CONTINUOUS INTEGRATION AND QUALITY ASSURANCE: A CASE STUDY OF TWO OPEN SOURCE PROJECTS

Jesper Holck
Copenhagen Business School
Department of Informatics
Howitzvej 60
DK-2000 Frederiksberg
Denmark
jeh.inf@cbs.dk

Niels Jørgensen
Roskilde University
Computer Science
Universitetsvej 1
DK-4000 Roskilde
Denmark
nielsj@ruc.dk

ABSTRACT

A decentralized variant of continuous integration can be defined in terms of two fundamental rules: (1) Developers' access to add contributions to the development version at any time, and (2) developers' obligation to integrate their own contributions properly. Decentralized, continuous integration may adapt well to organizations where developers work relatively independently, as in many open source projects. The approach raises the issue of how these organizations can exercise central control, as attaining the benefits of continuous integration requires that contributions are useful and satisfy the project's definition of successful integration. We have investigated the use of continuous integration in FreeBSD and Mozilla. Our account of quality assurance activities in the two open source projects distinguishes between Mintzberg's three complementary forms of central control: Standardization and control of work output, work processes, and worker skills. Our study indicates that two major challenges face projects using decentralized, continuous integration: (1) To balance the access to add contributions against the need to stabilize and mature the software prior to a release, and (2) to consider the developers' limited time and resources when interpreting their obligation to integrate their changes properly.

INTRODUCTION

In the term 'continuous integration', *integration* refers to assembly of software parts and *continuous* to the absence of time-constraints. Several development methods label certain activities as continuous integration, including Unified Process (Jacobson, Booch, & Rumbaugh, 1999), and eXtreme Programming, where it is one of 12 recommended best practices (Beck, 1999). Continuous integration may supplement a phased approach, where 'continuous' refers to the way integration takes place during a phase of integration and tests; or it may be part of iterative methods with (as in XP) or without (as in FreeBSD and Mozilla) processes prescribing that modules and their interfaces are designed and documented prior to implementation.

By 'integration' we understand more specifically all activities required to assemble a complete system of software from its parts (Herbsleb & Grinter, 1999), and by 'continuous' we understand that developers integrate frequently and throughout all phases in the development life cycle. In this broader sense, continuous integration is used in open source as well as in commercial projects. The definition comprises both relatively controlled processes, as used in Microsoft (Cusumano & Selby, 1997) with a daily build cycle and central management of the build process (most notably of correcting a broken build), and less structured processes as used in Mozilla and FreeBSD. The latter, more decentralized approach has the following two characteristics: First, the developers' access to add software contributions to the development branch at any time, and second, the developers' obligation to integrate their own contributions properly. This approach is decentralized in the sense that both the decision of when to integrate and the responsibility for a successful integration are delegated to the individual. There is no sharp distinction between the centralized and decentralized approaches, and one of the projects in our study, Mozilla, can be characterized as borderline due to its semi-structured development process.

According to studies of commercial projects using daily builds and thus some form of continuous integration, advantages of continuous integration include reduced integration risks (errors are found early) and motivation (you see a working system) (Cusumano & Selby, 1997; Ebert, Parro, Suttels, & Kolarczyk, 2001; Olsson & Karlsson, 1999). Continuous integration is also an alternative to 'big bang' integration, where all modules are combined in one go, and which usually results in large numbers of errors, hard to isolate and correct owing to the vast expanse of the program (Pressman, 1992).

Disadvantages of continuous integration may include degeneration of architecture due to lack of focus on overall design and time spent on too frequent releases of too poor quality (Olsson & Karlsson, 1999).

Continuous integration, especially in a decentralized variant, may be of particular interest in open source projects due to the difficulty associated with imposing structured approaches in such projects and their participants. Some of the prevalent characteristics of open source projects are:

Few specifications: Many of the documents traditionally regarded as essential for coordination seem to be missing in most open source projects, including plans and schedules (Mockus, Fielding, & Herbsleb, 2002).

Geographical distribution: A study of Linux contributors showed that they "come from a truly worldwide community spanning many organizations" (Dempsey, Weiss, Jones, & Greenberg, 2002), and a recent on-line survey of 2,784 open source participants (Ghosh, Glott, Krieger, & Robles, 2002) showed that "most of the developers feature networks that consist of rather few people." Studies have shown that integration is particularly difficult in geographically distributed projects (Herbsleb & Grinter, 1999).

Volunteers: Most developers contribute to the projects in their free time – an estimated 60% are not paid for their work (Hars & Ou, 2001; Jørgensen, 2001) – and consequently they are less likely to accept to be ordered around and perform tedious tasks.

Self-directed egoists: This characterization of open source developers (Raymond, 2001) may be crude, but it is not too distant from the results of empirical studies. According to Lakhani et al. (2002), the two most important motivations for participating in an open source project were "intellectually stimulating" and "improves skill". For only 20% of the developers, "work with team" was a key motivator.

However, to attain the benefits of continuous integration, it is crucial that the project imposes some form of control on the contributions flowing into the development version of the project's software. For example, the benefit of easier defect diagnosis, because build errors must be the result of very recent contributions (McConnell, 1996a), is unobtainable if the build is broken from the outset.

