A preliminary examination of code review processes in open source projects

Peter C. Rigby and Daniel M. German
Software Engineering Group, Dept. of Computer Science
University of Victoria
{pcr, dmg}@cs.uvic.ca

1. INTRODUCTION

This paper represents a first attempt to understand the code review processes used by open source projects. Although there have been many studies of open source projects [1, 2, 4], these studies have focused on the entire development process and community of the project or projects. We are not aware of any paper that has examined the review process of open source projects in depth or compared the review processes used among projects. We examined the stated and observed code review processes used by 11 open source projects; the four most interesting projects are discussed. Additionally, we examined the mature, well-known Apache server project in depth. We extracted the developer and commit mailing lists into a database in order to reconstruct and understand the review and patch processes of the Apache project. The paper is broken into six sections. The remainder of this section introduces our research questions. The second section introduces our research methodology and data extraction techniques. The third section discusses the actors in the process, introduces a general patch process, and contrasts the formal and observed processes used by GCC, Linux, Mozilla, and Apache. The fourth section discusses some observed review patterns. The fifth section quantitatively answers our research questions using the Apache data (although we discuss our findings, due to time and other considerations, we leave the development of hypotheses to future work). In the final section we present our conclusions and future work.

1.1 Research Questions

What is the patch process and review process used by the projects?
The review process is part of the patch process, as such one cannot describe the review process without understanding the patch process and project structure. We provide a qualitative description of these processes and the project structure in the next section.

What types of review does a the project use?
We expect that projects do not use only the traditional, formal style of review, but also other lightweight reviews.

Why are patches rejected? What percentage of patches are rejected?
We seek to understand what makes a good and bad patch.

Who performs the review? Are the top developers also the top reviewers?
We want to understand if the best developers are also the best reviewers. Although code review ownership is not addressed in this paper, in future we would like to determine whether reviewers always review code pertaining to a certain set of files or a given area of expertise.

When are reviews performed? What is the frequency of review?
In a proprietary setting reviews are performed at predetermined times in the development cycle. Does a similar phenomenon occur in OSS review?

How long do reviews take to perform? How does the patch size affect the review?
Traditional review usually takes no more than two hours and involves a complete solution to a problem. Is this the case in OSS reviews?

How does merit-based trust among actors affect the review? Are more trusted individuals reviewed less often? How much feedback is provided in the review?
Open source projects are based on trust, only developers who are trusted have commit privileges. However, trust may result in fewer reviews of core developers producing more defects in the product.

What kinds of non-source code patches are reviewed? How does the kind of patch affect the review?

What affect does reviewing have on other elements of the patch process? What is the relationship between reviewing and testing?
Both testing and review are defect detection techniques, as such they compete for developer attention. Additionally, some projects are more difficult to create automated tests than others, which will effect the relative amount of review and testing?

2. METHODOLOGY AND DATA SOURCES

In order to create an accurate picture of the code review process, we first examined the qualitative patch statements. These statements were created by the project to guide developers both external and internal to the project. They helped us focus our later investigation of qualitative review patterns and quantitative analyses. Our qualitative research allowed us to identify recurring patterns in the data that might be difficult to detect quantitatively and helped to assert our quantitative conclusion by providing examples. Both the project statements and the qualitative investigation of development

artifacts guided the development of our research questions.

In our quantitative section we examine the Apache project's developer and version control mailing lists based on our research questions. The developer's mailing list contains general discussion related to the development of the server. We were most interested in patches that were reviewed and discussed on the mailing list. Since this mailing list was large (over 100,000 emails), it was difficult to resolve the names on the mailing list as many individuals had multiple email names (e.g., Peter Rigby vs. Peter C. Rigby vs. pcr) and many email addresses. There were also some inconsistencies in the data, such as emails that belonged to a discussion, a thread, being separated from that thread. These inconsistencies were removed and now involve less than 10% of the emails. The commit (version control) mailing list receives an email every time a commit is made to the version control system. The commit contains the difference between the old and new file (i.e., the lines that have changed, and some surrounding context); the patch submitter's name; the reviewer's name; the time the commit was made; a log describing what was changed; a list of modified, added, and removed files; and a subject that begins "cvs [or svn] commit:" followed by the list of files. There were often more than one submitter and reviewer. The commit mailing list was much easier to work with because user names are unique and many of the entries are generated by a computer, making the data far more consistent than the developer's mailing list.

The data was extracted into a mysql database using Perl scripts. Once in the database we fixed inconsistencies with the data using (sometimes iteratively) SQL statements. A database is significantly easier to work with, German [3], than creating a data file from Perl scripts for each research question as was done by Mockus et al. [4]. Querying the database made analysis significantly quicker and more accurate. We hope to use this data to validate our own results against Mockus' results.

3. PATCH STATEMENTS

A patch is a change, fix, or enhancement to the product, including source code and documentation. Patches are an integral part of the OSS development process. Most projects have a statement that describes their requirements for a patch and how the patch is processed. In contrast, few projects have a statement describing how they conduct code reviews, but most mention code review in their patch statement. We describe the elements of the patch statement and project structure that are related to code review.

3.1 Roles

The actors in the patch process include the following: a bug reporter, a contributor, a tester, a reviewer, and a committer. An individual may play one or more roles on any given patch and the roles may be played by any number of individuals. Since this paper is an investigation of code review processes, we mainly focus on code review, but we also discuss aspects of bug reporting and testing when they pertain to code review.

