# A project model for the FreeBSD Project

A Candidatus Scientiarum Thesis at

the University of Oslo, Institute for Informatics

by Niklas Saers

# Table of Contents

## List of Figures

# Preface

This thesis is the result of my Candidatus Scientiarum degree at the Department of Informatics at the Faculty of Mathematics and Natural Sciences at the University of Oslo, in the period January 2002 to May 2003.

I have been a FreeBSD user since 1996, and while I am not currently developing free software, I consider myself a part of the BSD community. In my work I have been contributing to a number of other open source projects since 1997.

I would like to thank the librarians at the Department of Informatics for their great help in finding supporting literature.

I would like to thank my mentors Ingvil Hovig and Tone Bratteteig for their help during my thesis.

I would like to thank Dag-Erling Smørgrav for his kind support, both in terms of providing information and helping with the development of tools for statistics generation.

I would like to thank my family for their support, proof-reading and comments.

I would like to thank the FreeBSD Organization at the rpt.FreeBSD.org cluster for providing disk space and computing resources.

I would like to thank all my interviewees for their time and helpful answers.

I would like to thank all those who have read and commented on my work throughout the writing process for their kind words and their useful feedback.

I would like to thank Tatsumi Hosokawa for letting me use his artwork of the FreeBSD mascot for the cover.

Finally, I would like to thank all my friends who have joined me for coffees and heard me rant about my thesis. Without your help for venting, I could never have finished this thesis.

# Abstract

This thesis provides a baseline on which a methodology for the FreeBSD Project can be built. The three results of this thesis are:

A descriptive "project model" for the FreeBSD Project

A set of "quality goals" for the project model

A "comparison" between the quality goals and the project model, giving us the quality of the project model.

The project model is based on project documents, interviews, mail archives and the experience of the author in working with the project. The quality goals are based on this as well as supporting literature.

The discussion of project issues is backed up by a strong repertoire of theory in the field of software engineering.

The main findings of this thesis are:

The FreeBSD Project scores well on most of the defined quality goals

There are issues regarding the project organisation that needs to be addressed

# Chapter 1

# Introduction

## 1.1. The aim of this research

When adding people to a project, the amount they need to talk to one-another follows the curve (2^n)-1 where n is the number of people in the project [Brooks, 1995]. Tools can combat this communication overhead, and one tool that is considered effective is a methodology. A methodology is a set of normative documents saying how things should be done in the project, and the processes used are chosen because they are thought to be more effective and efficient for this project than other processes.

Every software development effort uses a methodology in some way or another, but it may not be very well articulated. Some have only a set of guidelines. Having a well articulated methodology, however, may have advantages such as making the project easier to "sell" to uninformed third parties that the project needs to approach for funding, support or to sell its services towards. It becomes easier in the sense that the third party can understand what makes the project tick, and this gives it credibility to the third party. This is in accordance with [Giddens, 1997: 32] who says that trust is gained through confidence in the person or organisation's predictability.

This research will study the FreeBSD Project, a project where the project members work on a voluntary basis to create a free, BSD-based UNIX-like operating system. This project has experienced a rapid growth of project members, and does not have a well documented methodology. The first aim of this research is to describe the FreeBSD Project, and I do this through creating a project model. A project model describes how the project is run, without making judgement of how it should be run. This model will serve as a basis for discussion about the current processes, and can over time be transformed into or lay the basis for a methodology. In terms of process improvement it will provide a baseline to which the project can be compared as it evolves and improves.

Although the project model is not normative like a methodology, it will still help reduce the communication overhead, reducing time needed to find out how processes are done and who should be contacted for which issues. In order to reduce the time effectively, it should be fast and easy to look up information in.

In order to discuss the current processes, I have chosen to measure its qualities. We will do this through stating a set of quality goals for the project model and measure how well the goals are achieved. Through comparing these findings to the quality goals, we will be able to say how well this project model works in fulfilling the goals.

It is outside the scope of the thesis to discuss alternative project models or recommend improvements to the current project model. It is also outside the scope of this thesis to provide a full methodology for the FreeBSD Project. As the FreeBSD project is based on voluntary work, cost will not be discussed in this thesis.

The scope of my study has been following the FreeBSD Project from April 1st, 2002 to April 1st, 2003. I will select a few subprojects that I, on the basis of interviews and reading from mailing lists, find suitable to describe, and thus not describe every subproject to the FreeBSD Project.

