10 Two Case Studies of Open Source Software Development: Apache and Mozilla

Audris Mockus, Roy T. Fielding, and James D. Herbsleb

The open source software (OSS) "movement" has received enormous attention in the last several years. It is often characterized as a fundamentally new way to develop software (Di Bona et al. 1999; Raymond 2001) that poses a serious challenge (Vixie 1999) 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, and 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 Perens 1999). 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 it widely available for distribution.

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

- OSS systems are frequently built by large numbers (i.e., hundreds or even thousands) of volunteers. It is worth noting, though, that currently a number of OSS projects are supported by companies and some participants are not volunteers.
- Work is not assigned; people undertake the work they choose to undertake.

▪ There is no explicit system-level design, or even detailed design (Vixie 1999).
▪ 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 e-mail 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 colocated development (Herbsleb and Grinter 1999), yet OSS represents 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—Raymond (2001) calls this "Linus's Law." Code is written with more care and creativity, because developers are working only on things for which they have a real passion (Raymond 2001).

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 (Krochmal 1999). 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 (one of the OSS software projects under consideration in this case study) is, according to the Netcraft survey, the most widely deployed Web server at the time of this writing. It accounts for nearly 70% of the 54 million Web sites queried in the Netcraft data collection. In fact, the Apache server has led in "market share" each year since it first appeared in the survey in 1996. By any standard, Apache is very successful.

Although 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, for example, Bollinger et al. 1999 and McConnell 1999). 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.

Introduction

This chapter presents two case studies of the development and maintenance of major OSS projects: the Apache server and Mozilla. We address key questions about their development processes, and about the software that is the result of those processes. We first studied the Apache project, and based on our results, framed a number of hypotheses that we conjectured would be true generally of open source developments. In our second study, which we began after the analyses and hypothesis formation were completed, we examined comparable data from the Mozilla project. The data provide support for several of our original hypotheses.

Our research 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 to Q4) is aimed at understanding basic parameters of the process by which Apache and Mozilla came to exist.

Q1: What were the processes used to develop Apache and Mozilla?
In answer to this question, we construct brief qualitative descriptions of the Apache and Mozilla development processes.

Q2: How many people wrote code for new functionality? How many people reported problems? How many people repaired defects?
We want to see how large the development communities were, 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? That is, 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 each 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 regarding 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 to Q6) concerns the outcomes of these processes. 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 and Mozilla code?
We compute defects per thousand lines of code, and defects per delta in order to compare different operationalizations of the defect density measure.

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

In the following section, we describe our research methodology for both the Apache and Mozilla projects. This is followed in the third section by the results from the study of the Apache project, and hypotheses derived from those results. The fourth section presents our results from the study of the Mozilla project, and a discussion of those results in light of our previous hypotheses.

Methodology and Data Sources

In order to produce an accurate description of the open source development processes, we wrote a draft of description of each process, then had it reviewed by members of the core OSS development teams. For the Apache project, one of the authors (RTF), who has been a member of the core development team from the beginning of the Apache project, wrote the draft description. We then circulated it among all other core members and incorporated the comments of one member who provided feedback. For Mozilla, we wrote a draft based on many published accounts of the Mozilla process.[1] We sent this draft to the Chief Lizard Wrangler, who checked the draft for accuracy and provided comments. The descriptions in the next section are the final product of this process. The commercial

development process is well known to two of the authors (AM and JDH) from years of experience in the organization, in addition to scores of interviews with developers. We present a brief description of the commercial process at the end of this section.

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 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. (To facilitate future studies, the scripts used to extract the data are available for download at http://mockus.org/oss.)

Apache Data Sources

Developer E-mail List (EMAIL) Anyone with an interest in working on Apache development could 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 in February 1995. Our analysis is based on all e-mail archives retrieved on May 20, 1999.

We wrote Perl scripts to extract the date, the sender identity, the message subject, and the message body, which was further processed to obtain details on code changes and problem reports (see later discussion). Manual inspection resolved such things as multiple e-mail 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. Every MR automatically generates an e-mail message stored in the *apache-cvs* archive that 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 information: 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.

Some changes were made in response to problems that were reported. For each MR that was generated as a result of a problem report (PR), we obtained the PR number. We refer to changes made as a result of a PR as "fixes," and changes made without a problem report as "code submissions." According to a core participant of Apache, the information on contributors and PRs was entered at least 90 percent 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.

Mozilla Data Sources

The quantitative data were obtained from CVS archives for Mozilla and from the Bugzilla problem tracking system.

Deltas were extracted from the CVS archive running the CVS log on every file in the repository. MRs were constructed by gathering all deltas that share login and comment and are recorded within a single three-minute interval. The comment acknowledges people who submitted the code and contains relevant PR numbers (if any). As before, we refer to MRs containing PRs as "fixes," and the remaining MRs as "code submissions."

The product is broken down into directories */layout*, */mailnews*, and so on. Files required to build a browser and mail reader are distributed among them. We have selected several directories that correspond to modules in Mozilla (so that each one has an owner) and that are similar in size to the Apache project (that is, that generate between 3 thousand and 12 thousand delta per year). Abbreviated descriptions of directories taken from Mozilla documentation (Howard 2000) follow:

▪ */js* contains code for tokenizing, parsing, interpreting, and executing JavaScript scripts.
▪ */layout* contains code for the layout engine that decides how to divide up the "window real estate" among all the pieces of content.
▪ */editor* contains code used for the HTML editor (i.e., Composer in Mozilla Classic), for plain-text and HTML mail composition and for text fields and text areas throughout the product.
▪ */intl* contains code for supporting localization.
▪ */rdf* contains code for accessing various data and organizing their relationships according to Resource Description Framework (RDF), which is an open standard.
▪ */netwerk* contains code for low-level access to the network (using sockets and file and memory caches) as well as higher-level access (using various protocols such as http, ftp, gopher, and castanet).
▪ */xpinstall* contains the code for implementing the SmartUpdate feature from Mozilla Classic.

We refer to developers with e-mail domains *netscape.com* and *mozilla.org* as *internal developers*, and all others we call *external developers*. It is worth noting that some of the 12 people with the *mozilla.org* e-mail address are not affiliated with Netscape. We attempted to match email to full names to eliminate cases where people changed e-mail addresses over the considered period or used several different e-mail addresses, or when there was a spelling mistake.

To retrieve problem report data, we used scripts that would first retrieve all problem report numbers from Bugzilla, and then retrieve the details and the status changes of each problem report. In the analysis, we consider only three status changes for a problem report. A report is first CREATED, then it is RESOLVED, either by a fix or other action. (There are multiple reasons possibly; however, we discriminated only between FIXED and the rest in the following analysis.) After inspection, the report reaches the state of VERIFIED if it passes, or is reopened again if it does not pass. Only reports including code changes are inspected. Each report has a priority associated with it, with values P1 through P5. PRs also include the field "Product," with "Browser" being the most frequent value, occurring in 80 percent of PRs.

Data for Commercial Projects

The change history of the files in the five commercial projects was maintained using the Extended Change Management System (ECMS) (Midha 1997) for initiating and tracking changes, and the Source Code

Control System (SCCS) (Rochkind 1975) for managing different versions of the files.

We present a simplified description of the data collected by ECMS and SCCS that are relevant to our study. SCCS, like most version control systems, operates over a set of source code files. An atomic change, or *delta*, to the program text consists of the lines that were deleted and those that were added in order to make the change. Deltas are usually computed by a file-differencing algorithm (such as UNIX *diff*), invoked by SCCS, which compares an older version of a file with the current version.

SCCS records the following attributes for each change: the file with which it is associated, the date and time the change was "checked in," and the name and login of the developer who made it. Additionally, the SCCS database records each delta as a tuple including the actual source code that was changed (lines deleted and lines added), the login of the developer, the MR number (discussed later), and the date and time of the change.