3.2 General Patch Process

No project used exactly the same patch process as another project. Indeed, many newer and smaller projects did not have a formal statement for the processing of patches. Mature projects had statements that appear to have evolved over the lifetime of the project. However, most projects' processes contained the elements described in the general patch process below.

1. someone reports a bug (defect) or requests a new feature
2. it becomes a priority (e.g., a lot of people find/want it or it is critical)
3. there is a discussion about the bug
4. someone posts a patch (sometimes just a work around in the case of a bug) there is often some uncertainty about the patch.
5. people try the patch, (lazy testing) (if it fails, no need to review a broken patch), but ultimately a successful patch is produced
6. the patch is reviewed and formally tested (sometimes minimally)
7. once the patch is deemed acceptable, it is committed and released in the next version

3.3 Commonalities in Review Processes

To avoid repetition, we discuss elements of review that were common among the four projects. First, every project had a coding standard. Second, projects required contributors to update documentation, especially for new functionality. Third, projects emphasized that patches must be independent (e.g., don't send major functionality changes with simple formatting changes), complete, and small. All projects agreed that smaller patches were generally easier and quicker to review than larger patches. Fourth, patches were often ignored and lost; it is the responsibility of the contributor to resend the patch at defined intervals. All patches that are sent from external contributors (i.e., developers without commit privileges) are reviewed.

3.4 GCC - Write Policies

The GNU Compiler Collection's (GCC) process for submitting a patch for review follows the general pattern. GCC requires more extensive testing [1] than most projects and provides a test suite that can simulate other hardware environments. GCC requires that a detailed commit log, linked to all problem reports, be included with the patch. If the patch affects many different aspects of the compiler, it will have to be sent to all affected mailing lists (all patches are sent to the gcc-patches mailing list). GCC has a long patch resend time period of two weeks.

GCC has developed a strict set of policies for granting commit privileges. These policies are not enforced by subversion, GCC's version control tool. Rather, their implementation is dependent on the integrity and understanding of the 148 [2] developers who have commit privileges. The project relies on its steering committee to discipline uncooperative members.

GCC's policies create four categories of write privileges. 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

[1] `http://gcc.gnu.org/contribute.html\#testing` December 2005

[2] See GCC'S MAINTAINERS file version 1.439

of code their contribution modifies. The final **free for all** approval applies to obvious changes to artifacts like documentation or the website [3]. Consistent with Mockus et al.'s [4] results, the core group (developers who can commit code to any part of the project) consists of 11 developers. The core group is supported by a much larger group of 137 developers who are responsible for ports to particular hardware environments or regularly provide patches.

GCC's write policies imply that code must at least be reviewed by the maintainer of the code. The maintainer of a given section of code is not required to be reviewed by anyone, nor are developers with global write privileges.

3.5 Linux - Pyramid of Trust

The Linux Operating System Kernel's review process is informal [4]. 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 Linus 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 Linus 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 Linux source 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. Linus 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 Alan 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.

Interestingly, Linus does not provide feedback on patches. Instead, it is the contributor or a reviewer of the patch who is responsible for repeatedly sending the patch to Linus until the patch appears in Linus's released patch list. Other maintainers will provide feedback on patches. For example, Alan Cox will always respond, even if the response is terse. This lack of responsiveness coupled with "lazy testing" can lead to long patch process times.

3.6 Mozilla - Two Reviewers

The Mozilla project is a suite of applications that were originally developed by Netscape Inc. under a proprietary license. In January 1998 [5] 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. Compared to the other projects we examined, Mozilla had the highest level of code review. "Code review is our basic mechanism for validating the design and implementation of patches." [5]

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.

Mozilla performs daily builds for Linux, Windows, and Mac OSX at 8am PST. Developers who have committed code in a daily build are required to be available to fix any build problems associated with their code. The committer is responsible for any third party (external) code, but the reviewer is not. The project's CVS tree, Mozilla's uses the CVS version control tool, is closed until all build problems are fixed. Although Mozilla has very few automated tests, there are a number of manual tests including "smoke tests". Mozilla requires that the build pass the "smoke tests", which are run by quality assurance volunteers, before it will open the CVS tree.

There are 116 developers working on 96 [6] Mozilla modules. On average, developers work on 2.7 different modules; the mean number of reviewers (owners and peers) per module is 3.2. Currently, 26 [7] developers are authorized to perform "super" reviews.

Mozilla reviewers will respond within 24 hours with a schedule for the review. This project is strongly against checking in uncertain code, because "Checking code in seems to cause a psychological effect that the patch is done, it's time to go on to the next thing, and requested changes are new and extra [and undesirable] work." [8] One would expect this statement to lead to long patch times.

