A Case Study of Open Source Software Development: The Apache Server

Audris Mockus
Bell Labs, 263 Shuman Blvd.
Naperville, IL 60566 USA
+1 630 713 4070
audris@research.bell-labs.com

Roy T. Fielding
Information & Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
fielding@ics.uci.edu

James Herbsleb
Bell Labs, 263 Shuman Blvd.
Naperville, IL 60566 USA
+1 630 713 1869
herbsleb@research.bell-labs.com

ABSTRACT

According to its proponents, open source style software development has the capacity to compete successfully, and perhaps in many cases displace, traditional commercial development methods. In order to begin investigating such claims, we examine the development process of a major open source application, the Apache web server. By using email archives of source code change history and problem reports we quantify aspects of developer participation, core team size, code ownership, productivity, defect density, and problem resolution interval for this OSS project. This analysis reveals a unique process, which performs well on important measures. We conclude that hybrid forms of development that borrow the most effective techniques from both the OSS and commercial worlds may lead to high performance software processes.

Keywords

software process, defect density, repair interval, code ownership, open source

1 INTRODUCTION

The open source software "movement" has received enormous attention in the last several years. It is often characterized as a fundamentally new way to develop software [6, 15] that poses a serious challenge [16] to the commercial software businesses that dominate most software markets today. The challenge is not the sort posed by a new competitor that operates according to the same rules but threatens to do it faster, better, cheaper. The OSS challenge is often described as much more fundamental, and goes to the basic motivations, economics, market structure, and philosophy of the institutions that develop, market, and use software.

The basic tenets of OSS development are clear enough, although the details can certainly be difficult to pin down precisely (see [14]). OSS, most people would agree, has as its underpinning certain legal and pragmatic arrangements that ensure that the source code for an OSS development will be generally available. Open source developments typically have a central person or body that selects some subset of the developed code for the "official" releases and makes them widely available for distribution.

These basic arrangements to ensure freely available source code have led to a development process that is radically different, according to OSS proponents, from the usual, industrial style of development. The main differences usually mentioned are

- OSS systems are built by potentially large numbers (i.e., hundreds or even thousands) of volunteers.
- Work is not assigned; people undertake the work they choose to undertake.
- There is no explicit system-level design, or even detailed design [16].
- There is no project plan, schedule, or list of deliverables.

Taken together, these differences suggest an extreme case of geographically distributed development, where developers work in arbitrary locations, rarely or never meet face to face, and coordinate their activity almost exclusively by means of email and bulletin boards. What is perhaps most surprising about the process is that it lacks many of the traditional mechanisms used to coordinate software development, such as plans, system-level design, schedules, and defined processes. These "coordination mechanisms" are generally considered to be even more important for geographically distributed development than for co-located development [9], yet here is an extreme case of distributed development that appears to eschew them all.

Despite the very substantial weakening of traditional ways of coordinating work, the results from OSS development are often claimed to be equivalent, or even superior to software developed more traditionally. It is claimed, for example, that defects are found and fixed very quickly because there are "many eyeballs" looking for the problems (Eric Raymond calls this "Linus's Law" [15]). Code is written with more care and creativity, because developers are working only on things for which they have a real passion [15].

It can no longer be doubted that OSS development has produced software of high quality and functionality. The

Linux operating system has recently enjoyed major commercial success, and is regarded by many as a serious competitor to commercial operating systems such as Windows [10]. Much of the software for the infrastructure of the internet, including the well known **bind, Apache,** and **sendmail** programs, were also developed in this fashion.

The Apache server (the OSS software under consideration in this case study) is, according to the Netcraft survey [13] the most widely deployed web server at the time of this writing. It accounts for over half of the 7 million or so web sites queried in the Netcraft data collection. In fact, the Apache server has grown in "market share" each year since it first appeared in the survey in 1996. By any standard, Apache is very successful.

While this existence proof means that OSS processes can, beyond a doubt, produce high quality and widely deployed software, the exact means by which this has happened, and the prospects for repeating OSS successes, are frequently debated (see, e.g., [12, 3]). Proponents claim that OSS software stacks up well against commercially developed software both in quality and in the level of support that users receive, although we are not aware of any convincing empirical studies that bear on such claims. If OSS really does pose a major challenge to the economics and the methods of commercial development, it is vital to understand it and to evaluate it.

This paper presents a case study of the development and maintenance of a major OSS project, the Apache server. We address key questions about the Apache development process, and about the software that is the result of that process. In the remainder of this section, we present our specific research questions. In Section 2, we describe our research methodology, followed by a description of the Apache development process in Section 3. Section 4 presents our quantitative results on participant and developer roles, product defect density, and user perspective of the Apache process and product. Finally, we present our conclusions and our hypotheses for future research in Section 5.

1.1 Research Questions

Our questions focus on two key sets of properties of OSS development. It is remarkable that large numbers of people manage to work together successfully to create high quality, widely used products. Our first set of questions (Q1-Q4) is aimed at understanding basic parameters of the process by which Apache came to exist.

Q1: What was the process used to develop Apache?

In answer to this question, we construct a brief qualitative description of Apache development.