In order to make a change to a software system, a developer might have to modify many files. ECMS groups atomic changes to the source code recorded by SCCS (over potentially many files) into logical changes referred to as Modification Requests (MRs). There is typically one developer per MR. An MR may have an English-language abstract associated with it, provided by the developer, describing the purpose of the change. The open time of the MR is recorded in ECMS. We use the time of the last delta of an MR as the MR close time. Some projects contain information about the project phase in which the MR is opened. We use it to identify MRs that fix post-feature test and postrelease defects.

Commercial Development Process

Here we describe the commercial development process used in the five comparison projects. We chose these projects because they had the time span and size of the same order of magnitude as Apache, and we have studied them previously, so we were intimately familiar with the processes involved and had access to their change data. In all projects, the changes to the source code follow a well-defined process. New software features that enhance the functionality of the product are the fundamental design unit by which the systems are extended. Changes that implement a feature or solve a problem are sent to the development organization and go through a rigorous design process. At the end of the design process, the work is assigned to developers in the form of Modification Requests, which list the work to be done to each module. To perform the changes, a developer makes the required modifications to the code, checks whether the changes

are satisfactory (within a limited context; that is, without a full system build), and then submits the MR. Code inspections, feature tests, integration, system tests, and release to customer follow. Each of these stages may generate fix MRs, which are assigned to a developer by a supervisor who assigns work according to developer availability and the type of expertise required. In all of the considered projects, the developers had ownership of the code modules.

The five considered projects were related to various aspects of telecommunications. Project A involved software for a network element in an optical backbone network such as SONET or SDH. Project B involved call handling software for a wireless network. The product was written in C and C++ languages. The changes used in the analysis pertain to two years of mostly porting work to make legacy software run on a new real-time operating system. Projects C, D, and E represent operations administration and maintenance support software for telecommunications products. These projects were smaller in scale than projects A and B.

Study 1: The Apache Project

The Apache Development Process

Q1: What was the process used to develop Apache?

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, Apache developers replaced the old server code base in July 1995 with a new architecture designed by Robert Thau. Then all existing features, and many new ones, were ported to the new architecture and it was made available for beta test sites, eventually leading to the formal release of Apache httpd 1.0 in January 1996.

The Apache software development process is a result of both the nature of the project and the backgrounds of the project leaders, as described by Fielding (1999). 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.

Roles and Responsibilities The Apache Group (AG), the informal organization of people responsible for guiding the development of the Apache

HTTP Server Project, consisted entirely of volunteers, each having at least one other “real” job that competed for their time. For this reason, 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 e-mail lists exclusively to communicate with each other, and a minimal quorum voting system for resolving conflicts.

The selection and roles of core developers are described in Fielding 1999. AG members are people who have contributed for an extended period of time, usually more than six months, and are nominated for membership and then voted on by the existing members. AG started with 8 members (the founders), had 12 through most of the period covered, and now has 25. What we refer to as the set of “core developers” is not identical to the set of AG members; core developers at any point in time include the subset of AG that is active in development (usually 4 to 6 in any given week) and the developers who are on the cusp of being nominated to AG membership (usually 2 to 3).

Each AG member can vote on the inclusion of any code change, and has commit access to CVS (if he or she desires it). Each AG member is expected to use his or her judgment about committing code to the base, but there is no rule prohibiting any AG member from committing code to any part of the server. Votes are generally reserved for major changes that would affect other developers who are adding or changing functionality.

Although there is no single development process, each Apache core developer iterates through a common series of actions while working on the software source. These actions include discovering that a problem exists or new functionality is needed, determining whether a volunteer will work on the issue, 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 might involve many iterations before reaching a conclusion, although it is generally preferred that the entire set of changes needed to solve a particular problem or add a particular enhancement be applied in a single commit.

Identifying Work to Be Done There are many avenues through which the Apache community can report problems and propose enhancements. Change requests are reported on the developer mailing list, the problem reporting system (BUGDB), and the Usenet newsgroups associated with the

Apache products. The developer discussion list is where new features and patches for bugs are discussed and BUGDB is where bugs are reported (usually with no patch). Change requests 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 request or contain a patch to solve 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 (*STATUS*) is stored in each product's repository, containing a list of high-priority problems, open issues among the developers, and release plans.

The second area for reporting problems or requesting enhancements is in the project's BUGDB, which allows anyone with Web or e-mail access to enter and categorize requests by severity and topic area. Once entered, the request is posted to a separate mailing list and can be appended to via e-mail 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 requests: removing mistaken or misdirected problem reports, answering requests 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 avenue for reporting problems and requesting enhancements 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 promising enhancements and real problems, and to take the time to report them to the BUGDB or forward them directly to the developer mailing list. In general, only problems reported on released versions of the server are recorded in BUGDB.

In order for a proposed change actually to be made, an AG member must ultimately be persuaded it is needed or desirable. "Showstoppers"—that is, problems that are sufficiently serious (in the view of a majority of AG members) that a release cannot go forward until they are solved—are always addressed. Other proposed changes are discussed on the developer

mailing list, and if an AG member is convinced that it is important, an effort is made to get the work done.

Assigning and Performing Development Work Once a problem or enhancement has found favor with the AG, the next step is to find a volunteer who will work on that problem. Core developers tend to work on problems that are identified with areas of the code with which they are most familiar. Some work on the product's core services, and 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 core 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 core developers have greater respect for the opinions of those with experience in the area being changed. As a result, new core 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.

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

Prerelease Testing Once a solution has been identified, the developer makes changes to a local copy of the source code and tests the changes on his or her own server. This level of testing is more or less comparable to unit test, and perhaps feature test in a commercial development, although the thoroughness of the test depends on the judgment and expertise of the developer. There is no additional testing (e.g., regression, system test) required prior to release, although review is required before or after committing the change (see next section).

Inspections After unit testing, the core developer either commits the changes directly (if the Apache guidelines under revision with Apache

Group 2004 call for a commit-then-review process) or produces a "patch" and posts it to the developer mailing list for review. In general, changes to a stable release require review before being committed, whereas changes to development releases are reviewed after the change is committed. If approved, the patch can be committed to the source by any of the developers, although in most cases it is preferred that the originator of the change also perform the commit.

As described previously, 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. Most core developers do in fact review all changes. 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.

Managing Releases 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.

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 (Basili and Weiss 1984). However, they might be unfamiliar to many readers since the

software metrics are not in wide use—see, for example, Carleton et al. 1992 and Fenton 1994. For this reason and to give the reader some sense of what kinds of results might be expected, we provide data from several commercial projects. 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 that are distributed by third parties and thus not included in our study. Many of these modules include more lines of code than the core server.

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 (fixes) and those that were not (code submissions). We found that 182 people contributed to 695 fixes, and 249 people contributed to 6,092 code submissions.

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 3,060 different people submitted 3,975 problem reports, whereas 458 individuals submitted 591 reports that subsequently caused a change to the Apache code or documentation. The remaining reports did not lead to a change because they did not contain sufficient detail to reproduce the defect, the defect was already fixed or raised, the issue was related to incorrect configuration of the product, or the defect was deemed to be not sufficiently important to be fixed. Many of the reports were in regard to operating system faults that were fixed by the system vendor, and a few others were simply invalid reports due to spam directed at the bug reporting system's e-mail interface. There were 2,654 individuals who submitted 3,384 reports that we could not trace to a code change.

How Was Work Distributed within the Development Community?

Q3: Were these functions carried out by distinct groups of people? That is, 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 10.1 plots the cumulative proportion of code changes (vertical axis) versus the top *N* contributors to the code base (horizontal axis).