All four projects use the Bugzilla bug tracking tool which is developed by Mozilla. Mozilla's code review process is more highly integrated with bugzilla than the other projects. A flag in bugzilla is set whenever a patch needs to be reviewed, and a request is automatically sent to the reviewer's mailing list. In the past, most open source projects used a mailing list to communicate about bugs, reviews, and all other software artifacts. This developer's mailing list made it difficult to track individual problems. Bugzilla has successfully integrated bug reporting, testing, and code review in a single

[3] `http://gcc.gnu.org/svnwrite.html\#policies` December 2005

[4] For more information see `http://www.kernel.org/pub/linux/docs/lkml/index.html` December 2005

[5] `http://www.mozilla.org/hacking/code-review-faq.html` December 2005

[6] `http://www.mozilla.org/owners.html` December 2005

[7] `http://www.mozilla.org/hacking/reviewers.html` December 2005

[8] `http://www.mozilla.org/hacking/code-review-faq.html` December 2005

place, leaving the mailing list for open discussions. The support for code review is not explicit and leaves much room for improvement.

3.7 Apache - Post-commit Review

The Apache httpd server project, the focus of our quantitative section, has a 69% of the server market share [9]. The Apache code review system depends largely on its voting system. This sophisticated system has evolved from the part-time, asynchronous nature of the Apache community. Most patches are voted upon using a **commit-then-review** policy. A developer with commit privileges is allowed to commit a patch to the repository without review. If problems arise from the patch or it is in some way not agreeable to a voting member, the patch can be can be vetoed and must be reverted to a previous version of the code. For significant changes, such as a new feature or a new idea, the change must be **reviewed-then-committed**. Two days' notice must be given to allow core members to review and vote on the issue. Since this kind of change requires a consensus, at least three positive (+1) votes and no negative (-1) veto votes must be attained for a change to be accepted. There are other types of voting systems employed by Apache, but they are mainly used by the Apache foundation and during the release process [10].

The Apache website [11] recognizes 54 current core developers. Of the projects under examination, Apache has the largest number of committers who can modify any section of the code. This reduces the amount of review required, but also increases the risk of defective code being added to the project. Apache's commit policies create two barriers to 'buggy' code. The first barrier is to require significant changes be to reviewed and agreed on by consensus. The second barrier is to allow vetoes of committed code. For the latter to be effective, developers must examine the commit mailing list to ensure that unacceptable code is not committed.

Apache no longer requires developers to post every patch to the mailing list because many were unintentionally ignored. Patches are now posted to the bugzilla bug repository. If a patch requires review and discussion, the patch is also posted to the developers' mailing list.

Apache committers are responsible for what they commit. They generally review, test, and release it on their own system before committing. Apache has an entire subproject devoted to developing its test suite [12].

3.8 Automated Testing vs. Review

What effect does reviewing have on other elements of the patch process? What is the relationship between reviewing and testing?

We leave quantitative answers to this question for future work. We noticed that projects that have many automated tests performed less review. For example, both GCC and Apache have extensive test suites and committers who do not need to be formally reviewed by anyone. Linux represents an extreme in which only one person has commit privileges. However, the closer (higher) you are to the top in the pyramid of trust the fewer reviews you must pass through. Linux relies on informal tests. Mozilla reviews all patches by two different individuals. They do not have many automated tests and rely mostly on manual tests. With Linux (operating system) and Mozilla (GUI), it is difficult to automate tests.

[9] http://news.netcraft.com/archives/2005/04/01/april_2005_web_server_survey.html April 2005

[10] http://www.apache.org/foundation/voting.html December 2005

[11] http://httpd.apache.org/contributors/ December 2005

[12] http://httpd.apache.org/test/

4. ANECDOTAL PATTERNS

After examining the formal review and patch statements, we qualitatively examined the mailing lists of Mozilla and Apache. The main difficulty with developing review patterns before first quantitatively evaluating the projects is that it is difficult to find interesting patches through random selection. Patterns become more interesting when backed by quantitative findings. For example, after quantitative analysis, one could examine all the patches with the top number of responses (the most heavily discussed patches). We present some interesting patterns [13]

4.1 Accepted then Rejected

In the Apache review process, a patch is posted to the mailing list for review. Often the first reviewer would approve the patch, but a second or even third reviewer would veto it and the patch would be dropped or reworked.

4.2 External Contributor Rewrite

In all projects, code that comes from external sources (i.e., contributors who do not have commit privileges) is reviewed. We noticed a recurring pattern in which an external patch would be below the acceptable quality required by the project. Instead of providing the contributor with review comments, as was often done, the committer would simply rewrite the patch providing the contributor with a sample of good code.

4.3 Committer as Mediator

Instead of being the principle actor, the committer acts as a knowledgeable mediator/moderator between parties interested in solving a particular problem. This pattern is especially prominent with bugzilla because the discussion of the problem need not be posted to the developer's mailing list (which can be daunting). In this pattern, the committer usually arrives after some discussion of the bug has already taken place and potentially after someone has created a patch. The committer negotiates between those who have a bug and are willing to test a patch and those who are skilled and frustrated enough with the bug to create a patch. The committer provides deeper insight into the ongoing discussion. We expect that this pattern will be more prominent with obscure and non-critical problems.

4.4 Focused Lazy Testing

Instead of using formal test suites, individuals who have a certain problem will test uncommitted patches and provide feedback. If the patch does not work it saves a core developer from performing a review. The testing is focused because only the patch associated with the bug is tested and lazy because no formal tests are run (the patch is just tried out).

5. APACHE

In this section, we discuss our results from a preliminary quantitative evaluation of the Apache project.

[13] Other patterns can be found at guinness.cs.uvic.ca:8080/~pcr/OSSCR/ in the reports section.