In summary, this thesis aims to:

- Create a project model for the FreeBSD Project
- Create a set of metrics to measure the quality of the project model
- Compare the project model with the metrics to say if the project model holds the desired qualities.

## 1.2. Motivation

While the main object of this thesis is motivated by reducing the communications overhead and increasing the marketing opportunities in this particular project, there are a number of other motivations as well.

[Cockburn, 1998] provides the following figure for how effective communication is through different mediums:

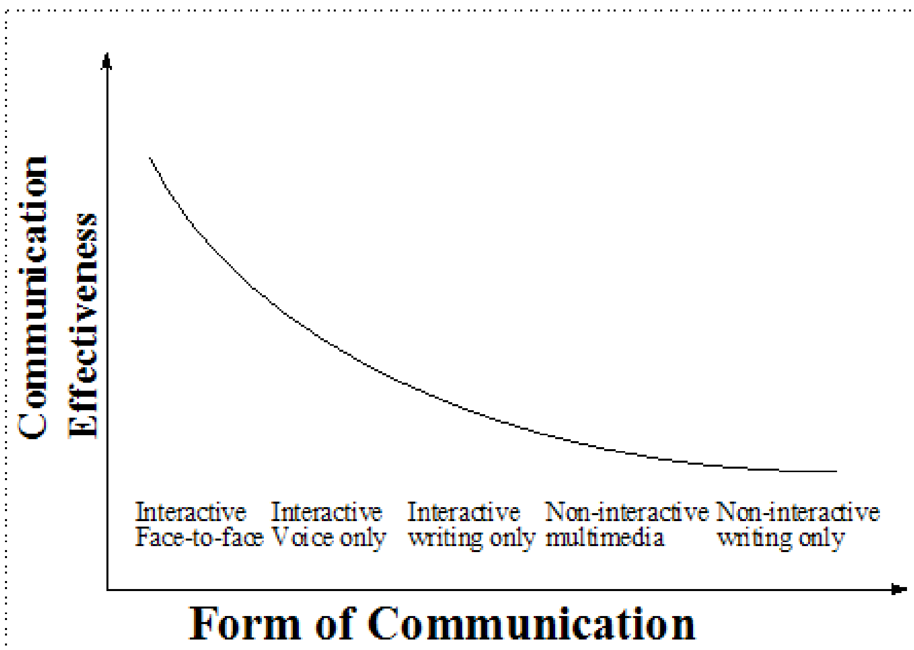


**Figure 1. Effect of communication through different mediums**

Since most of the communication in the FreeBSD Project is mailing lists, its effectiveness is that of "interactive writing only". Less effective communication means more communication to communicate what is needed. Reducing the communications overhead through a project model is therefore a real time saver for the people in the project.

[Cockburn, 1998] also provides a model for when what kind of methodology is needed.

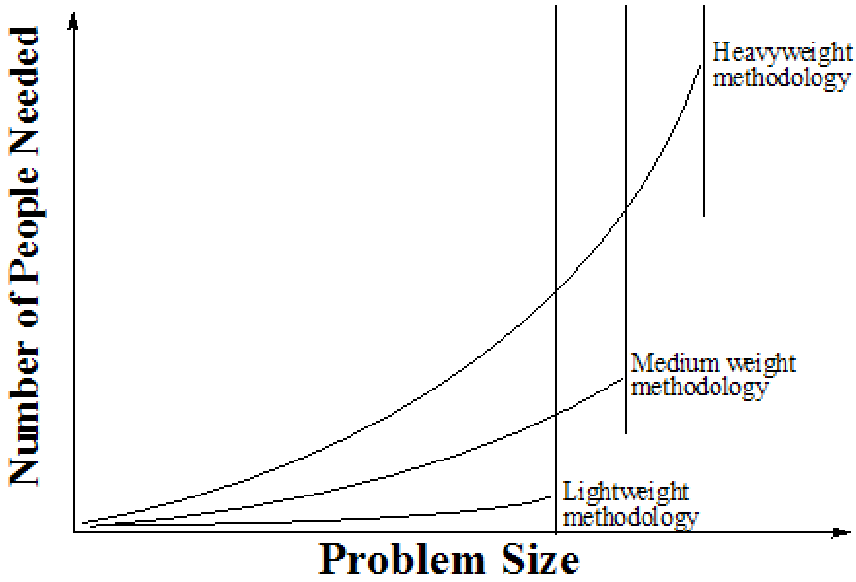


**Figure 2. Methodology weight needed for different sized projects**

In Section 4.2 we will see that there has been an increase in active developers in the FreeBSD project during the past few years. With them come more ideas of what the project should be, and thus an increase in problem size. From my interpretation of [Cockburn, 1998] this means that while the project could use a lightweight, tacitly understood methodology a few years back, it is time to develop a project model now.