The contributors are ordered by the number of MRs from largest to smallest. The solid line in figure 10.1 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 10.1 shows that the top 15 developers contributed more than 83 percent of the MRs and deltas, 88 percent of added lines, and 91 percent of deleted lines. Very little code and, presumably, correspondingly small effort is spent by noncore developers (for simplicity, in this section we

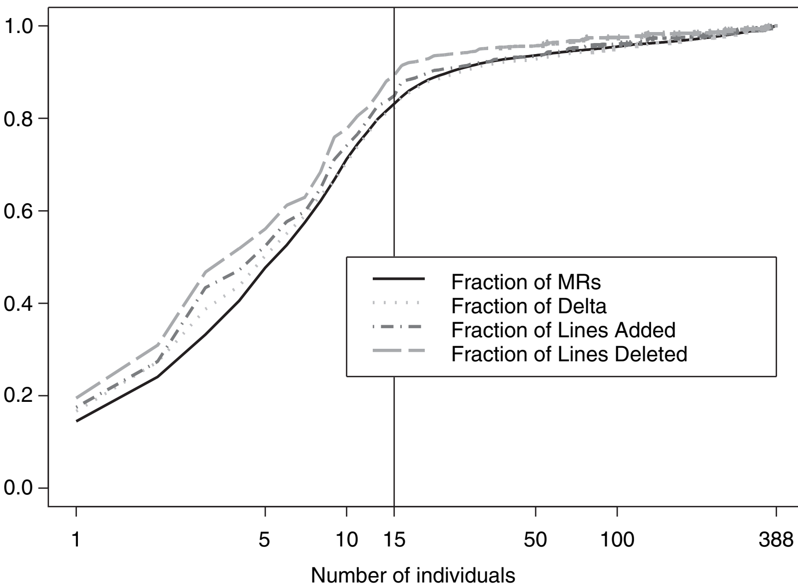

Figure 10.1
Cumulative distribution of contributions to the code base

refer to all the developers outside the top 15 group as *noncore*). The MRs done by core developers are substantially larger, as measured by lines of code added, than those done by the noncore group. This difference is statistically significant. The distribution of the MR fraction is significantly (p-value < 0.01) smaller (high values of the distribution function are achieved for smaller values of the argument) than the distribution of added lines using the Kolmogorov-Smirnov test. The Kolmogorov-Smirnov test is a nonparametric test that uses empirical distribution functions (such as shown in figure 10.1). We used a one-sided test with a null hypothesis that the distribution of the fraction of MRs is not less than the distribution of the fraction of added lines. Each of the two samples under comparison contained 388 observations representing the fraction of MRs and the fraction of lines added by each developer.

Next, we looked separately at fixes only. There was a large (p-value $<$ 0.01) difference between distributions of fixes and code submissions. (We used a two-sample test with samples of the fraction of MRs for fixes and code submissions. There were 182 observations in the fix sample and 249 observations in the code submission sample.) Fixes are shown in figure 10.2. The scales and developer order are the same as in figure 10.1.

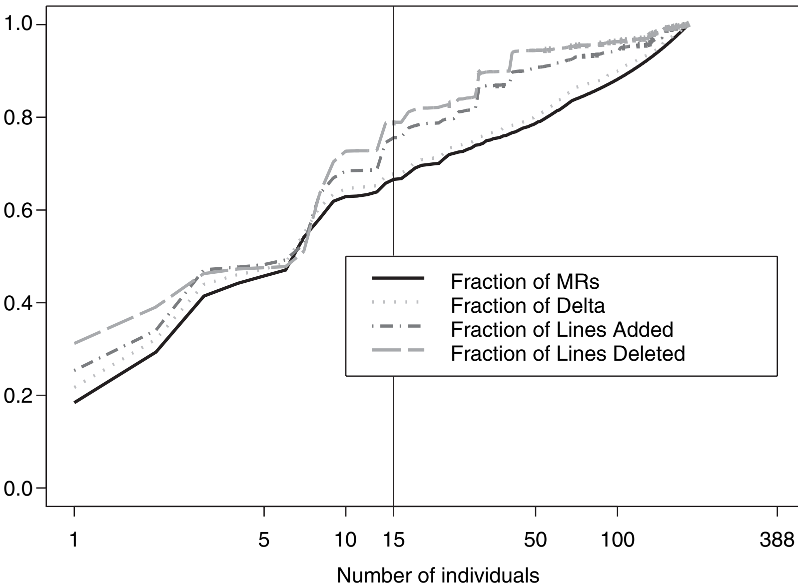

Figure 10.2
Cumulative distribution of fixes

Figure 10.2 shows that participation of the wider development community is more significant in defect repair than in the development of new functionality. The core of 15 developers produced only 66 percent of the fixes. The participation rate was 26 developers per 100 fixes and 4 developers per 100 code submissions, that is, more than six times lower for fixes. 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 January 1, 1998. Forty-nine distinct developers contributed more than one fix 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 code submission in the 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 10.1 presents basic data about this comparison group. All projects come from the telecommunications domain (see earlier sections). The first two projects were written mostly in the C language, and the last three mostly in C++.

Table 10.1
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

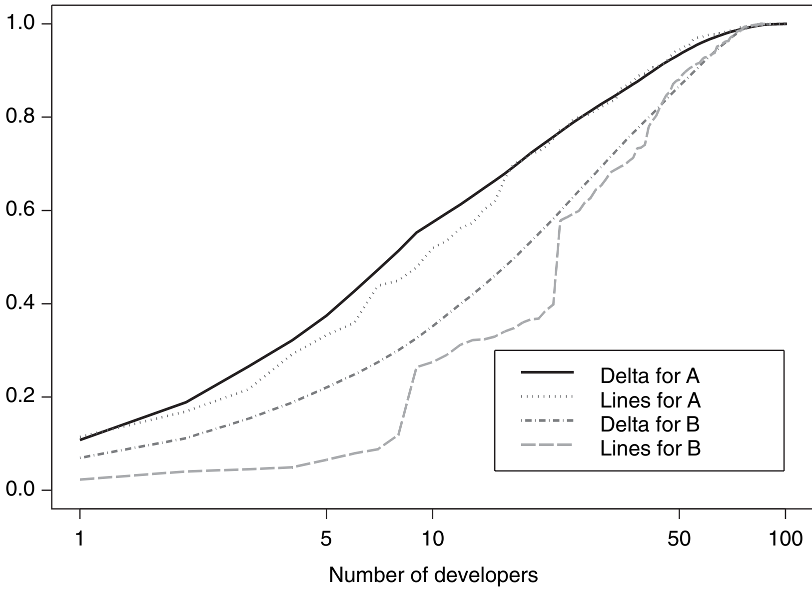

Figure 10.3
Cumulative distribution of the contributions in two commercial projects

Figure 10.3 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 B contributed 77 percent of the delta (compared to 83 percent for Apache) and 68 percent of the code (compared to 88 percent). Even more extreme differences emerge in porting of a legacy product done by project A. 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 percent of MRs and 88 percent of lines added. We chose these proportions because they were the proportions we observed empirically for the summed contributions of the 15 core Apache developers.

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 (see table 10.2). Measured in thousands of lines of code (KLOC) per year, they achieve a level of production that is within a factor of 1.5 of the top full-time devel-

Table 10.2
Comparison of code productivity of top Apache developers and 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

opers 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 in Apache are much smaller, in lines of code added, 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 3,975 distinct problem reports. The top 15 problem reporters submitted only 213 or 5 percent of PRs. Almost 2,600 developers submitted one report, 306 submitted 2, 85 submitted 3, and the maximum number of PRs submitted by one person was 32.

Of the top 15 problem reporters only 3 are also core developers. It shows that the significant role of system tester is reserved almost exclusively for the wide community of Apache users.

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 2 (and 20 had at least 4) developers making more than 10 percent 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.

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 with four commercial projects.

How to Measure Defect Density One frequently used measure is postrelease defects per thousand lines of delivered code. This measure has several major problems, though. 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 were 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; that is, postrelease defects, and not build and testing problems. The Apache PRs are very similar in this respect to counts of postrelease 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 presystem-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).

Defect Density Results Table 10.3 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 presystem test defects for project A. The defect data for Apache was obtained from BUGDB, and for commercial projects from ECMS as described previously. Only defects resulting in a code change are presented in table 10.3.

The defect density in commercial projects A, C, D, and E varies substantially. Although the user-perceived defect density of the Apache

Table 10.3
Comparison of Defect Density Measures

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

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.

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 Apache PR resolution interval is approximated by its empirical distribution function that maps the interval in days to proportion of PRs resolved within that interval. Fifty percent of PRs are resolved within a day, 75 percent within 42 days, and 90 percent within 140 days. Further investigation showed that these numbers depend on priority, time period, and whether 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 lack of impact 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.

