J Manage Governance (2007) 11:119–128
DOI 10.1007/s10997-007-9026-5

ORIGINAL PAPER

Developer autonomy in the FreeBSD open source project

Niels Jørgensen

Published online: 9 June 2007

Abstract Delegation of authority is the basic approach to coordination of work in FreeBSD, an open source software (OSS) project that develops and maintains an operating system. This essay combines a software engineering perspective with a knowledge creation perspective to discuss three central mechanisms in FreeBSD: (1) motivation, (2) quality control by frequent building, and (3) bugfixing by parallel debugging. It is argued that frequent building is the project's key coordinating mechanism, and that build breakdowns play a constructive role in the processes of knowledge creation in FreeBSD.

Keywords Frequent building · Knowledge creation · Open source · Quality assurance

1 Introduction

The governance structure of the FreeBSD open source project has two important features. On the one hand, the project's formal organization is *democratic* in the sense that FreeBSD's project leadership, the so-called core team, is elected bi-annually by the project's developers. In the July 2006 elections, out of 308 developers entitled to vote, 208 cast their votes, electing nine core team members (FreeBSD 2006).

On the other hand, the governance structure is *decentralized* as far as daily work with change integration is concerned: the decision to add a change to the source code repository is made by the individual developer (with commit privileges, cf. below) who has worked on the change. FreeBSD lets more than 300 developers add

N. Jørgensen (✉)
Computer Science Department, Roskilde University, P.O. Box 260, 4000 Roskilde, Denmark
e-mail: nielsj@ruc.dk

changes to the repository on a 24/7 basis. Once a change has been added to the repository, it is part of the project's official release, which is publicly available for download. In this sense, the FreeBSD developers are highly autonomous. This is contrary to a centralized decision making process, where a change must be sanctioned by an authority above the developer, which is the typical approach in classical software engineering. For example, Pressman and Ince (2000) suggest a process with a change request, a change evaluation, and—if approval is granted by the change control authority—a change order (sic). In its approach to adding changes, FreeBSD also distinguishes itself from other open source projects such as Linux. Linux is controlled by a single person, Linus Torvalds. He, together with a small group of people to which he delegates authority (his 'lieutenants'), adds other people's contributions to the repository.

This essay discusses FreeBSD's development process from two main perspectives. First, a *software engineering perspective* sheds light on FreeBSD's effort to coordinate its seemingly chaotic process. The project lacks the power to allocate workforce according to a classical approach, where work is orchestrated into phases of (typically) design, code, integrate, and test. Instead, FreeBSD relies on frequent building for quality control. Building is the process of transforming human-readable source files to machine-executable files. Frequent building is a well-known part of several new approaches in software engineering, including rapid development (McConnell 1996) and agile development (Beck et al. 2001), and has been used by Microsoft, Netscape, and other commercial software developing firms (MacCormack et al. 2001). In these approaches, features or changes are added frequently. Frequent changes require frequent builds, since the software must be re-built when the source is modified.

Second, a *knowledge creation perspective* may be useful in understanding the impact of frequent building on FreeBSD's knowledge creation processes. Lee and Cole (2003) proposed a model of community-based knowledge creation as a means of capturing the innovation process in Linux. The characteristics of the model apply to FreeBSD as well. This essay proposes the concept of breakdowns as discussed by Nonaka (1994) as an additional perspective for understanding the knowledge creation process in FreeBSD. While Lee and Cole's model focuses mainly on the individual new feature, the concept of breakdowns may be useful in understanding how knowledge is created about the software as a whole, because breakdowns may occur due to interdependencies between different parts of the software, and may lead to understanding these better.

The essay first provides some details about FreeBSD. The next three sections discuss three mechanisms that can be identified as central in the project: developer autonomy as a motivation factor, quality assurance by frequent building, and bugfixing by parallel debugging. A final section concludes.