We use Mintzberg's work on organizational archetypes and different mechanisms of control (Mintzberg, 1979) to categorize the activities in FreeBSD and Mozilla, which we have found are related to quality assurance and continuous integration. As indicated above, in some ways open source projects lack structure and therefore might resemble Mintzberg's adhocracy archetype: a flat organization of specialists, forming small project groups from task to task, and supposedly the most appropriate organization for modern organizations that depend on their employees' creative work. However, we find it rather intriguing and illuminating to examine the *structured* aspects of the two open source projects. In support of this, we may add that there are strong elements of stability in the projects: throughout their lives they have used the same technological infrastructure, and the projects are primarily developing new versions of the same product (an operating system and a web browser suite, respectively). Therefore, we have organized our account of quality assurance and continuous integration in the two projects to follow Mintzberg's distinction between establishing central control via standardization of worker skills, work processes, and work output. Each of these forms of control is predominant in one of Mintzberg's archetypes: the professional bureaucracy, the machine bureaucracy, and the divisionalized organization.

The rest of the paper is organized as follows: First we describe the two projects in our case study and how we have made the survey. Subsequently, we examine the projects' quality assurance activities focusing on control of worker skills, work processes, and work output, respectively. Finally, in the last section we conclude and discuss our findings.

FreeBSD, Mozilla, and our survey

FreeBSD and Mozilla organize their source code repository (from now on: the *repository*) around a main branch (the *trunk*) into which most changes are inserted and one (Mozilla) or more (FreeBSD) additional branches hosting the projects' production releases. See table 1 for basic facts about the two projects.

Table 1. Basic facts on FreeBSD and Mozilla

Name	FreeBSD	Mozilla
Product	Operating system	Web browser suite
Major product qualities	Robustness, security	Independence, open interfaces, user interface
Major platforms	Intel x86, Alpha, SPARC, PC-98	Windows, Linux, MacOS
Approximate size of trunk	29,000 files, 11 million lines	40,000 files, 6 million lines
Activity on trunk in October 2002	118 persons *committed* 2,063 changes	107 persons *committed* 2,856 changes
50% of these commits made by	12 developers	7 developers
Project management	9 person *Core Team*	11 person *Mozilla.org Staff*

The repositories are stored at central locations and can be reached via the Internet (www.freebsd.org and www.mozilla.org). Developers contribute to the projects by continually and in parallel updating (changing, adding, and deleting) repository files; a process controlled by means of CVS augmented with extra tools. Each file change (*commit*) creates a new version; all old file versions remain accessible, and thus it is always possible to go back to an older version of a file if problems or errors occur.

The software product that results from *building* with the newest versions of all files is called the *development version*; because files are updated continuously, the development version will also be constantly evolving. The aggregation of all files related to the development version is called the *trunk.*

Because both projects maintain production releases and therefore have to isolate new development from maintenance of previous releases, there is a need to create branches in the repository. At a certain time, a branch is created as a (logical) copy of the most recent file versions from the trunk; subsequently, changes to the trunk will not directly influence the branch and vice versa, and developers must specify which branch they want to use when downloading or committing source files.

In addition to the various development and production versions, both projects' software is also adapted to a wide range of different platforms (the table only shows the most important). As a consequence, developers not only need to differentiate between releases (residing on different branches), they must also be able to commit changes intended for specific platforms, which is accomplished by using conditional compilation rather than branching.

Both projects employ several staff and management functions, including top-level management boards (*Mozilla.org staff* and FreeBSD *Core Team*). But most of the development work in both projects takes place in one-man projects, where developers contribute new or revised source code, largely working by themselves (Jørgensen, 2001).

We have pulled statistical data from FreeBSD's and Mozilla's repositories in October 2002, studied the projects' public mailing lists, and drawn on a survey of FreeBSD committers performed in November 2000 and also the basis for (Jørgensen, 2001). The survey obtained 72 replies, corresponding to just over 35% of the project's committers at that time. The developers were asked 29 multiple-choice questions and also asked for further comments. Finally we have conducted e-mail-based interviews with selected developers from FreeBSD and Mozilla.

CONTROLLING QUALITY THROUGH SKILLS: PROMOTION BASED ON MERITS

Mozilla and FreeBSD carefully control which developers are allowed to commit changes to their software. To get commit privileges, in both projects you must first demonstrate your competence, typically by making high-quality contributions over a period of time.

The accreditation procedures for new committers are related to what Mintzberg designates 'standardization of skills', the key coordination mechanism in professional bureaucracies, often illustrated by examples like hospitals and universities (Mintzberg, 1979). In comparison with these, the standardizations of skills in FreeBSD and Mozilla are rather simple. Essentially, they only consist of filtering the admitted developers into the single category of 'professionals': the committers.

Promotion of external contributors to committers

Mozilla has a formal bureaucratic procedure for accrediting committers involving a formal application, acceptance from a *voucher* (person who already has commit privileges), and a written acceptance from three of the so-called 'super reviewers' (*Getting CVS Write Access to Mozilla*, 2003). In FreeBSD, you acquire the right to commit to the source code repository by the Core Team based on a request from one or more committers, but the procedure does not seem to be as formal as in Mozilla. By only granting commit-privileges to developers that have demonstrated programming as well as interpersonal skills and interest in the project, FreeBSD and Mozilla reduce the risks of low-quality contributions. The trial period, which contributors must go through before they become committers, can be seen as an apprenticeship where the persons gradually learn and adapt to the projects' procedures, rituals, and culture. The contributors will only be given commit privileges if the adaptation appears successful:

> This process serves multiple purposes; after all, the FreeBSD community is made up of people who do the work. For committers, the work consists of creating useful and correct patches. If you don't consistently and regularly create good patches, there's no point in giving you commit access, now is there? … By the time you've submitted several dozen PRs [problem reports], you'll either work well with the FreeBSD team or everyone will understand that you and the team just can't get along *(Lucas, 2002).*

Because developers without commit privileges have to find committers that will approve their contributions and perform the actual changes to the repository, the committers have a strong motivation for delegating commit privileges to qualified developers:

> To a committer, taking patches from PRs is a trivial annoyance. Contributions are certainly appreciated, but they must be read, evaluated, and tested. … If you submit enough useful and correct PRs, eventually some committer will get sick of taking care of your work and will ask you if you want to be able to commit them yourself *(Lucas, 2002).*

This pattern of developers, gradually becoming more involved in FreeBSD or Mozilla, bears resemblance to professional communities evolving by gradually letting peripheral participants become fully qualified members, as described by Lave and Wenger (1991) in their study of what they term legitimate peripheral participation; an analogy also noted by Nakakoji et al. (2002). According to a FreeBSD developer, the accreditation procedures seem to work very well:

> By and large, most of the committers are better programmers than the people I interview and hire in Silicon Valley *(FreeBSD developer)*.

Controlling the committers

A critical concern in the projects is to ensure that developers with commit privileges also commit good source code contributions. Both projects reserve the right to delete a change from the trunk and to revoke a developer's commit privileges completely. Removal of a change is technically a simple task provided that the change is independent of all subsequently committed changes, which is usually the case. In FreeBSD, there is a well-defined process for deleting a change already committed:

> Any disputed change must be backed out […] if requested by a maintainer. […] This may be hard to swallow in times of conflict […] If the change turns out to be the best after all, it can easily be brought back *(*The FreeBSD Committers' Big List of Rules*)*.

Moreover, a consensus between committers working in a certain area can be overridden for security reasons:

> Security related changes may override a maintainer's wishes at the Security Officer's discretion *(*The FreeBSD Committers' Big List of Rules*)*.

The ultimate sanction is to revoke the commit privileges, and FreeBSD's internal rules specify a procedure for doing this temporarily or permanently (*The FreeBSD Committers' Big List of Rules*). Under normal circumstances, this requires three core team members to act in unison, and subsequently the core team is required to participate in a public hearing about the matter if requested by the developer in question. It is our estimate that the ultimate sanction, analogous to firing a hired developer, is used less than once a year and that it serves as a means to resolve collaborative issues rather than to maintain a high level of coding skills.

CONTROLLING QUALITY THROUGH PROCESS: LAISSEZ FAIRE

Prior to the point where their new source code contribution is integrated into the development version, committers in both FreeBSD and Mozilla are free to choose their own approach to develop the change. On one hand, this freedom is well suited to the voluntary nature of most of the development work. On the other, the lack of systematic approaches, e.g. thorough analysis and design activities, may increase previously identified important risks associated with the daily build and continuous integration (Olsson & Karlsson, 1999): excessively proactive development (developers failing to think before they act), architectural degeneration, and 'quick and dirty' changes.

In general, there is a relatively small amount of control via standardization of work processes in FreeBSD and Mozilla. The Mintzberg archetype for organizations relying on standardization of work as a key coordinating mechanism is the machine bureaucracy, and example organizations include McDonald's and automobile manufacturers (Mintzberg, 1979). This section describes the process rules and guidelines that apply to the developer's pre-integration activities, including the requirements for public announcement, discussion, and review. The section also discusses the pre-integration activities for which there are no such rules or guidelines, in particular design.

Work assignment

The life cycle of a change begins when a developer decides to start working on a task. FreeBSD and Mozilla's rules for work assignment are extremely liberal: in general, committers are free to pick any task they wish. The projects encourage developers to pick tasks that appeal to them:

> Look through the open PRs, and see if anything there takes your interest *(Hubbard, 2003).*

This is in line with Raymond's thesis about open source projects relying on developers inclination to "scratch their personal itch" (Raymond, 2001).

The use of continuous integration is an important condition that makes it relatively easy for developers to define and choose on which task they want to work. As the decision when to integrate is delegated to the individual committer, there can be no specific order in which the various contributions are supposed to be integrated in the development version. Thus, developers are largely free to choose, implement and commit any contribution to the repository with only a limited effort to coordinate with other developers.

This independence is an obvious advantage in an open source project: From the point of view of the project as a whole, it reduces the need for a plan in which tasks are defined, allocated, and scheduled; a plan that would require a considerable effort to produce and maintain. From the developer's point of view, the freedom to choose a task and integrate a change quickly may be highly motivating:

> ... there is a tremendous sense of satisfaction to the 'see bug, fix bug, see bug fix get incorporated so that the fix helps others' cycle *(FreeBSD developer)*.

This may also apply to developers who are paid by a company:

> I use FreeBSD at work. It is annoying to take a FreeBSD release and then apply local changes every time. When […] my changes […] are in the main release […] I can install a standard FreeBSD release […] at work and use it right away *(FreeBSD developer)*.

The developers' free choice of task assignments are, however, balanced by various recommendations and rules. To see who is working on what, Mozilla recommends that if the task, on which a developer chooses to work, has not already been reported as a bug, the developer should do this first:

> Enter the task you're planning to work on as enhancement requests and Bugzilla will help you track them and allow others to see what you plan to work on *(*bugs*, 2003)*.