In our second approach for operationalizing priority, we categorized the modules into groups according to how many users depended on them. PRs were then categorized by the module to which they pertained. Such categories tend to reflect priorities, since they reflect number of users (and developers) affected. Figure 10.4 shows comparisons among such groups of modules. The horizontal axis shows the interval in days and the vertical axis shows the 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” includes features that are not as widely deployed. From a customer’s point of view, “Core” and “Most

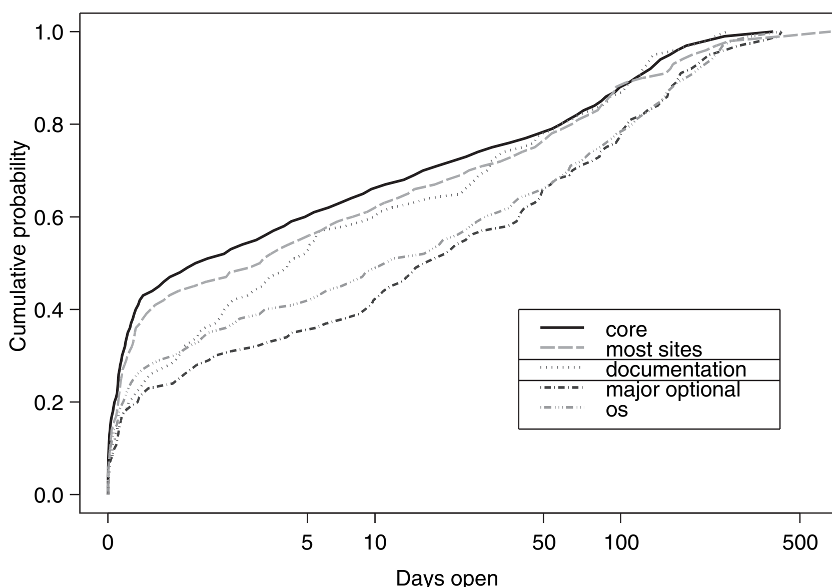

Figure 10.4
Proportion of changes closed within given number of days

sites" PRs should be solved as quickly as possible, and the "OS" and "Major optional" should generally receive lower priority.

The data (figure 10.4) 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 < 0.01 using the Kolmogorov-Smirnov test), whereas the trends within groups do not differ significantly. The documentation PRs show mixed behavior, with "low-priority" behavior for intervals under five days and "high-priority" behavior otherwise. This may be explained by a lack of urgency for documentation problems (the product still operates), despite being very important.

Reduction in Resolution Interval To investigate whether the problem resolution interval improves over time, we broke the problems into two groups according to the time they were posted (before or after January 1, 1997). The interval was significantly shorter in the second period (p-value < 0.01). This change indicates that this important aspect of customer support improved over time, despite the dramatic increase in the number of users.

Hypotheses

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
- 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. In the following section, we present results from Study 2, another case study, which allows us to test several of these hypotheses. All the hypotheses can be tested by replicating these studies using archival data from other OSS developments.

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

We base this hypothesis both on our empirical findings in this case and 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 to 15 people.

Hypothesis 2: For projects that are so large that 10 to 15 developers cannot write 80 percent 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, such as 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 of 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; for example, Red Hat for Linux, ActivePerl for Perl, and CYGWIN 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.

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 these types of work typically have fewer dependencies among participants than does the development of new functionality, potentially much larger groups can work on them. In 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 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; that is, 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 the correct and desirable behavior. Since the lack of domain knowledge is

one of the chief problems in large software projects (Curtis, Krasner, and Iscoe 1988), one of the main sources of error is eliminated when domain experts write the software. It remains to be seen whether 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 (Raymond 2001), 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, a thought which led us to collect and analyze the data reported in Study 2.

Subsequent to our formulation of these hypotheses, we decided to replicate this analysis on another open source project. We wanted to test these hypotheses, where possible, and we particularly wanted to look at a hybrid commercial/OSS project in order to improve our understanding of how they could be combined, and what the results of such a combination would be. Recent developments in the marketplace brought forth several such hybrid projects, most notably the Mozilla browser, based on the commercial Netscape browser source code.

In the next section, we use the methodology described earlier to characterize Mozilla development, to answer the same basic questions about the development process, and insofar as possible, test the hypotheses we developed in Study 1.

Study 2: The Mozilla Project

Mozilla has a process with commercial roots. In the face of stiff competition, Netscape announced in January, 1998 that their Communicator product would be available free of charge, and that the source code would also be free of charge. Their stated hope was to emulate the successful

development approach of projects such as Linux. The group mozilla.org was chartered to act as a central point of contact and "benevolent dictator" for the open source effort. Compared to the Apache project, the work in the Mozilla project is much more diverse: it supports many technologies including development tools (CVS, Bugzilla, Bonsai, Tinderbox) that are not part of the Web browser. It also builds toolkit-type applications, some of which are used to build a variety of products, such as Komodo from ActiveState. At the time of writing, it is unclear how well Netscape's open source strategy has succeeded.

There are many ways in which characteristics of open source and commercial development might be combined, and Mozilla represents only a single point in a rather large space of possibilities. It must be kept in mind, therefore, that very different results might be obtained from different hybridization strategies. In our conclusions, we describe what we see as the strengths and weaknesses of the Mozilla approach, and suggest other strategies that seem promising.

We base our description of the Mozilla development process on references[2] with a view from the inside (Baker 2000; Paquin and Tabb 1998), from the outside (Oeschger and Boswell 2000), and from a historic perspective (Hecker 1999; Zawinski 1999).

The Mozilla Development Process

Q1: What was the process used to develop Mozilla?

Mozilla initially had difficulty attracting the level of outside contributions that was expected. Mitchell Baker, "Chief Lizard Wrangler" of mozilla.org, expressed the view that "the public expectations for the Mozilla project were set astoundingly high. The number of volunteers participating in the Mozilla project did not meet those expectations. But there has been an important group of volunteers providing critical contributions to the project since long before the code was ready to use." After one year, one of the project leaders quit, citing lack of outside interest because of the large size, cumbersome architecture, absence of a working product, and lack of adequate support from Netscape.

However, after the documentation was improved, tutorials were written, and the development tools and processes refined, participation started slowly to increase. Some documents now available address the entire range of outsider problems (such as Oeschger and Boswell 2000). Also, the fact that the development tools were exported to be used in commercial software projects at Hewlett-Packard, Oracle, Red Hat, and Sun Microsystems

(Williams 2000), is evidence of their high quality and scalability. At the time of this writing, Mozilla is approaching its first release—1.0.

Mozilla has substantial documentation on the architecture and the technologies used, and has instructions for building and testing. It also has Web tools to provide code cross-reference (LXR) and change presentation (Bonsai) systems. A brief point-by-point comparison of the Apache and Mozilla processes is presented in table 10.8 in the appendix to this chapter. Next, we describe the necessary details.

Roles and Responsibilities Mozilla is currently operated by the mozilla.org staff (12 members at the time of this writing) who coordinate and guide the project, provide process, and engage in some coding. Only about four of the core members spend a significant part of their time writing code for the browser application. Others have roles dedicated to such things as community QA, milestone releases, Web site tools and maintenance, and tools such as Bugzilla that assist developers. Although the external participation (beyond Netscape) has increased over the years, even some external people (from Sun Microsystems, for example) are working full-time, for pay, on the project.

Decision-making authority for various modules is delegated to individuals in the development community who are close to that particular code. People with an established record of good quality code can attempt to obtain commit access to the CVS repository. Directories and files within a particular module can be added or changed by getting the permission of the module owner. Adding a new module requires the permission of mozilla.org. Much responsibility is delegated by means of distributed commit access and module ownership; however, mozilla.org has the ultimate decision-making authority, and retains the right to designate and remove module owners, and to resolve all conflicts that arise.

Identifying Work to Be Done Mozilla.org maintains a roadmap document (Eich 2001) that specifies what will be included in future releases, as well as dates for which releases are scheduled. Mozilla.org determines content and timing, but goes to considerable lengths to ensure that the development community is able to comment on and participate in these decisions.