5.1 Core Group Changes

The Apache core group has changed dramatically over the lifetime of the project. Of the core group that Mockus et al. [4] studied between February 1996 and May 1999, only one member is still part of the top 20 committers in 2005; four members remain in the periphery of the project. This change is not a recent phenomenon, as many of the leading Apache developers in 1998 made way for other developers in 1999 and left the core group in 2000. Examining the top committer for each year, one notices that this individual remains on top for two years and gradually fades out over the next few years. In every instance, the top committer has at least one of his top two years with more than double the number of commits of the second highest committer. It will be interesting to use a metric that takes the movement of all developers, not just the top developer, into account. We expect that these changes lead to an influx of new and innovative ideas as well as changes in the development process. The original statistics that we gathered were project lifetime statistics, from 1997 to 2004 (1996 and 2005 are incomplete), but due to core group changes, these may be misleading. The task of analyzing and interpreting individual years is a much more involved and difficult task than analyzing the lifetime of the project. Where possible, we present yearly and monthly data.

5.2 Types of Review

Another complication is that the Apache project has many different types of review. The three identified types are pre-commit (formal review-then-commit), post-commit (commit-then-review), and secondary review. Pre-commit review is only recorded when a patch is committed; this means that reviewed revisions and rejected patches are not recorded. Post-commit review is never formally recorded, but we can deduce when it occurs by looking in the mailing list for replies to a committed patch. Since a reply to a patch likely represents someone finding a problem with the patch, we only know how many post-commit reviews contained problems. We do not know how many post-commit patches were reviewed with no problems found. Furthermore, we do not know how many different individuals post-commit reviewed a patch. Secondary review occurs when an individual does not review the patch themselves, but reads the comments of the reviewer, and makes an additional comment themselves. This type of review is only recorded as replies to a patch in the mailing list; it can apply to both types of reviews and is mainly left to future work.

5.3 Accept? Reject?

What percentage of reviewed patches are accepted? Rejected?

Over the lifetime of the project, 5747 patches were submitted for pre-commit review. Of these patches we were able to trace only 2522 to patch commits (44%). We expect that many patches were submitted more than once when they were ignored or if they required patch revisions. To determine why 56% of pre-commit reviewed patches were rejected, we intend to do more detailed data analysis including the use of more sophisticated message threading techniques and manual classification of a smaller time period.

Over the lifetime of the project, 9% of post-reviewed commits are found to have problems. We assume, based on qualitative information and post-commit review frequency (see below), that close to 100% of the patches are post-commit reviewed. This implies that 91% of patches are accepted. Interestingly, 5% of pre-commit reviewed patches are found to still contain a problem when post-commit review is performed. We believe that this is because pre-review patches are generally larger, more complex, and often submitted by external contributors.

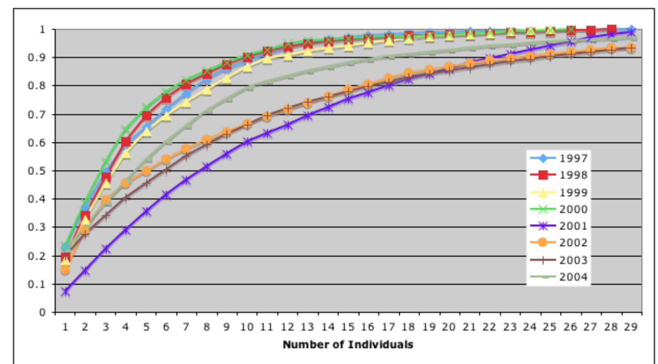

Figure 1: The cumulative distribution of pre-commit reviews by year

5.4 Reviewer Characteristics

Who performs the review?

Pre-commit review. Over the lifetime of the Apache project, reviews were performed by 130 individuals; however, many of these people only reviewed a single patch. In 1999, Mockus et al. [4] found that the Apache project had a core group size of 15 developers; this core group made 83% of the commits. Analyzing the same time period, we found that the top 15 reviewers performed 93% of the reviews. During this time, there were 55 individual reviewers and approximately 300 [4] individual [14] patch contributors. It would appear that the review group is a subset of the core group. Indeed, 10 reviewers are responsible for 84% of the reviews. In figure 1, it can be seen that between 1997 and 2000 the core reviews remains small. However, from 2001 to 2002, the review group grows to 20 reviewers performing 84% of the reviews. In 2004, the review group appears to have shrunk to 12 reviewers. We are not certain about the cause of this apparent fluctuation in reviewer group size. Currently, we cannot correlate it with commit core group size because we have not resolved the patch submitter names. However, figure 2 shows the total number of commits (inflated since submitter name is not resolved) and reviews over the lifetime of the project. This figure demonstrates that over the lifetime of the project, the group of committers and reviews is almost the same size. The core group size is much larger, with 80% of the commits done by 26 individuals, than Mockus's original finding of 15 developers (likely from core group changes).

The previous results pertain only to pre-commit reviews. Since post-commit and secondary review require name resolution on the mailing list (over 100,000 emails), we leave the determination of who performs these reviews to future work.

Are the top developers (committers) also the top reviewers?