FreeBSD has no similar formal requirement that developers should announce their current task.

There are at least two exceptions to developers' free choice of task assignments. The first exception is soft in the sense that it is a recommendation rather than a rule: the encouragement to work on important bugs. The general call in Mozilla is to "stay focused on the most important problems [i.e. bugs]" (*The Seamonkey Engineering Bible*, 2003), especially stressing the issue before production releases. If in doubt, the developer is recommended to choose to work on one of the bugs reported in the bug-tracking system, and inform possible stakeholders of his or her plan:

> Start with the PRs that have not been assigned to anyone else. If a PR is assigned to someone else, but it looks like something you can handle, email the person it is assigned to and ask if you can work on it – they might already have a patch ready to be tested, or further ideas that you can discuss with them *(Hubbard, 2003)*.

The second exception is an obligation to consult the person responsible for a given code area. Both projects have a notion of code ownership in the sense that most files have a *maintainer* (FreeBSD) or *module owner* (Mozilla), often responsible for entire directories or applications:

> The maintainer owns and is responsible for that code. This means that he is responsible for fixing bugs and answering problems reports […] *(Kamp, 1996)*.

Code ownership is a mechanism for coordination via a consensus process:

> [A commit should happen] only once something resembling consensus has been reached *(*The FreeBSD Committers' Big List of Rules*)*.

The requirements to announce and discuss one's (intended) choice of work tasks publicly help to mitigate obvious risks:

- The risk of doing duplicate (and hence wasted) work because different developers unknowingly might be working in parallel on the same problem. Even though developer resources are gratis in open source projects, they are not unlimited and should be utilized efficiently.

- The risk of spending time on changes that will not be considered improvements by the community. This is a waste of the developer's resources and may result in extra work for others if the changes need to be backed out of the repository.
- The risk of integration problems because two or more developers want to make incompatible changes to the same module.

Work breakdown

Work is broken down into small tasks that are not too difficult to integrate. Neither project has any rule pertaining to the granularity of work breakdown; rather working with small changes is a consequence of the obligation to integrate one's own contributions. An interesting consequence of this approach is the difficulties associated with developing large, new features that are not easily broken down into independent pieces.
An example of a very large task is the recent work in FreeBSD on Symmetric Multi-Processing (SMP) enabling the operating system to utilize multiple processors. The SMP effort was organized as a subproject with its own project manager. The subproject considered encapsulating its work on a separate branch, but rejected this in fear of 'big bang' integration problems as experienced in another open source operating system project, BSD/OS:

> … they [BSD/OS] went the route of doing the SMP development on a branch, and the divergence between the trunk and the branch quickly became unmanageable. […] We are completely standing the kernel on its head, and the amount of code changes is the largest of any FreeBSD kernel project taken on thus far. To have done this much development on a branch would have been infeasible. *(FreeBSD SMP project manager, 2000).*

So the SMP subproject chose to add their radical kernel changes incrementally, but the problem of preserving the development version in a working state was a huge challenge and seen as a heavy burden – the task can be compared to transforming a van into a sports car while driving. Changes were added incrementally over several years, but other developers' work was seriously affected, especially in the autumn of 2000 when the SMP work severely 'destabilized' the trunk causing build failures and other errors due to dependencies with other, concurrent work on the trunk.
The obligation to preserve the development in a working state could be seen as implying an implicit rule saying: avoid the introduction of large and complex new features. However, it should be noted that concurrent development on an operating system kernel is inherently difficult, so this implicit rule may apply to other approaches to integration as well.

Design

Neither FreeBSD nor Mozilla requires design to precede coding in the sense of writing, discussing, or approving design documents prior to coding. Indeed, in practice there is typically no design document for the individual change: 31 of the 72 committers surveyed in FreeBSD responded that they had never distributed a design document (defined as a separate document, distinct from a source file). Some documentation of the design of the systems' basic architecture is accessible, though.
However, the lack of an established practice to use design documents as a coordination mechanism should be viewed in the context of the projects as largely maintenance-oriented. FreeBSD has inherited a largely unaltered, basic architectural design from its predecessors, the first versions of which were developed in the late 1970s (*About FreeBSD's Technological Advances*). Mozilla, being a much younger project with roots that only go back to the mid-90s, is more in need of providing its own design documentation. For example, the project has developed its own component model (XPCOM) and uses a software layer originally developed by Netscape, which provides a platform-independent interface to multithreading (NSPR). These and other complex, project-specific parts of Mozilla are described at an introductory level (*Hacking Mozilla*) and in more detail (*Core Architecture*) in a series of publicly available documents.

Review

There seems to be no requirement or tradition for design reviews, but both projects require review of source code changes prior to commit. If the developer is a committer, the review may be the only occasion where others are involved in approving the developer's work prior to commit.

FreeBSD's Committer's Guide rule no. 2 states "discuss any significant change before committing" (*The FreeBSD Committers' Big List of Rules*), and 86% of the committers surveyed said that they actually received feedback on their latest change when submitting it to review. Mozilla has detailed rules requiring all changes to be reviewed by another committer, and in most cases to be 'super-reviewed' as well. The 'super-review' is done by one or more of a designated group of strong hackers and examines the quality of the code itself, its potential effects on other areas of the tree, its use of interfaces, and its adherence to Mozilla coding guidelines (Eich & Baker, 2003). To enforce reviews, Mozilla requires that a committer always should state the names of the contribution's reviewers when adding source code to the repository.