Q2: How many people wrote code for new Apache functionality? How many people reported problems? How many people repaired defects?

We want to see how large the Apache development community is, and identify how many people actually occupied each of these traditional development and support roles.

Q3: Were these functions carried out by distinct groups of people, i.e., did people primarily assume a single role? Did large numbers of people participate somewhat equally in these activities, or did a small number of people do most of the work?

Within the Apache development community, what division of labor resulted from the OSS "people choose the work they do" policy? We want to construct a profile of participation in the ongoing work.

Q4: Where did the code contributors work in the code? Was strict code ownership enforced on a file or module level?

One worry of the "chaotic" OSS style of development is that people will make uncoordinated changes, particularly to the same file or module, that interfere with one another. How does the development community avoid this?

Our second set of questions (Q5-Q6) concerns the outcomes of this Apache process. We examine the software from a customer's point of view, with respect to the defect density of the released code, and the time to repair defects, especially those likely to significantly affect many customers.

Q5: What is the defect density of Apache code?

We compute defects per thousand lines of code, and defects per delta in order to compare different operationalizations of defect density.

Q6: How long did it take to resolve problems? Were higher priority problems resolved faster than low priority problems? Has resolution interval decreased over time?

We measured this interval because it is very important from a customer perspective to have problems resolved quickly.

2 METHODOLOGY AND DATA SOURCES

In order to produce an accurate description of the Apache development process, one of the authors (RTF), who has been a member of the core development team from the beginning of the Apache project wrote a draft description. This draft was then circulated among other core members, who checked it for accuracy and filled in missing details. The description in the next section is the final product of this process.

In order to address our quantitative research questions, we obtained key measures of project evolution from several sources of archival data that had been preserved throughout the history of the Apache project. The development and testing teams in OSS projects consist of individuals who rarely if ever meet face to face, or even via transitory media such as the telephone. One consequence of this is that

virtually all information on the OSS project is recorded in electronic form. Many other OSS projects archive similar data, so the techniques used here can be replicated on any such project. (A detailed description, including scripts used to extract the data are available from the authors on request.)

We used the following archival sources of data:

Developer email list (EMAIL). Anyone with an interest in working on Apache development can join the developer mailing list, which was archived monthly. It contains many different sorts of messages, including technical discussions, proposed changes, and automatic notification messages about changes in the code and problem reports. There were nearly 50,000 messages posted to the list during the period starting February, 1995. Our analysis is based on all email archives retrieved on May 20, 1999.

We wrote Perl scripts to extract date, sender identity, message subject, and the message body that was further processed to obtain details on code changes and problem reports (see below). Manual inspection was used to resolve such things as multiple email addresses in cases where all automated techniques failed.

Concurrent Version Control archive (CVS). The CVS commit transaction represents a basic change similar to the Modification Request (MR) in a commercial development environment. (We will refer to such changes as MRs.) Every commit automatically generates an email message stored in the apache-cvs archive, which we used to reconstruct the CVS data (the first recorded change was made on February 22, 1996. The version 1.0 of Apache released in January 1996 had a separate CVS database). The message body in the CVS mail archive corresponds to one MR and contains the following tuple: date and time of the change, developer login, files touched, numbers of lines added and deleted for each file, and a short abstract describing the change. We further processed the abstract to identify people who submitted and/or reviewed the change and to obtain the Problem Report (PR) number for changes made as a result of a problem report. According to a core participant of Apache, the information on contributors and PRs was entered at least 90% of the time. All changes to the code and documentation were used in the subsequent analysis.

Problem reporting database (BUGDB). As in CVS, each BUGDB transaction generates a message to BUGDB stored in a separate BUGDB archive. We used this archive to reconstruct BUGDB. For each message, we extracted the PR number, affected module, status (open, suspended, analyzed, feedback, closed), name of the submitter, date, and comment.

We used the data elements extracted from these archival sources to construct a number of measures on each change to the code, and on each problem report. We used the process description as a basis to interpret those measures. Where possible, we then further validated the measures by comparing several operational definitions, and by checking our interpretations with project participants. Each measure is defined in the following sections within the text of the analysis where it is used.

3 THE APACHE DEVELOPMENT PROCESS

Q1: What was the process used to develop Apache?

The Apache software development process is a result of both the nature of the project and the backgrounds of the project leaders, as described in [8]. Apache began with a conscious attempt to solve the process issues first, before development even started, because it was clear from the very beginning that a geographically distributed set of volunteers, without any traditional organizational ties, would require a unique development process in order to make decisions.

The Apache Group (AG), the informal organization of core people responsible for guiding the development of the Apache HTTP Server Project, consisted entirely of volunteers. None of the developers could devote large blocks of time to the project in a consistent or planned manner, therefore requiring a development and decision-making process that emphasized decentralized workspaces and asynchronous communication. AG used email lists exclusively to communicate with each other, and a minimal quorum voting system for resolving conflicts.

Apache began in February 1995 as a combined effort to coordinate existing fixes to the NCSA httpd program developed by Rob McCool. After several months of adding features and small fixes, AG replaced the old server code base in July 1995 with a new architecture designed by Robert Thau. AG then ported all existing features, and many new ones, to the new architecture and made it available for beta test sites, eventually leading to the formal release of Apache httpd 1.0 in January 1996.

