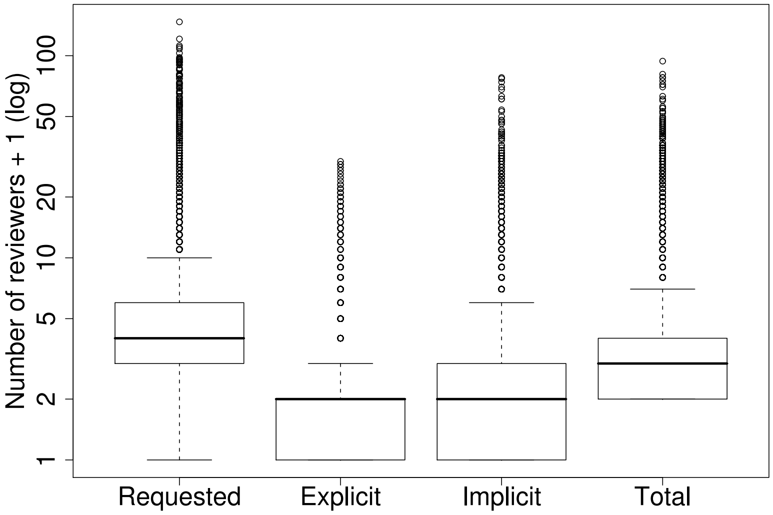


Figure 5.5: Explicit vs. Implicit Responses: The number of people that the review request was sent to (Requested). Of those requested, the number who responded (Explicit). The number of people who responded even though they were not explicitly requested, *i.e.* self-selection (Implicit). The (Total) number of people who responded.

As such, Linux appears to maintain the advantages of broadcasting reviews, while allowing authors to notify reviewers they believe should examine their contribution.

5.8 Threats to Credibility

The “truth value” of qualitative research should not be assessed in the same “valid” vs “invalid” manner of quantitative research [106]. Instead we examine the internal and external credibility of our research. Internal credibility assesses how meaningful and dependable our findings are within the phenomenon and context under study. While external credibility assesses the degree to which our findings generalize across different populations and contexts.

5.8.1 Internal Credibility

While there are many aspects to internal credibility the goal is to examine the "credibility of interpretations and conclusions within the underlying setting or group." [106] We assess the fit, applicability, and level of triangulation.

Fit Do the findings resonate with the interviewed core developers? [106] A summary of the research findings was sent to all participants. We also asked them to search for their interviewee code to ensure that they had made was not misinterpreted or taken out of context. Of the ten interviewees A1, LG, LK, F1, and KF responded. The interviewees were interested in and agreed with the findings, with LG and LK pointing out two minor issues (a missing footnote and a typo, respectively).

Applicability "Do the Findings offer new insights?" [32]

Kroah-Hartman found our results interesting and has agreed to help in the process of creating a shorter version of this work that would target practitioners.

> "Very nice dissertation, I'd like to see the final copy when you are done with it if you don't mind, it's really interesting"
>
> Kroah-Hartman, Linux, Interview$_{LK}$

Triangulation "involves the use of multiple and different methods, investigators, sources, and theories to obtain corroborating evidence" [106]. Since we examined publicly available data on six successful OSS projects, it should be possible to replicate our study on these or other projects. We used multiple data sets (*i.e.* archival and interview data), and multiple methods (*i.e.* grounded theory and various measures of review) to triangulate our findings.

5.8.2 External Credibility

External credibility refers to the degree that the findings of a study can be generalized across different populations of persons, settings, contexts, and times." [106]

To improve the generalizability of our results, the six case studies were chosen in a sequence that tested weaker aspects of our themes [142]. However, we examined only large, mature projects, so future work is required to examine the review processes of smaller, less successful projects.

Furthermore, KDE and GNOME were the only projects that included non-infrastructure software. Both projects use tracker based RTC, while CTR was still conducted on the mailing list. Studies of peer review on projects developing end-user software, *e.g.,* Firefox and Eclipse [21, 104], indicated that when defects are reported by non-developers, improved defect and review tracking tools are required. We only interviewed core developers. While these developers do the majority of the work on an OSS project [96], attending to the needs of and interviewing outside developers is left to future work. Future work may indicate the need for a tool that supports a hybrid review process, efficiently combining broadcasting with review tracking.

5.9 Summary and Conclusion

In this chapter, we conducted six case studies, manually coding hundreds of reviews, interviewing core developers, and measuring several aspects of OSS review processes. Our goal was to understand the mechanisms and behaviours that facilitate peer review in an open source setting. We summarize our results below.

(1) How do experts decide to review contributions in a broadcast of general development discussion?
Developers use filtering, progressive detail within patches, full and interleaved histories of discussion, and recipient building based on past interest to find contributions that fall within their area of interest and obligation.

(2) Why are patches ignored? What is the impact on the project?
Developers prefer to postpone reviews rather than rush through them. Therefore, patches that fail to generate interest among the core development team tend to be ignored until they become interesting. It follows that issue tracking does not completely solve the problem of ignored patches.

(3) Once a patch has been found and a review begins, which stakeholders are involved? How do these stakeholders interact to reach a decision?
Roles are not assigned in OSS development. This allows competent and objective stakeholders to play a role that accords with their skillset and leads to group problem solving. The style of interaction among stakeholders depends on whether the patch can be assessed from a purely technical perspective or whether project scope must also be considered.

(4) What happens to the decision making process when very large numbers of stakeholders are involved in a review?

The latter type of patch can lead to politicized, long, opinionated and unproductive discussion, *i.e.* bike shed painting discussion. Core developers typically try to avoid these debates.

(5) Can large projects effectively use broadcast based review?
While investigating scale issues, we found that three large projects, FreeBSD, KDE, and GNOME, effectively used multiple topic specific lists to separate discussions. Developers are typically subscribed to multiple lists, which results in lists being bridged when a discussion crosscuts topics. The Linux Kernel mailing list is an order of magnitude larger than the topic specific lists on the projects we examined. On this list, explicit requests to reviewers who the author believes should see a patch are employed, alongside the general broadcast.

OSS development continues to surprise software engineering researchers with its unique and often unintuitive approaches to software development that nevertheless result in mature and successful software products [95, 55]. Peer review is no exception. In the next chapter, we draw comparisons between broadcast based peer review and other styles of peer review, such as bugtracker based peer review and traditional software inspection. In Chapter 6, we integrate the findings across all our cases and study methodologies and we adapt and discuss the theory of peer review we presented in Chapter 3.

Chapter 6

Discussion and Conclusions

In this final chapter, we describe our main contribution: a theory of OSS peer review. The theory, Section 6.1, is set in the context of our findings and derived from multiple cases, sources of data, and methodologies. A secondary contribution of this work is the integration and comparison of the existing literature and our OSS review findings. Section 6.2 compares our findings on OSS peer review to the software inspection and Agile development literature. We also assess the benefits and weaknesses of using review tools rather than broadcasting reviews over email. Future work is suggested in the form of possible transfer between the methods used in industry and those used in OSS as well as potential simple tool modifications. Section 6.3 is as a very brief summary of our findings and future work written for practitioners. Section 6.4 summarizes and provides concluding remarks for this dissertation.

6.1 Theory of OSS Peer Review

The main contribution of this work is a theory that describes OSS peer review. Three stages of research – review processes, parameters and models, and mechanisms and behaviours – were involved in the development of this theory. Below we revisit the research questions, data, and methodology for each stage. We then present our theory and describe its development based on our findings.

Review Processes (Chapter 2): What are the review processes used on successful OSS projects? We answered this question by examining the review processes of 25 successful OSS projects. We also selected Apache, Subversion, Linux, FreeBSD, KDE, and Gnome for further analysis.

Parameters and Models (Chapter 3 and 4): By measuring archival data from six high-profile OSS projects we were able to create statistical models of review efficiency and effectiveness. Our research questions addressed:

- the frequency of reviews and the relationship between the frequency of reviews and development activity,
- the level of participation in reviews and the size of the review group,
- the level of experience and expertise of authors and reviewers,
- the size of the artifact under review,
- the complexity of the artifact under review,
- the review interval (*i.e.* the calendar time to perform a review), and
- the number of issues found per review.

Mechanisms and Behaviours (Chapter 5): Using grounded theory, we analyzed 500 instances of review and interviewed 10 core developers across the six projects. We addressed the following aspects of review:

- the techniques used to find relevant patches for review,
- the impact of ignored reviews,
- the review outcomes, stakeholders, and interactions,
- the effect of too many opinions during a review, and
- scalability issues involved in broadcasting reviews to large projects.

Case studies can provide an excellent background from which to develop new theories. The theory we developed in our original case study of Apache [119] appears to generalize, with minor modification, across all of the examined projects. Below, we restate some of our case study findings as components of a theory and suggest why each component may lead to a successful peer review technique. The following statement encapsulates our understanding of how OSS review functions.

(1) Early, frequent reviews (2) of small, independent, complete contributions (3) that, despite being asynchronously broadcast to a large group of stakeholders, are conducted by a small group of self-selected experts, (4) resulting in an efficient and effective peer review technique.

We dissect this statement below, showing the evidence that we have gathered, how this evidence is related to existing literature, and what remains as future work.

1. Early, frequent reviews

The longer a defect remains in an artifact, the more embedded it will become and the more it will cost to fix. This rationale is at the core of the 35-year-old Fagan inspection technique [39]. We have seen comparatively high review frequencies for all OSS projects in Section 3.3. Indeed, the frequencies are so high that we consider OSS review as a form of "continuous asynchronous review". We have also seen a short interval that indicates quick feedback (see Section 3.8). This frequent review and feedback also incorporates re-review of changes within the same discussion thread. While re-review in a separate inspection meeting can be costly [110], it has been shown to reduce defects in traditional inspection [136].

2. of small, independent, complete contributions

Breaking larger problems into smaller, independent chunks that can be verified is the essence of divide-and-conquer. Morera and Budescu [98] provide a summary of the findings from the psychological literature on the divide-and-conquer principle. This strategy also eases review by reducing the complexity of a contribution (See Section 3.7). Section 3.6 illustrates the small contribution size in OSS projects, which has been recognized by a number of researchers [34, 95, 135] and is smaller than those seen on industrial projects [96] and in inspection on Lucent [110]. However, we are the first to link small contribution size and the complexity of a change to peer review policy. The review process discussion in Section 2.2 and statements from our interviewees in Section 5.3.2 provides support for the idea that OSS developers will review only small, independent, complete solutions.

Although there are many advantages to small contributions, one potential disadvantage is fragmentation. Fragmentation could be a serious issue in industrial development, which may require assurances that all aspects of a product have been reviewed before a release. A system for tracking these small contributions (see discussion in Section 6.2.3.3) could make this accountability possible and eliminate concerns about unreviewed code: "commit-then-whatever".

3. that, despite being asynchronously broadcast to a large group of stakeholders, are conducted by a small group of self-selected experts

The mailing list broadcasts contributions to a potentially large group of individuals. A smaller group of reviewers conducts reviews periodically. To find patches for review in what can be an overwhelming quantity of information (See Figure 5.4 which shows scale), core developers use simple techniques, such as email filtering and subject scanning (See Section 5.3). With the exception of the top reviewers, most individuals participate in very few reviews per month. An even smaller group of reviewers, between one and two, actually participates in any given review (see Section 3.4). Small contribution sizes coupled with self-selection should allow the number of individuals per review to be minimized. The larger a contribution, the more likely it is to require a broad range of expertise and communication between experts. It is possible to increase the level of expertise in a group by adding more individuals to the review [122], but this solution can be expensive. Optimizing the number of reviewers per contribution could lead to overall cost savings.

The level of *expertise* is an important predictor of review effectiveness[110, 122]. OSS practitioners, Fielding [41], Raymond [114], and Fogel [43], discuss how OSS developers self-select the tasks they work on. In Section 5.3, our interviewees stated that they choose contributions to review based on their interest and, in the case of the subsystem maintainers, a sense of obligation. This finding is consistent with Lakhani and Wolf's [80] survey of OSS developers that found the main motivation for participating was intrinsic and enjoyment based. This self-selection likely leads to people with the right expertise reviewing a contribution. As we saw in our statistical models in Section 4.2, the level of expertise is an important predictor of the number of issues found and the length of the review interval. Further, although our findings showed that reviewers had more experience and expertise than authors, both groups had been with the project for approximately one year and in the case of reviewers, usually multiple years.

One risk with self-selection is that it is possible for contributions to be *ignored*. We found, in Section 5.4, that reviews of patches that fail to interest the core development team are postponed rather than hurriedly undertaken. Core developers are empowered to review code that is interesting and useful instead of simply reviewing the code of all developers. This selection process means that some patches will be ignored and never included in the system. In the case of CTR, self-selection could lead to unreviewed code being included in the system. If self-selection of reviews were used in industry, a review that was not naturally selected could be assigned to a developer. Another risk with self-selection and broadcasting is that *too many stakeholders* may be involved with a review. When many uninvolved stakeholders raise trivial and non-technical issues, a review can become deadlocked. In Section 5.6, our measures show that while non-core developers partake in reviews, there are few instances where they outnumber the core team. This result is also supported by our interviewees, who said the “bike shed” [73] effect rarely occurred during review.

In a very large, diverse, or poorly modularized project, a broadcast mechanism can become too busy (*i.e.* developers are interested in a small proportion of the information). In Section 5.7, we examine the techniques used by projects to overcome *scaling* issues. One solution is to use multiple mailing lists to limit the broadcast to developers working on related modules. However, the advantage of a broadcast is that developers outside of the core-group can contribute when the core-group is lacking the required expertise for a particular problem. We found that some developers subscribe to multiple lists and bridge together what would otherwise be isolated workgroups (See Section 5.7.1). A second solution is to send messages directly to potential reviewers as well as the entire mailing list. While developers continue to respond to broadcast patches, this technique works effectively on the massive Linux Kernel Mailing List because developers differentiate and respond more readily to messages sent directly to them (See Sections 5.7.2 and 5.3.4).

4. leads to an efficient and effective peer review technique

The review interval in OSS is on the order of a few hours to a couple of days, making peer review in OSS very efficient. While it is difficult to model review interval (see Section 4.2.1), this interval is drastically shorter than that of traditional inspection, which is on the order of weeks [122]. In terms of effectiveness, we have examined large, successful OSS projects that have reasonably low defect rates. While it is difficult to measure the exact number of true defects found during a review, we found that there are between one and two issues discussed per review. More issues will be found in complex contributions that receive wide community participation, and that have reviewers with high levels of expertise and experience (See Section 4.2.2).

When comparing the two main review types used in OSS, RTC and CTR, we see that they are significantly more like each other than any industrial technique [119]. In terms of efficacy, RTC is slower, while CTR finds fewer issues. However, these differences are likely due to the different contexts in which they are used (See Section 2.2).

The number of issues found is known to be a limited measure of review effectiveness [71]. OSS and asynchronous reviews provide opportunities for the other benefits of peer review, such as education of new developers, group discussion of fixes to a defect, and abstract or wandering discussions. With traditional inspections, the discussion takes place in a co-located meeting and centers around defects, while all other discussion, such as the solution to the defect, are banned [1, 138]. In contrast, asynchronous reviews ease the time constraints that are present in a review meeting, allowing for a more open discussion. In Section 5.5.2, we identified how reviewers and other stakeholders interact in OSS development. While formal roles do not exist as they do in traditional inspection, reviewers take on a series of personas (*e.g.,* mentor and grumpy cynic). Outside stakeholders, who can sometimes provide competent and objective code and criticisms, are excluded from the traditional inspection meeting.

6.2 Implications and Future Work

In the previous section we stated our theory and discussed the evidence we have for each aspect of it. In this section, we discuss the implications of our findings in the context of the existing literature on peer review and software development. The contribution of this section is a comparison of OSS peer review with software inspection and Agile methods. We also compare our findings on email based review to tool supported review. Future work emerges as we discuss possible transfers between OSS review and other review processes and tools.

6.2.1 Comparison with Inspection

There are few similarities between OSS review and Fagan inspections [39]. However, many of the advances and research findings over the last 35 years for software inspections are reflected by OSS peer review practices. In some cases, OSS review can be seen as taking an inspection finding to the extreme. In this section, we compare the inspection findings to our OSS review findings on the style of review meeting, formality of process, and the artifact and under review.

6.2.1.1 Process and Meetings

Fagan inspection is a formal process that relies on rigid roles and steps, with the single goal of finding defects [38]. Most of the work on inspection processes has made minor variations to Fagan's process (*e.g.,* [90, 76]) but kept most of the formality, measurability, and rigidity intact [77, 79, 138].

An important exception to this trend was the work done by Votta that facilitated *asynchronous inspection*. In contrast to Fagan-style inspections, where defect detection is performed only during the meeting phase, Votta [82] showed that almost all defects can be found during the individual preparation phase, where a reviewer prepares for an inspection meeting by thoroughly studying the portion of code that will be discussed. Not only were few additional defects found during synchronous meetings, but scheduling these meetings accounted for 20% of the inspection interval. This result allowed researchers to challenge the necessity of meetings. Eick *et al.* [36] found that 90% of the defects in a work product could be found in the preparation phase. Perry *et al.* [109] conducted a controlled experiment in an industrial setting to test the necessity of a meeting and found that meeting-less inspections, according to the style of Votta, discovered the same number of defects as synchronous Fagan inspections. A similar result was obtained by Johnson *et al.* [72]. There are many tools to support inspection in an asynchronous environment (*e.g.,* [91, 87]). Mirroring these findings, OSS reviews are conducted in an asynchronous setting.

The rigid goal of finding defects and measuring success based on the number of defects found per source line lead to a mentality of "Raise issues, don't resolve them." [71] This mentality has the effect of limiting the ability of the group to collectively problem solve and to mentor an artifact's author [122]. In OSS review, author, reviewers, and other stakeholders freely discuss the best solution, not the existence of defects – there are instances where a reviewer re-writes the code and the author now learns from and becomes a reviewer of the new code.

The optimal number of inspectors involved in a meeting has long been contentious (*e.g.,* [23, 136]). Reviews are expensive because they require reviewers to read, understand, and critique an artifact. Any reduction in the number of reviewers that does not lead to a reduction in the number of defects found will result in cost savings. Buck [23] found no difference in the number of defects found by teams of three, four, and five individuals. Bisant and Lyle [16] proposed two person inspection teams that eliminated the moderator. In examining the sources of variation in inspections, Porter *et al.* [110] found that two reviewers discovered as many defects as four reviewers. The consensus seems to be that two inspectors find an optimal number of defects [122]. In OSS review, the median number of reviewers is two; however, since the patch and review discussions are broadcast to a large group of stakeholders, there is the potential to involve a large number of interested reviewers if necessary.

While most studies, including Votta's work on the necessity of a meeting, focused on the impact of process variations, Porter *et al.* [110] examined both the process and the inputs to the process (*e.g.,* reviewer expertise, and artifact complexity). In terms of the number of defects found, Porter *et al.* concluded that the best predictor was the level of expertise of the reviewers. Varying the processes had a negligible impact on the number of defects found. This finding is echoed by others (*e.g.,* [122, 77]). While OSS review practices evolved organically to suit the needs of the development team, they mirror Porter *et al.*'s finding – OSS review has minimal process but relies heavily on self-selecting experts.

6.2.1.2 Artifact Under Review

Traditional inspection processes required completed artifacts to be inspected at specific checkpoints. Traditionally, the development of these artifacts was done by individuals or relatively isolated groups of developers. The work could take many months and involve little or no external feedback until it was completed [82]. Once complete, these large artifacts could take weeks to inspect. While the importance of shorter development iterations has become increasingly recognized [81], OSS has much smaller change sizes than the industrial projects examined by, for example, Mockus *et al.* [96]. These extremely small artifact changes (*e.g.,* between 11 and 32 lines of code modified) and the short time in which the review occurs (hours to days) allows for early, frequent feedback – "continuous asynchronous review".

Aside from the issue of timely feedback, large artifacts were also unfamiliar to the developers tasked with inspecting them, making preparing for and inspecting an artifact an unpleasant task. Not only was it tedious, but as Parnas and Weiss [108] found, inspectors were not adequately prepared and artifacts were being poorly inspected. Parnas and Weiss suggested active reviews that increased the quality of inspections by making inspectors more involved in the review. Porter and Votta [111] and Basili *et al.* [8] and many other researchers developed and validated software reading techniques that focused developers on different scenarios and perspectives. While researchers continue to question the effectiveness of these scenarios and checklists [58], the main idea was to ensure that reviewers paid attention during a review and that they focused on the areas for which they had expertise. OSS reviews take this need for focused expertise in a different direction. First, reviews of small artifacts increases the likelihood that developers will be able to understand and focus on the whole artifact. Second, the high frequency of reviews means that developers are regularly and incrementally involved in the review of an artifact, likely lessening the time required to understand the artifact under review. Third, many systems and subsystems involve multiple co-developers who already understand the system and only need to “learn” the change to the artifact under review. Finally, outsiders can learn about the system and provide expertise that is missing from the core team. Our findings also show that OSS reviewers review code that they are interested in or obliged to review because they maintain that section of code. They are invested in the change under review and, in some cases, they will have to maintain the change if it is committed. This contrasts with traditional inspectors who heavily rely on extrinsic motivation to review code they have no active interest in.

In summary, both the inspection literature and our findings regarding OSS review imply that the formality and rigidity of Fagan inspections is unnecessary. Peer review practices that are conducted asynchronously, that empower experts, that have timely feedback through small contributions, and that allow developers to focus on their area of expertise are more efficient and at least as effective as more formal peer review techniques.

6.2.2 Comparison with Agile Development

OSS has much in common with Agile development [9, 81, 64, 17] and the Agile manifesto [10]. For example, both prefer working software over documentation, both empower individuals rather than having a rigid process, both respond to change by working in small increments rather than rigidly following a plan, and both work closely with the customer [1] rather than negotiating contracts.