CONTROLLING QUALITY THROUGH WORK OUTPUT: DON'T BREAK THE BUILD

The major event in the life cycle of a change in FreeBSD and Mozilla is the commit of the change to the central repository. Prior to the commit, preliminary versions of the change have resided in the developer's private repository, most likely on his or her own computer. The commit is the delivery of the developer's work output to the project as a whole, and defines the point in time where it must meet the project's standards.

Mintzberg's archetype for organizations relying on control of work output as a key coordinating mechanism is the divisionalized organization exemplified e.g. by large multinational companies with relatively autonomous divisions, responsible for their own products (Mintzberg, 1979). Control of work output in FreeBSD and Mozilla of software changes produced by their 'divisions', the individual developers, is merely qualitative rather than quantitative as in companies that also attempt to control the productivity of divisions.

First, we discuss the standard defined by the projects and how compliance may be verified, and then we discuss how the projects modify the two basic rules of continuous integration: the developer's build-obligation and the developer's commit access.

The standard that contributions must satisfy

Both in FreeBSD and Mozilla it is emphasized that changes committed must keep the build working and comply with the projects' coding guidelines. Mozilla requires that developers run a number of simple tests before committing (Duddi, 1999), and is very clear about the requirement not to break the build:

> Breaking (run time, compile time, or link time) the tree is not ok. It costs lots of money (more than you can justify wasting) to have hundreds of engineers sitting idle waiting for a good tree to pull *(*Working with the Seamonkey Tree*, 2002).*

In FreeBSD, the requirements are given as part of the explanation to rule no. 10, "Test your changes before committing them":

> If your changes are to the kernel, make sure you can still compile [the kernel]. If your changes are anywhere else, make sure you can still [compile everything but the kernel] *(*The FreeBSD Committers' Big List of Rules*).*

Extensive coding guidelines exist in both projects. Examples from Mozilla include a code style guide (*Mozilla Coding Style Guide*, 2003) and a portability guide (Williams, Collins, & Blizzard, 2003). FreeBSD provides a "Kernel source file style guide", code guidelines to facilitate software internationalization and a security guide with preventive rules such as:

> Never trust any source of input [...] never use gets() or sprintf(), period *(*FreeBSD Security Guide*, 1997).*

As far as we understand, the projects do not have other requirements that pertain to code contributions. For example, there is no requirement such as Extreme Programming's demand for pair programmers to write test programs before the actual coding is performed (Beck, 1999). (As an aside, we will mention that both projects include subprojects, for example FreeBSD's documentation project, producing other kinds of deliverables. However, the activities of these subprojects are outside the scope of this paper).

Verification

The build process is fully automated, and therefore the verification that contributions meet the build-requirement is straightforward. In FreeBSD, there is an automated routine for building the trunk twice a day on the major processor architectures, the so-called Tinderbox-builds, the result of which are shown on a webpage (http://www.rtp.freebsd.org/~des).
In Mozilla, there is a well-defined process for a daily verification procedure using a cluster of build machines (representing all targeted platforms). At 8 AM (PST) each working day, the build machines download the newest source code, build it, and execute a small number of regression tests.
However, in both projects most build errors will be detected by currently active developers, reporting the problem to one or more mailing lists, even before the Tinderbox builds (in FreeBSD) or the daily build verification (in Mozilla). Broken builds have immediate consequences for the active developers, because neither project operates with a so-called 'holding area'. In order to preserve the development version in a sound state allowing developers to rely on it for testing their own code, McConnell (1996b) recommends that projects using daily builds create a copy of the development version through which all changes must pass on their way to the (proper) development version to filter away changes not properly tested. Neither project has any such filtering of the stream of changes flowing into the trunk.
Verification that code contributions comply with coding guidelines is facilitated by the visibility of contributions:

- The repositories are browsable, providing easy public access to all sources files.
- In both projects, an automatic mail message is sent to other developers immediately upon commit of a change.

This visibility encourages developers to strive to produce code that will be perceived to have high quality. In responding to the statement "Knowing that my contributions may be read by highly competent developers has encouraged me to improve my coding skill", 57% of the 72 FreeBSD committers surveyed said "yes, significantly", and 29 % "yes, somewhat". One committer added: "Embarrassment is a powerful thing."
The visibility of the code is in part due to the project being open source, but also to the approach of continuous integration: A large number of developers are working with the most recent version of the trunk, and monitor the changes made because their work depends on them.
Given, on the one hand, the projects' reliance on the quality of the committed changes, and on the other hand, the frequent occurrences of broken builds, it is remarkable that a committed change rarely is removed from the repository. When it happens, it is normally not due to a broken build, but to disagreement about whether the change, correctly implemented or not, is in fact an improvement of the software.

Balancing the don't break the build rule

It appears to us that the projects' reason essentially is as follows when faced with one of the frequent build-breaking changes: As long as the change, when corrected, will improve the trunk, we prefer to keep it in the trunk and correct is as fast as possible rather than exercise the right to delete the change and throw it back to the developer, possibly to an uncertain faith.

