# Putting it All in the Trunk

# Incremental Software Development in the FreeBSD Open Source Project

Niels Jørgensen

Computer Science Department, Roskilde University
PO Box 260, DK-4000 Roskilde, Denmark
nielsj@nielsj.dk

**Abstract.** Software development in the FreeBSD project is incremental in the sense that a stream of contributions, including bugfixes and minor and major new features, go into a single branch in the repository, the development branch (or "trunk"), and are required to preserve the software in a working state. This creates a succession of development releases, akin to the practice of frequent releasing argued by Eric S. Raymond in his essay *The Cathedral and the Bazaar* to be the key to the success of Linux and other open source projects. The positive and negative impact of this approach is discussed based on a survey involving 72 project participants. On the positive side, the data indicate that it creates a highly motivating "see bug, fix bug, see bug fixed in new release" life cycle, and helps integrate bugfixing with new development. On the negative side, the data indicates that the highly incremental model does not support the development of complex new features.

**Keywords:** open source, incremental development, maintenance, motivation.

## 1 Introduction

Raymond coined the phrase "Release early, release often" (*Raymond, 1998*) to characterize open source development. The FreeBSD project, which produces an operating system, employs frequent releasing in a radical sense, with many new development releases each day. The main purpose of the study reported in this paper is to investigate the pros and cons of the project's approach to frequent releasing. The approach may be characterized as highly incremental, in the sense that all changes to the source code are made as small contributions ("increments") to the development branch, which is intended to be always in a working state, so that it is useful as a basis for further development.

Frequent releasing is a prerequisite of parallel debugging, commonly recognized also as a key feature of open source projects (*Feller and Fitzgerald, 2000*).

> In the bazaar view [..] you assume that bugs are generally shallow phenomena - or, at least, that they turn shallow pretty quick when exposed to a thousand eager co-developers poinding on every single new release. (*Raymond, 1998*)

One interesting question pertaining to parallel debugging is whether there are complex defects for which it does not work equally well as for simple ones. Another is whether parallel debugging is the only method used to fight bugs. By classical software engineering principles, the approach to software defects is, of course, not merely a question of finding and correcting them. One purpose of a dedicated design phase preceding implementation, as advocated eg. by Parnas and Clements (*Parnas and Clements, 1985*), is to prevent errors in the first place by good design. A precondition for this classical argument to apply is that there are design problems of some complexity. McConnell offered the lack of any real new design problems as an explanation of Linux' success despite the project's apparent "code, then fix" approach:

> By the time Linux came around, requirements and architecture defects had already been flushed out during the development of many previous generations of Unix. (*McConnell, 1999*)

This argument may apply to FreeBSD even more than Linux, because FreeBSD is a direct descendant of the Unix variant developed at Berkeley University from 1967 and onwards. On the other hand, the project does enhance the operating system with various new and complex features. Thus the classical arguments for a dedicated design phase may apply, and it may be of interest to investigate if and how the project deviates from the classical approach by omitting design before coding.

Frequent releasing in FreeBSD is due to delegation of commit (check in) privileges to approximately 200 developers. Whenever they commit a change, a new development release is created instantly. This delegation of authority spurred my interest in FreeBSD. The non-hierarchical approach seems to be in the spirit of open source, and may support quick and un-bureaucratic integration of changes.

Previous academic studies have investigated projects where an individual is in control of the repository, as in the Linux project (*Kuwabara, 2000*), or privileged access is restricted to a small group, as in the Apache project (*Mockus et al., 2000*). In Hars and Ou's study (*Hars and Ou, 2001*) of participants in open source projects, which showed that altruism and improvement of programming skills were among the major motivating factors, the development model was not a parameter. The present study investigates motivation as depending on specific elements of the development model, in particular the developer's authority to add contributions directly to the development release.

Section 2 describes the survey underlying this study. Sections 3 introduces the output produced by FreeBSD, and Section 4 the FreeBSD organization. Section 5 investigates FreeBSD's incremental development model in detail, and Section 6 concludes.

## 2 The survey

Data was collected in November 2000 using a Web-based questionnaire directed at the FreeBSD developers with repository commit privileges. There were 72 replies, that is, just over 35% of the "committers". The questionnaire contained 29 multiple-choice questions. For each question I asked for further comments. Subsequently I conducted interviews with eight respondents via email; I am referring to this qualitative input as *comments* and *interviews*, respectively.

In general, the questions were directed at the respondent's most recent work. For example, when was his most recent contribution ? 61% within the last week, and another 22% within the last four weeks.