Anyone can report bugs or request enhancements. The process and hints are presented in Mozilla Project. The bug reporting and enhancement request process uses the Bugzilla problem-reporting tool, and requires requesters to set up an account on the system. Bugzilla also has tools that

allow the bug reporter to see the most recent bugs, and if desired, to search the entire database of problem reports. Potential bug reporters are urged to use these tools to avoid duplicate bug reports. In addition, bug reporters are urged to come up with the simplest Web page that would reproduce the bug, in order to expedite and simplify the bug's resolution. Bugzilla provides a detailed form to report problems or describe the desired enhancement.

Assigning and Performing Development Work The mozilla.org members who write browser code appear to focus on areas where they have expertise and where work is most needed to support upcoming releases. The development community can browse Bugzilla to identify bugs or enhancements on which they would like to work. Fixes are often submitted as attachments to Bugzilla problem reports. Developers can mark Bugzilla items with a "helpwanted" keyword if they think an item is worth doing but don't themselves have the resources or all the required expertise. Discussions can also be found in Mozilla news groups, which may give development community members ideas about where to contribute. Mozilla.org members may use the Mozilla Web pages to note particular areas where help is needed. When working on a particular Bugzilla item, developers are encouraged to record that fact in Bugzilla in order to avoid duplication of effort.

Prerelease Testing Mozilla.org performs a daily build, and runs a daily minimal "smoke test" on the build for several major platforms, in order to ensure the build is sufficiently stable to allow development work on it to proceed. If the build fails, "people get hassled until they fix the bits they broke." If the smoke test identifies bugs, they are posted daily so that developers are aware of any serious problems in the build.

Mozilla currently has six product area test teams that take responsibility for testing various parts or aspects of the product, such as standards compliance, the mail/news client, and internationalization. Netscape personnel are heavily represented among the test teams, but the teams also include mozilla.org personnel and many others. The test teams maintain test cases and test plans, as well as other materials such as guidelines for verifying bugs and troubleshooting guides.

Inspections Mozilla uses two stages of code inspections: module owners review a patch in the context of the module and a smaller designated group (referred to as *superreviewers*, who are technically highly accomplished)

review a patch for its interaction with the code base as a whole before it is checked in.

Managing Releases Mozilla runs a continuous build process (Tinderbox) that shows what parts of the code have issues for certain builds and under certain platforms. It highlights the changes and their authors. It also produces binaries nightly and issues "Milestones" approximately monthly. As Baker (2000) points out:

> [T]he Milestone releases involve more than Tinderbox. They involve project management decisions, usually a code freeze for a few days, a milestone branch, eliminating "stop-ship" bugs on the branch and a bit of polishing. The decision when a branch is ready to be released as a Milestone is a human one, not an automated Tinderbox process. These Milestone decisions are made by a designated group, known as "drivers@mozilla.org," with input from the community.

Quantitative Results

In this section, we report results that address the same six basic questions we answered with respect to Apache in the previous section. There are some differences between the projects that must be understood in order to compare Mozilla to Apache in ways that make sense.

First, Mozilla is a *much* bigger project. As shown in table 10.4, Apache had about 6,000 MRs, 18,000 delta, and 220,000 lines of code added. In contrast, Mozilla consists of 78 modules (according to the Mozilla Project at the time of this writing), some of which are much larger than the entire Apache project. The following analyses are based on seven of the Mozilla modules.

The Size of the Mozilla Development Community

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

By examining all change login and comment records in CVS, we found 486 people who contributed code and 412 who contributed code to PR fixes that were incorporated. Numbers of contributors to individual modules are presented in table 10.5.

Table 10.5 presents numbers of people who contributed code submissions, problem fixes, and who reported problems. Because some problem reports do not correspond to a module in cases when the fix was not created or committed, we provide numbers for people who reported problems resulting in a fix and estimate of the total number using the overall

Table 10.4
Sizes of Apache, five commercial projects, and seven Mozilla modules

	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
/layout	12.7	42	800	2.6	174
/js	4.6	14	308	2.6	127
/rdf	4.1	12	274	2	123
/netwerk	3.2	10	221	1.6	106
/editor	2.9	8	203	2	118
/intl	2	5	118	1.8	87
/xpinstall	1.9	5	113	1.7	102

Table 10.5
Population of contributors to seven Mozilla modules

	Number of people whose code submissions were included in the code base	Number of people whose fixes were added to code base	Number of people who reported bugs that resulted in code changes	Number of people who reported problems (estimated)
/layout	174	129	623	3035
/js	127	51	147	716
/rdf	123	79	196	955
/netwerk	106	74	252	1228
/editor	118	85	176	857
/intl	87	47	119	579
/xpinstall	102	64	141	687

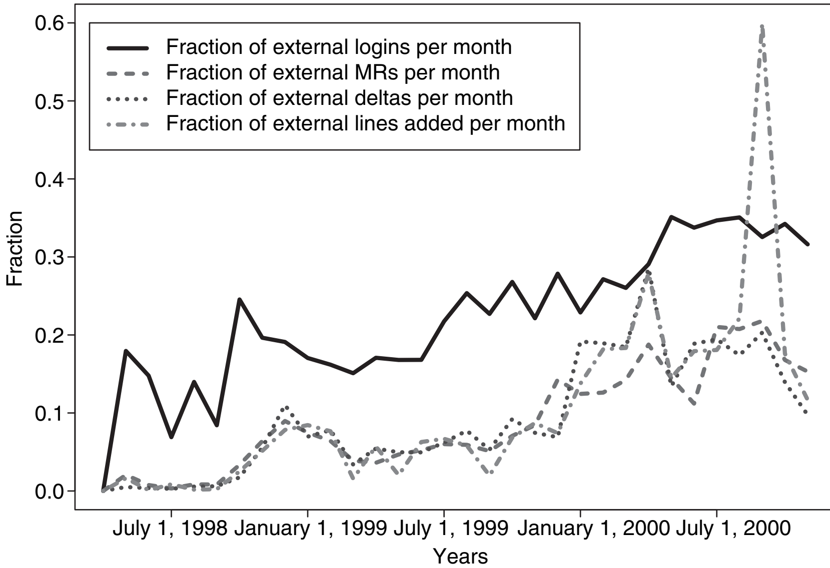

Figure 10.5
Trends of external participation in Mozilla project

ratio in Mozilla of the total number of people who reported PRs divided by the number of people who reported PRs that resulted in code changes. Based on the Bugzilla database, 6,837 people reported about 58,000 PRs, and 1,403 people reported 11,616 PRs that can be traced to changes to the code. To estimate the total number of people reporting PRs for a module (rightmost column), we multiplied the preceding column by 6,837/1,403.

External Participation Because Mozilla began as a commercial project and only later adopted an open source approach; in order to understand the impact of this change, it is essential to understand the scope and nature of external participation. To this end, we examined the extent and the impact of external participation in code contributions, fix contributions, and defect reporting.

Figure 10.5 plots external participation over time. The measures include the fraction of external developers and the fraction of MRs, delta, and number of added lines contributed monthly by external developers.

Figure 10.5 shows gradually increasing participation over time, leveling off in the second half of 2000. It is worth noting that outside participants tend, on average, to contribute fewer changes and less code relative to

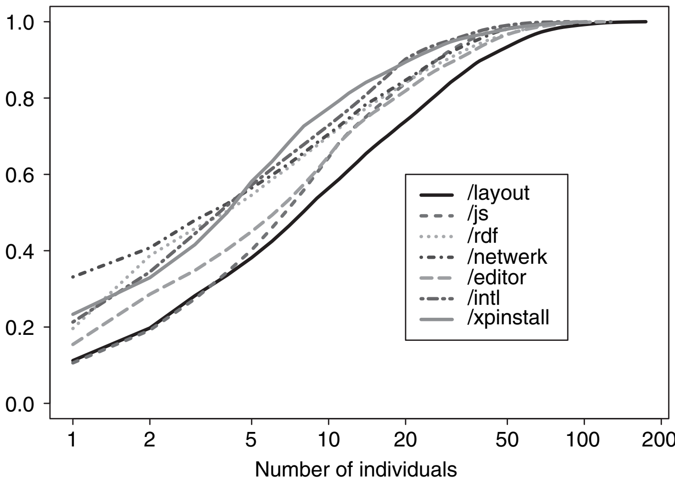

Figure 10.6
The cumulative distribution of contributions to the code base for five Mozilla modules

internal participants. It might reflect the part-time nature of the external participation.