> I can remember one instance where I broke the build every 2-3 days for a period of time; that was necessary [due to the nature of the work]. That was tolerated – I didn't get a single complaint *(FreeBSD committer, 2000).*

In Mozilla, the daily build verification (see the previous section) is highly organized. Almost as crucial as preventing broken builds is the requirement to be available after a commit that (sic) does break the build. Developers that have committed changes since the previous day's verification are said to be 'on the hook':

> If you are on the hook, your top priority is to be available to the build team to fix bustages. […] You are findable. You are either at your desk, or pageable, checking e-mail constantly, or on IRC so that you can be found immediately and can respond to any problems in your code *(*Hacking Mozilla with Bonsai*).*

There are several good reasons for balancing the 'don't break the build' rule with other considerations. Correcting a broken build can be highly challenging, since the failure may be due to dependencies on files or modules outside the area of the developer's primary expertise. Moreover, some changes may be difficult to test in the first place.

Interpretation of the 'don't break the build' rule is particularly important with respect to the effort that a developer should invest to prevent broken builds on *any* platform. As mentioned, both FreeBSD and Mozilla are developed for many platforms, of which 4 and 3 are particularly prioritized. Due to platform differences, a build may succeed on one and fail on another, which we refer to as a partially broken build. However, most developers only have access to a single platform, and therefore it may be impracticable to perform trial builds on each prioritized platform before check-in. In practice, the Mozilla project in general accepts a large part of the responsibility of correcting partially broken builds. One of the reasons is the unattractive alternative of accepting a source code change on some platforms, but not on others (those that build with the change vs. those that don't).

As a general rule, the repository in Mozilla is closed during the daily verification build until it has been terminated successfully. This means that no commits (except as part of the corrective effort) are allowed until all three prioritized platforms pass the test. This may last from two to several hours. The reason for 'closing the tree' for all platforms is described as follows:

> During the development of Netscape Navigator and Netscape Communicator it was argued many times that […] we should care less about a particular set of platforms and fix regressions on these "second-class" platforms later. We tried this once. The reason why we don't have Netscape Communicator on Win16 was the result of putting off the recovery of that platform until later. After a couple of weeks recovery became impossible. […] The problems will stack up […] as the codebase moves forward and it never catches up *(Yeh, 1999).*

In part, the problem of broken builds on other platforms is solved by making a set of central build machines available, to which sources can be uploaded and subjected to a trial build prior to commit. However, this is tedious and is not enforced as a general rule in either project. As a middle road, Mozilla provides the previously mentioned portability guides with rules and recommendations for producing cross-platform software (e.g., Williams et al., 2003).

In principle, FreeBSD and Mozilla delegate to the developers the responsibility to integrate their own contributions, but in practice it is a major challenge to strike a reasonable balance and to some degree accept that developers from time to time break the build and thus disrupt other developers' work. Indeed, making an absolute requirement that committed changes should be error-free would be absurd and defy the purpose of using the trunk for community testing, as discussed below.

Balancing the access to contribute.

In both projects, a stabilization period is explicitly declared for several weeks prior to major production releases. This is an important exception from developers' access to commit changes and may create tension in the projects. We will discuss the process leading to major production releases (e.g., FreeBSD's 5.0 of January 2003, and Mozilla's 1.0 of June 2002). In addition, the projects also create minor production releases (FreeBSD 4.6, 4.7, etc.; Mozilla 1.1, 1.2, etc.).

The purpose of a stabilization period is to limit the changes allowed to be committed. During stabilization, only changes seen as necessary or useful for the purpose of stabilization are allowed, most notably bug-fixes. In FreeBSD, the stabilization period for 5.0 lasted for two months. The first month was less strict allowing new features on a case-by-case basis at the release engineering team's discretion. The second month was more strict and only allowed commits if they were bug-fixes.

When the software is considered to be sufficiently stable, it is declared a production release. This is possible because prior to stabilization the software is already in a working state, which makes the use of special integration and testing teams unnecessary. It is a major advantage of continuous integration if a brief period of stabilization is indeed sufficient for changing 'work in progress' to 'production release'. However, the required stabilization effort may be huge and seen as diverging resources from more important or interesting new development.

In Mozilla, the stabilization period prior to the 1.0 release can be seen as lasting more than 8 months, beginning with the 0.9.6 release (in November 2001) upon which the "the trunk is closed to all but a relative few bug fixes, and everyone is focused on testing" (Eich, 2002). There is indication of pressure from Mozilla developers to relax commit restrictions:

> […] we're not looking for new features; we want stability, performance […], tolerably few bugs […]. Features cost us time […] those implementing the features […] could instead help fix 1.0 bugs […]. If you think you must have a feature by 1.0, please be prepared to say why to drivers, and be prepared to hear "we can't support work on that feature until after 1.0 has branched" in reply *(Eich, 2002).*

To allow for the resumption of new development on the trunk, the final two months of stabilization for Mozilla's 1.0 release took place on a separate branch (created April 2002). The isolation of bug-fixing from new development is in some sense a departure from the strict adherence to continuous integration, and indicates the following dilemma associated with stabilization:

- Creating a separate stabilization branch allows for resumption of new development, which will otherwise be halted when destabilizing changes are prohibited on the trunk.
- However, a separate stabilization branch requires that bug-fixes to the trunk are also made to the branch (and vice versa); it doubles the tasks related to managing a branch (assigned to branch drivers in Mozilla), and divides the pool of user/developers between the branches:

  > This branch [Mozilla 1.0] obviously entails overhead in driving, merging, reviewing, and testing *(Eich, 2002).*

The dilemma makes it important and difficult to decide if and when to branch stabilization away from new development.