A group of questions dealt with frequent releasing and parallel debugging, for instance, how often in the past week had the respondent obtained a new release, and when was the last time there had been a bug report pertaining to the respondent's code ?

A second group of questions focused on reviewing, for example, when was the last time the respondent had distributed a design document for reviewing, and what was the number of people that provided feedback ?

The last group of questions asked the respondent to characterize various possible sources of learning and motivation inside the project. For example, to a number of questions of the form "I have improved my technical skills by xxx" the respondent was asked to choose between "yes, significantly", "yes, somewhat", and "no, not particularly".

The survey's questions and answers are available via the Web at www.ruc.dk/~nielsj/research/freebsd/freebsd.html.

## 3 The product and services provided by the FreeBSD project

A FreeBSD release contains an operating system *kernel* and *utilities* plus a very large collection of ported applications (*ports*).

A FreeBSD port is a piece of third party, open source software - for example, programming languages, web servers and browsers, and databases. A port is the adaptation of the software for FreeBSD, accomplished in the form of a small software interface, a “wrapper”. The latest official release of FreeBSD before the survey (4.2 of November 2000) contained more than 4.300 ports. For example, there are approximately 50 Java-related ports, including several Java compilers, Bytecode interpreters, and Just-in-time compilers,

In providing ports, FreeBSD parallels a Linux distribution - rather than Linux as such which is essentially a Unix operating system kernel.

In 1999 and 2000, four and five new releases of the system were distributed on CDs, and made available for download via the Internet. The release notes accompanying release 4.2 lists 75 diverse items, many of which really comprise several changes. The changes may be grouped using Swanson’s classification of software changes in maintenance work (*Swanson, 1976*):

**Adaptive changes** including new hardware device drivers and updates of ports. The idea of using port wrappers is that the same wrapper can be used for subsequent versions of the ported software. However, given the large number of ports, a significant amount of work is required to verify that the wrappers adapt to new versions of the ported software, and modify them if necessary.

**Corrective changes.** Many error corrections including 10 security fixes, each of which correspond to a so-called security advisory (see below).

**Perfective changes.** Changes that improve performance, for example of the socket interface to TCP/IP networking, or introduce new minor features such as an extra optional parameter to the `sed` (Stream Editor) program.

### The required design effort

Perfective changes that introduce major new features are normally released as part of major releases, such as 4.0, as opposed to the minor releases (4.1, 4.2, etc.). During the succession of major releases (1.0 - 4.0) produced since the project’s inception in 1991, the basic design has remained unaltered, as stated rather directly in the project’s webpages:

> FreeBSD’s distinguished roots derive from the latest BSD software releases from the [..] University of California, Berkeley. The book The Design and Implementation of 4.4BSD Operating System, written by the 4.4BSD system architects, thus describes much of FreeBSD’s core functionality in detail. [The book is (*McKusick et al., 1996*)]

New requirements derive mainly from the Internet and processor technology that may turn relatively cheap computers into powerful Internet-servers. The same factors have driven Linux development, at least to some degree, as indicated in (*Torvalds, 1999*). FreeBSD’s focus on performance-critical networking is expressed at the top page of the FreeBSD website in something resembling a mission statement for a company:

> FreeBSD makes an ideal Internet or Intranet server. It provides robust network services, even under the heaviest of loads, and uses memory efficiently to maintain good response times for hundreds, or even thousands, of simultaneous user processes.

During the time of the survey there was extensive work on a feature known as Symmetric Multiprocessing, to become part of major release 5.0. Symmetric Multiprocessing (SMP) is crucial for the exploitation of new cost-effective PCs with multiple processors. An operating system kernel with SMP is able to allocate different threads to execute simultaneously on the various processors of such a PC. Specifically, the goal for release 5.0 was to introduce SMP for threads running in so-called kernel mode, that is, threads that are executing system calls. The 4.x releases enables SMP only for threads running in user mode, and so prohibits full exploitation of PCs with multiple processors.

Thus, while the work that FreeBSD's development model should support is highly maintenance-oriented, and in general does not require work dedicated to the design of new features, there is also some development of complex new features. In case of SMP, while the basic concepts and problems are well understood, there is no well-established design for the feature in general, and of course not for its integration into the given FreeBSD kernel.

### The "copy-centrist" license and the BSD family

FreeBSD is part of an informal association of open source projects (NetBSD and OpenBSD) and California-based companies that sell various products and services related to the operating system originally developed at Berkeley. The family of BSD operating systems, of which FreeBSD has the most installations, was found to run 15% of the approximately 1.3 million Internet servers covered in a survey from April '99 (*Zoebelin, 1999*).