While there are certainly differences between the two methodologies, the most striking is that Agile is done by small, co-located teams of developers, while OSS projects can scale to large, distributed teams that rarely, if ever, meet in a co-located setting.

One factor that makes it difficult to conduct Agile development with large, distributed teams is the dependency on what Cockburn calls “osmotic” communication: “information flows into the background hearing of members of the team, so that they pick up relevant information as though by osmosis ... Then, when one person asks a question, others in the room can either tune in or tune out, contributing to the discussion or continuing with their work” [25]. According to Hossain, Herbsleb, and others, this style of communication is very difficult to create in distributed development [65, 60].

The broadcast on a mailing list also depends on “osmotic” communication, but works in large, distributed development environments. All software development discussions,

[1] For OSS projects, the developer is usually also the “customer” or user

including patch submissions and questions, are posted to the mailing list. Developers "tune" discussion in and out by using mail filters and scanning subject lines to decide what is interesting to them and what is not. Developers can gain peripheral knowledge of the activities of others and can participate in discussion by adding a single, specific comment or actively contributing detailed reviews and code. While there are many similar principles between Agile and OSS development, assessing the appropriateness of using broadcast based communication in an industrial, Agile environment is left to future work. However, the possibility of having "continuous," asynchronous communication (*i.e.* very fast feedback) and peer review in an Agile setting is intriguing.

6.2.3 Comparison of Email and Tool Based Review

Despite the existence of many peer review tools (*e.g.,* Gerrit[50], Review Board [116], Gitorious [51], and Code Collaborator [27]), the projects we examined and the core developers we interviewed all used email for review, with KDE also using Review Board [116] and GNOME also using Bugzilla [24]. The purpose of this section is to examine the strengths and weaknesses of both email and tool based review. We do not enumerate all of the review tools [2] or the features of these tools to compare them to email (which has no specific reviewing features), but instead examine both at a level of fit with developers.

Current tools for software review provide a rich, but rigid environment for tracking reviews, linking with version control, marking up changes and discussing patches. In contrast, Email provides a flexible, but informal environment for discussing reviews and other issues. With tools, reviews are conducted in a specialized environment that is separate from general development discussion. With email, full review discussions are broadcast along with all other development discussion. We discuss the tradeoffs of both approaches in terms of the interface and features, the communication medium, and the ability to track reviews.

6.2.3.1 Interface and Features

Our interviewees were generally satisfied with email based review; however, one common weakness that they identified was the lack of linking between patches and the version control system$_{LI,KS,KF,GM}$ [3]. Tools such as Gerrit[50] and Review Board [116] provide this linkage.

[2] We also do not consider older tools (See Macdonald and Miller [87]) or tools that were only used in a research environment

[3] Each code represents one of our interviewees. See Chapter 5 for more details

For example, a diff made against an older version of a file can be updated by selecting the most recent version within a tool, or an approved change can be immediately committed into the version control system.

In contrast, email does not provide these links, but does provide a more flexible environment for dealing with diffs. For example, while tools have the ability to create in-line comments (*i.e.* comments interleaved within a diff), they lack the ability to remove code that does not pertain to the issue under discussion. Using email, developers remove any code or comments that do not relate to the issue they have raised. In this manner, the reviewer and author are focused on a particular issue, and the histories of particular issues are easily followed.

Tools also force a particular style of review and an interface that is unfamiliar to new users$_{F1,KF,GM}$. While almost all users and developers have experience using email, some developers were concerned that unknown tools would reduce input from outsiders. Another concern with tools is that they usually isolate reviews into a separate environment from general development discussion. To partially address these concerns, one developer suggested integrating tools with email by allowing reviewers "to reply by email, rather than having to use the web interface (and yet having the reply visible in the web page for others)."$_{KF}$ Future work is needed to access the feasibility and complexity of this approach.

6.2.3.2 Communication Medium

One strength of OSS development is that reviews are broadcast within general development discussion so that all developers see all contributions and reviewers can self-select patches to review. Separating reviewing from the general development discussion could have the effect of loosing feedback from peripherally related but competent individuals. If a broadcast is not used, some form of patch triage and reviewer assignment must be used.

However this loss of broadcast can be advantageous in certain environments. Separate review tools are especially useful when non-core issues are being discussed by a specific group of developers$_{A1}$. A non-core issue or obscure use case might interests only a small segment of the development team. Discussing this issue in a separate forum will mean that core-developers are not distracted. Furthermore, outside developers who are only interested in a particular area will not have to search through the barrage of general development discussion to find the specific issues they are interested in.

On very large and diverse projects, *e.g.,* KDE and GNOME, there is a clear need to isolate work groups that have different "core" issues. Also, while email may be familiar to most developers, users, and outsiders, a broadcast mailing list, due to the volume of messages, will likely be more overwhelming for newcomers than becoming accustomed to a specific tool. Future work is needed to understand how best to manage the tradeoffs between isolation of reviews in tools or specific mailing lists, and broadcast of reviews within general development discussion.

6.2.3.3 Review Tracking and Traceability

Tools provide an environment that facilitates review and patch tracking. For example, a tool can track whether a patch has been reviewed and who approved or rejected the patch. Surprisingly, none of the interviewees mentioned tracking as an area that should be improved in OSS peer review. However, all interviewees were core developers who can commit code changes. We did not interview non-core developers. Future work involving non-core developers may indicate the need for better tracking so that their patches are not ignored.

In an industrial setting where management may need to track and measure the progress of a project closely, tracking tools can be vital. While tools can be explicitly designed to meet the tracking needs of a project, much of this tracking does not come for free; developers and reviewers are required to enter the tracking information into a specific tool. In contrast, the measures of OSS peer review that we developed through our analysis of archival data (such as number of developers involved in a review, and the amount of time a review took to complete) required no extra effort from developers or reviewers to obtain. However, there are certain attributes, for example, the number of defects found during review and whether a patch was committed, that are not implicitly recorded and cannot be determined without significant manual analysis.

As future work, we suggest the exploration of an alternative to tool based tracking where "meta-data" is added to email reviews. For example, when an email review thread is concluded, the author or a reviewer could send an email via a template which includes, for example, "Bugs found: X" or "Code accepted: <commit identifier>" and other easily parsible attributes. These attributes could be displayed as a summary for management or in a modified email environment (several email readers could be modified in different ways, depending on the needs of the developers). Our measures, such as the complexity of the current change, along with this explicit additional "meta" information could be displayed along with the author, subject and other standard email fields. The tool could also hide or "fold" the "meta" information so that it would not distract reviewers and authors in the body of the email. While this meta-data clearly requires additional effort from developers, the goal of this future work would be to modify the successful OSS techniques as little as possible while still providing tracking information that may be useful for industrial managers. This approach would keep review discussion within the general broadcast of development discussion, avoiding the segregation of reviews into unfamiliar, rigid tools.

6.3 Practitioner's Summary

Below we provide a brief summary of points for a practitioner to consider, should he or she wish to implement aspects of OSS peer review:

- Conducting reviews asynchronously is as effective as meeting in a co-located environment and is more efficient.
- Asynchronous reviews allow for team discussion of solutions to defects. They also allow passive developers to "listen" and learn from the discussion.
- The earlier a defect is found the better. Conduct reviews frequently – "continuous" asynchronous review. Ideally, reviews are conducted within hours of posting the change. This style of review is reminiscent of pair programming.
- Changes should be small and incremental, but they should also be logically independent and complete. Changes should consist of around 15 to 30 lines of changed code. Small changes should allow developers to periodically review code throughout the day.
- Reviews should be conducted by invested experts, ideally co-developers, who already understand the context in which the change is being made.
- Allow experts to select changes they are competent and interested in reviewing. Two reviewers are sufficient to find almost all defects. Assign reviews that are not naturally selected by at least two individuals. Is it cost effective to have an expert review an incompetent developer's change?

- The notion of “osmotic” communication, whereby co-located Agile developers hear discussions in the background, allows for a general awareness of what the team is doing and an opportunity for developers not otherwise involved to provide potentially critical information to the team. Broadcasting discussions and changes to a distributed development team works under the same principle, but allows teams to be large and geographically dispersed.

- Review tools provide traceability of reviews and a rich environment that can be integrated with the version control system. However, when compared with email, there is a tradeoff in flexibility and familiarity. Tools also separate review discussions from general development discussions.

While this work provides evidence for each of the above statements in an OSS context, practitioners should be aware that there is a need for future work that assess the appropriateness of combining broadcasting with reviewer assignment and tracking tools in an industrial setting.

6.4 Concluding Remarks

In this work we developed a theory of OSS peer review by empirically examining the review processes of successful OSS projects (Chapter 2), quantifying and statistically modelling the parameters of peer review (Chapters 3 and 4), and developing a grounded understanding of the behaviours and mechanisms that underlie OSS peer review (Chapter 5).

OSS development continues to surprise software engineering researchers with its unique and often unintuitive approaches to software development that nevertheless result in mature and successful software products [96, 55]. Peer review is no exception. Organic, developer driven, community-wide processes, mechanisms, and behaviours that use simple tools that rely heavily on the human can effectively manage large quantities of broadcast reviews and other development discussions. Future work is necessary to understand whether OSS practices can help Agile development scale to large distributed settings, and if the simple tools and practices used in the OSS community transfer to other development settings.

Appendices

Appendix A

Project Classification and Review Policies

As we discussed in the introduction, this dissertation is only interested in examining the review processes of successful, mature projects. Even with this limitation there are thousands of potential OSS case studies, and the accurate determination of success in OSS is difficult to quantify [48]. We use two approaches to identifying potential case studies. First, we examine iconic, high profile projects which have obvious measures of success (*e.g.,* dominance of a particular software domain). Second, we use an online "catalogue" of thousands of OSS applications to select a more diverse sample of successful projects [44]. We manually classify the review processes of 25 OSS projects. In this appendix, for each we provide a description of the project, the type of project, the review types and policies used on the project, and any other observations, such as the size of the development team and the governance structure. In Section 2.2 and Table 2.1, we describe the different types of we saw accross all the projects we examined.

A.1 High Profile Projects

While we examined the review processes for 25 projects, these high profile projects encapsulated all the types of review we saw on the larger sample and had defined policies and large development archives we could examine. The following six projects are the focus of this work: the Apache httpd server, the Subversion version control system, the FreeBSD and Linux operating systems, and the KDE and Gnome desktop environment projects. While we discuss the review policies of the Mozilla and Eclipse projects in this section, the detailed examination of these projects has been done by other researchers [21, 69, 104]

A.1.1 Apache

"Apache is the world's most popular HTTP server, being quite possibly the best around in terms of functionality, efficiency, security and speed." [44]

In terms of success, in June 2010, Apache dominated the server market running on 54% of the world's serves (the sample was done on over 200 million servers) [102].

Project Type [44]: Internet, Web, HTTP Servers

Types of Review: RTC, "lazy" RTC, and CTR.

Other Observations: The Apache server project was started by a group of globally distributed server administrators who voluntarily shared fixes to the original code base of the NCSA server [41]. Since the original developers were all volunteers who had never met in a face-to-face manner, it would seem natural that they would examine each other's code before including it in their own local server. As the project evolved, a shared version control repository was created and a trusted group of developers controlled access to this repository [41]. For the purposes of the Apache project, trusted or core developers, are those who have demonstrated their competence and have been given voting and commit privileges. Their review technique, known as *review-then-commit*, was formalized and accepted as Apache policy [2]: before any code can be added to the shared repository (i.e., committed) it must receive a positive review by at least three core-group members.

Reviewing all code before it is committed can become a very onerous, time consuming process. This frustration with "strong" RTC is illustrated through an email quotation from one of the core developers of the Apache project in 1998:

> This is a beta, and I'm tired of posting bloody one-line bug fix [contributions]
> and waiting for [reviews] ... Making a change to Apache, regardless of
> what the change is, is a very heavyweight affair.

After a long, heated email discussion, a new style of review was accepted as Apache policy, namely *commit-then-review*. CTR is used when a trusted developer feels confident in what he or she is committing. RTC is now only used when core developers are less certain of their contribution (e.g., it is complex) or when contributions are made by non-core developers (i.e., developers without commit privileges). According to Fielding, one of the founding members of the Apache project, Apache core developers are expected to review all commits and report any issues to the mailing list [95]. However, some core developers involved in the policy debate in 1998 are less convinced that this post-commit review will happen in a volunteer environment; one developer dubs the policy “commit-then-whatever”.

In summary, the RTC and CTR review policies create two barriers to “buggy” code. The first barrier is to require significant changes and any changes from non-core developers to be reviewed before being committed (RTC). The second barrier is to require reviews of committed contributions (CTR). For the latter process to be effective, developers must examine the commit mailing list to ensure that unacceptable contributions are fixed or removed.

A.1.2 Subversion

“Subversion is a version control system. Originally designed to be a compelling replacement for CVS in the open source community, it has far exceeded that goal and seen widespread adoption in both open source and corporate environments. The Subversion project produces Subversion’s core libraries (written in C), a fully functional command line client (svn), repository administration programs, API bindings for various languages (Perl, Python, Java, Ruby, etc.), and various additional tools and scripts.” [44]

In 2007, Forrester Wave [124] compared 11 source control management systems on 123 different criteria and found that Subversion was the clear leader in terms of stand-alone source control management systems.

Project Type [44]: Version Control

Types of Review: Subversion uses an Apache style of RTC and CTR.

Other Observations: Some of the Apache developers joined the Subversion project and had an influence on the project’s review policies.

A.1.3 Linux

“Linux is a clone of the Unix kernel, written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX and Single UNIX Specification compliance. It has all the features you would expect in a modern fully-fledged Unix kernel, including true multitasking, virtual memory, shared libraries, demand loading, shared copy-on-write executables, proper memory management, and TCP/IP networking.” [44]

Project Type [44]: Operating System

Review Type and Other Observations: The Linux Operating System Kernel's review process is informal (For more information see http://www.kernel.org/pub/linux/docs/lkml/index.html December 2005). Patches are sent to the appropriate developer in the 'MAINTAINERS' file and the mailing list. To avoid "flaming," contributors can privately send works-in-progress to the appropriate maintainer. Once a patch is considered "obviously correct," the patch can be sent to Torvalds for further review and potential inclusion in the kernel. For a patch to be "obviously correct," it must be either small and obvious, critical and obvious, or large and/or complex and reviewed and tested. If a patch does not meet the "obviously correct" criteria, the patch will not be committed. Since test cases for kernel development are difficult to write, Linux relies on informal testing. This testing, "lazy testing," consists of having people try your patch and reporting whether it worked. Since lazy testing can take a long time, the contributor may have to rewrite the patch against the latest kernel to have it accepted.

The Linux project is unique among the examined projects in that there is only one true committer: Linus Torvalds. Since it is not possible for Torvalds to review every patch, he partially depends on his "lieutenants," the people he trusts, to determine if a patch is correct. In turn, the lieutenants, such as Alan Cox, trust other regular committers, and these regular contributors trust other less well-known contributors. This "pyramid of trust," picture which is largely dependent on the git version control tool, allows each developer to maintain their own version of the software and to be the ultimate judge of what is committed and what is not. In practice Torvalds maintains the current version of the kernel, and he is "very conservative about what he lets in," while maintainers of older kernel versions, such as Cox, accept more experimental patches. If a patch proves to be stable, these maintainers send the patch to Linus for inclusion in the current kernel.

A.1.4 FreeBSD

"Briefly, FreeBSD is a UNIX operating system based on U.C. Berkeley's 4.4BSD-lite release for the i386 platform (and recently the alpha platform). It is also based indirectly on William Jolitz's port of U.C. Berkeley's Net/2 to the i386, known as 386BSD, though very little of the 386BSD code remains. A fuller description of what FreeBSD is and how it can work for you may be found on the FreeBSD home page." [44]

The project is mature and successful project; it is often used to run servers by large webhosting companies. In 2004, approximately 2.5 Million websites ran FreeBSD [101] and Netcraft serveys of the most reliable webhosting companies has FreeBSD servers at the top [102].

Project Type [44]: Operating system

Types of Review: FreeBSD has three types of review: Review-then-commit, commit-then-review, and pre-release review. RTC can be used by any developer, CTR can only be used by developers who have commit privileges, and pre-release review is a massive review of all new code directly before a release. We do not study the pre-release review in this work.

Other Observations: Individuals who provide code contributions to the project belong to three groups: core-member, committer, and contributor. The first two groups have commit privileges. The core-members are voted into the core by active committers for a two year term; this group has been between seven and nine members. Contributors that have done significant work can be voted into the committer group. A new committer is initially assigned a mentor (i.e., an active committer) who is responsible for reviewing all of the individual's commits.

Since FreeBSD has a large code base, there are acknowledged individuals who maintain aspects of the project. When a developer wants to commit code to an area he or she does not maintain, the developer must first ask for a review (i.e., RTC).

A.1.5 KDE

"KDE is a powerful graphical desktop environment for Unix workstations. It combines ease of use, contemporary functionality and outstanding graphical design with the technological superiority of the Unix operating system. KDE is a completely new desktop, incorporating a large suite of applications for Unix workstations. While KDE includes a window manager, file manager, panel, control center and many other components that one would expect to be part of a contemporary desktop environment, the true strength of this exceptional environment lies in the interoperability of its components." [44]

In terms of success, the list of awards KDE has won can be found at http://www.kde.org/community/awards/.

Project Type [44]: Desktop Environment

Types of Review: KDE uses both RTC and CTR. While email RTC was used on KDE, the project now uses the tracking tool ReviewBoard (http://reviewboard.kde.org) to manage RTC reviews.

A.1.6 GNOME

"The GNOME project provides two things: The GNOME desktop environment, an intuitive and attractive desktop for users, and the GNOME development platform, an extensive framework for building applications that integrate into the rest of the desktop." See http://www.gnome.org/about/.

There are roughly a few million non-technical users and thousands of technical ones [53]. There are many industrial sponsors of the GNOME project; among the 19 projects on the GNOME advisory board are IBM, Novell, Hewlett-Packard, Google, and the Mozilla Foundation http://foundation.gnome.org/about/ (visited in 2008).

Project Type [44]: Desktop Environment

Types of Review and Other Observations: Although subprojects within GNOME may each use a slightly different process, patches are typically sent to GNOME's Bugzilla repository and not to the mailing list. While email RTC was used, Bugzilla is more commonly used currently; however, CTR continues to be used (Personal communication with Germán). German [46] provides discussion of the history and project structure of GNOME.

A.1.7 Mozilla

The following summary is from Rigby and German [118]. The Mozilla project is a suite of applications that were originally developed by Netscape Inc. under a proprietary license. In January 1998 [114] Netscape released the source code under the Netscape Public License and later under the more successful Mozilla Public License. The application suite includes a web browser and an email client. Mozilla also includes software development tools like bugzilla.

Project Type [44]: Internet, Communications

Types of Review and Other Observations: Mozilla uses a "strong" form of RTC that is dependent on the bugtracker Bugzilla. "Code review is our basic mechanism for validating the design and implementation of patches"(http://www.mozilla.org/hacking/code-review-faq.html December 2005). Since many of the Mozilla projects are dependent on each other, every commit to the Mozilla repository is reviewed by at least two independent reviewers. The first type of review is conducted by the module owner or the module owner's peer. This review catches domain-specific problems. A patch that changes code in more than one module must receive a review from each module. The second type of review is called a "super review". The goal of this review is to find integration and infrastructural problems that may effect other modules or the user interface. "Super review" is not required for independent projects, such as bugzilla. Simple, obvious patches can request a "rubber stamp" (quick) "super review". By requiring both types of review, Mozilla ensures that someone with domain expertise and someone else with overall module and interface knowledge have approved the patch [118].

A.1.8 Eclipse

"Eclipse is a kind of universal tool platform - an open extensible IDE for anything and nothing in particular. The real value of Eclipse comes from tool plug-ins that "teach" Eclipse how to work with things - java files, web content, graphics, video, etc. Eclipse allows you to independently develop tools that integrate with other people's tools seamlessly." [44]

Project Type [44]: Software Development

Types of Review and Other Observations: Eclipse uses an Bugzilla RTC. The Eclipse project recieves sustaintly development recources from IBM (http://wiki.eclipse.org/Development_Resources).

A.2 Freshmeat

Freshmeat catalogues thousands of OSS project [44]. Users can view these projects and subscribe to updates relating to the projects. Freshmeat calculates a number of metrics. Popularity is measured using the last 90 days of data, according to the following formula [45]:

$$((recordhits + URLhits) * (subscriptions + 1))^{1/2} \tag{A.1}$$

where record hits is the number of hits the project receives on the Freshmeat website, URL hits is the number of click-throughs to the projects main website, and subscriptions is the number of people subscribed to this project's updates on Freshmeat.

We categorize the top 19 projects on Freshmeat as of September 24, 2009.

A.2.1 1, gcc

"The GNU Compiler Collection contains frontends for C, C++, Objective-C, Fortran, Java, and Ada as well as libraries for these languages." [44]

Project Type [44]: Software Development, Compilers, Debuggers

Types of Review: GCC's write policies imply that code must at least be reviewed by the maintainer of the code (RTC). The maintainer of a given section of code is not required to be reviewed by anyone, nor are developers with global write privileges [118].

GCC's policies create four categories of write privileges that affect the level of review. Developers with **global write privileges** do not need approval to check in code to any part of the compiler; there are 11 developers with global write privileges. The second category consists of developers with **local write privileges**; they are authorized to commit changes to sections of code they maintain without approval. Developers in this category are generally responsible for ports to other hardware platforms. The third category consists of regular contributors who can only **write after approval** from the maintainer of the section of code their contribution modifies. The final **free for all** approval applies to obvious changes to artifacts like documentation or the website (http://gcc.gnu.org/svnwrite.html#policies December 2005).