Much larger external participation may be found in problem reporting. About 95 percent of the 6,873 people who created PRs were external, and they reported 53 percent of the 58,000 PRs.

Q3: Were these functions carried out by distinct groups of people; that is, 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?

Figure 10.6 shows cumulative distribution contributions (as for Apache in figure 10.1). The developer participation does not appear to vary as much as in the Apache project. In particular, Mozilla development had much larger core groups relative to the total number of participants. The participation curve for Mozilla is more similar to the curves of commercial projects presented in figure 10.3.

The problem reporting participation was very uniform in Apache, but contributions vary substantially in Mozilla, with 50 percent of PRs reported by just 113 people, with the top person reporting over 1,000 PRs (compared to Apache, where the top reporter submitted only 32 PRs). Forty-six of these 113 PR submitters did not contribute any code, and only 25 of the 113 were external. Unlike Apache, where testing was conducted almost

Table 10.6
Comparison of productivity of the "top" developers in selected Mozilla modules

Module	KMR/dev/year	KLOCA/dev/year	Size of core team
/layout	0.17	11	35
/js	0.13	16	24
/rdf	0.11	11	26
/netwerk	0.13	8.4	24
/editor	0.09	8	25
/intl	0.08	7	22
/xpinstall	0.07	6	22

exclusively by the larger community, and not the core developers, there is very substantial internal problem reporting in Mozilla, with a significant group of dedicated testers. Nevertheless, external participants also contribute substantially to problem reporting.

Given that most of the core developers work full-time on the project, we might expect the productivity figures to be similar to commercial projects (which, when measured in deltas or lines added, were considerably higher than for Apache). In fact, the productivity of Netscape developers does appear to be quite high, and even exceeds the productivity of the commercial projects that we consider (see table 10.6).

As before, we defined core or "top" developers in each module as groups of the most productive developers that contributed 83 percent of MRs and 88 percent of lines added. There was one person in the "core" teams of all seven selected modules and 38 developers in at least two "core" teams. Almost two-thirds (64 out of 102) of the developers were in only a single core team of the selected modules.

Although the productivity numbers might be different due to numerous differences between projects, the data certainly appear to suggest that productivity in this particular hybrid project is comparable to or better than the commercial projects we examined.

Code Ownership

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

For the Apache project, we noted that the process did not include any "official" code ownership; that is, there was no rule that required an owner to sign off in order to commit code to an owned file or module. We looked

at who actually committed code to various modules in order to try to determine whether a sort of de facto code ownership had arisen in which one person actually committed all or nearly all the code for a given module. As we reported, we did not find a clear ownership pattern.

In Mozilla, on the other hand, code ownership is enforced. According to Howard 2000 and the Mozilla Project, the module owner is responsible for fielding bug reports, enhancement requests, and patch submissions in order to facilitate good development. Also, before code is checked in, it must be reviewed by the appropriate module owner and possibly others. To manage check in privileges, Mozilla uses a Web-based tool called *despot*.

Because of this pattern of "enforced ownership," we did not believe that we would gain much by looking at who actually contributed code to which module, since those contributions all had to be reviewed and approved by the module owner. Where there is deliberate, planned code ownership, there seemed to be no purpose to seeing if de facto ownership had arisen.

Defects

Q5: What is the defect density of Mozilla code?

Because Mozilla has yet to have a nonbeta release, all PRs may be considered to be post-feature-test (i.e., prerelease). The defect density appears to be similar to, or even slightly lower than Apache (see table 10.7). The

Table 10.7
Comparison of post-feature-test defect density measures

Module	#PR/KDelta	#PR/KLOC added
Apache	40.8	2.6
C	164	5.7
D	196	6.0
E	256	6.9
/layout	51	2.8
/js	19	0.7
/rdf	27	1.4
/netwerk	42	3.1
/editor	44	2.5
/intl	20	1.6
/xpinstall	56	4.0

defect density, whether measured per delta or per thousand lines of code, is much smaller than the commercial projects, if one counts all defects found after the feature test. The highest defect density module has substantially lower defect density than any of the commercial projects, post-feature-test. Compared to the postrelease defect densities of the commercial products, on the other hand, Mozilla has much higher defect densities (see table 10.3).

Since the Mozilla project has yet to issue its first nonbeta release, we cannot assess postrelease defect density at the time of this writing. Although these Mozilla results are encouraging, they are difficult to interpret definitively. Without data on postrelease defects, it is difficult to know whether the post-feature-test densities are low because there really are relatively few defects in the code, or because the code has not been exercised thoroughly enough. As we reported earlier, though, more than 6,000 people have reported at least one problem with Mozilla, so we are inclined to believe that the low defect densities probably reflect relatively low defect code, rather than code that has not been exercised.

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?

Out of all 57,966 PRs entered in the Bugzilla database, 99 percent have a valid creation date and status change date; 85 percent of these have passed through the state RESOLVED and 46 percent of these have resolution FIXED, indicating that a fix was checked into the codebase; 83 percent FIXED bugs have passed through the state VERIFIED, indicating that inspectors agreed with the fix.

Figure 10.7 plots the cumulative distribution of the interval for all resolved PRs broken down by whether the PR resolution is FIXED, by priority, by the module, and by date (made before or after January 1, 2000). All four figures show that the median resolution interval is much longer than for Apache. We should note that half of the FIXED PRs had 43 percent or more of their resolution interval spent after the stage RESOLVED and before the stage VERIFIED. It means that mandatory inspection of changes in Mozilla almost doubles the PR resolution interval. But this increase does not completely account for the difference between Apache and Mozilla intervals; half of the observed Mozilla interval is still significantly longer than the Apache interval.

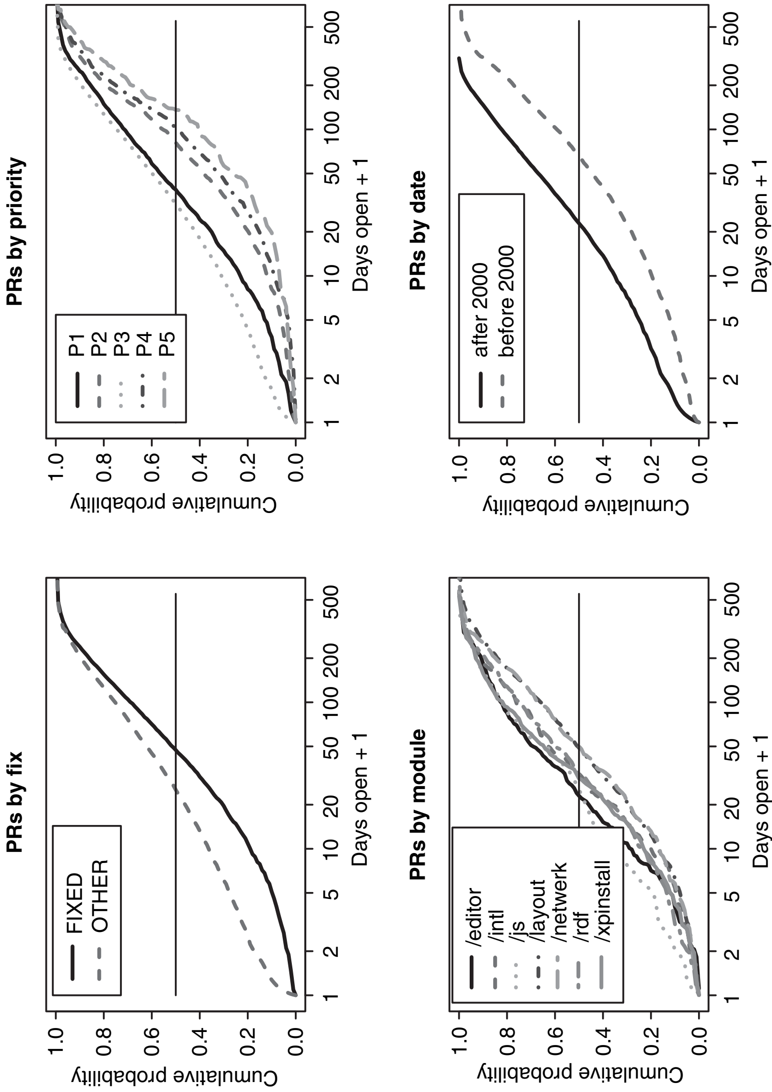

Figure 10.7
Problem resolution interval

Half of the PRs that result in fixes or changes are resolved in less than 30 days, and half of the PRs that do not result in fixes are resolved in less than 15 days. This roughly corresponds to the inspection overhead (inspections are only done for FIXED PRs).