2 FreeBSD

FreeBSD, like Linux, is a Unix-type operating system. While Linux was built from scratch by Linus Torvalds and others, FreeBSD is a derivative of BSD Unix, a major

Unix variant developed at Berkeley University from 1977 onwards. It is distributed with a BSD open source license, which is a pragmatic license that allows users to modify the software and distribute it as either open or closed code. FreeBSD is widely used on the Internet, including at large sites such as Yahoo. Also, a derivative of FreeBSD is used in Darwin, the kernel of Apple's OS X operating system.

The FreeBSD developers with write access to the repository (308 in all) are called committers. Other developers' changes are added by a committer. Technically, a developer's right to commit changes is based on his access rights to the source code repository. The repository is stored on machines in California connected to the Internet. A change is a modification to one or several lines, in one or several files. Changes are added using commands from the revision control program CVS. The CVS user interface includes a command for downloading an up-to-date version of the repository, and a command for uploading a set of modified files back into the repository, thereby adding a change.

Part of the data about FreeBSD presented in the sequel was collected in 2000–2001 using email interviews and a webbased survey directed at FreeBSD's committers (approximately 200 at the time). The survey received replies from 72 developers. Results were published in Jørgensen (2001).

The release notes associated with release 6.0 of October 2005 are an indication of the evolutionary nature of work on FreeBSD. The notes describe the enhancements made since release 5.4 of May, 2005, numbering approximately 225 entries. Each entry describes a new feature, bugfix, or other improvements. The new features listed are typically small improvements such as *''The* uaudio *driver now has some added functionality, including volume control on more inputs and recording capability on some devices''. A typical bugfix entry is: ''Several programming errors in*cvs*, which could potentially cause arbitrary code to be executed on CVS servers, have been corrected''* (FreeBSD 2005).

3 Developer autonomy as a motivating factor

FreeBSD's developers appear to be extremely satisfied with the project's decentralized approach to change integration. As many as 81% of the respondents in my survey said that they were encouraged a lot by the access to add changes directly. Developers said, for example, ''*I don't feel I'm at the whim of a single person''*, and ''*I have submitted code fixes to other projects and been ignored. That was no fun at all.''*

In detailing why they like FreeBSD's decentralized approach, several developers pointed out that it made work easier for them: ''*It is frequently easier to make a change to the code base directly than to explain the change so someone else can do it.''* And another said: *''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''*.

These comments can be interpreted as follows. First and foremost committers feel they enjoy high autonomy. Moreover, decentralized governance saves them

from some boring and tedious activities. In terms of the Job Characteristics Model as developed by Hackman and Oldham (cf. Hertel, this issue), skill variety and possibly even task identity may be assumed to be high. So altogether, in several important ways the FreeBSD approach to code integration contributes to high internal work motivation.

The literature about open source developers' motives has mainly focused on internal and external motivations (see Hertel, this issue). In the light of the findings above, it would be worthwhile to also include the governance structure of work as a motivating factor. Presumably, it will be an important factor for both joining and staying in an OSS project.

4 Quality assurance by frequent building

Given the absence in FreeBSD of centralized control of changes added to the software, the project needs alternative mechanisms for quality control. Raymond (2000) discusses two complementary mechanisms in open source projects: hierarchy and parallel debugging. The latter mechanism will be discussed in the next section. This section discusses the first mechanism, which relies on what Lee and Cole (2003) call a two-tier hierarchy and aims at reducing the complexity that arises when many developers work on the same software. The basic principle is that changes made by lower tier developers must be sanctioned by upper tier developers.

Coordination inside an upper tier of such a hierarchy is manageable if it is small, consisting of a few people only. When the size of the upper tier expands, a coordination mechanism inside the upper tier is required. In Apache, controversies over software changes were settled by voting in the Apache core group, which numbered from eight to about two dozen members between 1995 and 1999 (see Mockus et al. 2000). Voting is still used in the current Apache HTTP server project (Apache 2007). In Linux, with over 100 maintainers in the upper tier, a hierarchy *inside* the upper layer has been established, where Linus and his lieutenants ultimately decide (Lee and Cole 2003).