Other Observations: GCC has developed a strict set of policies for granting commit privileges. These policies are not enforced by SVN. Rather, their implementation is dependent on the integrity and understanding of the 148 (See GCC'S MAINTAINERS file version 1.439) developers who have commit privileges. The project relies on its steering committee to discipline uncooperative members. There are 11 core developers who never have to get write permission before committing code.

A.2.2 2, Apache

See Appendix A.1.1 above.

A.2.3 3, cdrtools

“cdrtools (formerly cdrecord) creates home-burned CDs/DVDs with a CDR/CDRW/DVD recorder.” [44]

Project Type [44]: Software Distribution, Archiving, multimedia, Sound/Audio, CD Audio, CD Writing, CD Ripping, Boot

Types of Review: There is no review policy.

Other Observations: The project is hosted on a berilOS at http://developer.berlios.de/projects/cdrecord/. berilOS is similar to SourceForge in the services it provides. An examination of the bug tracker and mailing list show that cdrtools is a mature project with almost no activity on the mailing lists. There are three developers with administration privileges.

A.2.4 4, Linux

See Section A.1.3.

A.2.5 5, Postgresql

“PostgreSQL is a robust relational database system with more than 20 years of active development that runs on all major operating systems.” [44]

Project Type [44]: Database Database, Engines/Servers

Types of Review: **RTC** Postgresql has a detailed patch submission guide at http://wiki.postgresql.org/wiki/Submitting_a_Patch. The guide is similar to Apache’s RTC with the following exceptions. 1) Developers are encouraged to discuss changes on the developers’ list before actually writing code 2) Patches are tracked and queued using commitfest https://commitfest.postgresql.org/ 3) non-core reviewers are encouraged to perform testing and reviews and provide feedback to demonstrate to a core developer that the patch is likely of a sufficiently high quality to review, directions for less experienced reviewers is available at http://wiki.postgresql.org/wiki/Reviewing_a_Patch.

Other Observations: German [47] finds that a small number of committers, two in this case, are responsible for the majority of commits. However, these developers are supported by a large number of other developers reporting bugs and providing patches.

A.2.6 6, VLC

"VLC media player is a media player, streamer, and encoder for Unix, Windows, Mac OS X, BeOS, QNX, and PocketPC." [44]

Project Type [44]: multimedia, Video, Display, Sound/Audio, Players

Types of Review: VLC uses an RTC style of review. There does not appear to be a commit mailing list and no mention of core-developers reviewing each others' code after it is committed. The policy for submitting patches to the developers' list is available at http://wiki.videolan.org/Sending_Patches. VLC requires large patches to be sent as patchsets created by git.

Other Observations: VLC is run by an autonomous not for profit organization. This foundation has a board of directors that determines the direction of the project.

A.2.7 7, MPlayer

"MPlayer is a movie and animation player that supports a wide range of codecs and file formats" [44]

Project Type [44]: multimedia, Video, Conversion, Display

Types of Review: MPlayer uses both RTC and CTR. MPlayer's RTC policy is identical to Apache's http://www.mplayerhq.hu/DOCS/tech/patches.txt, and there are discussions of committed patches on the cvs-log (actually SVN) list (see http://www.mplayerhq.hu/design7/mailing_lists.html.

Other Observations: Like the Apache project, developers who have made significant contributions will be given commit privileges.

A.2.8 8, Clam AntiVirus

"Clam AntiVirus is an anti-virus toolkit designed especially for email scanning on mail gateways." [44]

Project Type [44]: Utilities, Communications, Email, Filters, Security

Types of Review: There is no explicit policy of peer review in Clam AV. There are, however, a developers' list and a bug database. Clam AV must also deal with reviewing of potential virus, which is done at http://www.clamav.net/sendvirus/.

Other Observations: Clam AV was acquired by Sourcefire in August 2007 (http://investor.sourcefire.com/phoenix.zhtml?c=204582&p=irol-newsArticle&ID=1041607) which now owns the trademarks and copyright of the source code. Although Clam AV remains under the GPL, the core developers are now paid by Sourcefire. However, contributions from outside developers are owned by those developers. The core development team (14 members) and the list of developers (8 members) who are being considered for inclusion is available at http://www.clamav.net/about/team.

A.2.9 9, MySQL

"MySQL is a widely used and fast SQL database server." [44]

Project Type [44]: Database, Database Engines/Servers

Types of Review: MySQL follows an RTC policy for both internal and external developers (http://forge.mysql.com/wiki/Code_Review_Process). Code is checked into the developers local Bazaar version control repository and a worklog (an internal tool), bug report, and/or email is sent to the developers list. The author must find at least two reviewers. After review, the code will make it through various branches to trunk (http://forge.mysql.com/wiki/Development_Cycle). A detailed checklist is provided at http://forge.mysql.com/wiki/Code_Review_Process.

Other Observations: At the time of this analysis, MySQL is owned by Sun. External developers must sign the SCA (http://forge.mysql.com/wiki/Sun_Contributor_Agreement) that gives copyright ownership to Sun and the original contributor.

A.2.10 10, PHP

"PHP is a widely-used Open Source general-purpose scripting language that is especially suited for Web development and can be embedded into HTML." [44]

Project Type [44]: Software Development, Interpreters, Internet, Web, Dynamic Content

Types of Review: PHP does not have an official review policy; however, its internals list contains patch discussions, RTC, and it has an SVN commit mailing list, CTR, (http://

www.php.net/mailing-lists.php). There is also a bug database, a test framework, and a quality control team that does extensive testing before releases (http://qa.php.net/). The QA team's goal is "to support the PHP core developers by providing them with timely quality assurance."

Other Observations: Although the PHP Group owns the trademarks and copyright of PHP's code and logos, we were unable to find formal policies or a list of members for this group.

A.2.11 11, phpMyAdmin

"phpMyAdmin is a tool intended to handle the administration of MySQL over the Web." [44]

Project Type [44]: Database, Front-Ends, Systems Administration

Types of Review: phpMyAdmin has only a basic outline for submitting patches (http://wiki.phpmyadmin.net/pma/FAQ_7.3). This outline resembles an Apache style of RTC. There is no official CTR policy; however, there is an SVN commit mailing list.

Other Observations: phpMyAdmin extensively uses the services provided by SourceForge to manage its development. The project currently has 15 core developers http://www.phpmyadmin.net/home_page/about.php.

A.2.12 12, ntop

"ntop is a network probe that shows network usage in a way similar to what top does for processes." [44]

Project Type [44]: Emulators

Types of Review: There is no official policy for reviews in ntop. External developers can submit patches to the developers mailing list (RTC http://www.ntop.org/needHelp.html). The project uses Trac "a web-based software project management and bug/issue tracking system emphasizing ease of use and low ceremony" http://www.ntop.org/trac/about. Trac provides a "Timeline" which shows the latest changes to the SVN repository (CTR).

Other Observations: The AUTHORS file in the ntop sources acknowledges over 100 developers (http://www.ntop.org/trac/browser/trunk/ntop/AUTHORS).

ntop also provides commercial support for its applications.

A.2.13 13, TightVNC

“TightVNC is a VNC distribution with many new features, improvements, and bugfixes over VNC.” [44]

Project Type [44]: Systems Administration, Networking

Types of Review: TightVNC has no official policy of peer review. There is also no SVN commit mailing list.

Other Observations: TightVNC makes light use of SorceForge’s management facilities and has 5 registered project administrators.

A.2.14 14, GTK+

“GTK, which stands for the Gimp ToolKit, is a library for creating graphical user interfaces.” [44]

Project Type [44]: Desktop Environment, Software Development, Libraries, Application Frameworks, Widget Sets

Types of Review: GTK+ does not have a formal peer review policy and does not have a mailing list that broadcasts git commits. GTK+ is part of the GNOME project which recommends patches be send to the Bugzilla repository and not the mailing list (See Section A.1.6 on GNOME).

Other Observations: The GTK+ core development team consists of nine developers (`http://www.gtk.org/development.html`). Each week the team means in a public IRC channel.

A.2.15 15, libjpeg

“libjpeg is a library for handling the JPEG (JFIF) image format.” [44]

Project Type [44]: multimedia, Graphics, Graphics Conversion, Software Development, Libraries

Types of Review: No available information.

Other Observations: Although many projects depend on libjpeg, both in OSS and industry, the details surrounding the project are thin http://www.ijg.org/. This lack of information is likely because the project performs the limited set of function well and needs very little modification.

A.2.16 16, WebGUI

"WebGUI is a content management framework built to allow average business users to build and maintain complex Web sites." [44]

Project Type [44]: Internet, Web, Dynamic Content, Site Management, Office/Business, Software Development, Libraries, Application Frameworks, Widget Sets

Types of Review: WebGUI, which is developed by the Plain Black Corporation, has no mention of peer review or patch processes. There are also no mailing lists. Discussion is done on a forum and through a bugtracker. It appears that the corporation controls and does most of the development internally.

Other Observations: WebGUI was developed by the Plain Black Corporation and released under the GPL in 2001 (http://www.plainblack.com/plain-black-corporation). The corporation controls the direction of WebGUI and sells support services to a large variety of organizations.

A.2.17 17, Nmap Security Scanner

"Nmap ("Network Mapper") is a utility for network exploration, administration, and security auditing." [44]

Project Type [44]: Security, Networking, Firewalls

Types of Review: Bugs and patches are sent to the NMap developers' list (RTC) http://nmap.org/svn/HACKING. There does not appear to be a commit mailing list (no CTR).

Other Observations: Network Mapper appears to be controlled by a single developer, Fyodor, who is supported by a larger community that discusses issues on the developers' list http://nmap.org/book/man-bugs.html.

A.2.18 18, DokuWiki

“DokuWiki is a standards-compliant, simple-to-use Wiki mainly aimed at creating documentation of any kind.” [44]

Project Type [44]: Text Processing, Markup, HTML/XHTML, Software Development, Documentation, Information Management, Document Repositories, Internet, Web, Dynamic Content, Wiki

Types of Review: DokuWiki uses both RTC and CTR. There is no formal RTC policy, but patches should be sent to the developers’ list (http://www.dokuwiki.org/devel:patches). The Darcs version control system sends out, on a daily basis, links to and a summary of all source changes to the developers’ list for CTR. For example see email at http://www.freelists.org/post/dokuwiki/darcs-changes-20090120. Bug-tracking is also used.

Other Observations: DokuWiki was developed by Gohr who continues development with support from other developers http://en.wikipedia.org/wiki/DokuWiki.

A.2.19 19, Samba

“Samba is a software suite that provides seamless file and print services to SMB/CIFS clients.” [44]

Project Type [44]: Communications, File Sharing

Types of Review: Samba uses a basic style of RTC and CTR http://www.samba.org/samba/devel/TODO.html. The project recommends that developers without commit privileges find a core developer who they can begin to discuss their patch with as early as possible. Patches are sent to the developers’ mailing list, while commits to the central git repository are sent to a commits mailing list.

Other Observations: At any given time there are between 10 and 12 core committing developers (http://www.samba.org/samba/team/). The samba team consists of 30 individuals who have write access to the shared git repository.

Appendix B

Mining Procedure

We used the follow sequence of steps, each step represents a series of scripts:

1. Download the appropriate mailing lists
2. Create databases and extract emails
3. Run test queries to check for data inconsistencies and other errors
4. Re-thread messages (See Section B.1)
5. Parse out any mime encoded email into plaintext (most patches are already plaintext)
6. Parse commit mailing list
7. Parse all commits and diffs for size, files changed, and other attributes
8. All diffs on the mailing list are RTC and all commits with responses are CTRs
9. Randomly select a number of reviews to make sure that they are "real" reviews
10. Parse all "from" email addresses and all recipient (*e.g.,* "Cc") addresses
11. Use Bird *et al.*'s [14] name aliasing tool to resolve multiple email addresses to a single individual. Aliasing is a largely manual process. On large projects only alias addresses who have send more than 30 messages. We have been able to successfully alias up to 10K distinct email addresses
12. Run scripts to extract the measures. For example, the parameters of review discussed in Section 3.1.
13. Randomly select reviews and ensure that measures are accurate
14. Use R scripts to analyze the results [113]

B.1 Re-threading emails

We created scripts to extract the mailing list data into a database. A mail script extracted the mail headers including sender, in-reply-to, and date headers. The date header was normalized to GMT time. Once in the database we threaded messages by following the references and in-reply-to headers. For more information see RFC 2822 [115].

Since the references and in-reply-to headers are not required, some of the messages (replies to commits are left to later discussion) did not have a valid reply header. These messages were re-threaded using the date and subject of the invalid message. Valid messages with a similar subject ("re:" was striped off) that where within a seven day period prior of the invalid message were selected as potential parents. If this process left more than one potential parent, we selected the one closest in date to the invalid message. Some threading algorithms [1] use only subject matching. This technique is dangerous when subjects are common. For example, the subject "patch" was frequently used in the early days of Apache development, so threading on this subject alone would result in messages being incorrectly associated with each other. However, some message subjects still started with "re:" but did not have a parent message. Examining some messages further, indicated that their parent message was not in the mailing list. We suspect that some messages started in private or on mailing lists we do not have, but when it becomes necessary for a group, the mailing list is included in the reply. However, this explanation is one of many.

Since replies to a commit (CTR) will appear on the mailing list as a thread without a parent, we threaded these messages separately. We re-threaded messages associated with a commit as follows. First, all messages with a reply header that corresponded to a message (a commit) on the commit mailing list were threaded. Second, since the subject is usually the names of first few files involved in the commit and developers often made a series of commits to similar set of files within a short period of time, using subjects alone was not initially useful. Instead we choose a 3 day range and only re-threaded messages with one parent that had a matching subject. Finally, we used the algorithm mentioned above to re-thread siblings; in this case, the first reply to the commit is considered the parent. These techniques combined reduced the number of messages without a valid parent. Since the subject is automatically created and is a subset of the files modified, it is not indicative of the actual problem or even the files under discussion. We used a more conservative approach

[1] See http://www.jwz.org/doc/threading.html November 2006

when re-threading these messages.

B.2 File path matching problems

To calculate the file path based expertise measure in Section 3.5.2, we must extract the path names of change diffs committed to the version control system as well as patch diffs sent to the mailing list. Unfortunately, the path committed diffs in the version control system often differs from the path used by an individual to create a patch diff. In some projects, the file path in the version control system was contained within the path for the created diff. This allowed us to find the version control path substring within the patch diffs. This technique proved effective for Apache, Linux, and KDE. For Subversion, FreeBSD, and Gnome this technique matched less than 50% of the paths. Future work may improve upon these path matching techniques; however, for this measure we examine only the former three projects.

Appendix C

Models of Efficiency and Effectiveness

This appendix contains ten models of efficiency, *i.e.* review interval, and ten models of effectiveness, *i.e.* number of issues found. There are 20 models in total because we examine six projects, four of which have two review types, and we have two response variables. The discussion of these models can be found in Chapter 4.2.

C.1 Models of Efficiency

On the subsequent pages we provide the analysis of variance tables and diagnostics (constant variance and normal errors) plots of our efficiency models. To model efficiency, measured by the continuous variable review interval, we use a multiple linear regression model with constant variance and normal errors [30, pg. 120]. The discussion of these models can be found in Section 4.2.1.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	32.37	32.37	11.59	0.0007
log(auth work + 1)	1	72.19	72.19	25.85	0.0000
log(rev age + 1)	1	477.65	477.65	171.00	0.0000
log(rev work + 1)	1	337.53	337.53	120.83	0.0000
log(files + 1)	1	282.06	282.06	100.98	0.0000
log(churn + 1)	1	233.54	233.54	83.61	0.0000
log(diffs)	1	216.74	216.74	77.59	0.0000
log(replies)	1	1957.00	1957.00	700.61	0.0000
log(files + 1):log(churn + 1)	1	0.81	0.81	0.29	0.5895
Residuals	3422	9558.62	2.79		

Table C.1: Apache RTC: $R^2 = .27$

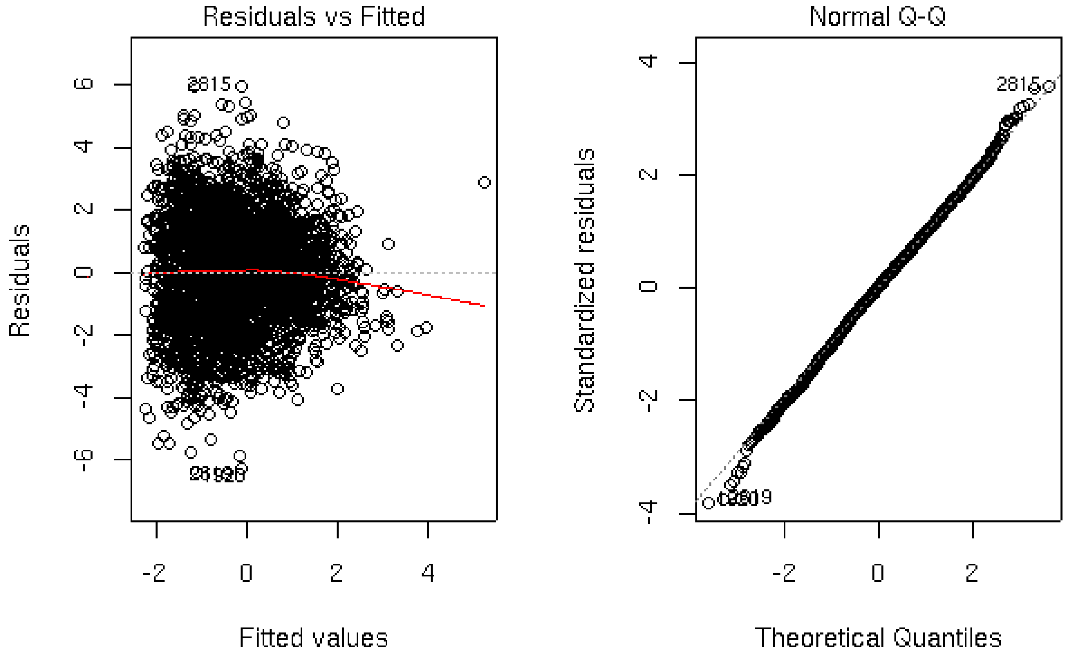


Figure C.1: Diagnostic plots for Apache RTC

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	31.66	31.66	10.64	0.0011
log(auth work + 1)	1	77.71	77.71	26.12	0.0000
log(rev age + 1)	1	79.43	79.43	26.69	0.0000
log(rev work + 1)	1	479.77	479.77	161.23	0.0000
log(files + 1)	1	452.00	452.00	151.89	0.0000
log(churn + 1)	1	520.33	520.33	174.85	0.0000
log(diffs)	1	344.36	344.36	115.72	0.0000
log(replies)	1	1640.97	1640.97	551.44	0.0000
log(files + 1):log(churn + 1)	1	6.40	6.40	2.15	0.1427
Residuals	2775	8257.76	2.98		

Table C.2: SVN RTC: $R^2 = .30$

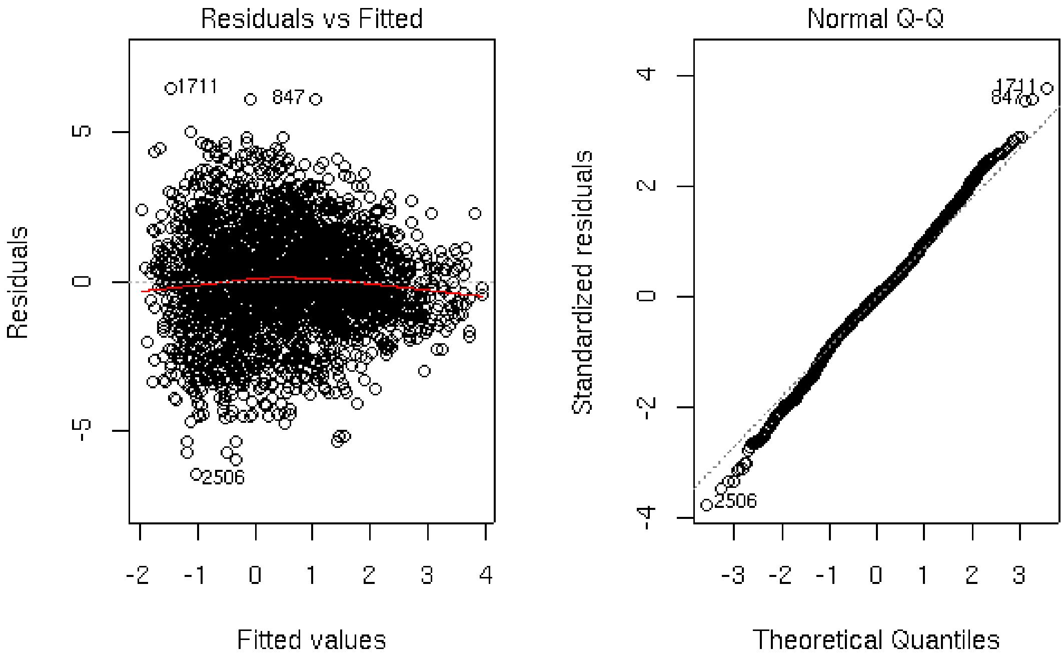


Figure C.2: Diagnostic plots for SVN RTC

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	24.53	24.53	8.12	0.0044
log(auth work + 1)	1	1180.94	1180.94	391.13	0.0000
log(rev age + 1)	1	1096.24	1096.24	363.08	0.0000
log(rev work + 1)	1	7091.73	7091.73	2348.79	0.0000
log(files + 1)	1	4091.21	4091.21	1355.01	0.0000
log(churn + 1)	1	2218.75	2218.75	734.85	0.0000
log(diffs)	1	2457.39	2457.39	813.89	0.0000
log(replies)	1	14967.96	14967.96	4957.41	0.0000
log(files + 1):log(churn + 1)	1	196.50	196.50	65.08	0.0000
Residuals	27657	83505.12	3.02		