The selection and roles of core developers are described in [8]. Each AG member can vote on the inclusion of any code change, and has write access to CVS (if they desire it). Members are people who have contributed for an extended period of time, usually more than 6 months, and are nominated for membership and then voted on by the existing members. We started with 8 members (the founders), had 12 through most of the period covered, and now have 25.

The "core developers" in any period include both the subset of AG that are active in development (usually 4-6 in any given week) and the developers who are on the cusp of being nominated (usually 2-3). That's why the "core" appears as 15 people during the period studied.

Although there is no single development process, each Apache developer iterates through a common series of actions while working on the software source. The actions

include discovering that a problem exists, determining whether a volunteer will work on it, identifying a solution, developing and testing the code within their local copy of the source, presenting the code changes to the AG for review, and committing the code and documentation to the repository. Depending on the scope of the change, this process may involve many iterations before reaching a conclusion, though it is generally preferred that the entire set of changes needed to solve a particular problem be applied in a single commit.

There are many avenues for discovering problems. Problems are reported on the developer mailing list, the problem reporting system (BUGDB), and the USENET Apache newsgroups. Problems on the mailing list are given the highest priority. Since the reporter is likely to be a member of the development community, the report is more likely to contain sufficient information to analyze the problem. These messages receive the attention of all active developers. Common mechanical problems, such as compilation or build problems, are typically found first by one of the core developers and either fixed immediately or reported and handled on the mailing list. In order to keep track of the project status, an agenda file is stored in each product's repository, containing a list of high priority problems, open issues, and release plans.

The second area for problem discovery is in the project's BUGDB, which allows anyone with Web or email access to enter and categorize problems by severity and topic area. Once entered, the problem report (PR) is posted to a separate mailing list and can be appended to via email replies, or edited directly by the core developers. Unfortunately, due to some annoying characteristics of the BUGDB technology, very few developers keep an active eye on the BUGDB. The project relies on one or two interested developers to perform periodic triage of the new reports: removing mistaken or misdirected reports, answering reports that can be answered quickly, and forwarding items to the developer mailing list if they are considered critical. When a problem from any source is repaired, the BUGDB is searched for reports associated with that problem so that they can be included in the change report and closed.

Another source for problem discovery is the discussion on Apache-related USENET newsgroups. However, the perceived noise level on those groups is so high that only a few Apache developers ever have time to read the news. In general, the Apache Group relies on interested volunteers and the community at large to recognize when a real problem is discovered and to take the time to report that problem to the BUGDB or forward it directly to the developer mailing list. In general, only problems reported on released versions of the server are recorded in BUGDB.

Once a problem has been discovered, the next step is to find a volunteer who will work on that problem. Developers tend to work on problems that are identified with areas of the code they are most familiar. Some work on the product's core services, while others work on particular features that they developed. The Apache software architecture is designed to separate the core functionality of the server, which every site needs, from the features, which are located in modules that can be selectively compiled and configured. The developers obtain an implicit "code ownership" of parts of the server that they are known to have created or to have maintained consistently. Although code ownership doesn't give them any special rights over change control, the other developers have greater respect for the opinions of those with experience in the area being changed. As a result, new developers tend to focus on areas where the former maintainer is no longer interested in working, or in the development of new architectures and features that have no preexisting claims (frontier building).

After deciding to work on a problem, the next step is attempting to identify a solution. In general, the primary difficulty at this stage is not finding a solution, it is in deciding which of various possibilities is the most appropriate solution. Even when the user provides a solution that works, it may have characteristics that are undesirable as a general solution or it may not be portable to other platforms. When several alternative solutions exist, the developer usually forwards the alternatives to the mailing list in order to get feedback from the rest of the group before developing a solution.

Once a solution has been identified, the developer makes changes to a local copy of the source code, tests the changes on their own server, and either commits the changes directly (if the Apache guidelines [1] call for a commit-then-review process) or produces a "patch" and posts it to the developer mailing list for review. If approved, the patch can be committed to the source by any of the developers, though in most cases it is preferred that the originator of the change also perform the commit.

As described above, each CVS commit results in a summary of the changes being automatically posted to the apache-cvs mailing list, including the commit log and a patch demonstrating the changes. All of the core developers are responsible for reviewing the apache-cvs mailing list to ensure that the changes are appropriate. In addition, since anyone can subscribe to the mailing list, the changes are reviewed by many people outside the core development community, which often results in useful feedback before the software is formally released as a package.

When the project nears a product release, one of the core developers volunteers to be the release manager, responsible for identifying the critical problems (if any) that prevent the release, determining when those problems have been repaired and the software has reached a stable

point, and controlling access to the repository so that developers don't inadvertently change things that should not be changed just prior to the release. The release manager creates a forcing effect in which many of the outstanding problem reports are identified and closed, changes suggested from outside the core developers are applied, and most loose ends are tied up. In essence, this amounts to "shaking the tree before raking up the leaves." The role of release manager is rotated among the core developers with the most experience with the project.