The FreeBSD type of open source license accommodates close cooperation with the commercial world. Similarly to the Apache license, the FreeBSD license derives from the X-license (*Perens, 1999*) and allows the user to incorporate the system into closed source products. The cooperation with the BSD siblings is visible in a number of ways:

One of the companies in the BSD family, BSDi, which maintains its own derivative of the Berkeley operating system, helped FreeBSD launch the effort to develop Symmetric Multiprocessing, by sharing its sources, design ideas, and lessons learned.

FreeBSD's release coordinator works for the company which is the official vendor of the FreeBSD releases. In general, a large part of the development work in FreeBSD is paid for: 43% of the respondents said an employer had paid for all or part of their time spent on their latest code contribution.

### Security-related services

It is extremely difficult to evaluate the quality of the software developed by FreeBSD to a more accurate level than simply stating that the operating system is working and has many users. For an operating system, especially for Internet servers, a crucial aspect of software quality is security. Although it is outside the scope of this paper to discuss the degree of security attained by FreeBSD, it is of interest to note that the project is extremely *committed* to security. As reported below in Section 5, this is also reflected in the project's guidelines for coding.

FreeBSD provides a kind of (free) security service in that the most recent releases are monitored for security. On a typical month, approximately half a dozen so-called security advisories are distributed to public mailing lists.

An example advisory is entitled *Bash creates insecure temporary files* (*FreeBSD, 2001a*). It explains what the bash program is (a Unix shell), describes a newly discovered security problem with the shell, assesses the impact (a possible denial of service attack) and finally describes both a temporary workaround (deletion of the program) and various ways of obtaining a corrected version of the shell.

For the bash shell error, corrected versions were made available in three versions for Intel x86 processors (for the latest 3.x, 4.x, and development releases) and two versions for the Compaq Alpha processor. Thus, FreeBSD's monitoring of its software for security entails a large amount of maintenance-oriented work.

## 4 Organizing the melting pot

The FreeBSD committers have write permissions to all branches of the source repository. As many as 1200 external contributors have also contributed source code or documentation to the FreeBSD project, in general on a less frequent basis; such contributions are added via a committer. Dozens of daily changes are added to FreeBSD's development branch, each of which creates a new development release. The project's tools for development support allow for such changes to be committed via the Internet by the execution of a single command, so well-defined and well-understood processes seem imperative in order to avoid chaos.

#### The core team

The project is headed by a core team with nine members, elected by and amongst the committers. In my interpretation, the team's most important areas of work are:

**Communication.** This includes definition and diffusion of processes, for example via the Committers' Guide (*FreeBSD, 2001b*). The guide's first rule is *Respect other committers.* It is motivated as follows:

> Being able to work together long term is this project's greatest asset, one far more important than any set of changes to the code [..].

An indication of the priority given to communication-related tasks is the Coordinator assignments. Approximately 15 responsibility areas have people assigned to them (some but not all are core team members). The assignments

include documentation project manager, internationalization, postmaster, public relations, usenet support, and webmaster.

**Management of the source repository.** A large number of coordinators have various aspects of configuration management as their responsibility area, including managers of the repository, releases, and the bug tracking system. The core team grants commit privileges to the repository.

> As a new committer there are a number of things you should do first. [..] Add yourself to the "Developers" section of the Handbook and remove yourself from the "Additional Contributors" Section. This is a relatively easy task, but remains a good first test of your CVS skills. (*FreeBSD, 2001b*).

A commit can be "rolled back", and a set of well-defined procedures exist for the core team's execution of it's right to revoke commit privileges.

**The maintainers**

The basic organizational "unit" is the individual contributor: Work is typically divided into tasks performed by a single person. 65% of the respondents said that their last task had been worked on largely by themselves only, with teams consisting of 2 and 3 committers each representing 14%.

Large portions of the source files, including most of the ports, are associated with a *maintainer*, either a committer or an external contributor.

> The maintainer owns and is responsible for that code. This means that he is responsible for fixing bugs and answer problem reports [..]. Changes to directories which have a maintainer shall be sent to the maintainer for review before being committed [..]. (*FreeBSD, 2001c*)

For a given area of the code, there may be an official maintainer, or the "de facto maintainer" may simply be the contributor of the last change:

> In cases where the "maintainer-ship" of something isn't clear, you can also look at the CVS logs for the file(s) in question and see if someone has been working recently or predominantly in that area. (*FreeBSD, 2001b*).