Pre-commit review. We examined the lifetime top committers and reviewers to determine if the same individuals are on top in both roles. The group size varied from five to 20 people, each time we

[14] Examining Mockus' scripts revealed that resolution of names was only performed on core group members, thus inflating the total number of contributors, but having little effect on the percentage based size of the core group

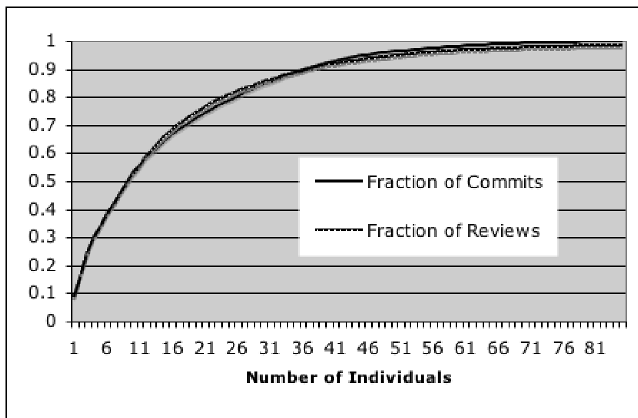

Figure 2: The cumulative distribution of reviews and commits for the lifetime of the project.

Common	Group Size	Percent
3	5	60
5	10	50
13	15	87
16	20	80

Table 1: Lifetime top reviewers and committers. The first column indicates which individuals are common to both roles. The second column is the total number of individuals (e.g., top five). The final column is the percentage of common individuals.

compare the review and commit group to determine how many individuals are common to both roles. Table 1 shows that at least 80% of individuals are common to both roles with a group size between 15 and 20 individuals. Surprisingly, the percentage does not continue to increase as the core group size increases from 15 to 20 (or from 20 to 26). This indicates that the review group is not necessarily a subset of the commit group. The two roles are even less related when we examine the top reviewers and committers. Table 1 illustrates that 60% or less of the individuals are common to both roles with a groups size of 10 to 15 individuals. Additionally, results regarding the top five reviewers and committers indicate that the reviewers have been with the project for an average of 7 years while the committers have been with the project for average of 4.75 years. The longevity of reviewers compared to committers indicates that strong reviewers stay with the project longer and potentially coach the newer developers. Further study of shorter time periods and individual people must be done before definite conclusions can be drawn.

5.5 Review Cycles

What is the frequency of review?

The median number of pre-commit reviews that were accepted is 262 per year (mean 272). The median number of reviews per month is 18 (mean 22). The actual number is likely higher, since these results do not account for rejected patches.

The median number of post-commit reviews that contained a problem is 258 per year (mean 304). The median number of reviews per month is 17 (mean 23).

Figure 3 shows the fluctuations of two review types at monthly intervals. Since more reviews will likely lead to more pre-commit review acceptance and post-commit reviews that find problems, this figure adequately shows the fluctuation in both review types. It appears that few months had high levels of both types of review. The years 2000, 2001, and 2002 (Apache 2.0 was released in 2002) had a sustained increase in commits over other years (See figure 4. These three years also have the highest levels of post-commit reviews and the lowest levels of pre-commit reviews. Qualitatively, we know that pre-commit reviewed patches are larger than post-commit reviewed patches. This size difference would produce more commits in years with a large number of post-commit reviews. We discuss other reasons for these fluctuations later. Formal correlational tests (e.g., Pearson correlations) need to be performed before we can draw definite conclusions.

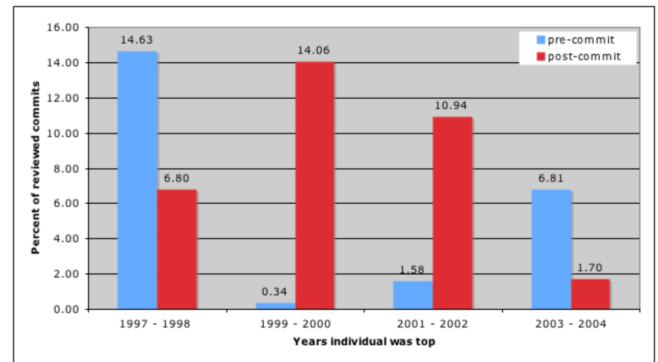

Figure 5: The top individual reviewers and the two years they were the top reviewer.

When are reviews performed?

The Apache policies state that changes to pre-release code can be post-commit reviewed, while changes to post-release code must be pre-commit reviewed. We have not examined the prevalence of each type of review in pre- and post release in each of the Apache branches. However, examination of figure 3 shows that the release of Apache 1.3 in June of 1998 and Apache 2.0 in April of 2002 were followed by an immediate decrease in post-commit review and a more gradual and less obvious increase in pre-commit review. It would be interesting to mark all releases on figure 3.

Apache policy also states that pre-commit review must be performed on new features and complex changes, while the post-commit review policy applies to all other patches. We do not believe that this is always applied in practice. Figure 5 show how often each type of review was performed for patches submitted by the top committer. It appears impossible that the top committer for 1999 and 2000 only contributed seven patches that required pre-commit review (i.e., were complex or were new features). Looking through the mailing list we found that this individual was aggressive toward his reviewers and toward people he pre-commit reviewed. We believe that the type of review performed is more dependent on the culture of the core group than on the type of patch.

5.6 Merit-based trust

How does merit-based trust between actors affect the review? Are more trusted individuals reviewed less often? How much feedback is provided in the review?