Table C.3: Linux RTC: $R^2 = .29$

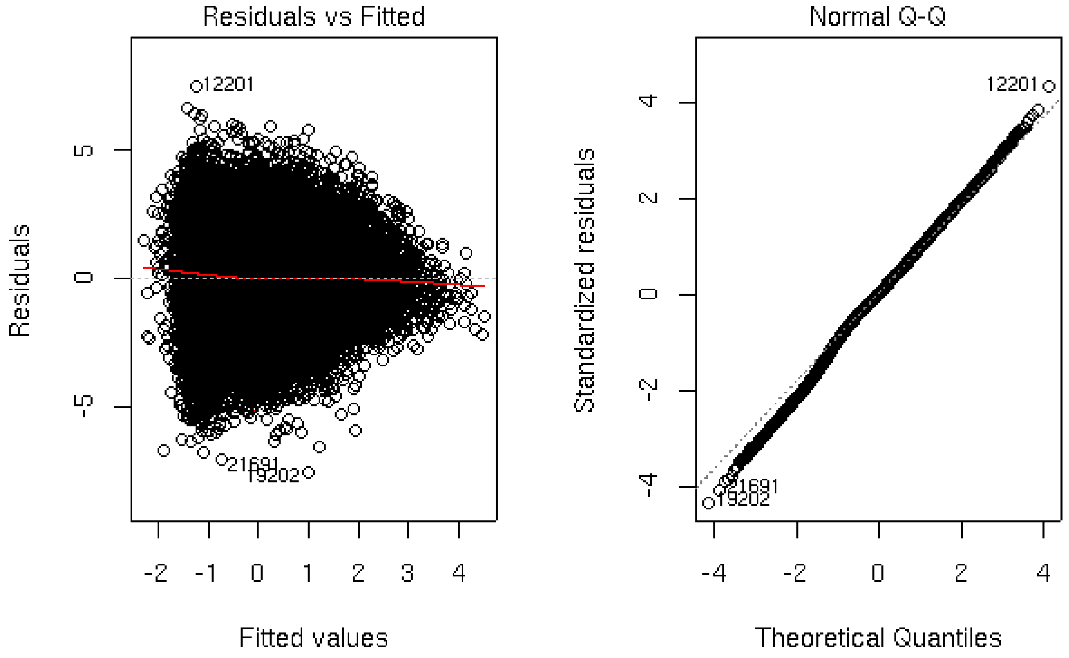


Figure C.3: Diagnostic plots for Linux RTC

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	73.36	73.36	24.97	0.0000
log(auth work + 1)	1	535.25	535.25	182.18	0.0000
log(rev age + 1)	1	5816.69	5816.69	1979.74	0.0000
log(rev work + 1)	1	339.76	339.76	115.64	0.0000
log(files + 1)	1	653.79	653.79	222.52	0.0000
log(churn + 1)	1	1521.87	1521.87	517.98	0.0000
log(diffs)	1	3511.03	3511.03	1195.00	0.0000
log(replies)	1	15554.42	15554.42	5294.03	0.0000
log(files + 1):log(churn + 1)	1	17.31	17.31	5.89	0.0152
Residuals	25394	74610.30	2.94		

Table C.4: FreeBSD RTC: $R^2 = .27$

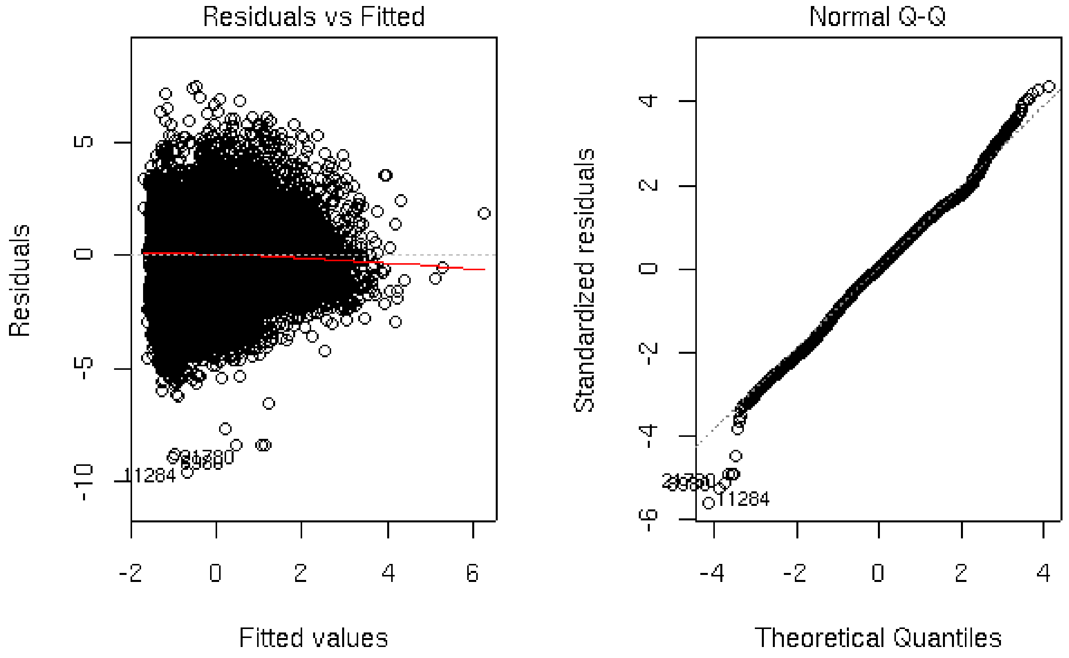


Figure C.4: Diagnostic plots for FreeBSD RTC

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	182.39	182.39	52.43	0.0000
log(auth work + 1)	1	3116.03	3116.03	895.77	0.0000
log(rev age + 1)	1	1568.05	1568.05	450.77	0.0000
log(rev work + 1)	1	2193.47	2193.47	630.57	0.0000
log(files + 1)	1	231.37	231.37	66.51	0.0000
log(churn + 1)	1	137.84	137.84	39.63	0.0000
log(diffs)	1	5203.06	5203.06	1495.74	0.0000
log(replies)	1	8191.15	8191.15	2354.74	0.0000
log(files + 1):log(churn + 1)	1	10.77	10.77	3.10	0.0785
Residuals	8146	28336.53	3.48		

Table C.5: KDE RTC: $R^2 = .43$

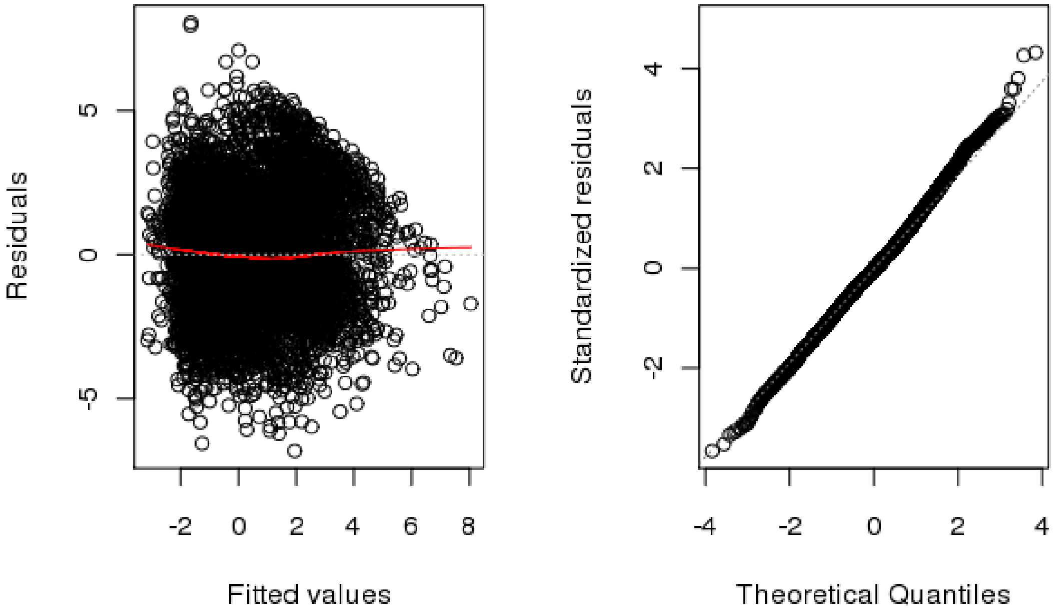


Figure C.5: Diagnostic plots for KDE RTC

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	7.79	7.79	2.52	0.1128
log(auth work + 1)	1	199.45	199.45	64.43	0.0000
log(rev age + 1)	1	407.11	407.11	131.51	0.0000
log(rev work + 1)	1	553.60	553.60	178.83	0.0000
log(files + 1)	1	607.32	607.32	196.18	0.0000
log(churn + 1)	1	868.08	868.08	280.41	0.0000
log(diffs)	1	909.43	909.43	293.77	0.0000
log(replies)	1	4815.35	4815.35	1555.47	0.0000
log(files + 1):log(churn + 1)	1	0.45	0.45	0.14	0.7035
Residuals	7931	24552.46	3.10		

Table C.6: Gnome RTC: $R^2 = .25$

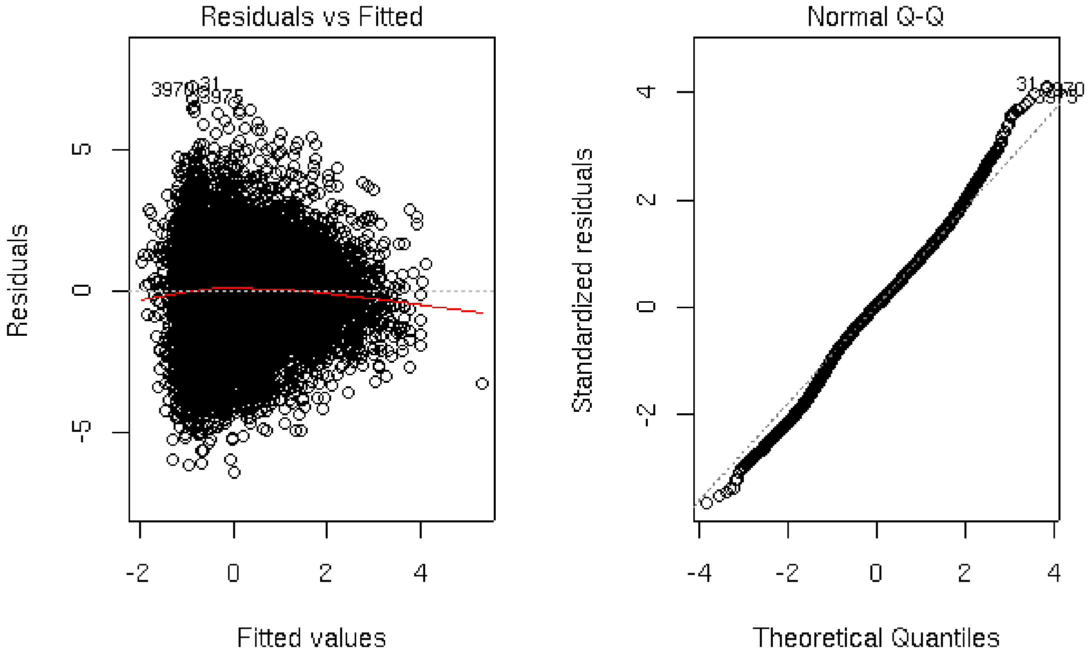


Figure C.6: Diagnostic plots for Gnome RTC

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	1.44	1.44	0.44	0.5079
log(auth work + 1)	1	44.55	44.55	13.60	0.0002
log(rev age + 1)	1	329.56	329.56	100.59	0.0000
log(rev work + 1)	1	128.51	128.51	39.22	0.0000
log(files + 1)	1	97.18	97.18	29.66	0.0000
log(churn + 1)	1	44.40	44.40	13.55	0.0002
log(diffs)	1	148.72	148.72	45.39	0.0000
log(replies)	1	2294.08	2294.08	700.20	0.0000
log(files + 1):log(churn + 1)	1	3.56	3.56	1.09	0.2975
Residuals	2523	8266.13	3.28		

Table C.7: Apache CTR: $R^2 = .27$

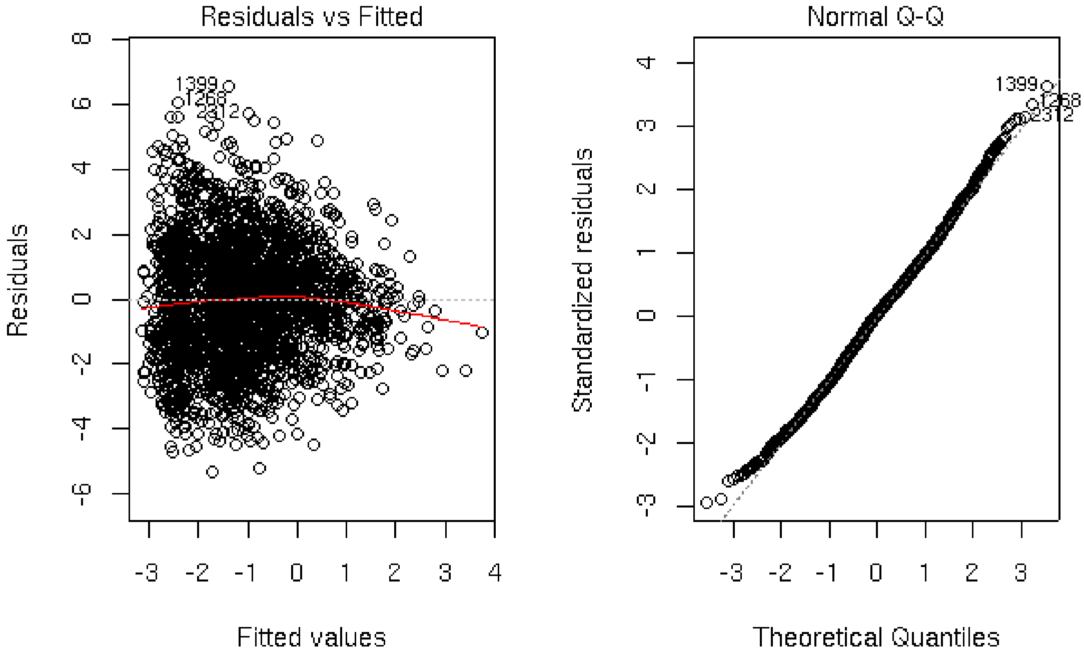


Figure C.7: Diagnostic plots for Apache CTR

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	1.80	1.80	0.56	0.4552
log(auth work + 1)	1	1.09	1.09	0.34	0.5609
log(rev age + 1)	1	123.49	123.49	38.21	0.0000
log(rev work + 1)	1	400.08	400.08	123.80	0.0000
log(files + 1)	1	172.68	172.68	53.43	0.0000
log(churn + 1)	1	57.31	57.31	17.74	0.0000
log(diffs)	1	104.58	104.58	32.36	0.0000
log(replies)	1	1216.22	1216.22	376.34	0.0000
log(files + 1):log(churn + 1)	1	11.79	11.79	3.65	0.0562
Residuals	2121	6854.52	3.23		

Table C.8: SVN CTR: $R^2 = .23$

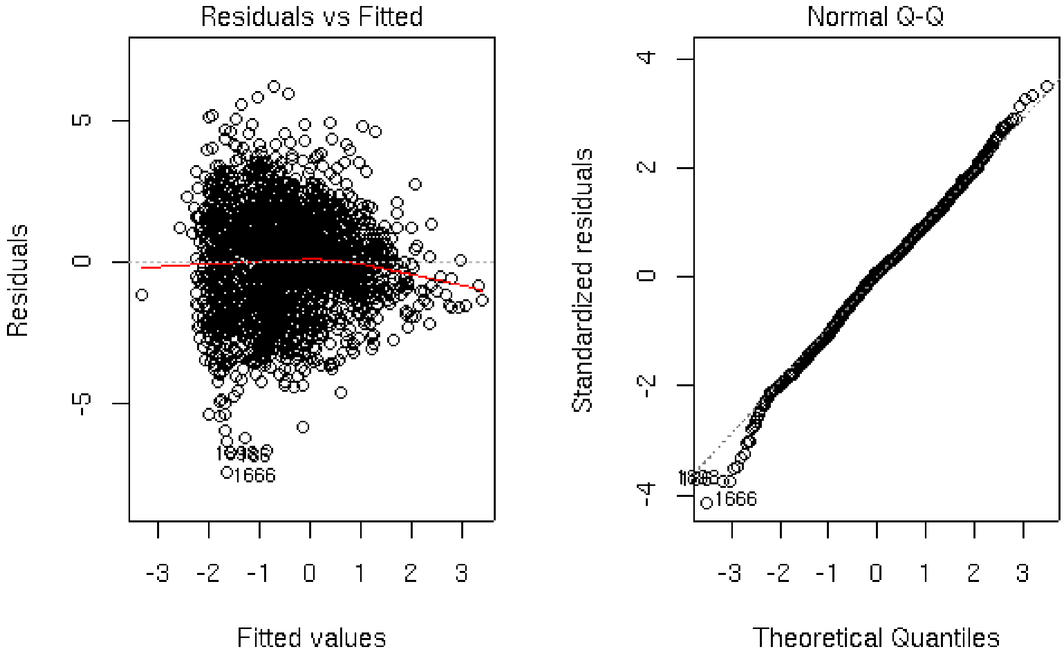


Figure C.8: Diagnostic plots for SVN CTR

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	110.89	110.89	32.29	0.0000
log(auth work + 1)	1	30.33	30.33	8.83	0.0030
log(rev age + 1)	1	3251.09	3251.09	946.77	0.0000
log(rev work + 1)	1	1479.78	1479.78	430.94	0.0000
log(files + 1)	1	254.93	254.93	74.24	0.0000
log(churn + 1)	1	402.85	402.85	117.32	0.0000
log(diffs)	1	2048.51	2048.51	596.56	0.0000
log(replies)	1	15369.67	15369.67	4475.91	0.0000
log(files + 1):log(churn + 1)	1	1.41	1.41	0.41	0.5223
Residuals	21986	75497.01	3.43		

Table C.9: FreeBSD CTR: $R^2 = .23$

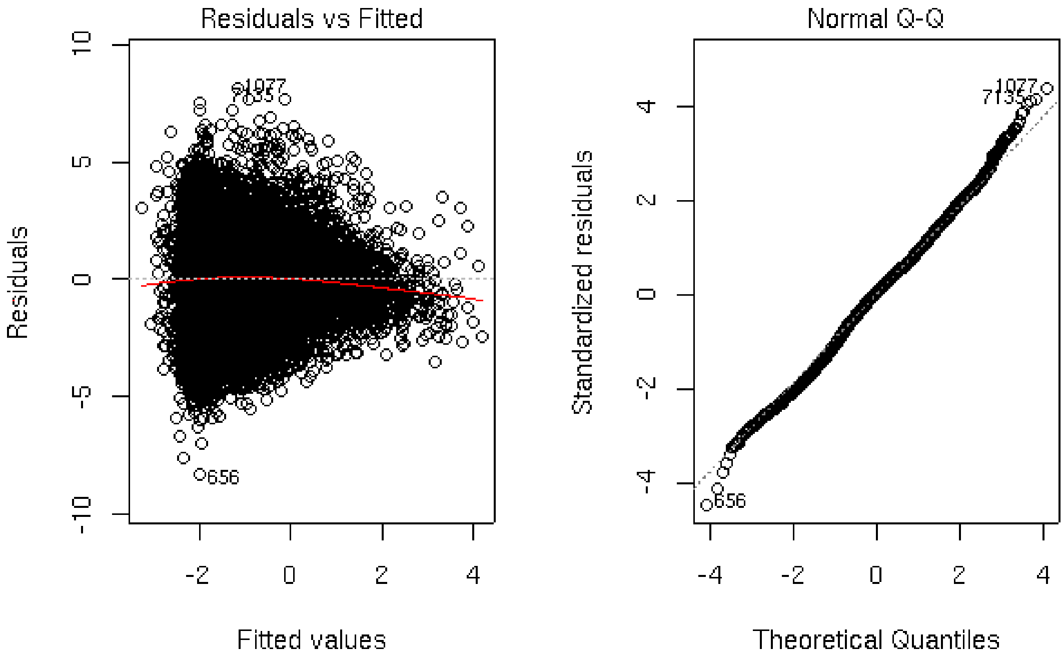


Figure C.9: Diagnostic plots for FreeBSD CTR

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	170.00	170.00	47.55	0.0000
log(auth work + 1)	1	18.88	18.88	5.28	0.0216
log(rev age + 1)	1	31.77	31.77	8.89	0.0029
log(rev work + 1)	1	300.14	300.14	83.96	0.0000
log(files + 1)	1	210.43	210.43	58.86	0.0000
log(churn + 1)	1	138.77	138.77	38.82	0.0000
log(diffs)	1	549.58	549.58	153.73	0.0000
log(replies)	1	11227.31	11227.31	3140.58	0.0000
log(files + 1):log(churn + 1)	1	10.11	10.11	2.83	0.0926
Residuals	14760	52765.70	3.57		