Maintainers are involved in maintenance in the broad sense including perfective changes, rather than forming a separate team dedicated to bugfixing. The respondents were asked to characterize their latest code contribution in terms of Swanson's three types of changes, supplemented with a fourth category of preventive changes as in (*Takang and Grubb, 1996*).

38% said that their last contribution was perfective, 29% corrective, 14% preventive, and 10% preventive. Many stressed that the choice of category was highly depending on the time of the survey.

> I do all the above [the four types of changes], my last commit just happened to be a bugfix/feature fix. (*Comment*)

> But if you had asked a different day, I would answer differently. (*Comment*)

## 5 Life cycle for changes

In working on an individual change, the FreeBSD committer is required to follow a set of guidelines laid down in the committer's guide (*FreeBSD, 2001b*), the FreeBSD Handbook (*FreeBSD, 2001c*), and the security guide (*FreeBSD, 2001d*). For the purpose of structuring the presentation of the guidelines and the survey data pertaining to them, in this section they are divided into six groups. This approach postulates a life cycle for changes as shown in Figure 1.

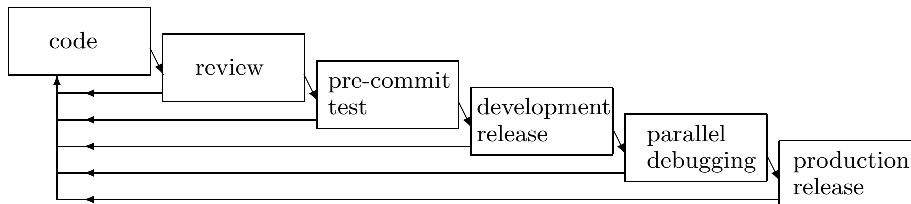


**Fig. 1.** The life cycle for changes. In general each stage is mandatory. Exceptions include skipping review of a trivial bugfix, and development release and parallel debugging of a bugfix urgently needed for production release.

Coding and pre-commit testing takes place in the committer's private repository. Development release of a change is the integration of it into the repository's development branch where it resides during parallel debugging. For production release a change is merged into the production branch. A change is merged into the previous production branch as well if it is a bugfix relevant for it, such as the security fix of the Bash shell discussed in Section 3. The branches are shown in Figure 2.

Significantly, the project does not have guidelines for allocating people to work on a change in the first place. The project maintains a prioritized list of tasks, but there is no obligation on behalf of a maintainer to pick up a task inside his area of responsibility.

Neither are there any guidelines pertaining particularly to large development efforts such as the enhancement of the kernel's Symmetric Multi-processing ability. It is up to the participants in a large development effort to create an appropriate internal organization; the life cycle applies merely at the level of the individual change. Development of SMP was organized as a subproject that proceeded in a way akin to classical principles: The project was launched upon a face-to-face meeting (at Yahoo's premises in California). For coordination of the approximately 10 active developers, it had a project manager and a plan that defined a work break-down in tasks and subtasks, and design was discussed and written down before coding.

**Code (stage 1)**

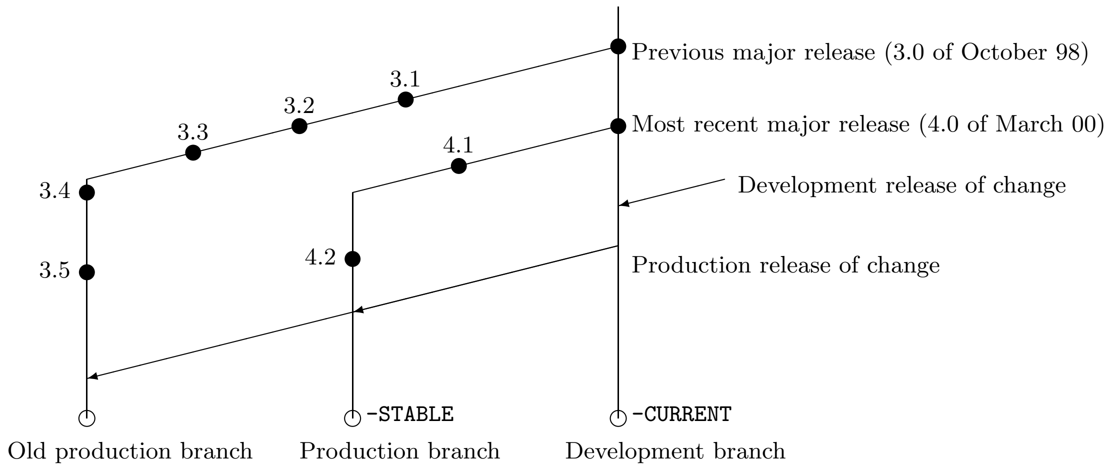


**Fig. 2.** The branches of FreeBSD's source repository. The filled circles are the major and minor releases available in November 2000. -CURRENT and -STABLE are the project's names for the current versions of the development and production branches.