Open source development is a little understood phenomenon. What has been researched has mainly been focussed around the most famous project, Linux. Many researchers talk about "the open source model" as if it were one model, and usually refer to the Bazaar model [Raymond, 2000] which Linux has been explained by. To gain a better understanding of open source, we need to examine more projects closely. FreeBSD, a popular open source project known as a rock-solid and feature rich operating system, is a good candidate. With only one article written on the FreeBSD projects approach to software development, this study is important as there is much to learn from how it runs its software development.

Finding more software process models that work well with the open source community allows us to choose between more software process models when working in an open source environment. This allows companies that want to explore the possibilities of open source to choose a model that is easier to integrate into their company. When projects want to go open source or open source projects are created, it is good to have a set of software process models to choose from and see what model would work well for what project and thus make an informed opinion on what model to choose.

FreeBSD has a long history, and is an offspring of BSD, an operating system that received much attention and use in academia until its end in 1994. Studying FreeBSD gives us the opportunity to contrast today’s practises with how they used to be.

The BSD projects come from a tradition of using hierarchical organisational structures. This contrasts the more flat and unstructured Bazaar model, and many people in the open source community perceive this as an elitist project. Even so, the project attracts new developers. The FreeBSD Project is therefore interesting to study to broaden the view on what organisational models fit into the open source culture.

BSD has since its beginning been an operating system in which research results have been quickly incorporated. Rather than being focused on backwards compatibility, it has worked as a test-bed for researchers in computer science. A kind of evolution has taken place where the best technologies have been kept and where new technologies all the time can be integrated and evaluated. The

FreeBSD Project has maintained this spirit, while maintaining its reputation of delivering a rock solid product. Other projects can learn from the processes that allow this.

The development of BSD pioneered both open source and internet based development. The FreeBSD Project is a continuation of this development. What has changed, and how does it organise its development today?

The open source BSD derivatives NetBSD and FreeBSD pioneered net based distribution and large-scale distribution on cheap CD-ROM media. The ease and low cost together with high availability has been seen as one of the success factors for open source projects. How does the FreeBSD Project handle its distribution today?

FreeBSD has a need to be understood by the corporate world. A quote from one of the project founders, Jordan K. Hubbard, illustrates this: "My biggest frustration is that more of [...] the corporate world [...] hasn't been quick to jump aboard with personnel and material resources. They don't see this, as I do, as a powerful collaborative model rather than just a bunch of guys doing a free OS. I look at the track record of more traditional collaborative efforts like the OSF [(Open Systems Foundation)] or the ACE [(Advance Computing Environment)] Consortium [...] and I already see far greater success with groups like the FreeBSD Project or the Linux International folks". [Laird et.al, 2001]

One of the main reasons for the resignation of both Jordan K. Hubbard and Mike Smith was that from being a fun project, FreeBSD became to them associated with enormous amounts of conflict and bureaucracy [Daemon News, 2002]. As the project grew larger, the communication needs increased. This is where [Cockburn, 1998] finds it necessary to use a methodology as opposed to a loose collection of techniques. A methodology needs a starting point describing what is today, and this project model will provide such a starting point.

If measured by the number of users, much of the code in BSD has had a large success. The influence on commercial Unix systems and Linux is tremendous: half of Linux' utilities come from BSD and most of the advantages pioneered by BSD have been incorporated into commercial Unix systems. Apple's new operating system, Mac OS/X, relies on the BSD derivative Darwin. Even Microsoft has copied many of its utilities and features, such as the FTP client distributed with Windows [Daemon News, 2001]. The network stack TCP/IP has become the network standard of the internet. This success suggests that we should examine its development closely and see what we can learn.