There is a significant relationship between interval and priority. Half of the PRs with priority P1 and P3 are resolved in 30 days or less, and half of priority P2 PRs are resolved in 80 days or less, whereas the median interval of P4 and P5 PRs exceeds 100 days. The recorded priority of PRs did not matter in the Apache context, but the "priority" implicitly determined by affected functionality had an effect on the interval. These results appear to indicate that Mozilla participants were generally sensitive to PR priority, although it is not clear why priority P3 PRs were resolved so quickly.

There is substantial variation in the PR resolution interval by module. The PRs have a median interval of 20 days for */editor* and */js* modules and 50 days for */layout* and */netwerk* modules. This is in contrast to Apache, where modules could be grouped by the number of users they affect. Furthermore, */editor* affects fewer users than */layout* (2-D graphics), yet resolution of the latter's problems is slower, unlike in Apache, where the resolution time decreased when the number affected users increased.

The resolution interval decreases drastically between the two periods, possibly because of the increasing involvement of external developers or maturity of the project. We observed a similar effect in Apache.

Hypotheses Revisited

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

Hypothesis 2: For projects that are so large that 10 to 15 developers cannot write 80 percent 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.

These hypotheses are supported by the Mozilla data. The essential insight that led to these hypotheses is that when several people work on the same code, there are many potential dependencies among their work items. Managing these dependencies can be accomplished informally by small groups of people who know and trust each other, and communicate frequently enough so that each is generally aware of what the others are doing.

At some point—perhaps around an upper limit of 10 to 15 people—this method of coordinating the work becomes inadequate. There are too many people involved for each to be sufficiently aware of the others. The core groups for the various modules in Mozilla (with module size comparable to Apache in the range of 3 to 12 thousand delta per year and of duration longer than one year) range from 22 to 36 people and so are clearly larger than we contemplated in these hypotheses. And, much as we predicted, a form of code ownership was adopted by the various Mozilla teams.

There are at least two ways, though, that the Mozilla findings cause us to modify these hypotheses. Although the size of the project caused the creation of multiple separated project "teams" as we had anticipated (e.g., Chatzilla and other projects that contribute code to an */extensions* directory), we observe code ownership on a module-by-module basis, so that the code owner must approve any submission to the owned files. This process uses ownership to create a mechanism whereby a single individual has sufficient knowledge and responsibility to guard against conflicts within the owned part of the code. There is no "core" group as in the Apache sense, where everyone in the privileged group is permitted to commit code anywhere.

This leads to a further point that not only did the Mozilla group use ownership in ways we did not quite expect, they used other mechanisms to coordinate the work that are independent of ownership. Specifically, they had a more concretely defined process, and they had a much stricter policy regarding inspections. Both of these mechanisms serve also to maintain coordination among different work items. Based on these additional findings, we would rephrase Hypotheses 1 and 2 as follows:

Hypothesis 1a: Open source developments will have a core of developers who control the code base, and will create approximately 80 percent or more of the new functionality. If this core group uses only informal ad hoc means of coordinating their work, the group will be no larger than 10 to 15 people.

Hypothesis 2a: If a project is so large that more than 10 to 15 people are required to complete 80 percent of the code in the desired time frame, then other mechanisms, rather than just informal ad hoc arrangements, will be required to coordinate the work. These mechanisms may include one or more of the following: explicit development processes, individual or group code ownership, and required inspections.

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.

For the modules that we report on in Mozilla, we observed large differences between the size of core team (22 to 35), the size of the communities that submit bug fixes that are incorporated into the code (47 to 129) and that find and report bugs that are fixed (119 to 623), and the estimated total population of people that report defects (600 to 3,000). These differences are substantial and in the direction of the hypothesis, but are not as large as in Apache. In particular, the group that adds new functionality is larger than we would have expected. This is likely due to the hybrid nature of the project, where the core developers are operating in a more industrial mode, and have been assigned to work full-time on the project. Since Mozilla does not deviate radically from the prediction, and since the prediction was meant to apply only to pure open source projects, we don't believe that it requires modification at this time.

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.

We were not able to test this hypothesis with the Mozilla data, since it did in fact achieve large numbers of contributors.

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

The defect density of the Mozilla code was comparable to the Apache code; hence we may tentatively regard this hypothesis as supported. In Mozilla, there appears to be a sizable group of people who specialize in reporting defects—an activity corresponding to testing activity in commercial projects. Additionally, as we mentioned previously, Mozilla has a half-dozen test teams that maintain test cases, test plans, and the like. The project also uses a sophisticated problem-reporting tool, Bugzilla, that keeps track of top problems to speed problem reporting and reduce duplicate reports, and maintains continuous multiplatform builds. Inspections, testing, and better tools to support defect reporting apparently compensate for larger and more complex code. We must be very cautious in interpreting these results, because it is possible that large numbers of defects will be found when the product is released.

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

The reasoning behind this hypothesis was that low defect densities are achieved because developers are users of the software and hence have considerable domain expertise. This puts them at a substantial advantage relative to many commercial developers who vary greatly in their domain expertise. This certainly appears to be true in the Mozilla case. Although we did not have data on Mozilla use by Mozilla developers, it is wildly implausible to suggest that the developers were not experienced browser users, and thus "domain experts" in the sense of this hypothesis.

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

In the hybrid Mozilla case, response times are much longer than in the case of Apache. This may be due to the more commercial-like aspects of development; that is, the need to inspect, to submit the code through the owner, and so on. It also uses a 30-day release (milestone) cycle that more closely resembles commercial processes than the somewhat more rapid Apache process. Furthermore, the Mozilla product is still in the beta stage, and that might partly explain slower response times. Hence, it is not clear that the Mozilla data bear on this hypothesis, as long as it is taken to apply only to OSS, not to hybrid projects.

It should be noted that rapid responses to customer problems together with low defect density may significantly increase the availability of OSS software by minimizing the number and shortening the duration of downtime of customer's systems.

Conclusion: Hybrid Hypotheses

As we pointed out in the introduction, there are many ways in which elements of commercial and open source processes could be combined, and Mozilla represents only a single point in that space. The essential differences have to do with coordination, selection, and assignment of the work.

Commercial development typically uses a number of coordination mechanisms to fit the work of each individual into the project as a whole (see for example Grinter, Herbsleb, and Perry 1999 and Herbsleb and Grinter 1999). Explicit mechanisms include such things as interface specifications, processes, plans, staffing profiles, and reviews. Implicit mechanisms include knowledge of who has expertise in what area, customs, and habits regarding how things are done. In addition, of course, it is possible to substitute communication for these mechanisms. So, for example, two

people could develop interacting modules with no interface specification, merely by staying in constant communication with each other. The "communication-only" approach does not scale, of course, as size and complexity quickly overwhelm communication channels. It is always necessary, though, as the default means of overcoming coordination problems, as a way to recover if unexpected events break down the existing coordination mechanisms, and to handle details that need to be worked out in real time.