The kernel, utilities, and ports wrappers are written almost exclusively in C. The so-called Kernel source file style guide describes the preferred use of the language. The guide is supplemented by a dedicated security guide (*FreeBSD, 2001d*) which explains basic security issues and sketches rules of thumb such as

> Never trust any source of input [..] never use gets() or sprintf(), period.
> If you do - we will send evil dwarfs after you.

The most significant aspect of the coding process is perhaps that it is straightforward to monitor whether the programming guidelines are followed, or to evaluate other aspects of code quality. An automatic mail message is sent summarizing every commit, and the repository is easily browsable via a Web-interface.

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% answered "yes, significantly", and 29% said "yes, somewhat". A committer summarized:

> Embarrassment is a powerful thing. (*Comment*)

The comments submitted to this question also indicate that many developers felt that although they had learned a lot by participating in FreeBSD, they were already fairly matured as software developers when they entered the project.

> The way you get granted commit privileges is by first making enough code contributions or bug fixes that everyone agrees you should be given direct write access to the source tree. This implies the code you've been submitting is of sufficiently impressive quality that no one objects and says, "Yuck, no, don't give him commit privileges. He writes ugly code." [..] by and large, most of the committers are better programmers than people I interview and hire in Silicon Valley. (*Interview*)

**Review (stage 2)**

The project strongly suggests (but does not absolutely require) that any contribution is reviewed. *Committers' Guide* rule 2 is "Discuss any significant change *before* committing":

> [A commit should happen] only once something resembling consensus has been reached. This doesn't mean that you ask permission before correcting every obvious syntax error [..]. The very best way of making sure that you're on the right track is to have your code reviewed by one or more other committers. [..] When in doubt, as for review !

The data indicate that there are frequent reviews, and that the main obstacle to an increase in review activity is that it is difficult to enlist reviewers.

Code reviewing is the most widespread. To solicit feedback, code is typically distributed via email. 57% had distributed code for reviewing within the last month, 85% within the last 3 months (and nobody never had). Almost everybody (86%) said they actually received feedback on their latest such distribution, most of them from two or more reviewers.

Typically, it requires some effort for a committer to persuade co-developers to review his work:

> I have to aggressively solicit feedback if I want comments. (*Comment*)

> If I don't get enough feedback, I can resort to directly mailing those committers who have shown an interest in the area [..] I can also hang out on IRC and ask for comments in 'real-time'.. (*Interview*)

One of the few respondents that had not received feedback (on his last proposal) said he felt that in general, the key step was to attract the first comment to a mailing list posting:

> If somebody, anybody, replies then [..] it is active. (*Interview*)

and added that especially for people, like himself, that are not among the project's "heavy weights", some effort was required to attract attention and get a discussion started.

A respondent indicated that for complex code, reviewing is the most difficult to solicit:

> The simpler the code is, the more comments you will get; for larger projects, the fewer constructive comments you receive. (*Interview*)

The soliciting of feedback on design, defined in the survey as distributing a separate design document, not a source file, was an activity undertaken by 26% of the respondents within the last three months (where 90% had distributed code). As many as 93% said they actually received feedback on their last such distribution.

It seems that the main barrier is more that it is difficult to get feedback on design, rather than unwillingness to work with design in the sense of writing and submitting design proposals.

> Getting solid, constructive comments on design is something like pulling teeth. (*Comment*)

> I did get feedback [on a design document] of the type “This looks good” [..] but very little useful feedback. (*Comment*)

### Pre-commit test (stage 3)

At the heart of FreeBSD’s development model is the requirement that a committer shall do pre-commit testing to prevent breaking the build in the development branch. The commit is required to comprise not just the new or modified file(s), but any interdependent file. In reality this implies that the committer is responsible for *integrating* his change before committing it.

*Committers’ Guide* rule 10 is “Test your changes before committing them”:

> This may sound obvious, but if it really were so obvious then we probably wouldn’t see so many cases of people clearly not doing this. 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 [the utilities and ports].