For FreeBSD, the distinction between committers and external developers is akin to a two-tier hierarchy, where the committers constitute the upper tier. Coordination is even more complex than in Linux because there are more than 300 committers. Instead of introducing a voting system or a hierarchy, FreeBSD delegates quality control to its autonomous developers and says to them: ''*Test your changes before adding them. Make sure you don't break the build*!'' (FreeBSD 2007a). This approach may fruitfully be analyzed from two perspectives: a software engineering perspective and a knowledge creation perspective.

4.1 A software engineering perspective

The process of building FreeBSD's source code takes approximately 1 h, depending on the build machine. The main step is invoking a compiler to compile source files. To control the quality of his change, the developer should add it to a copy of the full system on his own system and do a trial build, before adding the change to the

central repository. To break the build is to add a change that prevents the source code from being built.

The ''don't break the build''-rule assumes that there is already a working version of the software. From a software engineering perspective, adding changes to a working version of the software is a variety of what is known as incremental integration. One of the advantages of the approach is support for debugging of errors observed during integration: ''*When the product is built and tested every day, it's much easier to pinpoint why the product is broken on any given day. If the product worked on Day 17 and is broken on Day 18, something that happened between the builds on Days 17 and 18 broke the product.*'' (McConnell 1996, p. 406). This approach to incremental integration is markedly different from the classical approach to integration, which is a planned hierarchical approach, where developers work on pre-designed modules, which are eventually assembled in larger and larger units. This classical approach to integration is also used in other engineering disciplines, such as civil or aircraft engineering, and is sometimes referred to as the V-model (Auyang 2004). An orchestrated approach as the V-model would be difficult for FreeBSD with its volunteer participants.

If FreeBSD's software is *not* in a working state, work is delayed because a build test against the most recent software cannot be performed. On the other hand, the rationale of the approach is that broken builds do happen from time to time, which reflects the inherent complexity of software integration. Indeed, more than 30% of the developers in my survey said that a change they had committed had caused a broken build in the repository, within the last 3 months.

Broken builds may be due to outright negligence. Several of the 11 rules in the committer's guide address the issue of build breakages, and rule 10 spells out the sanctions: *''Breaking some of these rules can be grounds for suspension or, upon repeated offence, permanent removal of commit privileges''*. However, the project interprets its ''don't break the build''-rule with some flexibility. A developer I interviewed said that over a period of time, he had broken the build every two-three days, and that this was tolerated by the project due to the nature of his work. Conversely, there is an understanding that the testing conducted by a developer prior to commit should be more extensive than doing a build; a developer should normally also make sure that the newly built system can start and perform basic functions.

4.2 A knowledge creation perspective

Nonaka's theory of knowledge creation (Nonaka 1994) offers a view of build breakages as playing a constructive role, namely as a significant part of the project's knowledge creation process. A central concept in his theory is *breakdowns*: unexpected events in the individual's interaction with the environment. Their significance is that they may lead to reconsideration of existing thinking, assumptions, and systems of knowledge. It is a concept from Heidegger's philosophy of technology that Winograd and Flores (1986) suggested to be relevant in software design. In the sequel I use the breakdown concept to highlight the significance of broken builds in FreeBSD.

A broken build is a breakdown in a very literal sense. When the compiler is not able to compile the sources, something is wrong. Moreover, a broken build is a violation of the ''don't break the build''-rule quoted above, and so a violation of prescribed practice—although as noted, the rule is open to interpretation. When a build is broken in FreeBSD, particular individuals are in focus, namely the committers that committed the change that caused the break. Sixty-five per cent of the respondents in my FreeBSD survey said that their last task had been carried out largely by themselves alone, with teams consisting of two and three developers each representing 14%. So typically, this concerns a single individual. However, if the error is not corrected quickly by the 'guilty' committer, other committers may participate in a corrective effort.