In summary, this description helps to address some of the questions about how Apache development was organized, and provides essential background for understanding our quantitative results. In the next section, we take a closer look at the distribution of development, defect repair, and testing work in the Apache project, as well as the code and process from the point of view of customer concerns.

4 QUANTITATIVE RESULTS

In this section we present results from several quantitative analyses of the archival data from the APACHE project. The measures we derive from these data are well-suited to address our research questions [2]; however, they may be unfamiliar to many readers since they are not software metrics that are in wide use, e.g., [4,7]. For this reason, we provide data from several commercial projects, to give the reader some sense of what kinds of results might be expected. Although we picked several commercial projects that are reasonably close to APACHE, none is a perfect match, and the reader should not infer that the variation between these commercial projects and APACHE is due entirely to differences between commercial and OSS development processes.

It is important to note that the server is designed so that new functionality need not be distributed along with the core server. There are well over 100 feature-filled modules distributed by third parties, and thus not included in our study. Many of these modules include more lines of code than the core server.

4.1 The size of the Apache development community.

Q2: How many people wrote code for new Apache functionality? How many people reported problems? How many people repaired defects?

The participation in Apache development overall was quite wide, with almost 400 individuals contributing code that was incorporated into a comparatively small product. In order to see how many people contributed new functionality and how many were involved in repairing defects, we distinguished between changes that were made as a result of a problem report (PR changes) and those that were not (non-PR changes). We found that 182 people contributed to 695 PR changes, while 249 people contributed to 6092 non-PR changes.

We examined the BUGDB to determine the number of people who submitted problem reports. The problem reports come from a much wider group of participants. In fact, around 3060 different people submitted 3975 problem reports. 458 individuals submitted 591 reports that subsequently caused a change to the Apache code or documentation. 2654 individuals submitted 3384 reports that did not result in a change.

4.2 How was work distributed within the development community?

Q3: Were these functions carried out by distinct groups of people, i.e., did people primarily assume a single role? Did large numbers of people participate somewhat equally in these activities, or did a small number of people do most of the work?

First, we examine participation in generating code. Figure 1 plots the cumulative proportion of code changes (vertical axis) versus the top N contributors to the code base (horizontal axis).

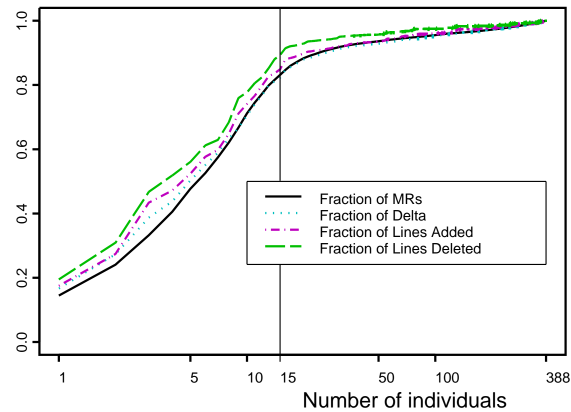

Figure 1. *The cumulative distribution of contributions to the code base.*

The contributors are ordered by the number of MRs from largest to smallest. The solid line in Figure 1 shows the cumulative proportion of changes against the number of contributors. The dotted and dashed lines show the cumulative proportion of added and deleted lines and the proportion of delta (an MR generates one delta for each of the files it changes). These measures capture various aspects of code contribution.

Figure 1 shows that the top 15 developers contributed more than 83% of the MRs and deltas, 88% of added lines and 91% of deleted lines. Very little code and, presumably, correspondingly small effort is spent by non-core developers (for simplicity, in this section we refer to all the developers outside the top 15 group as non-core). The MRs done by core developers are substantially larger than those done by the non-core group. This difference is statistically significant; the distribution of MR fraction is significantly

(p < 0.01) different from the distribution of added lines using Kolmogorov-Smirnov test.

Next, we looked separately at PR changes only. There was a large (p-value < 0.01) difference between distributions of PR and non-PR contributions. PR contributions are shown in Figure 2. The scales and developer order are the same as in Figure 1.

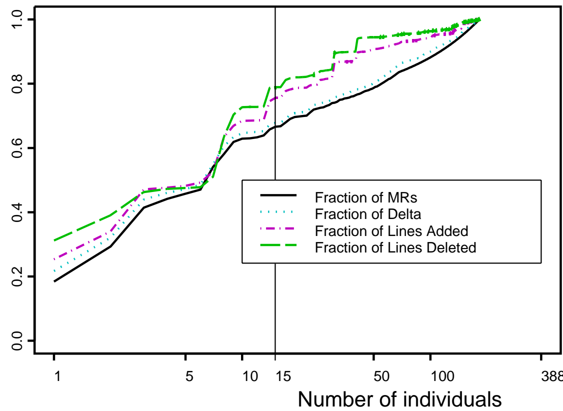

Figure 2. *Cumulative distribution of PR related changes.*

Figure 2 shows that participation of wider development community is more significant in defect repair than in the development of new functionality. Only 66% of the PR related changes were produced by the top 15 contributors. The participation rate was 26 developers per 100 PR changes and 4 developers per 100 non-PR changes, i.e., more than six times lower for PR changes. These results indicate that despite broad overall participation in the project, almost all new functionality is implemented and maintained by the core group.