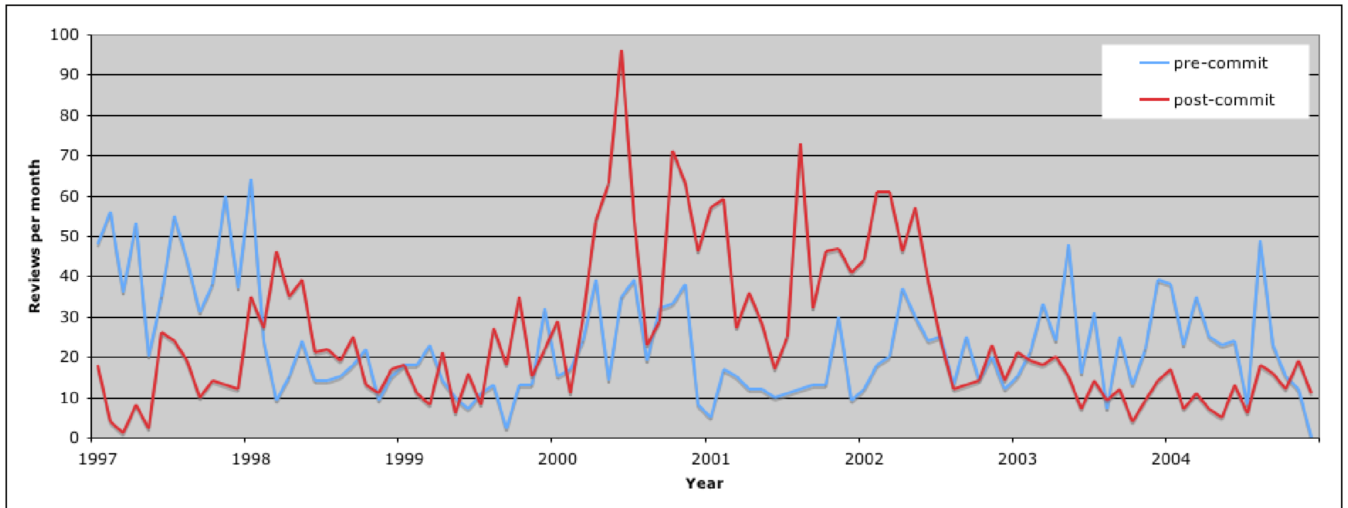

Figure 3: The blue line represents pre-commit reviews. The red line represents post-commit reviews.

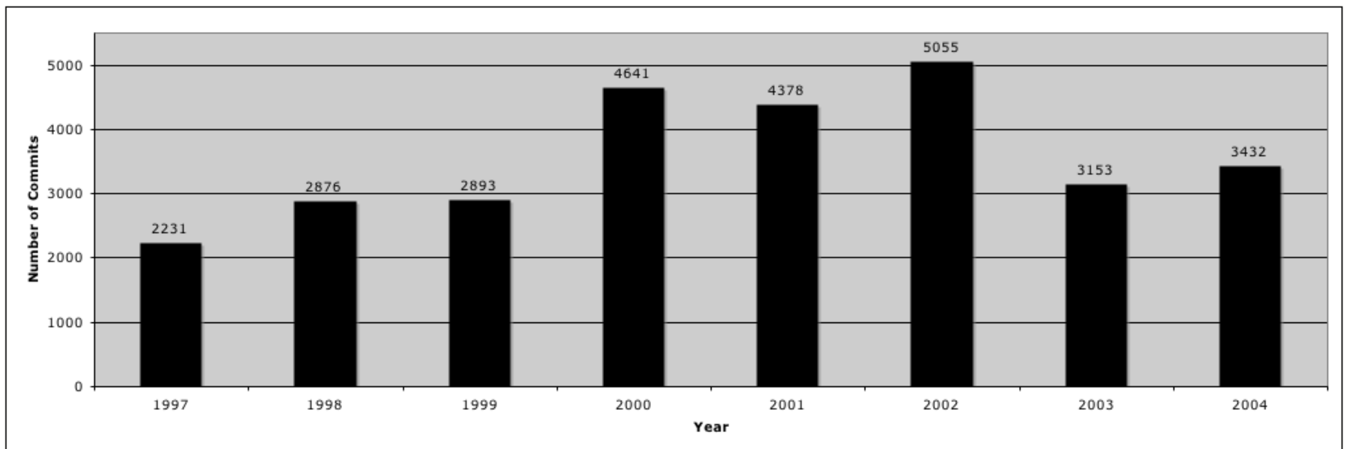

Figure 4: The number of raw commits per year.

The Apache policy of post-commit review applies only to trusted developers with commit privileges. This policy is left to the interpretation of the individual and the current understanding in the core group (See 5. All individuals who do not have commit privileges (i.e., are not in the core group) must be formally pre-commit reviewed. In future studies, we would like to understand how the trust affects the number of comments made in a review. We expect that newer, promising developers would be given more feedback than older developers with whom the reviewer has an established relationship.

5.7 Time and Patch Size

How long do reviews take to perform? How does the patch size affect the review?

Pre-commit review requires matching patches submitted to the mailing list with committed patches. This matching is difficult (rejected and resubmitted patches) and error prone; it is left to future work. However, we understand qualitatively that the more formal pre-commit review takes longer than post-commit review.