The learning process spurred by a broken build may involve understanding interdependencies between the changed part and other parts of the software. This is because a basic challenge related to integration is that a part may work well in isolation, but does not work together with the full system (McConnell 1996: 406). Interdependencies arise, for example, when the changed part defines a function that is used by other parts. A simplified example would be the following: when an extra parameter is added to a function defined in a part, there will be a build error if another part contains a function call without the extra parameter. According to a mail message issued to developers (FreeBSD 2007d), issues with function declarations are a major cause of broken builds. Another issue mentioned is that the build succeeds on the i386 Intel processor architecture, which is used on most of the developers' private machines, but fails on one of the other five processor types that the FreeBSD operating system supports. Other causes of broken builds include simultaneous commits of conflicting changes. An indication of the significance of broken builds in terms of learning is that 82% of the developers said that they had improved their technical skills by debugging build failures (this figure includes build failures on their private machines prior to commit).

While the learning processes associated with broken builds concern the system as a whole, the processes associated with earlier phases of work on a change in FreeBSD have a narrower scope. These early phases of work include initial coding and review of code proposals (and in some cases also bug reporting, to be discussed in the next section). Letting one's code be reviewed before committing is strongly recommended: ''*The very best way of making sure that you are on the right track is to have your code reviewed by one or more other committers*'' (FreeBSD 2007a). To solicit feedback, code is typically distributed via email. In the survey, 57% of the committers said they had distributed code for reviewing within the last month, and almost everybody (86%) said that they had obtained feedback. Typically, review comments are based merely on reading the code. Code reviewing is supported by a set of guidelines, including a kernel style guide (FreeBSD 2007b) and a security guide (FreeBSD 2007c). These guides describe the preferred use of the C programming language, and include, for example, a rule based on security considerations prohibiting the use of certain functions in the standard C library.

Thus, the learning processes related to broken builds are different from those related to earlier work on a change. The knowledge created in early work typically concerns the individual change; review feedback is based on reading the initial

code, and refers to general principles, such as those expressed in the project's guidelines. In contrast, the scope of the knowledge in subsequent work, which includes change integration, is the system as a whole; and feedback is based on observing the actual behavior of the system upon integration of the change.

A further characterization of the two types of learning processes can be made on the basis of Nonaka's (1994) distinction between two types of knowledge: explicit (codifiable, transmittable in systematic language) and implicit (hard to formalize, rooted in action and a specific context). In early phases of work, feedback from code reviewing is communicated from one developer to another in written form (by email), and to some extent, it is based on explicit, general knowledge as laid down in the guidelines. In contrast, when a change is integrated into the development version, the developers learn by interacting directly with the full system. The acquired knowledge is applied in the form of code corrections, but the knowledge itself is largely implicit and is not (and cannot be) communicated to others—but nevertheless, it is highly valuable in future work on change integration in the project.

The distinction between explicit and implicit knowledge may also be useful to characterize the difference between Linux and FreeBSD. While in FreeBSD, most changes are generated within the upper tier (the more than 300 developers), proposed changes in Linux are generated mainly by members of the lower tier, according to Lee and Cole (2003). Subsequently, the proposed changes are evaluated and selected by the Linux upper tier. Since the evaluation involves code reviewing as well as practical testing in the development version, it includes both types of learning processes. However, the Linux developer in the lower parts of the hierarchy does not participate in the evaluation effort, so his benefit in terms of learning is limited. In particular, the implicit part of the knowledge outcome of the evaluation is not communicated back to him. This suggests that the participatory, decentralized approach of FreeBSD is more supportive of the community's learning processes than the hierarchical approach of Linux.