We inspected the regularity of developer participation by considering two time intervals: before and after Jan 1, 1998. Forty-nine distinct developers contributed more than one PR change in the first period, and the same number again in the second period. Only 20 of them contributed at least two changes in both the first and second periods. One hundred and forty developers contributed at least one non-PR change in first period, and 120 in the second period. Of those, only 25 contributed during both periods. This indicates that only a few developers beyond the core group submit changes with any regularity.

Although developer contributions vary significantly in a commercial project, our experience has been that the variations are not as large as in the APACHE project. Since the cumulative fraction of contribution is not commonly available in the programmer productivity literature we present examples of several commercial projects that had a number of deltas within an order of magnitude of the number Apache had, and were developed over a similar period. Table 1 presents basic data about this comparison group. All projects come from the telecommunications domain. Project A is code for a wireless base station, project B is a port of legacy code for an optical network element, and projects C, D, and E represent various applications for operations, administration, and maintenance. The first two projects were written mostly in the C language, and the last three mostly in C++.

Table 1. *Statistics on Apache and five commercial projects.*

	MRs (K)	Delta (K)	Lines added (K)	Years	Developers
Apache	6	18	220	3	388
A	3.3	129	5,000	3	101
B	2.5	18	1,000	1.5	91
C	1.1	2.8	81	1.3	17
D	0.2	0.7	21	1.7	8
E	0.7	2.4	90	1.5	16

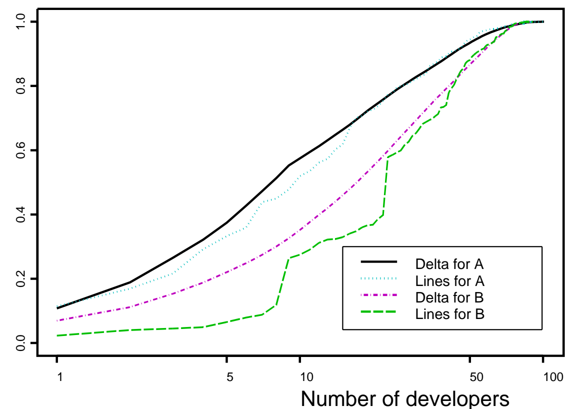

Figure 3.*Cumulative distribution of the contributions in two commercial projects.*

Figure 3 shows the cumulative fraction of changes for commercial projects A and B. To avoid clutter, and because they do not give additional insights, we do not show the curves for projects C, D, or E.

The top 15 developers in project A contributed 77 percent of the delta (compared to 83% for Apache) and 68 percent of the code (compared to 88%). Even more extreme differences emerge in porting of a legacy product done by project B. Here, only 46 and 33 percent of the delta and added lines are contributed by the top 15 developers.

We defined “top” developers in the commercial projects as groups of the most productive developers that contributed 83% of MRs (in the case of KMR/developer/year) and 88%

of lines added (in the case of KLOC/developer/year). We chose these proportions because they were the proportions we observed empirically for the summed contributions of the 15 core Apache developers.

Table 2. *Comparison of code productivity of the top Apache developers and the top developers in several commercial projects.*

	Apache	A	B	C	D	E
KMR/developer/year	.11	.03	.03	.09	.02	.06
KLOC/developer/year	4.3	38.6	11.7	6.1	5.4	10

If we look at the amount of code produced by the top Apache developers versus the top developers in the commercial projects, the Apache core developers appear to be very productive, given that Apache is a voluntary, part time activity and the relatively "lean" code of Apache. Measured in KLOC per year, they achieve a level of production that is within a factor of 1.5 of the top full-time developers in projects C and D. Moreover, the Apache core developers handle more MRs per year than the core developers on any of the commercial projects. (For reasons we do not fully understand, MRs are much smaller in Apache than in the commercial projects we examined.)

Given the many differences among these projects, we do not want to make strong claims about how productive the Apache core has been. Nevertheless, one is tempted to say that the data suggest rates of production that are at least in the same ballpark as commercial developments, especially considering the part-time nature of the undertaking.

Who reports problems?
Problem reporting is an essential part of any software project. In commercial projects the problems are mainly reported by build, test, and customer support teams. Who is performing these tasks in an OSS project?

The BUGDB had 3975 distinct problem reports. The top 15 problem reporters submitted only 213 or 5% of PRs. Almost 2600 developers submitted one report, 306 submitted two, 85 submitted three, and the maximum number of PRs submitted by one person was 32.

Of the top 15 problem reporters only three are also core developers. Because all problems that might affect end users tend to be reported in BUGDB, it shows that the significant role of system tester of the released code is reserved almost exclusively to the wide community of Apache users.

One would expect that some users, like administrators of web hosting shops, would be reporting most of the problems. Given the total number of websites (domain names) of over four million (according to the NetCraft survey [7]), this might indeed be so. The three thousand individuals reporting problems represent less than one percent of all Apache installations if we assume the number of actual servers to be one tenth of the number of websites (each server may host several websites).

4.3 Code Ownership

Q4: Where did the code contributors work in the code? Was strict code ownership enforced on a file or module level?