Post-commit reviews. The following quotations are from the discussion that ultimately led to Apache's acceptance of the commit-then-review (post-commit) policy. They illustrate many of the problems associated with pre-commit review. These quotes also qualitatively explain the sudden drop in pre-commit review experienced in early 1998 (See figure 3).

"This is a beta, and I'm tired of posting bloody one-line bug fix patches and waiting for votes." "Making a change to apache, regardless of what the change is, is a very heavyweight affair.", top reviewer in 1997 complaining about the review process in January of 1998.

"During the 1.2.5 release [before the acceptance of commit-then-review], [name removed] noted how many people are just +1ing [approving] code without really looking at it. I've done it. I'm sure we all have.", part of discussion in January of 1998.

"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.", part of discussion in January of 1998.

Post-commit reviews happen very soon after the patch is committed and are done surprisingly fast (even faster than the last quote would have us believe). Table 2) shows that 46% of post-commit reviews occurred in less than an hour and 84% occurred in less than 12 hours. To have response times like these, the Apache group must watch the commit mailing list very closely; this is why we believe that members of the core group review close to 100% of the patches. Additionally, given the short time frame, the size of the patch is likely very small when compared to longer-term reviews that examine a larger fix or problem. Mockus et al. [4] found that Apache had smaller patch sizes than the proprietary projects they examined; however, they did not understand why the patches were smaller. Furthermore, figures 4 and 3 indicate that the years with the largest number of commits also had higher levels of post-commit review. Since the patch is likely small, the review does not take very long, so the review does not tax the reviewer's attention.

Over the lifetime of the project, 9% of all commits received a post-commit review response (i.e., there was problem with the patch). If we assume that all commits are reviewed (high frequency and qualitatively), then this means that 9% of the time the reviewer found a problem in the patch, and 91% of the time the reviewer did not find a problem in the patch. If pre-commit review was used on all patches (as is the case with Mozilla), then nine times out of ten the developer would wait for a review only to hear that his or her patch was fine (false negative).

Time period	Cumulative percent
less 1 hour	45.8%
less 12 hours	83.8%
less 1 day	91.3%
less 1 week	97.9%
more 1 week	99.6

Table 2: The cumulative percent of responses to commits in a given time frame. Notice that over 90% of problems are found in less than a single day.

Traditionally, views toward code review require patches to contain a complete solution to a given problem or new feature. Reviews are taken very seriously and only performed at regular and usually long intervals. This long review cycle allows problems to become embedded in the code, making their removal difficult. In contrast, post-commit review requires solutions (or partial solutions) to small (or part of larger) problems. Since the code is immediately committed, the contributor and other developers can start using this code. If there is a problem in the code, it is noticed very soon after it is committed, and is relatively easy to remove, as there will be fewer dependencies on the problematic code.

Mockus et al. [4] found that unlike proprietary projects, the Apache project's *defect density* remained the same before and after a release. They mention in passing that this result "may indicate that fewer defects are injected into the code, or that other defect-finding activities such as inspections are conducted more frequently or more effectively." [4] Our quantitative result shows that indeed there are likely fewer defects injected into the code because reviews are conducted very frequently on small patches. There are efficiency gains because reviews are performed on small patches, because developers do not have to wait for a review (90% of the time the patch passes the review), and due to the relative ease and speed with which bad patches are caught and removed from the code (a vetoed patch must be either reversed, or an immediate fix accepted by the reviewer).

When compared to proprietary projects, Apache's pre-release defect densities are lower, but Apache's post-release defect densities (the same amount as pre-release defects) are higher. This difference can be explained by our finding that only 8.7% of problematic patches, which are found by post-commit review, are discovered after a single day has passed. This result also makes intuitive sense for two reasons. First, the reviewer has already reviewed and accepted the patch and will likely use other defect detection techniques before re-reviewing a patch. Second, as more time elapses, the number of small, fragmented patches to examine increases, making it difficult to find the offending patch (tool support could help). Defects that make it through post-commit review are mostly found by other defect detection techniques. Since post-release defect densities in Apache are high, it would appear that other open source defect detection strategies are less effective than traditional ones. However, we do not feel that the credit for low post-release

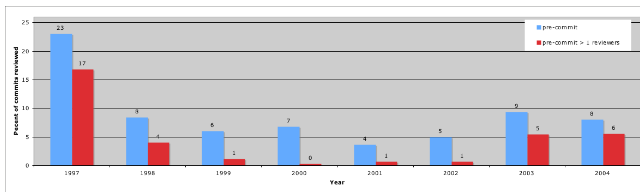

Figure 6: First column indicates percentage of commits reviewed per year. Second column indicates percentage of commits reviewed by two or more reviewers.

defect densities can be solely credited to post-commit review. Unlike proprietary projects, testers and users have access to and can post bugs about the current Apache source. Furthermore, it is difficult to obtain an accurate measure of defect density (and given that this is a report I have not had time to review the literature on defect densities in open source projects). A more recent study by Reasoning Inc. [15] indicated that the Apache project has a defect density comparable to proprietary software.