Table C.10: KDE CTR: $R^2 = .19$

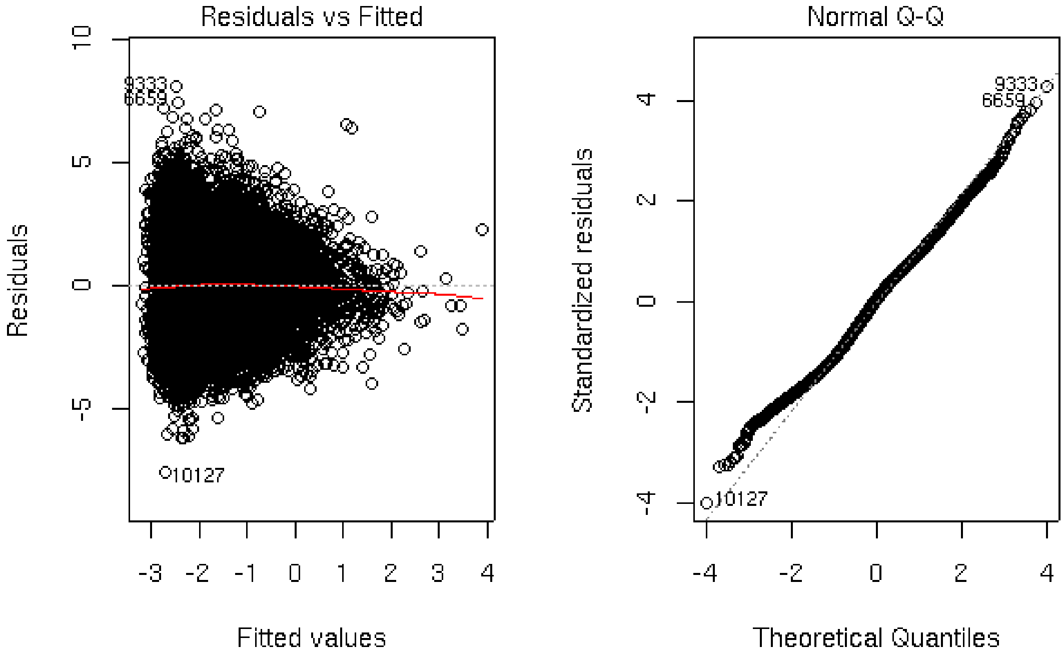


Figure C.10: Diagnostic plots for KDE CTR

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	14.17	14.17	1.87	0.1716
log(auth work + 1)	1	92.35	92.35	12.19	0.0005
log(rev age + 1)	1	473.99	473.99	62.57	0.0000
log(rev work + 1)	1	1118.48	1118.48	147.64	0.0000
log(files + 1)	1	823.45	823.45	108.70	0.0000
log(churn + 1)	1	183.63	183.63	24.24	0.0000
log(diffs)	1	1163.11	1163.11	153.53	0.0000
log(reviewers)	1	1368.27	1368.27	180.61	0.0000
log(files + 1):log(churn + 1)	1	262.33	262.33	34.63	0.0000
Residuals	3422	25924.05	7.58		

Table C.11: Apache RTC $R^2 = .29$

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	339.68	339.68	14.55	0.0001
log(auth work + 1)	1	7.38	7.38	0.32	0.5740
log(rev age + 1)	1	153.14	153.14	6.56	0.0105
log(rev work + 1)	1	6986.80	6986.80	299.18	0.0000
log(files + 1)	1	8115.03	8115.03	347.49	0.0000
log(churn + 1)	1	6719.32	6719.32	287.73	0.0000
log(diffs)	1	6031.21	6031.21	258.26	0.0000
log(reviewers)	1	1197.20	1197.20	51.26	0.0000
log(files + 1):log(churn + 1)	1	3179.84	3179.84	136.16	0.0000
Residuals	2775	64805.35	23.35		

Table C.12: SVN RTC $R^2 = .54$

C.2 Models of Effectiveness

On the subsequent pages we provide the analysis of deviance tables for our effectiveness models. To model effectiveness, measured by a count of the number of issues found, we use a generalized liner model (GLM) with Poisson errors and a logarithmic link function. A quasi-poisson model and a pseudo R-squared measure is used to correct for over- and under-dispersion [59] [89, pg. 219]. The discussion of these models can be found in Section 4.2.2.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	19.76	19.76	0.49	0.4819
log(auth work + 1)	1	896.91	896.91	22.44	0.0000
log(rev age + 1)	1	8619.96	8619.96	215.71	0.0000
log(rev work + 1)	1	87476.99	87476.99	2189.09	0.0000
log(files + 1)	1	102287.71	102287.71	2559.72	0.0000
log(churn + 1)	1	50443.55	50443.55	1262.34	0.0000
log(diffs)	1	51474.60	51474.60	1288.14	0.0000
log(reviewers)	1	104882.92	104882.92	2624.67	0.0000
log(files + 1):log(churn + 1)	1	10583.63	10583.63	264.85	0.0000
Residuals	27657	1105187.25	39.96		

Table C.13: Linux RTC $R^2 = .58$

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	575.78	575.78	151.01	0.0000
log(auth work + 1)	1	160.48	160.48	42.09	0.0000
log(rev age + 1)	1	6594.70	6594.70	1729.60	0.0000
log(rev work + 1)	1	1689.22	1689.22	443.03	0.0000
log(files + 1)	1	406.24	406.24	106.54	0.0000
log(churn + 1)	1	3030.80	3030.80	794.89	0.0000
log(diffs)	1	14779.84	14779.84	3876.31	0.0000
log(reviewers)	1	12948.61	12948.61	3396.04	0.0000
log(files + 1):log(churn + 1)	1	125.69	125.69	32.96	0.0000
Residuals	25394	96823.76	3.81		

Table C.14: FreeBSD RTC $R^2 = .46$

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	48.94	48.94	8.92	0.0028
log(auth work + 1)	1	166.03	166.03	30.26	0.0000
log(rev age + 1)	1	17.58	17.58	3.20	0.0735
log(rev work + 1)	1	1337.47	1337.47	243.79	0.0000
log(files + 1)	1	1929.02	1929.02	351.62	0.0000
log(churn + 1)	1	1036.43	1036.43	188.92	0.0000
log(diffs)	1	296.88	296.88	54.12	0.0000
log(reviewers)	1	3882.01	3882.01	707.60	0.0000
log(files + 1):log(churn + 1)	1	267.72	267.72	48.80	0.0000
Residuals	8146	44690.08	5.49		

Table C.15: KDE RTC $R^2 = .26$

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	24.21	24.21	0.28	0.5937
log(auth work + 1)	1	6095.71	6095.71	71.63	0.0000
log(rev age + 1)	1	2190.43	2190.43	25.74	0.0000
log(rev work + 1)	1	3682.75	3682.75	43.28	0.0000
log(files + 1)	1	4646.44	4646.44	54.60	0.0000
log(churn + 1)	1	10769.48	10769.48	126.55	0.0000
log(diffs)	1	14268.75	14268.75	167.67	0.0000
log(reviewers)	1	12031.40	12031.40	141.38	0.0000
log(files + 1):log(churn + 1)	1	2255.83	2255.83	26.51	0.0000
Residuals	7931	674917.17	85.10		

Table C.16: Gnome RTC $R^2 = .37$

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	0.51	0.51	0.21	0.6433
log(auth work + 1)	1	0.08	0.08	0.03	0.8532
log(rev age + 1)	1	4.36	4.36	1.84	0.1746
log(rev work + 1)	1	22.70	22.70	9.59	0.0020
log(files + 1)	1	52.99	52.99	22.39	0.0000
log(churn + 1)	1	97.28	97.28	41.10	0.0000
log(diffs)	1	283.91	283.91	119.96	0.0000
log(reviewers)	1	27.06	27.06	11.43	0.0007
log(files + 1):log(churn + 1)	1	4.40	4.40	1.86	0.1731
Residuals	2523	5971.22	2.37		

Table C.17: Apache CTR $R^2 = .18$

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	1.73	1.73	0.22	0.6415
log(auth work + 1)	1	5.11	5.11	0.64	0.4232
log(rev age + 1)	1	29.47	29.47	3.70	0.0545
log(rev work + 1)	1	310.65	310.65	39.00	0.0000
log(files + 1)	1	1207.76	1207.76	151.64	0.0000
log(churn + 1)	1	578.49	578.49	72.63	0.0000
log(diffs)	1	698.63	698.63	87.72	0.0000
log(reviewers)	1	207.57	207.57	26.06	0.0000
log(files + 1):log(churn + 1)	1	84.80	84.80	10.65	0.0011
Residuals	2121	16892.63	7.96		

Table C.18: SVN CTR $R^2 = .32$

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	6.58	6.58	19.06	0.0000
log(auth work + 1)	1	0.38	0.38	1.11	0.2924
log(rev age + 1)	1	135.07	135.07	391.46	0.0000
log(rev work + 1)	1	253.50	253.50	734.69	0.0000
log(files + 1)	1	175.90	175.90	509.79	0.0000
log(churn + 1)	1	16.79	16.79	48.67	0.0000
log(diffs)	1	2164.92	2164.92	6274.47	0.0000
log(reviewers)	1	476.63	476.63	1381.39	0.0000
log(files + 1):log(churn + 1)	1	3.50	3.50	10.14	0.0015
Residuals	21986	7585.98	0.35		

Table C.19: FreeBSD CTR
$R^2 = .40$

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(auth age + 1)	1	3.31	3.31	10.86	0.0010
log(auth work + 1)	1	2.27	2.27	7.46	0.0063
log(rev age + 1)	1	1.30	1.30	4.25	0.0392
log(rev work + 1)	1	35.08	35.08	115.13	0.0000
log(files + 1)	1	6.79	6.79	22.27	0.0000
log(churn + 1)	1	14.59	14.59	47.88	0.0000
log(diffs)	1	356.93	356.93	1171.37	0.0000
log(reviewers)	1	212.50	212.50	697.36	0.0000
log(files + 1):log(churn + 1)	1	6.20	6.20	20.34	0.0000
Residuals	14760	4497.58	0.30		

Table C.20: KDE CTR $R^2 = .16$

Appendix D

Interviews

This appendix contains our interview questions and examples of interview responses and codes.

D.1 Interview Questions

In this section, we provide an example of a request for an interview, we give the telephone or long version of the interview, and the shortened version of the interview.

D.1.1 Email Interview Requests

An email request for interview was sent to each potential participate. The goal of this request was to show that I had invested in studying peer review and was not simply "another research" who wanted their time.

Hi <PARTICIPANT NAME>,

I have been studying peer review processes in OSS projects for the past 4 years. Most of my analysis has involved extracting metrics (e.g., review interval) from mailing list archives. In order to triangulate my findings, I have been interviewing OSS developers. Since you have been involved in a large number of patch reviews, I would greatly appreciate it if I could talk to you on Skype for 15 to 30 minutes. If you prefer, I can email the questions and you can respond inline.

KDE <FOR EXAMPLE> is especially interesting to me because, unlike most of the other projects I have examined (e.g., Linux, Apache), it is an ecosystem of open source projects. I would like to understand how the many lists (over 100) interact.

Thanks, Peter

D.1.2 Telephone and Email Interview Questions

Below are is the semi-structured interview questions that were used for both telephone and email interviews. In the next section, we show how they were modified to reduce the effort required from interviewees.

There are 7 categories of questions in total. Please feel free to answer as many or as few as you like. Also please provide as much or as little detail as you like: "yes/no" is fine, but if you have an interesting insight or story to share, I would greatly appreciate it. If you feel I've missed an important aspect of peer review in Linux or that a question is leading, please feel free to add additional information outside of the questions.

Implied Consent

By filling in this email interview and sending it to me, YOUR FREE AND INFORMED CONSENT IS IMPLIED. Please see the end of this email for more information regarding this study.

Please choose one of the following two options regarding the ANONYMITY of your data:

a. Full ANONYMITY: We will not use your name during the analysis or when reporting results.

b. Waive your right to ANONYMITY: In this case we will acknowledge you in any quotation we use.

The Email ‘Interview’

1. The mailing list

Do you keep up with all discussion on the mailing list? Do you feel overwhelmed by the amount of list traffic?

Do you differentiate between general list traffic and email sent to you directly?

What techniques do you employ to deal with the large quantities of information?

How many lists are you subscribed to (approximately)?

In the case of multiple lists, do you/are there individuals who forward messages to other lists and keep them informed of important developments?

How do you remain informed of future discussion on a particular topic or review?

2. Finding Reviews

How do you locate reviews?

How important is the structure of the email? (e.g., subject, log, links, diffs)

Why do you join a review?

Does the interest and expertise influence your decision?

Do you feel obliged to review code in a particular area?

What impact does the business of your schedule have on what you review?

How important is the past discussion when you join an existing review discussion?

Does the person who wrote the code influence your decision to perform a review?

3. Describe the review

Which roles do you see during a review?

Are these roles pre-specified or do they emerge?

Do these roles morph as time goes on? e.g., reviewer become author

Do you stop once your found a defect or do you continue to review the whole artifact?

How is conflict resolved?

4. Large patches and git (e.g., features)

How do you deal with large patches? e.g., implementing a new feature or fixing a significant bug?

How do you deal with the conflicting requirement to have “small, independent, complete patches” while still being able to examine interrelated, larger patches?

How has the use of git by yourself and/or others affected the way you perform reviews?

5. Outsiders

What impact do developers and users outside of the core group have on peer reviews?

Is outsider input generally productive or counter productive?

What type of information do these outsiders provide?

6. Rejected patches

For what reason do you see new code/ideas getting rejected?

How important is maintaining existing working code vs adding new functionality?

How important is trust, competence, and interest building around a patch?

Normally, how many attempts are made before a review will be committed or rejected (min/max)?

Do you see many instances of re-review i.e., providing the same patch to people with similar problems?

How important are mailing list manners?

7. Tools and process

What tools do you use during peer review?

How would you change the process (and tools) involved in conducting peer reviews?

Any other feedback or insights are welcome

Thank You!

(See Section D.1.4 for the implied consent form)

D.1.3 Short Email Interview

Some of our initial interviewees said that the interview was too long and many did not respond. In this case, we cut down the length of the interview and resent it to these interviewees. This shorter interview (below) increased response rates.

The email 'interview' should take 10 to 15 minutes to answer. There are 7 categories of questions and 18 questions in total. Please feel free to answer as many or as few as you like. Also please provide as much or as little detail as you like: "yes/no" is fine, but if you have an interesting insight or story to share, I would greatly appreciate it (references to an email threads are also great). If you feel I've missed an important aspect of peer review in <INSERT PROJECT NAME> or that a question is leading, please feel free to add additional information outside of the questions.

Thank you, Peter Rigby

Implied Consent

By filling in this email interview and sending it to me, YOUR FREE AND INFORMED CONSENT IS IMPLIED. Please see the bottom of this email for more information regarding this study.

Please choose one of the following two options regarding the ANONYMITY of your data:

a. Full ANONYMITY: We will not use your name during the analysis or when reporting results.

b. Waive your right to ANONYMITY: In this case we will acknowledge you in any quotation we use.

The Email 'Interview'

1. The mailing list

How many lists are you subscribed to (approximately)?

What techniques do you employ to deal with the large quantities of information?

In the case of multiple lists, do you/are there individuals who forward messages to other lists and keep them informed of important developments?

How important is <INSERT BUGTRACKER NAME> in the review process?

2. Finding Reviews

How do you determine what to review?

How does time influence your decision to review and the quality of the review?

How important is the structure of the email that contains the patch? (e.g., subject, log, links, diffs)

3. Describe the review

How long does it take you to perform the review? (min/avg/max)

How do reviews generally proceed? (e.g., patch, comments, ..., reject/accept)

What roles do you see during a review? (e.g., mediator, author)

4. Large patches and version control

Do you deal with large patches in a different way than small patches?

Does the version control system affected the way you perform reviews?

5. Outsiders

What impact do developers and users outside of the core group have on peer reviews?

What type of information do these outsiders provide?

6. Rejected patches

For what reasons do you see new code/ideas getting rejected?

Normally, how many attempts are made before a patch will be committed or rejected (min/avg/max)?

7. Tools and process

What tools do you use during peer review?

How would you change the process (and tools) involved in conducting peer reviews?

Any other feedback or insights are welcome

Thank You!

(See Section D.1.4 for the implied consent form)

D.1.4 Implied Consent Form

You are invited to participate in a study entitled Peer Review in Open Source Software (OSS) Projects that is being conducted by Peter Rigby, Daniel German, and Peggy Storey.

Peter Rigby is a graduate student in the department of Computer Science at the University of Victoria and you may contact him if you have further questions through email pcr@uvic.ca or by phone 250-472-5878.

As a graduate student, I am required to conduct research as part of the requirements for a PhD in Computer Science. It is being conducted under the supervision of Dr. German and Dr. Storey. You may contact Dr. German through email dmg@uvic.ca or by phone

250-472-5790.

This research is being funded through an NSERC CGSD.

Purpose and Objectives The purpose of this research project is to systematically and comparatively examine the peer review techniques used by successful open source projects and to understand how these techniques compare to those used in other settings, such as industrial development. Using automated analysis, we have extracted the basic quantitative parameters of peer review (e.g., the frequency of review and the review interval). The purpose of the current study is to collect data pertaining to parameters that cannot be extracted automatically from a mailing list (e.g., the effort and time involved in reviewing), to discuss how the group interacts while conducting reviews, and to ensure that our results make sense to OSS practitioners. We are also interested in the review policies used by the project and how these were developed. The contribution of this work is to deepen our understanding of peer reviews in open source software development and to suggest how successful practices might be incorporated into other (e.g., new) OSS projects and modified for use in other development environments.

Importance of this Research Unlike other types of engineering, software engineering is still developing processes and metrics that yield timely, high-quality products. Overly formal, management driven processes are not always the solution. We want to understand how peer review, an important quality assurance mechanism, is conducted in the OSS community.

Participants Selection You are being asked to participate in this study because you are an OSS developer who has in the past conducted or who currently conducts regular peer reviews in an OSS project.

What is involved If you agree to voluntarily participate in this research, your participation will include answering this email interview questions.

Risks There are no known or anticipated risks to you by participating in this research.

Benefits The potential benefits of your participation in this research include reflection on the peer review process that your project uses, a better understanding of the peer review techniques used by OSS projects, and the potential for software projects to adopt the most efficient and effective peer review technique.

Voluntary Participation Your participation in this research must be completely voluntary. If you do decide to participate, you may withdraw at any time without any consequences or any explanation. If you do withdraw from the study, your data will not be used in the study and will be deleted.

Anonymity There are two possibilities regarding anonymity. Please see the beginning of this email to select the level you desire.

Confidentiality Your confidentiality and the confidentiality of the data will be protected by keeping it on a password protected computer that only the research team has access to.

Dissemination of Results It is anticipated that the results of this study will be shared with others in the following ways: published in articles, presented at scholarly meetings, and published in a thesis.

Disposal of Data Data from this study will be permanently deleted within three years of collection.

Contacts Individuals that may be contacted regarding this study include Peter Rigby (pcr@uvic.ca) and Daniel German (dmg@uvic.ca).

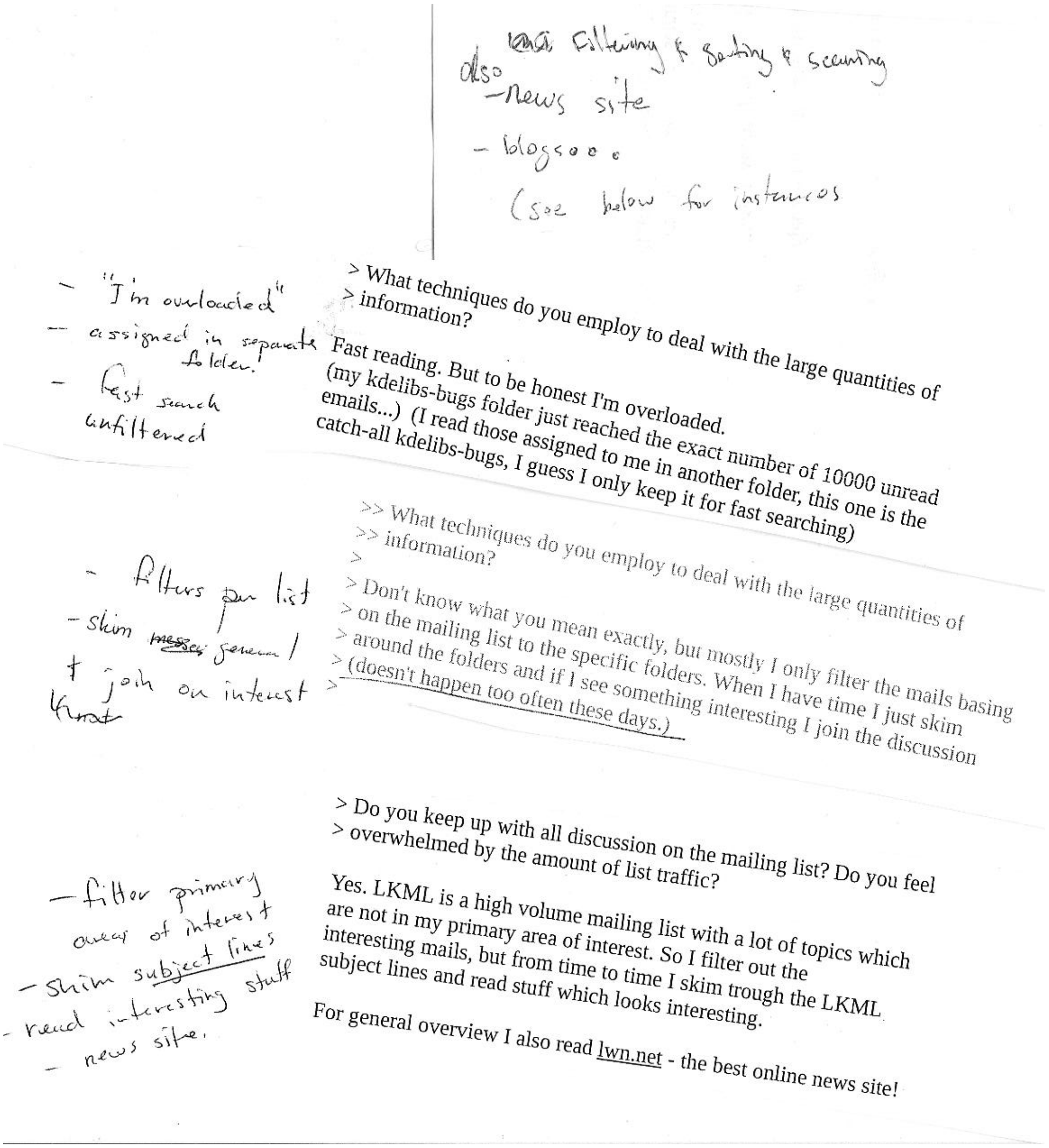
In addition, you may verify the ethical approval of this study, or raise any concerns you might have, by contacting the Human Research Ethics Office at the University of Victoria (250-472-4545 or).