Given the informal, distributed way in which Apache has been built, we wanted to investigate whether some form of "code ownership" has evolved. We thought it likely, for example, that for most of the Apache modules, a single person would write the vast majority of the code, with perhaps a few minor contributions from others. The large proportion of code written by the core group contributed to our expectation that these 15 developers most likely arranged something approximating a partition of the code, in order to keep from making conflicting changes.

An examination of persons making changes to the code failed to support this expectation. Out of 42 "*.c*" files with more than 30 changes, 40 had at least two (and 20 had at least four) developers making more than 10% of the changes. This pattern strongly suggests some other mechanism for coordinating contributions. It seems that rather than any single individual writing all the code for a given module, those in the core group have a sufficient level of mutual trust that they contribute code to various modules as needed.

This finding verifies the previous qualitative description of code "ownership" to be more a matter of recognition of expertise than one of strictly enforced ability to make commits to partitions of the code base.

4.4 Defects

Q5: What is the defect density of Apache code?

First we discuss issues related to measuring defect density in an OSS project and then present the results, including comparison to four commercial projects.

4.4.1 How to Measure Defect Density.
One frequently used measure is post-release defects per thousand lines of delivered code. This measure has at least three major problems, however. First, "bloaty" code is generally regarded as bad code, but it will have an artificially low defect rate. Second, many incremental deliveries contain most of the code from previous releases, with only a small fraction of the code being changed. If all the code is counted, this will artificially lower the defect rate. Third, it fails to take into account how thoroughly the code is exercised. If there are only a few instances of the application actually installed, or if it is exercised very infrequently, this will dramatically reduce the defect rate, which again produces an anomalous result.

We know of no general solution to this problem, but we

strive to present a well-rounded picture by calculating two different measures, and comparing Apache to several commercial projects on each of them. To take into account the incremental nature of deliveries we emulate the traditional measure with defects per thousand lines of code added (KLOCA) (instead of delivered code). To deal with the “bloaty” code issue we also compute defects per thousand deltas. To a large degree, the second measure ameliorates the "bloaty" code problem, because even if changes are unnecessarily verbose, this is less likely to affect the number of deltas (independent of size of delta). We do not have usage intensity data, but it is reasonable to assume that usage intensity was much lower for all the commercial applications. Hence we expect that our presented defect density numbers for Apache are somewhat higher than they would have been if the usage intensity of Apache was more similar to that of commercial projects. Defects, in all cases, are reported problems that resulted in actual changes to the code.

If we take a customer's point of view, we should be concerned primarily with defects visible to customers, i.e., post-release defects, and not build and testing problems. The Apache PRs are very similar in this respect to counts of post-release defects, in that they were raised only against official, stable releases of Apache, not against interim development "releases."

However, if we are looking at defects as a measure of how well the development process functions, a slightly different comparison is in order. There is no provision for systematic system test in OSS generally, and for the Apache project in particular. So the appropriate comparison would be to pre-system test commercial software. Thus, the defect count would include all defects found during the system test stage or after (all defects found after "feature test complete" in the jargon of the quality gate system).

4.4.2 Defect Density Results

Table 3 compares Apache to the previous commercial projects. Project B did not have enough time in the field to accumulate customer-reported problems and we do not have pre-system test defects for Project A.

We see that the two defect density measures in commercial projects A, C, D, and E are in good agreement (the defect density itself varies substantially, though). While the user-perceived defect density of the Apache product is inferior to that of the commercial products, the defect density of the code before system test is much lower. This latter comparison 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. It is also possible that the diversity of backgrounds of the developers participating in the OSS project have reduced the probability of defects (see, e.g., [11]).

Table 3. *Comparison of defect density measures.*

Measure	Apache	A	C	D	E
Post-release Defects/KLOCA	2.64	0.11	0.1	0.7	0.1
Post-release Defects/KDelta	40.8	4.3	14	28	10
Post-feature test Defects/KLOCA	2.64	*	5.7	6.0	6.9
Post-feature test Defects/KDelta	40.8	*	164	196	256

4.5 Time to resolve problem reports

Q6: How long did it take to resolve problems? Were high priority problems resolved faster than low priority problems? Has resolution interval decreased over time?

The distribution of PR resolution interval is approximated by its empirical distribution function that maps interval in days to proportion of PRs resolved within that interval. Fifty percent of PRs are resolved within a day, 75% within 42 days, and 90% within 140 days. Further investigation showed that these numbers depend on priority, time period, and whether or not the PR causes a change to the code.

Priority. We operationalized priority in two ways. First we used the priority field reported in the BUGDB database. Priority defined in this way has no effect on interval. This is very different from commercial development, where priority is usually strongly related to interval. In Apache BUGDB, the priority field is entered by a person reporting the problem and often does not correspond to the priority as perceived by the core developer team.