It must be noted that this type of post-commit review does not scale to large projects with less modular architectures than Apache's. It would appear (although I have not read the literature on peer and extreme programming) that post-commit review is, in effect, a type of distributed peer programming. In order to review code in such a short time period, reviewers must be intimately familiar with the code, and they must understand the effect of the code on the system as a whole. This type of distributed peer programming is potentially more efficient than traditional co-located peer programming (where one developer controls the input devices and the other comments and helps in navigation), because the peer must only examine what the other developer feels is the correct solution, not every step, including invalid ones, to the solution. If the peer does not feel the patch is correct, they can discuss the patch and give the developer navigational and other help. Furthermore, the advice is not limited to a single peer. Future experimentation is needed to validate these results

5.8 How many reviewers review a single patch?

Pre-commit review Although we do not know how many individuals unofficially review a patch (i.e., discuss the patch on the mailing list), we do know how many reviewers officially review patches that were accepted. For the lifetime of the project, 56% of patches were reviewed by one individual, 82% of reviews were performed by one or two reviewers, and only 18% of reviews had more than two reviewers. Figure 6 shows the percentage of commits that were reviewed in each year and the number of these commits that involved more than one reviewer. Again, it is obvious that certain years had high levels of involvement in the review process while other did not (Compare with figure 3.

Post-commit review. The median number of responses is two (mean 3.8). Normally, the first email describes the problem with the patch (is a response to the commit) and the second email is the response addressing the patch problem. Examining 80% of the reviews in ascending order revealed that these patches received 4 or fewer responses (not including the initial problem/review email). This finding shows that not only does the process of review happen quickly (see above), but the fix to the (small) patch is usually obvious and does not require a great deal of discussion. Since the number of individuals who discuss a patch must be equal to or less than the number of email messages, we expect that usually only two people discusses any given patch: the reviewer and the contributor. There are a few patches that have a large number of replies, the maximum number of responses is 80. Examining this email discussion, we discovered that the majority of the responses were not related to the patch, but some other problem that the patch had made obvious to the group. We expect that this type of tangential discussion or some controversial issue will be found in other patches with high response rates. Future work is required to validate these results and to determine the time between review and response to the problems found by the review.

[15] The results can be obtained at `http://www.reasoning.com/downloads.html` December 2005

5.9 Patch Kind

What kinds of non-source code patches are reviewed? How does the kind of patch affect the review?

These questions are mainly left to future work. However, as we noted in the patch statement section, Apache requires documentation to be submitted for new features and major changes. Since documentation is not critical to the functioning of the product, we expect that documentation is only reviewed when it comes from an external source, and then only minimally. Translations to other languages usually require at least two translators to perform the task. This process serves as review.

6. HYPOTHESES

Before developing hypotheses we need to fix some of the problems in our data, such as resolving the names in the mailing list and dealing with a number of minor inconsistencies, which are in less than 10% of our data. It would also be ideal to present our results to a group of interested researchers to obtain a different perspective on these findings before developing hypotheses. We leave the development and testing (on other projects) of hypotheses to future work.

7. CONCLUSION AND FUTURE WORK

This report is a first attempt at understanding the code review processes used by open source projects. In examining the "official" code review statements of four projects we found interesting similarities, such as the request for small, complete patches. We also found that differences in commit policy appeared to dictate the level of patch review. The amount, quality, and type of testing appeared to be related to how easy it is to automate tests. Fewer automated tests appears to force a project to do more formal code review. Through qualitative examination of development artifacts, we discovered a number of anecdotal patterns. The most interesting pattern was the "committer as mediator" pattern. This pattern occurs when a core group member acts as a knowledgeable mediator between those who report bugs and will try out new patches (lazy testers) and those who will invest the energy to create a patch.

We quantitatively answered our research questions by extracting the Apache project's developer and commit mailing lists into a database. Surprisingly, the core group has changed dramatically from 1996 to 2005, with a single, different top developer dominating the number of commits for two year periods. These changes affected the social and software development processes of the community, making lifetime analysis less meaningful. We found that the core reviewing group was similar, but not exactly the same as the committing core group. Pre-commit reviewers appear to have

been with the project for a longer period of time. We found that three different types of review exist: pre-commit, post-commit, and secondary review. Pre-commit review was similar to traditional formal review, while post-commit review appears unique to OSS development. Post-commit review allows trusted developers to commit patches without first being reviewed. When the patch is later reviewed, if it is found to contain an error, it must be fixed immediately or removed. Most errors (91.3% of the time) are found in less than a day allowing no time for a problematic patch to become embedded in the code. The process of post-commit review borrows elements of peer programming, but is potentially more efficient because peers only examines each other's final and best attempts. Post-commit review generally had a discussion of four emails indicating that likely only the reviewer and committer were involved. With pre-commit review few patches were reviewed by more than two individuals.

Throughout this paper we have mentioned areas we believe deserve future work; however, the most interesting are the post-commit review process, the changes to the core group, and the relationship between testing, review, and the number of defects in the software product.

8. REFERENCES

[1] A. Capiluppi, P. Lago, and M. Morisio. Characteristics of open source projects. volume 26, pages 317–327. Seventh European Conference on Software Maintenance and Reengineering, March 2003.

[2] D. M. German. Decentralized open source global software development, the GNOME experience. *Journal of Software Process: Improvement and Practice*, 8(4):201–215, 2004.

[3] D. M. German. Using software trails to reconstruct the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):367–384, 2004.

[4] 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, July 2002.

[5] E. S. Raymond. *The Cathedral and the Bazaar*. O'Reilly and Associates, 1999.