Despite the guide’s fairly clear statement of the “don’t break the build” rule, it seems there is a consensus that the rule can be excepted from in practice:

> 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. (*Interview*)

For a committer to build the system entails certain hardware and infra structure requirements: The committer must have access to a private copy of the latest version on the development branch, into which he can insert changes without interfering with the other developers. The high frequency of contributions means that a committer’s copy may quickly become out-of-date. 76% said that they had rebuilt their copy of the development release at least once in the week preceding their response (25% had done so on 5 days or more). Typically it takes 1-2 hours to build a full system.

In combination with the visibility of the source code, the objective of not breaking the build is an strong incentive for committers to improve their skills: 43% said they had improved their technical skills significantly by debugging build failures, and 39% said this was somewhat the case.

**Development release (stage 4)**

The committer has the authority to decide whether a change is ready for development release. He also carries out the act of releasing the change, since a new release is the effect of a commit to the development branch. There is a well-defined process for the case where, upon a commit, it turns out that an appropriate consensus had not been reached, despite the reviewing requirement.

> Any disputed change must be backed out [..] if requested by a maintainer. Security related changes may override a maintainer's wishes at the Security Officer's discretion. (*FreeBSD, 2001b*)

In practice, the delegation of the authority to commit changes means that the developer gets to see his contribution become part of the project's development release immediately upon completing the change.

The survey indicates this plays an important role in creating a motivating work environment. As many as 81% of the committers in the survey responded that they were encouraged a lot by having the ability to integrate code directly.

> I don't feel I'm at the whim of a single person; I can develop/commit under my own authority, and possibly be overridden by a general consensus (although this is rare). (*Comment*)

> [..] there is a tremendous satisfaction to the "see bug, fix bug, see bug fix get incorporated so that the fix helps others" cycle. This of course applies to things other than bugs.. (*Comment*)

In contrast, there may be no visible improvement of the development release as a result of changes committed in the early stages of a large development effort such as the SMP subproject. The SMP subproject manager summarized the results of the approximately half a year of work as follows:

> [..] we have done *huge* amounts of work at this point, but there are *no* performance benefits yet (performance is actually significantly worse). (*Interview, respondent's emphasis in email*)

The performance decrease was explained as a result of the incremental approach: The need to preserve the development release as a working system forced the SMP subproject to be extremely cautious and introduce a new locking mechanism in the kernel *before* actually launching the simultaneous kernel threads. The locking mechanism, which incurs a run-time overhead, served to encapsulate the fragments that the kernel was divided into, each of which was to be allowed to (eventually) have a thread running inside it.

In addition to decreased performance, the changes committed in the early stages of the SMP subproject rendered the development release somewhat unstable. This was predicted in advance - a warning was issued before the first SMP-related commit:

> `-CURRENT` will be destabilized for an extended period [..]. A tag [..] will be laid down before the initial check-in, and non-developers should either stick close to that tag [..] or expect large doses of pain.

The FreeBSD project as a whole was willing to accept some instability. The main problem, as seen from the SMP subproject, was the extra work required to maintain the development branch in a reasonable state.

> One of the things that worried me [..] was that we wouldn't have enough man power on the SMP project to keep up with the changes other developers were making. [..] the SMP changes touch huge amounts of code, so having others working on the code at the same time is somewhat disruptive [..]. (*Interview*)

Thus, the SMP effort incurred some instability in the development branch, making it less useful as the basis for all development. Preserving a working kernel was seen as a burden, and extra work was required to integrate, on an ongoing basis, with concurrent work in the kernel.

### Parallel debugging (stage 5)

When a change has been committed to the development branch, it is subjected to parallel debugging in a sense consistent with Raymond's notion (*Raymond, 1998*). Debugging is performed primarily by the committers and the external contributors of source code. For those who may find it difficult or inconvenient to download and build the source, the project provides the binaries:

> If you [..] have a good Internet connection, there is a machine `currentFreeBSD.org` which builds a full release once a day – every now and again, try and install the latest release from it and report any failures [..] (*FreeBSD, 2001c*)

There is a web-based tool for creating problem reports (PRs), submitting them in the project's PR database, and browsing them. People are encouraged not only to send in PR's but also to follow-up on them:

> If you know of any bugfixes which have been successfully applied to `-CURRENT` but have not been merged into `-STABLE` after a decent interval (normally a couple of weeks), send the committer a polite reminder. (*FreeBSD, 2001c*)

Some respondents indicated that they receive a constant flow of problem reports; nearly half the respondents said that within the last month, someone (other than themselves) had reported a problem related to "their" code; and nearly half said that within the last month there had been a bugfix to their code contributed by someone else. 82% had received either form of feedback within the last 3 months. This indicates that the parallel debugging organized around the development branch does create a significant amount of feedback to the contributors.