The second approach for operationalizing priority categorized the modules into groups according to how many users depend on them. PRs were then categorized by the module to which they pertain. Such categories tend to reflect priorities since they reflect number of users (and developers) affected. Figure 4 shows comparisons among such groups of modules. The horizontal axis shows interval in days and the vertical axis shows proportion of MRs resolved within that interval. “Core” represents the kernel, protocol, and other essential parts of the server that must be present in every installation. “Most Sites” represents widely-deployed features that most sites will choose to include. PRs affecting either “Core” or “Most Sites” should be given higher priority because they potentially involve many (or all) customers and could potentially cause major failures. On the other hand, “OS” includes problems specific to certain operating systems, and “Major Optional” include features that are not as widely deployed. From a customer's point of view, “Core” and “Most Sites” PRs should be solved as quickly as possible, while the “OS” and “Major Optional” should generally receive lower priority.

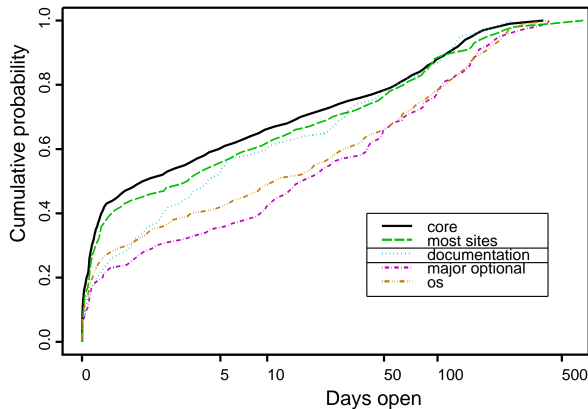

Figure 4. *Proportion of changes closed within given number of days.*

The data (Figure 4) show exactly this pattern, with much faster close times for the higher-priority problems. The differences between the trends in the two different groups are significant (p-value<.01 using Kolmogorov-Smirnov test), while the trends within groups do not differ significantly. The documentation PRs show mixed behavior, with "low priority" behavior for intervals under 5 days and "high priority" behavior, otherwise. This may be explained by the fact that documentation problems are not extremely urgent (the product still operates), yet very important.

Reduction in resolution interval. To investigate if the problem resolution interval improves over time, we broke the problems into two groups according to the time they were posted (before or after Jan 1, 1997). The interval was significantly shorter in the second period (p-value<.01). This indicates that this important aspect of customer support improved over time, despite the dramatic increase in the number of users.

5 HYPOTHESES AND REPLICATION

In this case study, we reported results relevant to each of our research questions. Specifically, we reported on

- the basic structure of the development process,
- the number of participants filling each of the major roles,
- the distinctiveness of the roles, and the importance of the core developers,
- suggestive, but not conclusive, comparisons of defect density and productivity with commercial projects, and
- customer support in OSS.

Case studies such as this provide excellent fodder for hypothesis development. It is generally inappropriate to generalize from a single case, but the analysis of a single case can provide important insights that lead to testable hypotheses. In this section, we cast some of our case study findings as hypotheses, and suggest explanations of why each hypothesis might be true of OSS in general. All the hypotheses can be tested by replicating this study using archival data from other OSS developments.

Hypotheses 1: Open source developments will have a core of developers who control the code base. This core will be no larger than 10-15 people, and will create approximately 80% or more of the new functionality.

We base this hypothesis both on our empirical findings in this case, and also on observations and common wisdom about maximum team size. The core developers must work closely together, each with fairly detailed knowledge of what other core members are doing. Without such knowledge they would frequently make incompatible changes to the code. Since they form essentially a single team, they can be overwhelmed by communication and coordination overhead issues that typically limit the size of effective teams to 10-15 people.

Hypothesis 2: For projects that are so large that 10-15 developers cannot write 80% of the code in a reasonable time frame, a strict code ownership policy will have to be adopted to separate the work of additional groups, creating, in effect, several related OSS projects.

The fixed maximum core team size obviously limits the output of features per unit time. To cope with this problem, a number of satellite projects, such as Apache-SSL, were started by interested parties. Some of these projects produced as much or more functionality than Apache itself. It seems likely that this pattern of core group and satellite groups that add unique functionality targeted to a particular group of users, will frequently be adopted in such cases.

In other OSS projects like Linux, the kernel functionality is also small compared to application and user interface functionalities. The nature of relationships between the core and satellite projects remains to be investigated; yet it might serve as an example how to break large monolithic commercial projects into smaller, more manageable pieces. We can see the examples where the integration of these related OSS products is performed by a commercial organization, e.g., RedHat for Linux, ActivePerl for Perl, and Cygnus for GNU tools.

Hypothesis 3: In successful open source developments, a group larger by an order of magnitude than the core will repair defects, and a yet larger group (by another order of magnitude) will report problems.

Hypothesis 4: Open source developments that have a strong core of developers but never achieve large numbers of contributors beyond that core will be able to create new functionality but will fail because of a lack of resources devoted to finding and repairing defects in the released code.

Many defect repairs can be performed with only a limited risk of interacting with other changes. Problem reporting can be done with no risk of harmful interaction at all.

Since this work has reduced dependencies among participants, potentially much larger groups can work on them. In a successful development, these activities will be performed by larger communities, freeing up time for the core developers to develop new functionality. Where an OSS development fails to stimulate wide participation, either the core will become overburdened with finding and repairing defects, or the code simply will never reach an acceptable level of quality.