By filling in this email interview and sending it to me, YOUR FREE AND INFORMED

CONSENT IS IMPLIED and indicates that you understand the above conditions of participation in this study and that you have had the opportunity to have your questions answered by the researchers.

D.2 Examples of Coding and Memoing Interviews

The following images represent the "pile" of evidence around the use of filtering by interviewees. Codes can be seen written in the margin. The theme name is written on a note at the beginning of Image D.1. More details about the filtering them can be found in Section 5.3.1.



also Filtering & Sorting & Searching
- news site
- blogs...
(see below for instances

- "I'm overloaded"
- assigned in separate folder!
- fast search unfiltered

> What techniques do you employ to deal with the large quantities of
> information?

Fast reading. But to be honest I'm overloaded.
(my kdelibs-bugs folder just reached the exact number of 10000 unread emails...) (I read those assigned to me in another folder, this one is the catch-all kdelibs-bugs, I guess I only keep it for fast searching)

- filters per list
- skim ~~messes~~ general / + join on interest
~~Kmat~~

>> What techniques do you employ to deal with the large quantities of
>> information?
>
> Don't know what you mean exactly, but mostly I only filter the mails basing
> on the mailing list to the specific folders. When I have time I just skim
> around the folders and if I see something interesting I join the discussion
> (doesn't happen too often these days.)
>

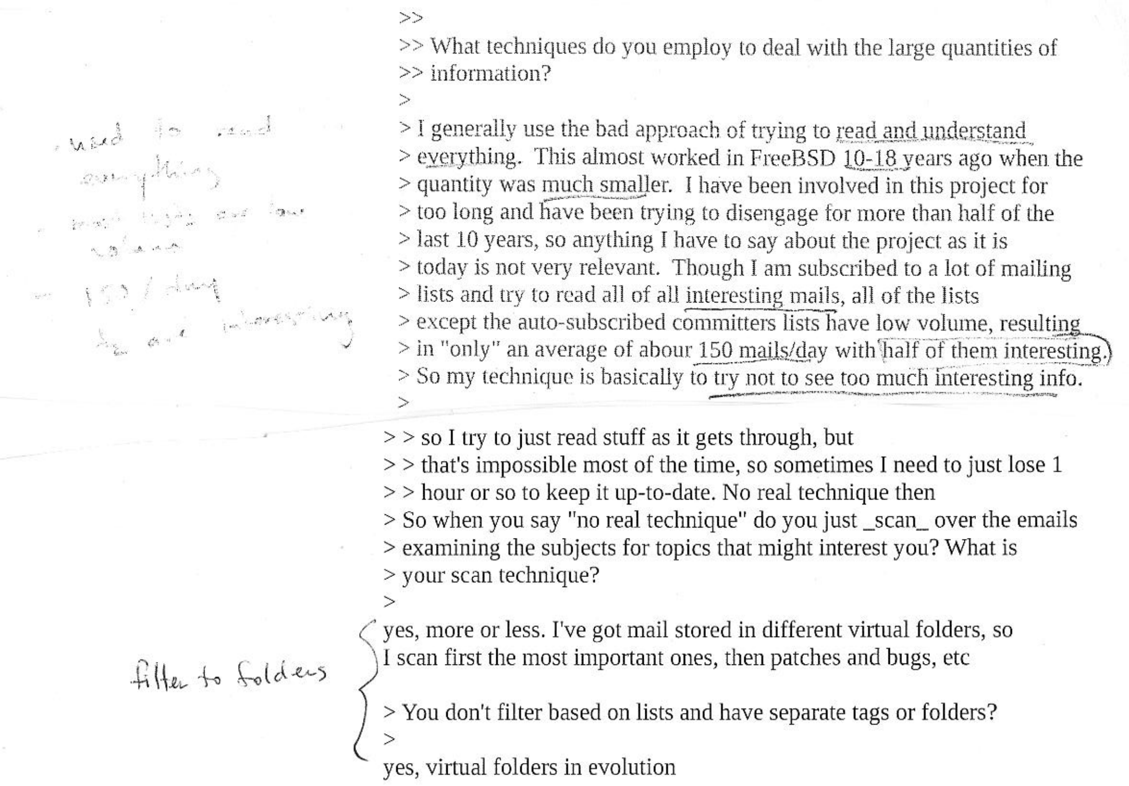
- filter primary areas of interest
- skim subject lines
- read interesting stuff
- news site.

> Do you keep up with all discussion on the mailing list? Do you feel
> overwhelmed by the amount of list traffic?

Yes. LKML is a high volume mailing list with a lot of topics which are not in my primary area of interest. So I filter out the interesting mails, but from time to time I skim trough the LKML subject lines and read stuff which looks interesting.

For general overview I also read lwn.net - the best online news site!

Figure D.1: Interview responses and coding for filtering theme (See Section 5.3.1)



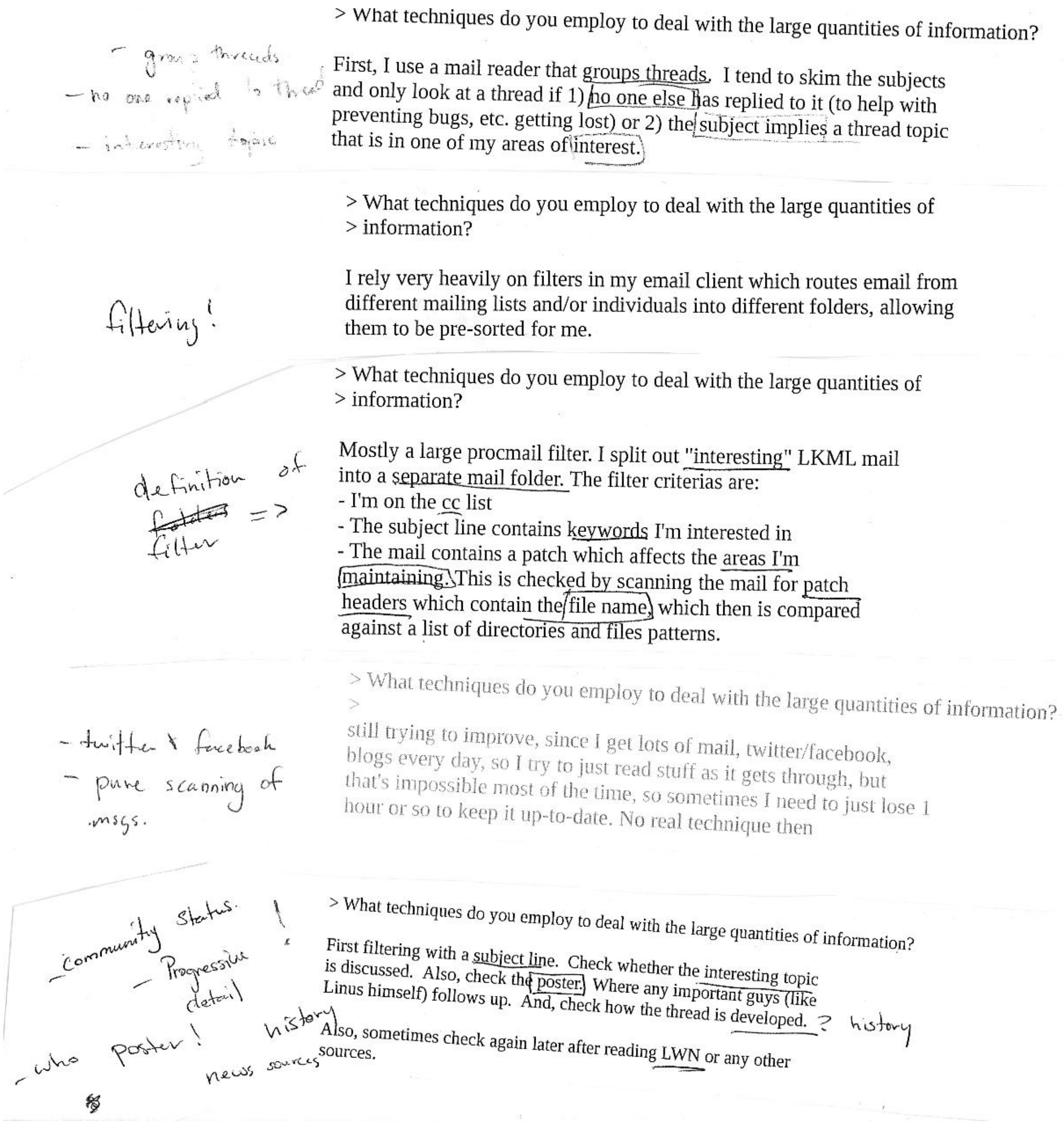
```
>>
>> What techniques do you employ to deal with the large quantities of
>> information?
>
> I generally use the bad approach of trying to read and understand
> everything.  This almost worked in FreeBSD 10-18 years ago when the
> quantity was much smaller.  I have been involved in this project for
> too long and have been trying to disengage for more than half of the
> last 10 years, so anything I have to say about the project as it is
> today is not very relevant.  Though I am subscribed to a lot of mailing
> lists and try to read all of all interesting mails, all of the lists
> except the auto-subscribed committers lists have low volume, resulting
> in "only" an average of abour 150 mails/day with half of them interesting.
> So my technique is basically to try not to see too much interesting info.
>
> > so I try to just read stuff as it gets through, but
> > that's impossible most of the time, so sometimes I need to just lose 1
> > hour or so to keep it up-to-date. No real technique then
> So when you say "no real technique" do you just _scan_ over the emails
> examining the subjects for topics that might interest you? What is
> your scan technique?
>
yes, more or less. I've got mail stored in different virtual folders, so
I scan first the most important ones, then patches and bugs, etc

> You don't filter based on lists and have separate tags or folders?
>
yes, virtual folders in evolution
```

- used to read everything
- most lists are low volume
- 150/day ½ are interesting

filter to folders

Figure D.2: Interview responses and coding for filtering theme (See Section 5.3.1)



> What techniques do you employ to deal with the large quantities of information?

First, I use a mail reader that groups threads. I tend to skim the subjects and only look at a thread if 1) no one else has replied to it (to help with preventing bugs, etc. getting lost) or 2) the subject implies a thread topic that is in one of my areas of interest.

- group threads
- no one replied to thread
- interesting topic

> What techniques do you employ to deal with the large quantities of
> information?

I rely very heavily on filters in my email client which routes email from different mailing lists and/or individuals into different folders, allowing them to be pre-sorted for me.

filtering!

> What techniques do you employ to deal with the large quantities of
> information?

Mostly a large procmail filter. I split out "interesting" LKML mail into a separate mail folder. The filter criterias are:
- I'm on the cc list
- The subject line contains keywords I'm interested in
- The mail contains a patch which affects the areas I'm maintaining. This is checked by scanning the mail for patch headers which contain the file name, which then is compared against a list of directories and files patterns.

definition of ~~folder~~ => filter

> What techniques do you employ to deal with the large quantities of information?
>
still trying to improve, since I get lots of mail, twitter/facebook, blogs every day, so I try to just read stuff as it gets through, but that's impossible most of the time, so sometimes I need to just lose 1 hour or so to keep it up-to-date. No real technique then

- twitter & facebook
- pure scanning of msgs.

> What techniques do you employ to deal with the large quantities of information?

First filtering with a subject line. Check whether the interesting topic is discussed. Also, check the poster. Where any important guys (like Linus himself) follows up. And, check how the thread is developed. ? history

Also, sometimes check again later after reading LWN or any other sources.

- community status. !
- Progressive detail
- who poster!
history
news sources

Figure D.3: Interview responses and coding for filtering theme (See Section 5.3.1)

Appendix E

Manual Analysis of Reviews

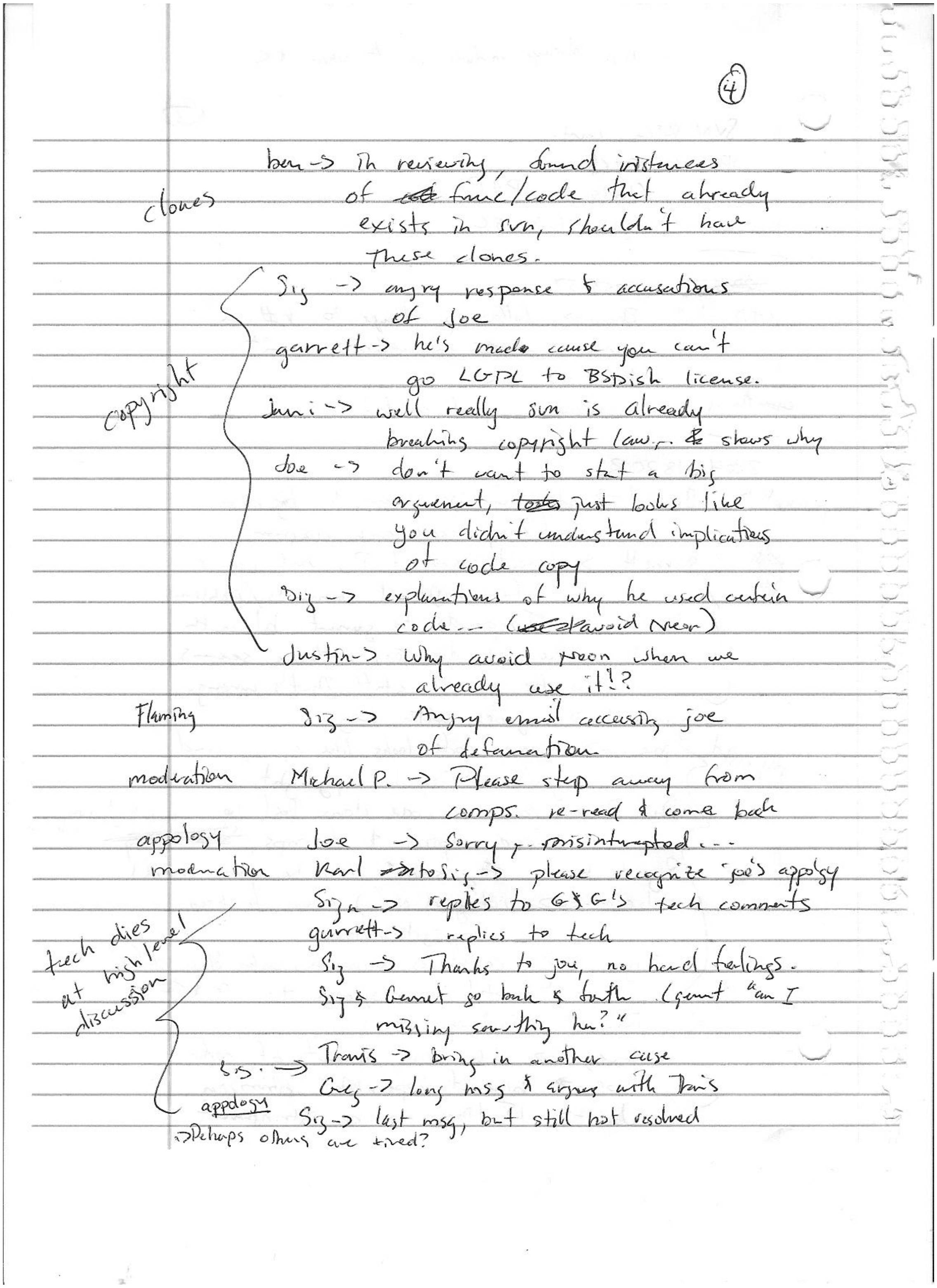
In this appendix, we provide an example of the coding and memoing techniques that we used for the manual analysis of 500 instance of peer reviews. For details on the procedure use, please see Section 5.2.

E.1 Examples of Coding of Peer Reviews

Figures E.1, E.2, and E.3 show the summaries or "field notes" or "explication de texte" of reviews on OSS projects. The coding of each review can be seen in the margin.

E.2 Examples of Memoing Peer Reviews

Figures E.4, E.5, and E.6 represent the sorted "pile" of evidence relating to the theme of ignored reviews. For more details on this theme please see Section 5.4.



4

clones — ben -> in reviewing, found instances of ~~cod~~ func/code that already exists in svn, shouldn't have these clones.

copyright —
Sig -> angry response & accusations of Joe
garrett -> he's mad cause you can't go LGPL to BSDish license.
Juni -> well really svn is already breaking copyright law... & shows why
Joe -> don't want to start a big argument, ~~tools~~ just looks like you didn't understand implications of code copy
Sig -> explanations of why he used certain code... (~~use~~/avoid neon)
Justin -> Why avoid neon when we already use it!?

Flaming — Sig -> Angry email accusing joe of defamation

moderation — Michael P. -> Please step away from comps. re-read & come back

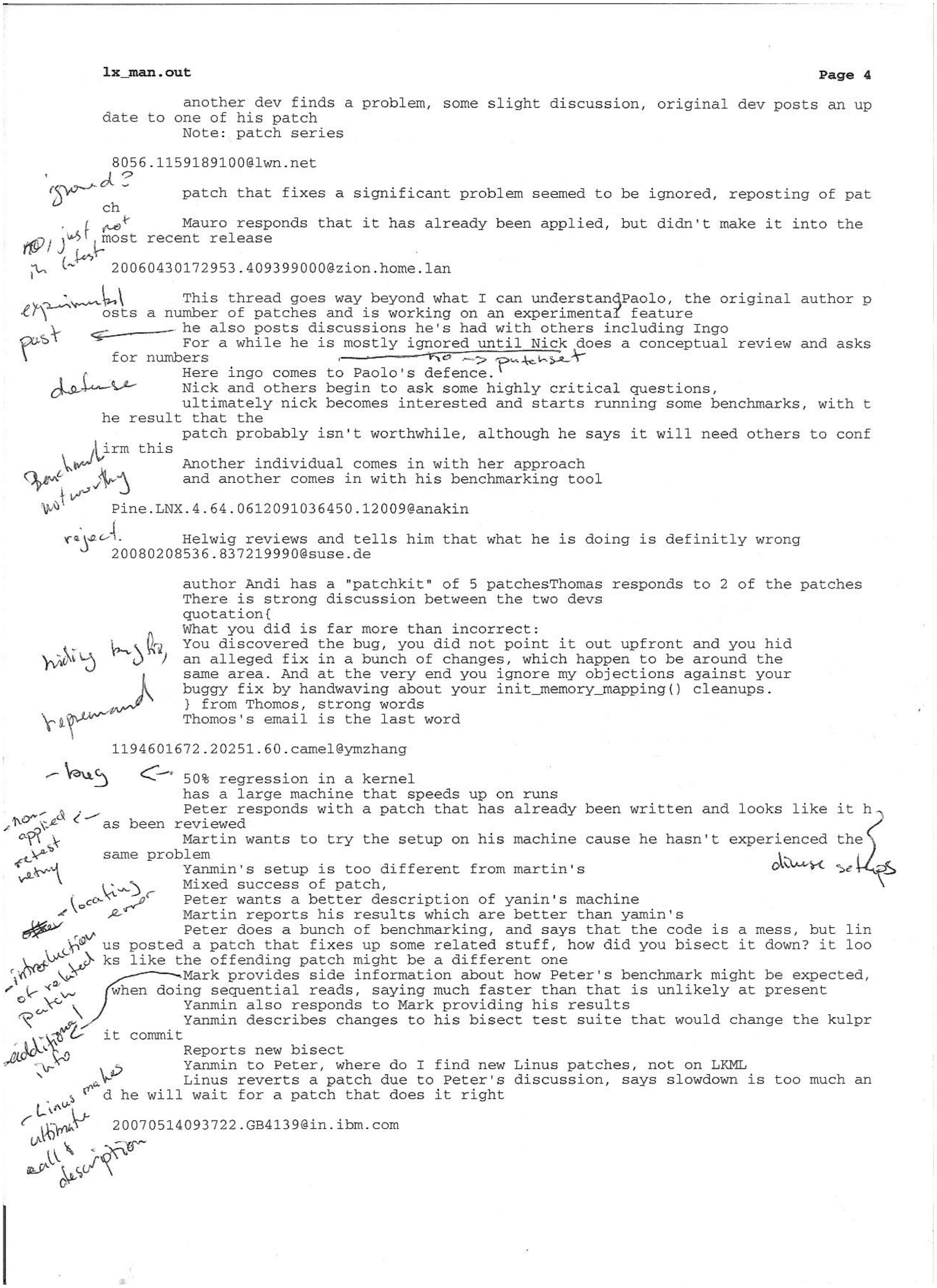
appology — Joe -> Sorry, misinterupted...

modenation — Karl ~~->~~ to Sig -> please recognize joe's appolgy

tech dies at high level discussion —
Sign -> replies to G & G's tech comments
garrett -> replies to tech
Sig -> Thanks to joe, no hard feelings.
Sig & Garret go back & forth (garret "am I missing something here?"
Sig. -> Travis -> bring in another case
Greg -> long msg & agrees with Travis

appology
Sig -> last msg, but still not resolved
-> Perhaps others are tired?