However, there is indication that the feedback generated by parallel debugging was of little value for the committers working on the complex SMP feature. The SMP project manager expressed that

> In actuality, the bug reports we've gotten from people have been of limited use. The problem is that obvious problems are quickly fixed, usually before anyone else notices them, and the subtle problems are too "unusual" for most of the other developers to diagnose [..] (*Interview*)

**Production release (stage 6)**

The project's process for production release of a change is based on delegation of responsibility to the project's 200 committers, similarly as for development release. However, a strong element of control is added via the veto power of the core team's release engineer.

For most changes, production release is in two steps. The first is the merge of the change from the development branch to the production branch. A change is merged by the committer when he believes it has been sufficiently tested in the development branch, without awaiting approval. However, the release engineer can ultimately require that a change is backed out from the production branch.

> Please [..] give the release engineer your full cooperation when it comes to the `-STABLE` branch. The management of `-STABLE` may frequently seem to be overly conservative, but also bear in mind the fact that conservatism is supposed to be the hallmark of `-STABLE` [..]. (*FreeBSD, 2001b*)

The second step towards production release is a code freeze period (see Figure 3). During code freeze the release engineer is vested with the authority to reject all changes other than bugfixes. A code freeze on the production branch ends when the release engineer declares it sufficiently stable so as to constitute the next minor release.

Changes that are part of a major new feature such as Symmetric Multiprocessing are not merged to the production branch. Instead they remain inside the development branch, where they eventually become part of the next major production release. After a code freeze of approximately 1 month, the release engineer declares the release residing in the development branch the new major production release.

Major releases are one of few events for which there is a deadline: the project attempts to provide one approximately every 18 months. This establishes a common goal for the entire project. In contrast, there is typically no deadline for the

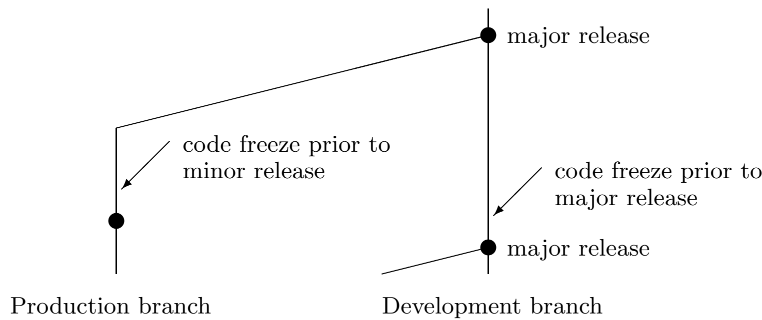


**Fig. 3.** Code freezes prior to major and minor releases.

individual change: 95% of the respondents said that there had been no deadline for the task that their last commit was a part of.

The authority vested in the role of the release engineer to reject changes during code freezes is seen as a necessary evil, particularly in case of freezes on the development branch, because this stalls new development:

> [..] we were spinning our thumbs. [..] It was a really boring month, but the 4.x release is probably the best release we've had [..]. (*Interview*)

## 6 Conclusion

The FreeBSD project organizes a huge effort which is mainly of a maintenance-oriented nature. An approach likely to have contributed to a successful allocation of man power to a type of work that might be perceived as boring or low-status is the integration of bugfixing with new development. These aspects of work are integrated organizationally by the delegation to so-called maintainers of the responsibility of the various areas of the source code, and in the development model by contributing all changes to the development branch.

| | |
|---|---|
| **code** | Instant visibility of committed code enables monitoring of code quality and motivates developers to improve skills. |
| **review** | Committers frequently seek and receive review. Main obstacle is lack of feedback, especially on design documents and complex code. |
| **pre-commit test** | The implied responsibility of the committer to integrate changes ("don't break the build") is a further learning incentive. |
| **development release** | The committer's commit authority creates a highly motivating "see bug, fix bug, see bug fixed" cycle for small changes. |
| **parallel debugging** | Highly prioritized by the project and produces a significant amount of feedback, but possibly mostly on simple problems. |
| **production release** | Prior to major/minor releases authority is delegated to a single individual, yet the committers retain their commit privileges. |

**Fig. 4.** Summary of findings of the investigation of FreeBSD's life cycle for changes.

The findings pertaining to the incremental change life cycle are summarized in Figure 4. The life cycle seems to support motivation and learning. In particular, the data indicate that the ability to integrate changes quickly and unbureaucratically in the development release stage is highly motivating. Also they suggest that being responsible for integration is an opportunity and incentive to learn during the pre-commit testing and coding stages.