5 Bugfixing by parallel debugging

Parallel debugging is a term coined by Eric Raymond to denote feedback to the project about errors that developers and other users observe as they use the software. It mitigates the need to allocate developers to debugging, which may be seen as less interesting and rewarding than new development. Parallel debugging produces a significant amount of feedback about the FreeBSD system. Nearly half the respondents said that, within the last month, someone else had reported a problem related to code they had worked on. This is consistent with the extent of the bug reporting activities observed in other open source projects, including Apache and Mozilla (Mockus et al. 2002).

Given the extensive feedback from parallel debugging, a crucial issue is its quality. Raymond's view on the significance of parallel debugging is summarized in his slogan *''given enough eye-balls, all bugs are shallow''* (Raymond 2000). This view implies that a community's diversified use of and feedback about the software

suffice as a basis for discovering the software's bugs, and that subsequent fixing of the bugs is relatively easy. In FreeBSD, however, there is indication that the feedback is sometimes quite insufficient. This was the case with FreeBSD's effort to introduce a feature called symmetric multiprocessing (SMP). The implementation of SMP is one of the most complex tasks ever undertaken by FreeBSD. SMP enables an operating system to allocate different program threads to execute simultaneously on a multi-processor PC, and this required substantial changes to the operating system's kernel. Work began around 2000 and was considered complete with the 6.0 release in 2005. The coordinator of work on SMP commented in 2001: ''*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 other developers to diagnose*.''

To implement symmetric multiprocessing, FreeBSD reused a basic architectural idea that had proven successful in another Unix-type open source project, BSD/OS (FreeBSD 2007e). The time required for the SMP effort reflects the complexity involved in embedding the general ideas in the specific context of FreeBSD. As von Krogh et al. (2005) have observed, such reuse of algorithms and/or methods is widespread in the open source world.

For quality control of the SMP effort, the developers relied on frequent building rather than on parallel debugging. Many immature changes related to SMP were added to the development version. The SMP developers themselves observed and corrected the broken builds and other problems they had caused, without relying on parallel debugging. In fact, broken builds were anticipated when the development effort was launched in 2000. The SMP coordinator said in a message to all developers that the development version of the software ''.. *will be destabilized for an extended period*'', indicating that the SMP developers assumed a right to interpret the ''don't break the build''-rule rather liberally.

6 Conclusions

FreeBSD's decentralized governance model appears to motivate the project's developers. This high motivation seems to spring from work characteristics like high autonomy and large skill variety. The project's main mechanism for quality assurance is frequent building, which is guided by the ''don't break the build''-rule. The project fears broken builds because they delay development, and when broken builds do occur, they spur an intense effort to correct the error. The committers generate most of the changes in FreeBSD, and any committer is expected to participate in the effort to correct a build he has broken. This effort may play a constructive role as a learning process that generates significant, and to a large extent implicit, knowledge about the system as a whole. Frequent building, with broken builds and their associated learning processes, may be a more important mechanism in FreeBSD than parallel debugging, since there is evidence that bugs may be observed by many, but the knowledge of key developers is required to fix

them. The hierarchy in Linux may exclude most of the contributors of changes from participation in learning processes such as those found in FreeBSD.

The applicability of FreeBSD's decentralized governance model to other projects may be limited for several reasons. If a project cannot reuse a design that is already known to be sound, an orchestrated design effort may be necessary. Also, if a project is able to enforce a strongly modularized design, developer knowledge of the system as a whole may be of less significance. In software engineering, modularization is the major approach to reduce software complexity, an approach that goes back to Parnas (1972) and is at the heart of modern object-oriented programming languages. Feller and Fitzgerald (2002: 171, 177) observe that several open source projects, including Linux and Mozilla, might suffer from weak modularization. This may apply to FreeBSD as well, and to some degree it may be caused by the project's lack of hierarchy to enforce and maintain strong modularization. This suggests that to some degree, FreeBSD's build process and the associated learning processes *compensate* for a lack of hierarchy in the organization and an entailed lack of modularity in the software.