Figure E.1: Summaries of SVN reviews with codes



```
lx_man.out                                                          Page 4
         another dev finds a problem, some slight discussion, original dev posts an up
date to one of his patch
         Note: patch series

 8056.1159189100@lwn.net
```

ignored?

```
         patch that fixes a significant problem seemed to be ignored, reposting of pat
ch
         Mauro responds that it has already been applied, but didn't make it into the
most recent release
```

~~no~~, just not in latest

```
 20060430172953.409399000@zion.home.lan
```

experimental

```
         This thread goes way beyond what I can understandPaolo, the original author p
osts a number of patches and is working on an experimental feature
         he also posts discussions he's had with others including Ingo
         For a while he is mostly ignored until Nick does a conceptual review and asks
 for numbers
         Here ingo comes to Paolo's defence.
         Nick and others begin to ask some highly critical questions,
         ultimately nick becomes interested and starts running some benchmarks, with t
he result that the
         patch probably isn't worthwhile, although he says it will need others to conf
irm this
         Another individual comes in with her approach
         and another comes in with his benchmarking tool
```

past

no -> patchset

defense

Benchmark not worthy

```
 Pine.LNX.4.64.0612091036450.12009@anakin

         Helwig reviews and tells him that what he is doing is definitly wrong
 20080208536.837219990@suse.de
```

reject.

```
         author Andi has a "patchkit" of 5 patchesThomas responds to 2 of the patches
         There is strong discussion between the two devs
         quotation{
         What you did is far more than incorrect:
         You discovered the bug, you did not point it out upfront and you hid
         an alleged fix in a bunch of changes, which happen to be around the
         same area. And at the very end you ignore my objections against your
         buggy fix by handwaving about your init_memory_mapping() cleanups.
         } from Thomos, strong words
         Thomos's email is the last word

 1194601672.20251.60.camel@ymzhang
```

hiding bug fix,

reprimand

```
         50% regression in a kernel
         has a large machine that speeds up on runs
         Peter responds with a patch that has already been written and looks like it h
as been reviewed
         Martin wants to try the setup on his machine cause he hasn't experienced the
same problem
         Yanmin's setup is too different from martin's
         Mixed success of patch,
         Peter wants a better description of yanin's machine
         Martin reports his results which are better than yamin's
         Peter does a bunch of benchmarking, and says that the code is a mess, but lin
us posted a patch that fixes up some related stuff, how did you bisect it down? it loo
ks like the offending patch might be a different one
         Mark provides side information about how Peter's benchmark might be expected,
 when doing sequential reads, saying much faster than that is unlikely at present
         Yanmin also responds to Mark providing his results
         Yanmin describes changes to his bisect test suite that would change the kulpr
it commit
         Reports new bisect
         Yanmin to Peter, where do I find new Linus patches, not on LKML
         Linus reverts a patch due to Peter's discussion, says slowdown is too much an
d he will wait for a patch that does it right

 20070514093722.GB4139@in.ibm.com
```

- bug

- non-applied retest retry

diverse setups

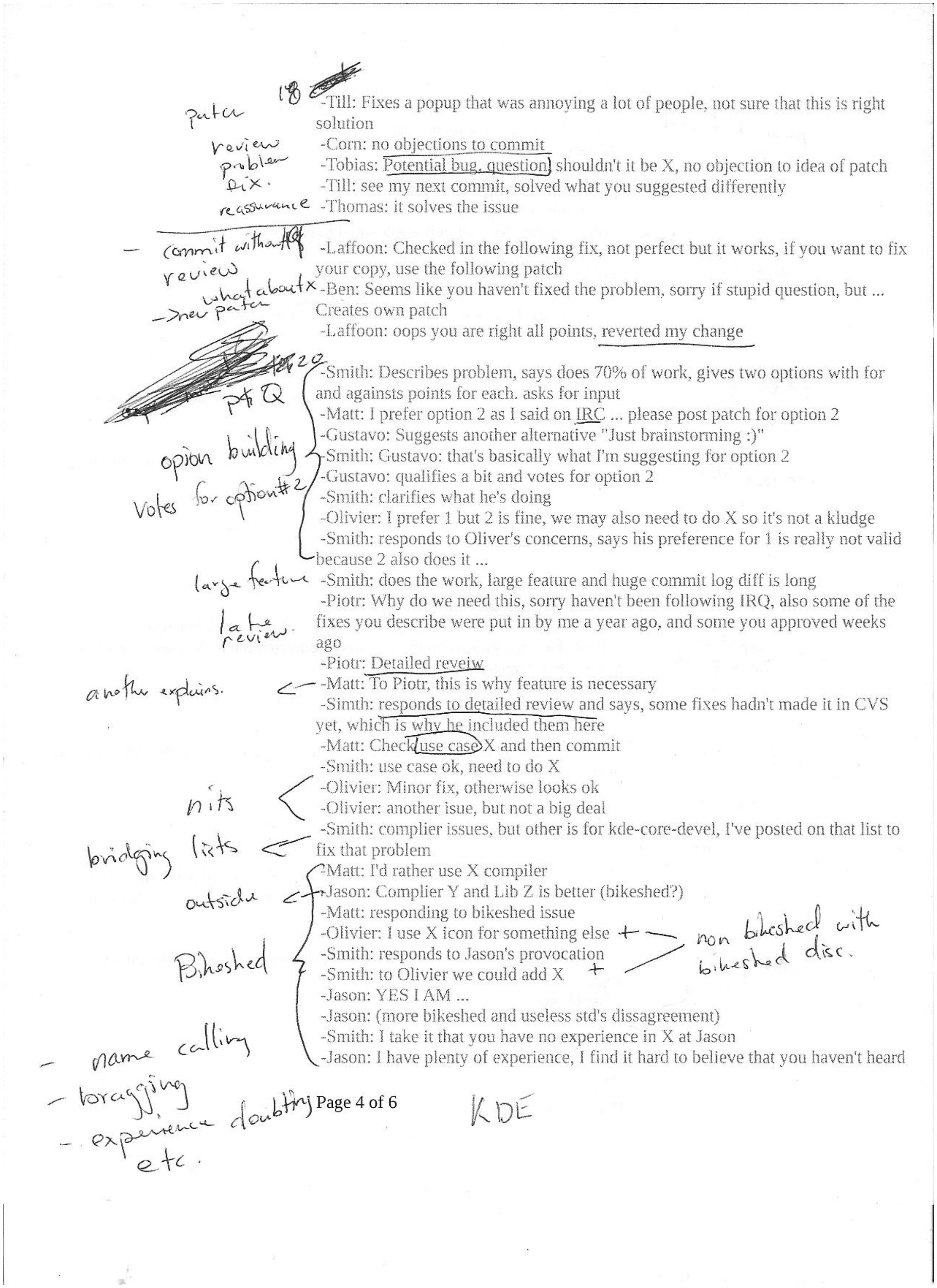
~~other~~ - locating error

- introduction of related patch

additional info

- Linus makes ultimate call & description

Figure E.2: Summaries of Linux reviews with codes



18

patch -Till: Fixes a popup that was annoying a lot of people, not sure that this is right solution

review -Corn: no objections to commit

problem -Tobias: Potential bug, question shouldn't it be X, no objection to idea of patch

fix. -Till: see my next commit, solved what you suggested differently

reassurance -Thomas: it solves the issue

– commit without review -Laffoon: Checked in the following fix, not perfect but it works, if you want to fix your copy, use the following patch

what about x -Ben: Seems like you haven't fixed the problem, sorry if stupid question, but ... Creates own patch

→ new patch

-Laffoon: oops you are right all points, reverted my change

20

pt Q -Smith: Describes problem, says does 70% of work, gives two options with for and againsts points for each. asks for input

-Matt: I prefer option 2 as I said on IRC ... please post patch for option 2

-Gustavo: Suggests another alternative "Just brainstorming :)"

opion building -Smith: Gustavo: that's basically what I'm suggesting for option 2

Votes for option#2 -Gustavo: qualifies a bit and votes for option 2

-Smith: clarifies what he's doing

-Olivier: I prefer 1 but 2 is fine, we may also need to do X so it's not a kludge

-Smith: responds to Oliver's concerns, says his preference for 1 is really not valid because 2 also does it ...

large feature -Smith: does the work, large feature and huge commit log diff is long

late review. -Piotr: Why do we need this, sorry haven't been following IRQ, also some of the fixes you describe were put in by me a year ago, and some you approved weeks ago

-Piotr: Detailed reveiw

another explains. ← -Matt: To Piotr, this is why feature is necessary

-Simth: responds to detailed review and says, some fixes hadn't made it in CVS yet, which is why he included them here

-Matt: Check use case X and then commit

-Smith: use case ok, need to do X

nits -Olivier: Minor fix, otherwise looks ok

-Olivier: another isue, but not a big deal

bridging lists ← -Smith: complier issues, but other is for kde-core-devel, I've posted on that list to fix that problem

-Matt: I'd rather use X compiler

outside ← -Jason: Complier Y and Lib Z is better (bikeshed?)

-Matt: responding to bikeshed issue

-Olivier: I use X icon for something else + non bikeshed with bikeshed disc.

Bikeshed -Smith: responds to Jason's provocation

-Smith: to Olivier we could add X +

-Jason: YES I AM ...

-Jason: (more bikeshed and useless std's dissagreement)

-Smith: I take it that you have no experience in X at Jason

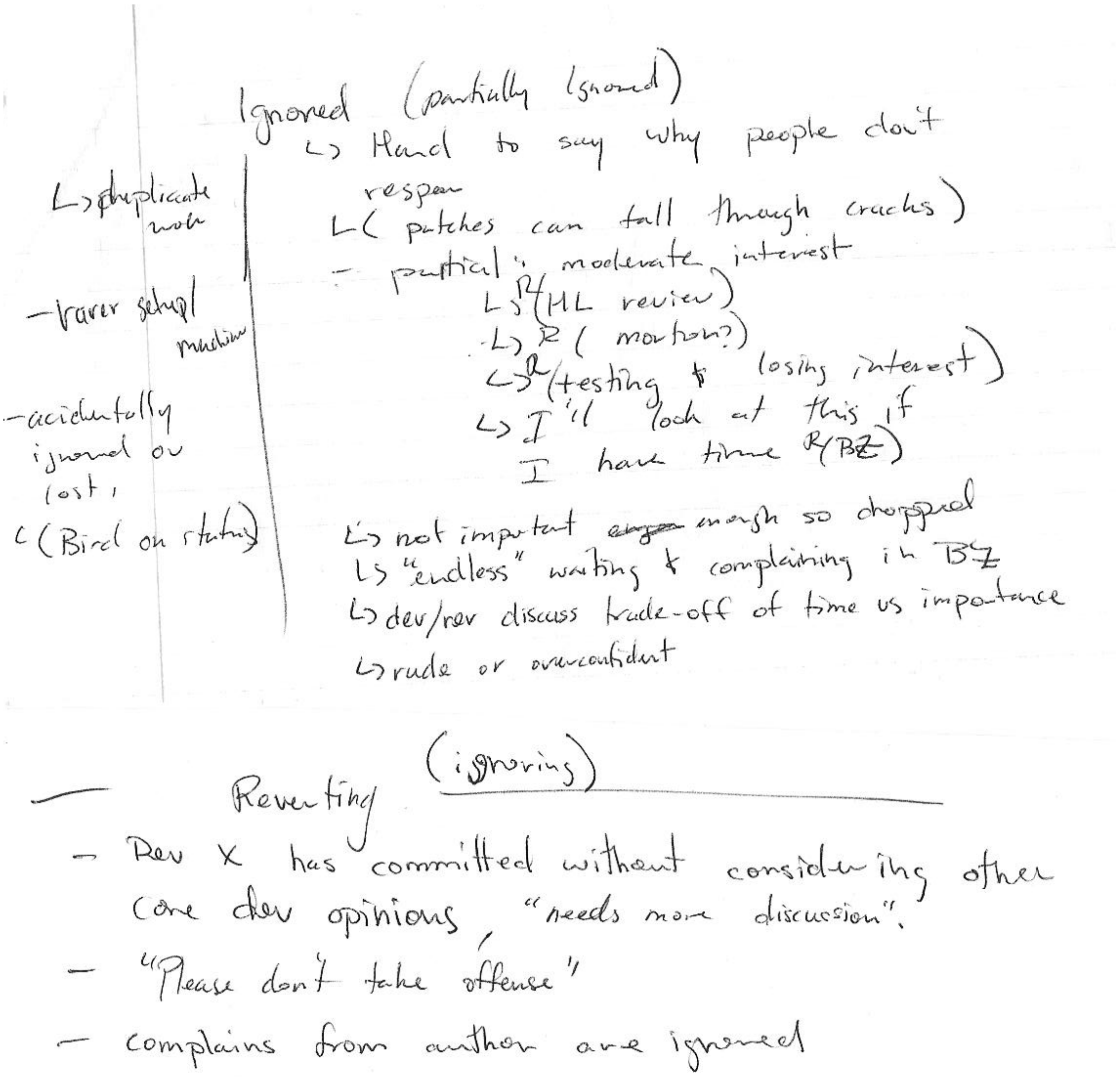
-Jason: I have plenty of experience, I find it hard to believe that you haven't heard

- name calling
- bragging
- experience doubting
etc.

Page 4 of 6

KDE

Figure E.3: Summaries of KDE reviews with codes



Ignored (partially Ignored)

↳ Hard to say why people don't respon

↳ (patches can fall through cracks)

– partial: moderate interest

↳ R (HL review)

↳ R (morton?)

↳ R (testing & losing interest)

↳ I'll look at this if I have time R (BZ)

↳ not important enough so dropped

↳ "endless" waiting & complaining in BZ

↳ dev/rev discuss trade-off of time vs importance

↳ rude or overconfident

↳ duplicate work

– rarer setup/machine

– accidentally ignored or lost,

↳ (Bird on status)

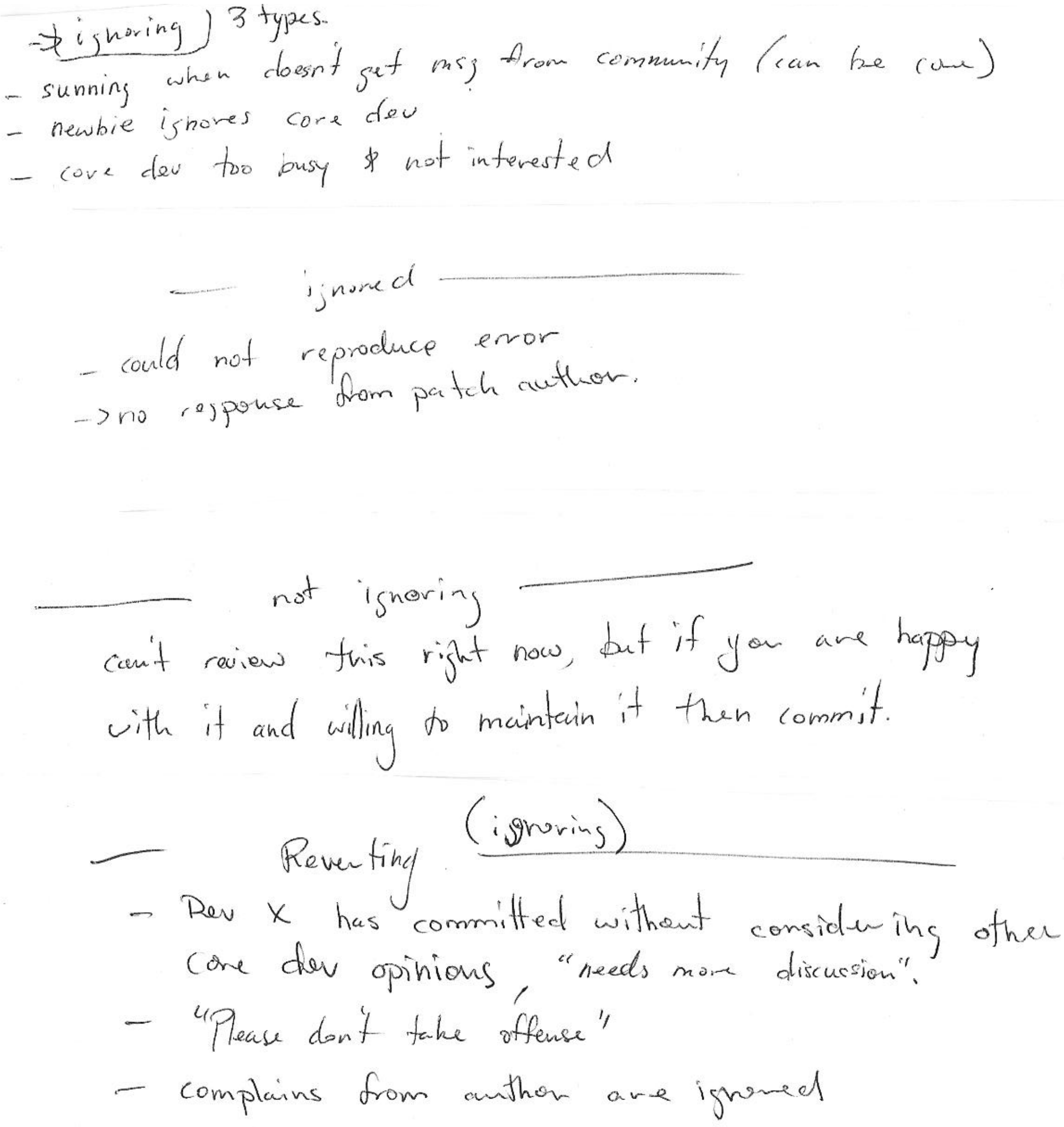
— Reverting (ignoring)

– Rev X has committed without considering other core dev opinions, "needs more discussion".

– "Please don't take offense"

– complains from author are ignored

Figure E.4: Memos for "ignored reviews" theme. See Section 5.4



→ ignoring) 3 types.
- sunning when doesn't get msg from community (can be core)
- newbie ignores core dev
- core dev too busy & not interested

ignored

- could not reproduce error
- → no response from patch author.

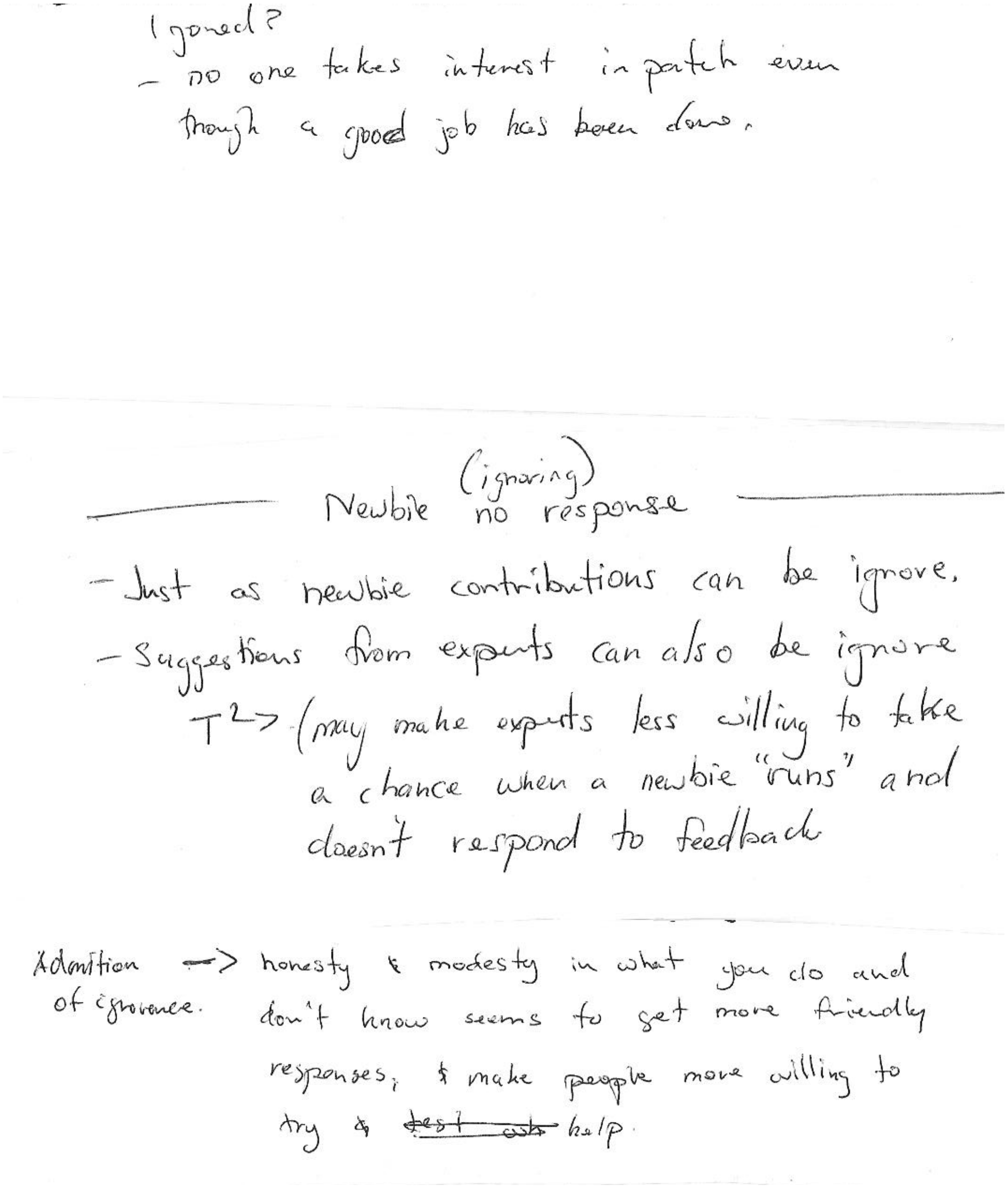
not ignoring

can't review this right now, but if you are happy with it and willing to maintain it then commit.