DISCUSSION AND CONCLUSION

Viewed from the perspective of quality assurance our case study of FreeBSD and Mozilla indicates that at a basic level the projects' approach to software integration actually works: in spite of their difficult-to-control and geographically distributed developers that pick tasks at their own convenience and do not write design documents, FreeBSD and Mozilla produce widely used software. It appears that the process of continuous integration, as used in the projects in a decentralized variant, to some degree replaces traditional software engineering coordination mechanisms like plans and design documents.

Incessant access to commit

The first key principle of the projects' decentralized approach to continuous integration is the committers' access to add contributions to the development version at any time. This access appears to be highly motivating: developers are free – only limited by the need to reach consensus with module owners – to choose which tasks they want to work on, and they can commit changes without awaiting approval. This feels unbureaucratic and the developers can see that the result of their work quickly becomes part of the project's software.

Using Mintzberg's terminology, the freedom to commit is absence of work process standardization. Commit access is granted to developers according to their previous merits, which can be seen as

standardization of worker skills. Once you are in, you are free to choose your own work process. However, this overall liberal work process regime is supplemented with various work process recommendations such as to work on prioritized bugs, and announce and discuss work.
A major challenge for the projects, related to the incessant access to commit, is the conflict between stabilization and new development: A prolonged period of stabilization on the trunk entails the cost of holding back new development, but if the stabilization period is too short, the release may be of poor quality. Alternatively, stabilization can be encapsulated on a separate branch, but this entails the cost of dividing developer resources for testing and managing between community testing and stabilization. Prior to their most recent major releases, both projects did in fact decide to branch away stabilization from new development. These decisions were made reluctantly, because of the concern that branches may diverge to a point where useful changes cannot easily be merged from one branch to another, e.g. when a bug-fix on the stabilization branch is difficult or impossible to perform on the development branch. This concern underlies the decision in FreeBSD to avoid developing the new SMP feature on a separate branch, and the principle in Mozilla to never 'leave behind' any prioritized platform. The fact that the projects chose to branch prior to production release, despite these concerns, indicate that restricting commit access for a prolonged phase was seen as entailing a too high cost in terms of holding back new development.

Obligation to integrate own contributions

The second key principle of the projects' decentralized process is the developers' obligation to integrate their own contributions and most notably to avoid breaking the build of the development version. From a long-term perspective, maintaining the development version in a healthy condition is crucial because it allows for the project to produce a software version that is mature enough for production release, merely through 'stabilizing' the development version. This may be a relatively painless process if the software is already in a working condition, implying that unexpected delays due to integration problems are avoided. From a short-term perspective, a working development version is crucial for the ongoing development and debugging effort: new development is halted if local source code changes cannot be tested against the newest source code in the trunk; and debugging is halted if developers and users can not run the software. A disadvantage is that with the source code in the trunk changing constantly, implementing a change is like hitting a moving target, and requires that developers' local source code is synchronized frequently. Placing the responsibility for integration on the developer or team developing a change may have a conservative effect, because large and complex new features (e.g. changes of the basic architecture) are difficult to divide into smaller, independent tasks that can be easily integrated.
In Mintzberg's terminology, the projects' 'don't break the build' rule is standardization of work products. Failure to comply with the rule is visible immediately. In both projects the ability to commit changes which are perceived to be of good quality constitute the merits on the basis of which commit access is granted, thus linking standardization of work products and worker skills in a manner akin to apprenticeships.
A major challenge for the projects, relating to 'don't break the build', is to interpret this rule in a suitable, pragmatic manner. There may be good reasons why developers frequently do not comply fully with their integration obligation, and rather than simply deleting error-prone changes, the projects organize a project-level, common effort to correct broken builds. The most structured effort takes place in Mozilla, where a daily cycle blocks new commits until the trunk has been built successfully, and where committers of recent changes are compelled to participate in resolving any broken build situations. Major reasons why developers inadvertently violate the 'don't break the build' rule include the inherent intricacy of debugging complex systems and not having access to the range of different platforms on which they should (ideally) test their changes.

FUTURE RESEARCH

In future research, studies of commercial software projects may be of interest to clarify whether such projects can attain the advantages of decentralized continuous integration, and provide insight into different ways of attaining balanced interpretations of the access to contribute and the obligation to integrate. In commercial software development, both the motivational advantage of seeing changes integrate quickly and the reduced risk of release delays may be as important as in open source projects.

While it is important to establish balanced interpretations of the access to contribute and the obligation to integrate, perhaps the major challenge, in open source as well as commercial projects, is to establish a project culture with strong encouragement to produce high quality code, but also with tolerance and mutual support. Indeed, the approach of continuous integration may provide a basis for developing such a culture, since the quality of the project's most recent source code becomes a common point of continuous, project-wide focus.

REFERENCES

About FreeBSD's Technological Advances. Retrieved Dec. 1, 2002, from: http://www.freebsd.org/features.html

Beck, K. (1999). Extreme Programming Explained: Embrace Change. Boston, USA: Addison Wesley.

bugs (2003). Retrieved April 8, 2003, from: http://www.mozilla.org/bugs/

Core Architecture. Retrieved Dec. 1, 2002, from: http://www.mozilla.org/catalog/architecture/

Cusumano, M. A., & Selby, R. W. (1997). How Microsoft Builds Software. **CACM**, 40(6), 53-61.