Hypothesis 5: Defect density in open source releases will generally be lower than commercial code that has only been feature-tested, i.e., received a comparable level of testing.

Hypothesis 6: In successful open source developments, the developers will also be users of the software.

In general, open source developers are experienced users of the software they write. They are intimately familiar with the features they need, and what the correct and desirable behavior is. Since the lack of domain knowledge is one of the chief problems in large software projects [5], one of the main sources of error is eliminated when domain experts write the software. It remains to be seen if this advantage can completely compensate for the absence of system testing. In any event, where the developers are not also experienced users of the software, they are highly unlikely to have the necessary level of domain expertise or the necessary motivation to succeed as an OSS project.

Hypothesis 7: OSS developments exhibit very rapid responses to customer problems.

This observation stems both from the "many eyeballs implies shallow bugs" observation cited earlier [15], and the way that fixes are distributed. In the "free" world of OSS, patches can be made available to all customers nearly as soon as they are made. In commercial developments, by contrast, patches are generally bundled into new releases, and made available according to some predetermined schedule.

Taken together, these hypotheses, if confirmed with further research on OSS projects, suggest that OSS is a truly unique type of development process. It is tempting to suggest that commercial and OSS practices might be fruitfully hybridized in a number of ways. For example, it might prove very attractive to commercial developers to use the OSS style project structure. In such an arrangement, there is a core team of recognized experts, who alone have the power to commit code to an official release, and a much larger group who contribute voluntarily in various ways, and who may prove themselves diligent and skillful enough to be added to the core. Everyone, under this type of project management, is self-determining. The core members can commit code where they choose, the peripheral members submit changes of any sort they choose. These decisions appear to be guided only by a common desire to see the product developed successfully, to contribute in meaningful ways, and to be seen as an important contributor. While we are certain that this suggestion will be met with healthy skepticism, we see no inherent reason why commercial developments could not operate in a similar manner, subject of course to restrictions on size, and the necessity that developers must be users. Assuming that this arrangement would work in a commercial setting, there could be tremendous benefits to pairing the high motivation, low pre-system test defect rates, and fast response of OSS with a more commercially-oriented system test capability. Such cross-fertilization might pave the way to a true revolution in software development.

6 REFERENCES

1. Apache guidelines, at <http://dev.apache.org/guidelines.html>.

2. V. R. Basili and D. M. Weiss, “A Methodology for Collecting Valid Software Engineering Data,” *IEEE Transactions on Software Engineering*, vol. 10, no. 6, 1984, pp. 728-738.

3. T. Bollinger, R. Nelson, K. M. Self, and S. J. Turnbull, “Open-Source Methods: Peering Through the Clutter,” *IEEE Software*, vol. July/August, no. 4, 1999, pp. 8-11.

4. A. Carleton, et al., "Software Measurement for DoD Systems: Recommendations for Initial Core Measures," Software Engineering Institute, CMU/SEI-92-TR-19, 1992.

5. B. Curtis, H. Krasner, and N. Iscoe, “A Field Study of the Software Design Process for Large Systems,” *Communications of the ACM*, vol. 31, no. 11, 1988, pp. 1268-1287.

6. C. DiBona, S. Ockman, and M. Stone, *Open Sources: Voices from the Open Source Revolution*. Sebastopol, CA: O'Reilly, 1999.

7. Norman Fenton, “Software Measurement: A Necessary Scientific Basis,” *IEEE Transactions on Software Engineering*, vol. 20, no. 3, March 1994, pp. 199-206.

8. R. T. Fielding, “Shared Leadership in the Apache Project,” *Communications of the ACM*, vol. 42, no. 4, 1999, pp. 42-43.

9. J. D. Herbsleb and R. E. Grinter, “Splitting the Organization and Integrating the Code: Conway’s Law Revisited,” presented at 21st International Conference on Software Engineering (ICSE 99), Los Angeles, CA, 1999.

10. M. Krochmal, “Linux Interest Expanding,” in *TechWeb*, at <http://www.techweb.com/wire/story/TWB19990521S0021>, 1999.

11. B Littlewood and D Miller, “Conceptual Modeling of Coincident Failures in Multi-Version Software”, IEEE Transactions on Software Engineering, vol. 15, no. 12, Dec 1989, pp. 1596-1614.

12. S. McConnell, “Open-Source Methodology: Ready for Prime Time?,” *IEEE Software*, vol. July/August, no. 4, 1999, pp. 6-8.

13. Netcraft Survey, at <http://www.netcraft.com/survey>.

14. B. Perens, “The Open Source Definition,” in *Open Sources: Voices from the Open Source Revolution*, C. DiBona, S. Ockman, and M. Stone, Eds. Sebastopol, CA: O'Reilly, 1999, pp. 171-188.

15. E. S. Raymond, “The Cathedral and the Bazaar,” at <http://www.tuxedo.org/~esr/writings/cathedral-bazaar/>.

16. P. Vixie, “Software Engineering,” in *Open Sources: Voices from the Open Source Revolution*, C. DiBona, S. Ockman, and M. Stone, Eds. Sebastopol, CA: O'Reilly, 1999, pp. 91-100.