Apache adopts an approach to coordination that seems to work extremely well for a small project. The server itself is kept small. Any functionality beyond the basic server is added by means of various ancillary projects that interact with Apache only through Apache's well-defined interface. That interface serves to coordinate the efforts of the Apache developers with anyone building external functionality, and does so with minimal ongoing effort by the Apache core group. In fact, control over the interface is asymmetric, in that the external projects must generally be designed to what Apache provides. The coordination concerns of Apache are thus sharply limited by the stable asymmetrically controlled interface.

The coordination necessary *within* this sphere is such that it can be successfully handled by a small core team using primarily implicit mechanisms; for example, a knowledge of who has expertise in what area, and general communication about what is going on and who is doing what, when. When such mechanisms are sufficient to prevent coordination breakdowns, they are extremely efficient. Many people can contribute code simultaneously, and there is no waiting for approvals, permission, and so forth, from a single individual. The core people just do what needs to be done. The Apache results show the benefits in speed, productivity, and quality.

The benefit of the larger open source community for Apache is primarily in those areas where coordination is much less of an issue. Bug fixes occasionally become entangled in interdependencies; however, most of the effort in bug fixing is generally in tracking down the source of the problem. Investigation, of course, cannot cause coordination problems. The tasks of finding and reporting bugs are completely free of interdependencies, in the sense that they do not involve changing the code.

The Mozilla approach has some, but not all, of the Apache-style OSS benefits. The open source community has taken over a significant portion of the bug finding and fixing, as in Apache, helping with these low-interdependency tasks. However, the Mozilla modules are not as indepen-

dent from one another as the Apache server is from its ancillary projects. Because of the interdependence among modules, considerable effort (i.e., inspections) needs to be spent in order to ensure that the interdependencies do not cause problems. In addition, the modules are too large for a team of 10 to 15 to do 80 percent of the work in the desired time. Therefore, the relatively free-wheeling Apache style of communication and implicit coordination is likely not feasible. The larger Mozilla core teams must have more formal means of coordinating their work, which in their case means a single module owner who must approve all changes to the module. These characteristics produce high productivity and low defect density, much like Apache, but at relatively long development intervals.

The relatively high level of module interdependence may be a result of many factors. For example, the commercial legacy distinguishes Mozilla from Apache and many other purely open source projects. One might speculate that in commercial development, feature content is driven by market demands, and for many applications (such as browsers) the market generates great pressure for feature richness. When combined with extreme schedule pressure, it is not unreasonable to expect that the code complexity will be high and that modularity may suffer. This sort of legacy may well contribute to the difficulty of coordinating Mozilla and other commercial-legacy hybrid projects.

It may be possible to avoid this problem under various circumstances, such as:

- New hybrid projects that are set up like OSS projects, with small teams owning well-separated modules
- Projects with OSS legacy code
- Projects with a commercial legacy, but where modules are parsed in a way that minimizes module-spanning changes (see Mockus and Weiss 2001 for a technique that accomplishes this)

Given this discussion, one might speculate that overall, in OSS projects, low postrelease defect density and high productivity stem from effective use of the open source community for the low-interdependence bug finding and fixing tasks. Mozilla's apparent ability to achieve defect density levels like Apache's argues that even when an open source effort maintains much of the machinery of commercial development (including elements of planning, documenting the process and the product, explicit code ownership, inspections, and testing), there is substantial potential benefit. In

particular, defect density and productivity both seem to benefit from recruiting an open source community of testers and bug fixers. Speed, on the other hand, seems to require highly modularized software and small highly capable core teams and the informal style of coordination this permits.

Interestingly, the particular way that the core team in Apache (and, we assume, many other OSS projects) is formed might be another of the keys to their success. Core members must be persistent and very capable to achieve core status. They are also free, while they are earning their core status, to work on any task they choose. Presumably they will try to choose something that is both badly needed and where they have some specific interest. While working in this area, they must demonstrate a high level of capability, and they must also convince the existing core team that they would make a responsible, productive colleague. This setup is in contrast to that of most commercial development, where assignments are given out that may or may not correspond to a developer's interests or perceptions of what is needed.

We believe that for some kinds of software—in particular, those where developers are also highly knowledgeable users—it would be worth experimenting, in a commercial environment, with OSS-style "open" work assignments. This approach implicitly allows new features to be chosen by the developers/users rather than a marketing or product management organization.

We expect that time and future research will further test our hypotheses and will demonstrate new approaches that would elegantly combine the best technologies from all types of software development environments. Eventually, we expect such work to blur distinctions between the commercial and OSS processes reported in this article.

Appendix

Table 10.8
Comparison of Apache and Mozilla processes

	Apache	Mozilla
Scope	The Apache project we examined includes only the Apache server.	The Mozilla project includes the browser, as well as a number of development tools and a toolkit. Some of these projects are as large or larger than the Apache server.
Roles and responsibilities	The Apache Group (AG) currently has about 25 members, all of whom are volunteers. They can commit code anywhere in the server. The core development group includes the currently active AG members as well as others who are very active and under consideration for membership in AG.	Mozilla.org has 12 members, who are assigned to this work full-time. Several spend considerable time coding, but most play support and coordination roles. Many others have substantial responsibility—for example, as owners of the approximately 78 modules, and leaders of the 6 test teams. Many of the non-mozilla.org participants are also paid to spend time on Mozilla development.
Identifying work to be done	Since only the AG has commit access to the code, they control all changes. The process is an open one, however, in the sense that others can propose fixes and changes, comment on proposed changes, and advocate them to the AG.	Anyone can submit a problem report or request an enhancement, but mozilla.org controls the direction of the project. Much of this authority is delegated to module owners and test teams, but mozilla.org reserves the right to determine module ownership and to resolve conflicts.

Table 10.8
(continued)

	Apache	Mozilla
Assigning and performing development work	Anyone can submit patches, choosing to work on his or her own enhancements or fixes, or responding to the developer mailing list, news group, or BUGDB. Core developers have "unofficial" areas of expertise where they tend to do much of the work. Other core developers tend to defer to experts in each area.	Developers make heavy use of the Bugzilla change management tool to find problems or enhancements on which to work. They are asked to mark changes they choose to work on in order to avoid duplication of effort. Developers can use Bugzilla to request help on a particular change, and to submit their code.
Prerelease testing	Developers perform something like commercial unit and feature testing on a local copy.	Minimal "smoke screen" tests are performed on daily builds. There are six test teams assigned to parts of the product. They maintain test cases, guidelines, training materials, and so on, on the mozilla.org Web site.
Inspections	All AG members generally review all changes. They are also distributed to the entire, development community, who also frequently submit comments. In general, inspections are done before commits on stable releases, and after commits on development releases.	All changes undergo two stages of inspections, one at the module level, and one by a member of the highly qualified "super reviewer" group. Module owners must approve all changes in their modules.
Managing releases	The job of release manager rotates through experienced members of AG. Critical problems are identified; access to code is restricted. When the release manager determines that critical problems are resolved and code is stable, the code is released.	Mozilla has daily builds and "Milestone" releases approximately monthly. The code is frozen for a few days prior to a Milestone releases; critical problems are resolved. A designated group at mozilla.org is responsible for Milestone decisions.

Acknowledgments

We thank Mitchell Baker for reviewing the Mozilla process description and Manoj Kasichainula for reviewing the Apache process description. We also thank all the reviewers for their insightful comments.

This work was done while A. Mockus and J. D. Herbsleb were members of software Production Research Department at Lucent Technologies' Bell Laboratories. This article is a significant extension to the authors' paper "A case study of open source software development: the Apache server" that appeared in the Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, June 2000 (ICSE 2000), pp. 263–272.

Notes

1. Please see Ang and Eich 2000; Baker 2000; Eich 2001; Hecker 1999; Howard 2000; Mozilla Project; Oeschger and Boswell 2000; Paquin and Tabb 1998; Williams 2000; Yeh 1999; and Zawinski 1999.

2. Ang and Eich 2000; Baker 2000; Eich 2001; Hecker 1999; Howard 2000; Mozilla Project; Oeschger and Boswell 2000; Paquin and Tabb 1998; Williams 2000; Yeh 1999; and Zawinski 1999.