References

Apache (2007). *Apache HTTP server project.* Retrieved March 12, 2007, from http://www.httpd.apache.org/ABOUT_APACHE.html.

Auyang, S. Y. (2004). *Engineering—an endless frontier*. Cambridge, MA: Harvard University Press.

Beck, K. et al. (2001). *Manifesto for agile software development.* Retrieved March 12, 2007, from http://www.agilemanifesto.org/.

Feller, J., & Fitzgerald, B. (2002). *Understanding open source software development*. London: Addison-Wesley.

FreeBSD (2005). *FreeBSD/i386 6.0 release notes.* Retrieved March 12, 2007, from http://www.freebsd.org/releases/6.0R/relnotes-i386.html.

FreeBSD (2006). *Core team election 2006.* Retrieved March 12, 2007, from http://election.uk.freebsd.org.

FreeBSD (2007a). *Committer's guide*. Retrieved March 12, 2007, from http://www.freebsd.org/doc/en_US.ISO8859-1/articles/committers-guide/.

FreeBSD (2007b). *Kernel developer's manual: Style*. Retrieved March 12, 2007, from http://www.freebsd.org/cgi/man.cgi?query=style&apropos=0&sektion=0&manpath=FreeBSD+6.2-RELEASE&format=html.

FreeBSD (2007c). *FreeBSD security information.* Retrieved March 12, 2007, from http://www.freebsd.org/security/security.html.

FreeBSD (2007d). *FreeBSD ports that you maintain which are currently marked broken.* (Message on a FreeBSD mailing list issued on January 29, 2007). Retrieved March 12, 2007, from http://www.lists.freebsd.org/pipermail/freebsd-ports/2007-January/038375.html.

FreeBSD (2007e). *FreeBSD SMPng project.* Retrieved March 12, 2007, from http://www.freebsd.org/smp/.

Jørgensen, N. (2001). Putting it all in the trunk: Incremental software development in the FreeBSD open source project. *Information Systems Journal, 11*, 321–336.

Lee, G. K., & Cole, R. E. (2003). From a firm-based to a community-based model of knowledge-creation: The case of the Linux kernel development. *Organization Science, 14*(6), 633–649.

MacCormack, A., Verganti, R., & Iansiti, M. (2001). Developing products on ''internet time'': The anatomy of a flexible development process. *Management Science, 47*(1), 133–150.

McConnell, S. (1996). *Rapid development*. Redmond, Washington: Microsoft Press.

Mockus, A., Fielding, R. T., & Herbsleb, J. (2000). A case study of open source software development: The Apache server. Proceedings of the 22nd International Conference on Software Engineering (ICSE '00), Limerick, Ireland, June 2000, 263–272.

Mockus, A., Fielding, R. T., & Herbsleb, J. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM), 11*(3), 309–346.

Nonaka, I. (1994). A dynamic theory of organizational knowledge creation. *Organization Science, 5*(1), 14–37.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM, 15*(12), 1053–1058.

Pressman, R. S., & Ince, D. (2000). *Software engineering: A practitioner's approach. European Edition, (5/e)*. London: McGraw-Hill.

Raymond, E. S. (2000). *The cathedral and the bazaar.* Version 3.0. Retrieved March 12, 2007, from http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar.

Von Krogh, G., Spaeth, S., & Haefliger, S. (2005). Knowledge reuse in open source software: An exploratory study of 15 open source projects. Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05), Track 5.

Winograd, T., & Flores, F. (1986). *Understanding computers and cognition: A new foundation for design*. Norwood, N. J.: Ablex Corporation.

Author Biography

Niels Jørgensen is Associate Professor at the Computer Science Department, Roskilde University. His main interests are technology theory, open source, and ICT-security. His current research goal is to understand the engineering aspects of computing, and in particular to capture similarities and differences between software engineering and engineering in classical, manufacturing disciplines such as aircraft engineering.