The project's explicitly and eloquently defined guidelines, and statements in them such as "Being able to work together long term is this project's greatest asset, one far more important than any set of changes to the code" contradict perceptions such as those in (*Wilson, 1999*) of the open source community as being unaware of the need for software process. The findings related to the reviewing process suggest that the project could attain an increase in review activity by encouraging more strongly to *respond* to requests for reviews, and those seeking it to pursue it more aggressively.

The absence of a traditional management authority is highly valued by the project's participants, and respondent's comments such as "embarrassment is a powerful thing" indicate that is has been replaced by an effective, but also somewhat frightening peer regime.

The project's life cycle for changes may be, at least, insufficient as the basis for organizing work on complex new features such as the kernel's Symmetric Multiprocessing ability. The SMP subproject chose to use a classical approach with long-term planning and "design, then code". The option of choosing a hybrid classical/incremental approach may testify to the flexibility of the basic change life cycle. However, the subproject saw integration with concurrent work in the development branch as a burden, and the feedback it received by parallel debugging was felt to be of limited use. Thus the survey indicates that the incremental approach and the ensuing parallel debugging is no "silver bullet" to overcome the intrinsic difficulties of intensified, concurrent development.

**Acknowledgement.** This research was supported by O'Reilly and Associates, Inc., and the Development Center for Electronic Business and the IT-University, Copenhagen.

## References

Feller, J. and Fitzgerald, B. (2000). A Framework Analysis of the Open Source Software Development Paradigm. In *Proceedings of the 21st International Conference on Information Systems*, Brisbane, Australia, December 2000.

FreeBSD (2001a). Bash creates insecure temporary files.
FreeBSD Ports Security Advisory.
ftp://ftp.FreeBSD.org/pub/FreeBSD/CERT/advisories/FreeBSD-SA-01:03.bash1.asc

FreeBSD (2001b). The FreeBSD Committers' Big List of Rules.
http://www.freebsd.org/tutorials/committers-guide/index.html

FreeBSD (2001c). The FreeBSD Handbook.
http://www.freebsd.org/handbook/index.html

FreeBSD (2001d) FreeBSD Security Information.
http://www.freebsd.org/security/security.html

Hars, A., and Ou, S. (2001). Working for Free ? - Motivations of Participating in Open Source Projects. In *Proceedings of the 34th Hawaii International Conference on System Sciences, 2001.*

McConnell, S (1999). Open-Source Methodology: Ready for Prime Time ? In *IEEE Software*, **16** (4), 6-11.

McKusick, M. K., Bostic K., Karels M. J., Quarterman, J. (1996). The Design and Implementation of the 4.4BSD Operating System. Addison-Wesley.

Mockus, A., Fielding, R. T., Herbsleb, J. (2000). A Case Study of Open Source Software Development: The Apache Server. In *Proceedings of the International Conference of Software Engineering, 2000, Limerick, Ireland*

Ko Kuwabara (2000). Linux: A Bazaar at the Edge of Chaos. In *First Monday*, **5** (3).
http://www.firstmonday.org/issue5_3/kuwabara/index.html

Parnas, D. L. and Clements, P. C. (1985). A Rational Design Process: How and Why to Fake it. In *Proceedings of Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, Berlin.

Perens, B. (1999). The Open Source Definition. In *Open Source: Voices from the Open Source Revolution*, O'Reilly, 1999.

Raymond, E.S. (1998). The Cathedral and the Bazaar.
http://www.tuxedo.org/ esr/writings/cathedral-bazaar/

Swanson, E.B. (1976). The Dimension of Maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*, October 1976, 492-497.

Takang, A. A. and Gruub, P. A. (1996). Software Maintenance. Concepts and Practice. Thomson Computer Press.

Torvalds, L. (1999). The Linux Edge. In *Communications of the ACM*, **42** (4), 38-39.

Wilson, G. (1999). Is the Open-Source Community Setting a Bad Example ? In *IEEE Software*, January/February 1999, 23-25.

Zoebelein, H.U. (1999). The Internet Operating System Counter. http://www.leb.net/hzo/ioscount/

**About the Author**

Niels Jørgensen is Associate Professor at the Computer Science Department of Roskilde University. Niels caught interest in open source when he was a Nokia employee and co-workers pointed to Eric Raymond's work, and to tools from the open source world. His Ph.D. was about compile-time analysis (abstract interpretation) of constraint programming languages. Besides open source and optimized compilation, Niels' current research interest includes mobile Internet.