Dempsey, B., Weiss, D., Jones, P., & Greenberg, J. (2002). Who Is an Open Source Developer? A Qualitative Profile of a Community of Open Source Linux Developers. **CACM**, 45(2), 67-72.

Duddi, S. (1999). Pre-Checkin Tests. Retrieved May 1, 2003, from: http://www.mozilla.org/quality/precheckin-tests.html

Ebert, C., Parro, C. H., Suttels, R., & Kolarczyk, H. (2001). Improving Validation Activities in a Global Software Development. Paper presented at the **23rd International Conference on Software Engineering (ICSE '01)**, Toronto, Canada.

Eich, B. (2002). Mozilla 1.0 Manifesto. Retrieved Nov. 15, 2002, from: http://www.mozilla.org/roadmap/mozilla-1.0.html

Eich, B., & Baker, M. (2003). Mozilla 'Super-Review'. Retrieved April 21, 2003, from: http://www.mozilla.org/hacking/reviewers.html

The FreeBSD Committers' Big List of Rules. Retrieved Dec. 1, 2002, from: http://www.freebsd.org/doc/en/articles/committers-guide/rules.html

FreeBSD Security Guide (1997). Retrieved Oct. 31, 2003, from: http://www.pl.freebsd.org/security.html

Getting CVS Write Access to Mozilla (2003). Retrieved April 22, 2003, from: http://www.mozilla.org/hacking/getting-cvs-write-access.html

Ghosh, R. A., Glott, R., Krieger, B., & Robles, G. (2002). Part 4: Survey of Developers, **FLOSS** Final Report.

Hacking Mozilla. Retrieved Dec. 1, 2002, from: http://www.mozilla.org/hacking/

Hacking Mozilla with Bonsai. Retrieved Dec. 1, 2002, from: http://www.mozilla.org/hacking/bonsai.html

Hars, A., & Ou, S. (2001). Working for Free? Motivations of Participating in Open Source Projects. Paper presented at the **34th Hawaii International Conference on System Sciences**, Hawaii, USA.

Herbsleb, J. D., & Grinter, R. E. (1999). Splitting the Organization and Integrating the Code: Conway's Law Revisited. Paper presented at the **International Conference on Software Engineering (ICSE '99).**

Hubbard, J. (2003). Contributing to FreeBSD. Retrieved Oct. 31, 2003, from: http://www.freebsd.org/doc/en/articles/contributing/

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). **The Unified Software Development Process**. Indianapolis, USA: Addison-Wesley.

Jørgensen, N. (2001). Putting It All in the Trunk: Incremental Software Development in the FreeBSD Open Source Project. **Information Systems Journal**, 11(4), 321-336.

Kamp, P.-H. (1996). Source Tree Guidelines and Policies. Retrieved Dec. 1, 2002, from: http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/policies.html

Lakhani, K. R., Wolf, B., Bates, J., & DiBona, C. (2002). The Boston Consulting Group Hacker Survey. Retrieved May 8, 2003, from: http://www.osdn.com/bcg/bcg-0.73/

Lave, J., & Wenger, E. (1991). **Situated Learning: Legitimate Peripheral Participation (Learning in Doing).** Cambridge: Cambridge University Press.

Lucas, M. (2002). How to Become a FreeBSD Committer. Retrieved April 22, 2003, from: http://www.onlamp.com/lpt/a/1492

McConnell, S. (1996a). Daily Build and Smoke Test. **IEEE Software**, 13(4).

McConnell, S. (1996b). **Rapid Development: Taming Wild Software Schedules**. Microsoft Press International.

Mintzberg, H. (1979). **The Structuring of Organizations**. Englewood Cliffs, New Jersey: Prentice Hall.

Mockus, A., Fielding, R. T., & Herbsleb, J. D. (2002). Two Case Studies of Open Source Software Development: Apache and Mozilla. **ACM Transactions on Software Engineering and Methodology,** 11(3), 309-346.

Mozilla Coding Style Guide (2003). Retrieved Oct. 31, 2003, from: http://www.mozilla.org/hacking/mozilla-style-guide.html

Nakakoji, K., Yamamoto, Y., Nishinaka, Y., Kishida, K., & Ye, Y. (2002). Evolution Patterns of Open-Source Software Systems and Communities. Paper presented at the **International Workshop on Principles of Software Evolution (IWPSE)**, Orlando, Florida.

Olsson, K., & Karlsson, E.-A. (1999). **Daily Build - the Best of Both Worlds: Rapid Development and Control**. Lund, Sweden: Swedish Engineering Industries.

Pressman, R. S. (1992). **Software Engineering - A Practitioner's Approach** (3rd, international ed.). New York: McGraw-Hill.

Raymond, E. S. (2001). **The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary** (Revised ed.). Sebastopol, California, USA: O'Reilly & Associates, Inc.

The Seamonkey Engineering Bible (2003). Retrieved April 21, 2003, from: http://www.mozilla.org/projects/seamonkey/rules/bible.html

Williams, D., Collins, S., & Blizzard, C. (2003). C++ Portability Guide, version 0.8. Retrieved Oct. 31, 2003, from: http://www.mozilla.org/hacking/portable-cpp.html

Working with the Seamonkey Tree (2002). Retrieved Oct. 31, 2003, from: http://www.mozilla.org/hacking/working-with-seamonkey.html

Yeh, C. (1999). Mozilla Tree Verification Process. Retrieved April 11, 2003, from: http://www.mozilla.org/build/verification.html