Reverting (ignoring)

- Rev X has committed without considering other core dev opinions, "needs more discussion".
- "Please don't take offense"
- complains from author are ignored

Figure E.5: Memos for "ignored reviews" theme. See Section 5.4



Igoned?
- no one takes interest in patch even though a good job has been done.

Newbie (ignoring) no response

- Just as newbie contributions can be ignore,
- Suggestions from experts can also be ignore

T2 -> (may make experts less willing to take a chance when a newbie "runs" and doesn't respond to feedback

Admition of ignorence. -> honesty & modesty in what you do and don't know seems to get more friendly responses, & make people more willing to try & ~~test out~~ help.

Figure E.6: Memos for "ignored reviews" theme. See Section 5.4

Bibliography

[1] A. Ackerman, L. Buchwald, and F. Lewski. Software inspections: an effective verification process. *IEEE Software*, 6(3):31–36, 1989.

[2] Apache. The apache server project. Apache Software Foundation. http://httpd.apache.org/, Accessed December 2010.

[3] J. Asundi and R. Jayant. Patch review processes in open source software development communities: A comparative case study. In *HICSS: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, page 10, 2007.

[4] V. Balijepally, R. Mahapatra, S. Nerur, and K. Price. Are two heads better than one for software development? The productivity paradox of pair programming. *Management Information Systems Quarterly*, 33(1):7, 2009.

[5] K. Bankes and F. Saucer. Ford systems inspection experiences. In *Proceedings of the International Information Technology Quality Conference*. Quality Assurance Institute, 1995.

[6] H. Barnard and R. Collicott. Compas: A development-process support system. *AT & T TECH. J.*, 69(2):52–64, 1990.

[7] J. Barnard and A. Price. Managing code inspection information. *IEEE Software*, pages 59–69, 1994.

[8] V. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. Zelkowitz. The empirical investigation of perspective-based reading. *Empirical Software Engineering*, 1(2):133–164, 1996.

[9] K. Beck. *Extreme Programming Explained*. Extreme Programming Explained. Addison-Wesley, 2000.

[10] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. The Agile Manifesto. 2001.

[11] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995.

[12] J. Berkus. The 5 types of open source projects, 2007. http://www.powerpostgresql.com/5_types.

[13] C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in oss projects. In *MSR: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 4. IEEE Computer Society, 2007.

[14] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *MSR: Proceedings of the Third International Workshop on Mining Software Repositories*, pages 137–143. ACM Press, 2006.

[15] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu. Open borders? immigration in open source projects. In *MSR: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 8. IEEE Computer Society, 2007.

[16] D. Bisant and J. Lyle. A two-person inspection method to improve programming productivity. *IEEE Transactions on Software Engineering*, 15(10):1294–1304, 1989.

[17] B. Boehm. Get ready for agile methods, with care. *ACM Computer*, 35(1):64 –69, 2002.

[18] B. Boehm and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Professional, 2003.

[19] B. W. Boehm. Software engineering economics. *Software Engineering, IEEE Transactions on*, SE-10(1):4 –21, 1984.

[20] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *In ICSE: Proceedings of the 21st International Conference on Software Engineering*, pages 555–563, 1999.

[21] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *CSCW: Computer Supported Cooperative Work*, pages 301–310. ACM Press, 2010.

[22] L. Brothers, V. Sembugamoorthy, and M. Muller. ICICLE: groupware for code inspection. In *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, pages 169–181. ACM Press, 1990.

[23] F. Buck. Indicators of quality inspections. *IBM Syst. Commun. Division, Tech. Rep. TR*, 21, 1981.

[24] Bugzilla. Bug-tracking system. http://www.bugzilla.org/about/, Accessed December 2010.

[25] A. Cockburn. *Crystal clear a human-powered methodology for small teams*. Addison-Wesley Professional, 2004.

[26] A. Cockburn and J. Highsmith. Agile software development, the people factor. *Computer*, 34(11):131 –133, 2001.

[27] CodeCollaborator. A web-based tool that simplifies and expedites peer code reviews. http://smartbear.com/codecollab.php, Accessed December 2010.

[28] J. Cohen. *Best Kept Secrets of Peer Code Review*. Smart Bear Inc., 2006.

[29] J. Corbin and A. Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative sociology*, 13(1):3–21, 1990.

[30] M. Crawley. *Statistics: An Introduction Using R*. John Wiley and Sons, 2005.

[31] J. Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage Publications, Inc., 2009.

[32] B. Dagenais and M. Robillard. Creating and evolving developer documentation: Understanding the decisions of open source contributors. In *FSE: Proceedings of the 18th ACM Symposium on the Foundations of Software Engineering*. ACM Press, 2010.

[33] T. DeMarco and T. Lister. *Peopleware: productive projects and teams*. Dorset House Publishing Company, Incorporated, 1999.

[34] T. Dinh-Trong and J. Bieman. The FreeBSD Project: A Replication Case Study of Open Source Development. *IEEE Transactions on Software Engineering*, 31(6):481–494, 2005.

[35] P. Eggert, M. Haertel, D. Hayes, R. Stallman, and L. Tower. *diff – compare files line by line*. Free Software Foundation, Inc., 2002.

[36] S. G. Eick, C. R. Loader, M. D. Long, L. G. Votta, and S. V. Wiel. Estimating software fault content before coding. In *Proceedings of the 14th International Conference on Software Engineering*, pages 59–65, 1992.

[37] M. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, 1986.

[38] M. Fagan. A history of software inspections. *Software pioneers: contributions to software engineering, Springer-Verlag, Inc.*, pages 562–573, 2002.

[39] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

[40] R. T. Fielding. Shared leadership in the apache project. *ACM Communications*, 42(4):42–43, 1999.

[41] R. T. Fielding and G. Kaiser. The Apache HTTP Server Project. *IEEE Internet Computing*, 1(4):88–90, 1997.

[42] D. Fisher, A. J. Brush, E. Gleave, and M. A. Smith. Revisiting Whittaker & Sidner's "email overload" ten years later. In *CSCW: Proceedings of Computer Supported Cooperative Work*, pages 309–312. ACM Press, 2006.

[43] K. Fogel. *Producing Open Source Software*. O'Reilly, 2005.

[44] Freshmeat. About freshmeat: A catalog of software projects. http://freshmeat.net/about, Accessed September 2009.

[45] Freshmeat. How do you measure a project's popularity? http://help.freshmeat.net/faqs/statistics/how-do-you-measure-a-projects-popularity, Accessed September 2009.

[46] D. German. The GNOME project: a case study of open source, global software development. *Software Process Improvement and Practice*, 8(4):201–215, 2003.

[47] D. German. A study of the contributors of PostgreSQL. In *MSR: Proceedings of the Third International Workshop on Mining Software Repositories*, pages 163–164. ACM Press, 2006.

[48] D. German. Using software distributions to understand the relationship among free and open source software projects. In *MSR: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 8. IEEE, 2007.

[49] D. German and A. Hindle. Measuring fine-grained change in software: towards modification-aware change metrics. In *Software Metrics, 2005. 11th IEEE International Symposium*, page 10, 2005.

[50] Gerrit. Web based code review and project management for git based projects. http://code.google.com/p/gerrit/, Accessed December 2010.

[51] Gitorious. Open source infrastructure for hosting open source projects that use git. http://gitorious.org/about, Accessed December 2010.

[52] B. Glaser. *Doing grounded theory: Issues and discussions*. Sociology Press Mill Valley, CA, 1998.

[53] GNOME. Gnome foundation report. http://foundation.gnome.org/reports/gnome-annual-report-2007.pdf, 2007.

[54] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[55] C. Gutwin, R. Penner, and K. Schneider. Group awareness in distributed software development. In *CSCW: Proceedings of the 2004 ACM conference on Computer Supported Cooperative Work*, pages 72–81, 2004.

[56] J. Hannay, T. Dybå, E. Arisholm, and D. Sjøberg. The effectiveness of pair programming: A meta-analysis. *Information and Software Technology*, 51(7):1110–1122, 2009.

[57] L. Harjumaa and I. Tervonen. A WWW-based tool for software inspection. In *HICSS: 31st Annual Hawaii International Conference on System Sciences*, pages 379–388. IEEE Computer Society, 1998.

[58] L. Hatton. Testing the value of checklists in code inspections. *IEEE Software*, 25(4):82–88, 2008.

[59] H. Heinzl and M. Mittlböck. Pseudo r-squared measures for poisson regression models with over- or underdispersion. *Computational Statistics & Data Analysis*, 44(1-2):253 – 271, 2003.

[60] J. D. Herbsleb. Global software engineering: The future of socio-technical coordination. In *FOSE: In proceedings of the 2007 Future of Software Engineering*, pages 188–198. IEEE Computer Society, 2007.

[61] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. Towards a theoretical model for software growth. In *MSR: Proceedings of the Fourth International Workshop on Mining Software Repositories*, pages 21–28. IEEE Computer Society, 2007.

[62] G. Hertel, S. Niedner, and S. Herrmann. Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel. *Research policy*, 32(7):1159–1177, 2003.

[63] A. Hindle, M. W. Godfrey, and R. C. Holt. Reading beside the lines: Indentation as a proxy for complexity metric. In *ICPC: Proceedings of the 16th IEEE International*

Conference on Program Comprehension, pages 133–142. IEEE Computer Society, 2008.

[64] R. Hoda, J. Noble, and S. Marshall. Organizing self-organizing teams. In *ICSE: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 285–294. ACM Press, 2010.

[65] E. Hossain, M. Babar, and J. Verner. How can agile practices minimize global software development co-ordination risks? *Software Process Improvement*, pages 81–92, 2009.

[66] J. Howison, M. Conklin, and K. Crowston. Flossmole: A collaborative repository for floss research data and analyses. *International Journal of Information Technology and Web Engineering*, 1:17–26, 2006.

[67] H. Hulkko and P. Abrahamsson. A multiple case study on the impact of pair programming on product quality. In *ICSE: Proceedings of the 27th international conference on Software engineering*, pages 495–504, 2005.

[68] J. Iniesta. A set of metrics to support technical reviews. In *In proceedings of the Second International Conference on Software Quality Management*, volume 1, pages 579–594, 1994.

[69] G. Jeong, S. Kim, T. Zimmermann, and K. Yi. Improving code review by predicting reviewers and acceptance of patches. (ROSAEC-2009-006), 2009.

[70] J. P. Johnson. Collaboration, peer review and open source software. *Information Economics and Policy*, 18(4):477 – 497, 2006.

[71] P. M. Johnson. Reengineering inspection. *ACM Communications*, 41(2):49–52, 1998.

[72] P. M. Johnson and D. Tjahjono. Does every inspection really need a meeting? *Empirical Software Engineering*, 3(1):9–35, 1998.

[73] P.-H. Kamp. A bike shed (any colour will do) on greener grass... FreeBSD mailing list archive http://www.webcitation.org/5ZZaDOxyW, 1999.

[74] P. Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets 1*, 28:1–9, 2008.

[75] D. Kelly and T. Shepard. Qualitative observations from software code inspection experiments. In *CASCON: Proceedings of the conference of the Centre for Advanced Studies on Collaborative Research*, page 5. IBM Press, 2002.

[76] J. C. Knight and E. A. Myers. An improved inspection technique. *ACM Communications*, 36(11):51–61, 1993.

[77] S. Kollanus and J. Koskinen. Survey of software inspection research. *Open Software Engineering Journal*, 3:15–34, 2009.

[78] B. L. Huang, Boehm. How much software quality investment is enough: A value-based approach. *IEEE Software*, 23(5):88–95, 2006.

[79] O. Laitenberger and J. DeBaud. An encompassing life cycle centric survey of software inspection. *Journal of Systems and Software*, 50(1):5–31, 2000.

[80] K. Lakhani and R. Wolf. Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects. In *Perspectives on Free and Open Source Software*, pages 1–27. MIT Press, 2005.

[81] C. Larman and V. Basili. Iterative and incremental developments: a brief history. *Computer*, 36(6):47 – 56, 2003.

[82] J. Lawrence G. Votta. Does every inspection need a meeting? *SIGSOFT Softw. Eng. Notes*, 18(5):107–114, 1993.

[83] G. Lee and R. Cole. From a Firm-Based to a Community-Based Model of Knowledge Creation: The Case of the Linux Kernel Development. *Organization Science*, 14(6):633–649, 2003.

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

[85] M. Leszak, D. E. Perry, and D. Stoll. Classification and evaluation of defects in a project retrospective. *Journal of Systems and Software*, 61(3):173 – 187, 2002.

[86] S. Lussier. New tricks: how open source changed the way my team works. *IEEE Software*, 21(1):68–72, 2004.

[87] F. Macdonald and J. Miller. A comparison of computer support systems for software inspection. *Automated Software Engineering*, 6(3):291–313, July 1999.

[88] W. E. Mackay. More than just a communication system: diversity in the use of electronic mail. In *Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, CSCW '88, pages 344–353, New York, NY, USA, 1988. ACM.

[89] J. Maindonald and J. Braun. *Data analysis and graphics using R: an example-based approach*. Cambridge University Press, 2003.

[90] J. Martin and W. T. Tsai. N-Fold inspection: a requirements analysis technique. *ACM Communications*, 33(2):225–232, 1990.

[91] V. Mashayekhi, C. Feulner, and J. Riedl. Cais: collaborative asynchronous inspection of software. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 21–34. ACM Press, 1994.

[92] P. McCullagh and J. Nelder. Generalized linear models (Monographs on statistics and applied probability 37). *London: Chapman Hall*, 1989.

[93] D. Mishra and A. Mishra. Efficient software review process for small and medium enterprises. *Software, IET*, 1(4):132–142, 2007.

[94] A. Mockus. Missing data in software engineering. In F. Shull, J. Singer, and D. I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 185–200. Springer, 2008.

[95] A. Mockus, R. Fielding, and J. Herbsleb. A case study of open source software development: The apache server. *ICSE: Proceedings of the 22nd international conference on Software Engineering*, pages 262–273, 2000.

[96] A. Mockus, R. T. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, 2002.

[97] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE: Proceedings of the 24th International Conference on Software Engineering*, pages 503–512. ACM Press, 2002.

[98] O. Morera and D. Budescu. A Psychometric Analysis of the Divide-and-Conquer Principle in Multicriteria Decision Making. *Organizational Behavior and Human Decision Processes*, 75(3):187–206, 1998.

[99] M. M. Muller and W. F. Tichy. Case study: extreme programming in a university environment. In *ICSE: Proceedings of the 23rd International Conference on Software Engineering*, pages 537–544. IEEE Computer Society, 2001.

[100] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye. Evolution patterns of open-source software systems and communities. In *IWPSE: Proceedings of the*

International Workshop on Principles of Software Evolution, pages 76–85. ACM Press, 2002.

[101] Netcraft. Nearly 2.5 million active sites running freebsd. http://news.netcraft.com/archives/2009/06/02/most_reliable_hosting_company_sites_in_may_2009.html, July 2004.

[102] Netcraft. Most reliable hosting company sites. http://news.netcraft.com/archives/2009/06/02/most_reliable_hosting_company_sites_in_may_2009.html, May 2009.

[103] J. Nosek. The case for collaborative programming. *Communications of the ACM*, 41(3):105–8, 1998.

[104] M. Nurolahzade, S. M. Nasehi, S. H. Khandkar, and S. Rawal. The role of patch review in software evolution: an analysis of the mozilla firefox. In *International Workshop on Principles of Software Evolution*, pages 9–18, 2009.

[105] D. O'Neill. Issues in software inspection. *IEEE Softw.*, 14(1):18–19, 1997.

[106] A. Onwuegbuzie and N. Leech. Validity and qualitative research: An oxymoron? *Quality and quantity*, 41(2):233–249, 2007.

[107] C. N. Parkinson. *Parkinson's Law: The Pursuit of Progress*. John Murray, 1958.

[108] D. L. Parnas and D. M. Weiss. Active design reviews: principles and practices. In *ICSE: Proceedings of the 8th international conference on Software engineering*, pages 132–136. IEEE Computer Society Press, 1985.

[109] D. Perry, A. Porter, M. Wade, L. Votta, and J. Perpich. Reducing inspection interval in large-scale software development. *Software Engineering, IEEE Transactions on*, 28(7):695–705, 2002.

[110] A. Porter, H. Siy, A. Mockus, and L. Votta. Understanding the sources of variation in software inspections. *ACM Transactions Software Engineering Methodology*, 7(1):41–79, 1998.

[111] A. Porter and L. Votta. An experiment to assess different defect detection methods for software requirements inspections. In *Proceedings of the 16th international conference on Software engineering*, pages 103–112. IEEE Computer Society Press, 1994.

[112] A. A. Porter, H. P. Siy, and J. Lawrence G. Votta. Understanding the effects of developer activities on inspection interval. In *ICSE: Proceedings of the 19th International Conference on Software Engineering*, pages 128–138, 1997.

[113] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011. ISBN 3-900051-07-0.

[114] E. S. Raymond. *The Cathedral and the Bazaar*. O'Reilly and Associates, 1999.

[115] P. Resnick et al. RFC 2822: Internet message format, 2001.

[116] ReviewBoard. A powerful web-based code review tool that offers developers an easy way to handle code reviews. http://www.reviewboard.org/, Accessed December 2010.

[117] P. C. Rigby, E. T. Barr, C. Bird, D. M. German, and P. Devanbu. Collaboration and Governance with Distributed Version Control. *ACM Transactions on Software Engineering and Methodology, Submission number TOSEM-2009-0087*, page 33, 2009.

[118] P. C. Rigby and D. M. German. A preliminary examination of code review processes in open source projects. Technical Report DCS-305-IR, University of Victoria, January 2006.

[119] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: A case study of the apache server. In *ICSE: Proceedings of the 30th international conference on Software Engineering*, pages 541–550, 2008.

[120] P. C. Rigby and M.-A. Storey. Understanding broadcast based peer review on open source software projects. In *ICSE: To Appear in the Proceedings of the 33rd International Conference on Software Engineering*. ACM Press, 2011.

[121] Rob Hartill. Email, January 1998. http://mail-archives.apache.org/mod_mbox/httpd-dev/199801.mbox/%3CPine.NEB.3.96.980108232055.19805A-100000@localhost%3E.

[122] C. Sauer, D. R. Jeffery, L. Land, and P. Yetton. The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research. *IEEE Transactions Software Engineering*, 26(1):1–14, 2000.

[123] G. M. Schneider, J. Martin, and W. T. Tsai. An experimental study of fault detection in user requirements documents. *ACM Transactions Software Engineering Methodology*, 1(2):188–204, 1992.

[124] C. Schwaber, M. Gilpin, and J. Stone. Software change and configuration management. Forrester Wave, 2007.

[125] E. Shihab, Z. Jiang, and A. Hassan. On the use of internet relay chat (irc) meetings by developers of the gnome gtk+ project. In *MSR: In the Proceedings of the 6th IEEE*

International Working Conference on Mining Software Repositories, pages 107–110. IEEE Computer Society, 2009.

[126] D. Shipley and W. Schwalbe. *Send: The essential guide to email for office and home*. Alfred A. Knopf, 2007.

[127] H. P. Siy and L. G. Votta. Does the modern code inspection have value? In *Proceedings of the International Conference on Software Maintenance*, pages 281–289. IEEE Computer Society, 2001.

[128] SourceForge. SourceForge: Find and Develop Open Source Software. http://sourceforge.net/about, Accessed December 2010.

[129] D. Spinellis. A tale of four kernels. In *ICSE: Proceedings of the 30th International Conference on Software Engineering*, pages 381–390. IEEE Computer Society, 2008.

[130] J. Stark. Peer reviews as a quality management technique in open-source software development projects. Springer, 2002.

[131] J. Tomayko. A comparison of pair programming to inspections for software defect reduction. *Computer Science Education*, 12(3):213–222, 2002.

[132] L. Torvalds. “Re: IRQF_DISABLED problem [maintaining status quo unless change is obviously better]”. Linux Kernel Mailing List http://kerneltrap.org/mailarchive/linux-kernel/2007/7/26/122293, 2007.

[133] G. von Krogh, S. Spaeth, and K. Lakhani. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217–1241, 2003.

[134] G. Weinberg. *The psychology of computer programming*. Van Nostrand Reinhold, 1971.

[135] P. Weissgerber, D. Neu, and S. Diehl. Small patches get in! In *MSR: In the Proceedings of the 5th IEEE International Working Conference on Mining Software Repositories*, pages 67–76. ACM, 2008.

[136] E. Weller. Lessons from three years of inspection data. *IEEE Software*, 10(5):38–45, 1993.

[137] S. Whittaker and C. Sidner. Email overload: exploring personal information management of email. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 276–283, 1996.

[138] K. E. Wiegers. *Peer Reviews in Software: A Practical Guide*. Addison-Wesley Information Technology Series. Addison-Wesley, 2001.

[139] L. Williams and R. Kessler. *Pair programming illuminated*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[140] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair programming. *IEEE Software*, 17(4):19–25, 2000.

[141] Y. Yamauchi, M. Yokozawa, T. Shinohara, and T. Ishida. Collaboration with lean media: how open-source software succeeds. In *CSCW: Proceedings of Computer Supported Cooperative Work*, pages 329–338. ACM Press, 2000.

[142] R. K. Yin. *Case Study Research: Design and Methods*, volume 5 of *Applied Social Research Methods Series*. Sage Publications Inc., 2 edition, 1994.

[143] R. K. Yin. *Applications of Case Study Research*, volume 34 of *Applied Social Research Methods Series*. Sage Publications Inc., 2 edition, 2003.

[144] E. Yourdon. *Structured Walkthroughs*. Yourdon Press Computing Series. Prentice Hall, 1989.