Studying the evolution of the methodologies as the projects evolve can help us understand what factors make the methodologies change. If we have a plan for how our project will evolve, we can examine these factors in choosing what methodology to start out with. Since the costs for changing methodologies midway can be large, taking measures to ensure that few changes of methodologies are needed will save the project time and money.

## 1.3. Thesis overview

The case study we will examine is the FreeBSD Project. The FreeBSD project consists at the time of writing of 275 members with special privileges and many associated developers and users through mailing lists and community web-pages.

This thesis consists of three logical parts:

- "Framework" - this introduction, the context the context the thesis is framed within and the methods used for making this thesis
- "Aspects" - three aspects of the FreeBSD Project: the organisational structure, administration of the project and the development model used in the project. This will be followed by a discussion chapter, putting the aspects together and discussing the findings.
- "Project Model" - the resulting model for the FreeBSD Project and its evaluation

The "Background" chapter gives the context of both the FreeBSD Project and this thesis. The context of the FreeBSD Project is set through discussing the history of the original BSD and the historical events that led to the FreeBSD Project that is today. Then comes an introduction to the Open Source community that the FreeBSD Project is a part of. Having summarised FreeBSD's context, the theory that is the context for this thesis is presented.

In the "Research Methods" chapter, the different research methods used are detailed and discussed. After this, metrics are set for how to measure the quality of the project model, and finally the project model development process is discussed.

The next part consists of three aspects of the Project Model and a discussion tying these parts together. Detailing the entire project is outside the scope of this thesis. The three aspects "organisational structure", "administration" and "development model" are detailed more closely, being a more in-depth examination than the project model is. This is because the Project Model is intended to be easy to use, and not provide a detailed explanation of why things are the way they are.

Based on the three aspects and the following discussion, the research done in the way listed in the "research methods" chapter and developed in the way discussed in the research methods chapter, the project model is presented. This model should be accurate as of April 1st, 2003.

Finally we will discuss how the project model compares to the quality metrics set up, and discuss the project model. Lastly we will gather the conclusions found in each chapter and make the final conclusion.

# Chapter 2

# Historic and academic context of this thesis

This chapter serves an introduction into the literature that supports my topic. It will also describe the context of the FreeBSD Project. It will, after having clarified key terms, describe the history of the FreeBSD Project and the open source community it is a part of, then discuss aspects of project organisation, and finally come to terms with how we will understand quality.

## 2.1. Concepts used in this thesis

Many of the words in the research aim are used daily where some words have more than one meaning depending on the context in which they are used. To remove ambiguity I will define how these words will be understood in this thesis.

A "project" is according to [PMI, 2000] a planned, “temporary endeavour undertaken to create a unique product, service or result". In this research, a project will refer to a logical unit of software development effort. Projects can contain sub-projects that are a project in their own right, and which outcome is planned to form a part of the main project.

A "model" simplifies something by addressing only the features important for the context of the model. It can be either descriptive, saying how that something is right now, or normative, saying how that something should be. Models are made to give an overview. The descriptive model does so on the basis of experience, while the normative model gives an overview of how that something should be and implies an evaluation that this is better than the alternatives. The descriptive model implies no such thing.

A "methodology" is a normative model, guidelining what processes and deliverables should be used for a project and in what sequence they should come. It includes what tools should be used to build the deliverables and templates for the deliverables. [Smevold, 2001: 21-24]

A "project model" is a descriptive model of how a project is organised. It can be based on monitoring the project statistically, interviewing project members, observation, document analysis, or a combination.

According to [Sommerville, 2001], a "software process" is "a set of activities and associated results which produce a software product". The four common activities to all software processes are:

- Specification - The definition of required functionality and constraints on the software
- Development - The software must be created to meet the specifications
- Validation - Confirm that the software developed is what the customer wants
- Evolution/Maintenance - The software must be updated to meet changing needs from the customer

It is worth noting that although the software processes have these features in common, the results are not the same for these activities. For instance, a software process that relies heavily upon iterations will have a different result for the specification activity than one relying on a set sequence of actions.

A "software process model" is a simplified description of the software development process, an abstraction that can describe one or more software processes. This is used both to communicate the software process without communicating it in its entirety, to classify the software process and as the basis for creating a software process for a new development effort. Examples are the waterfall model, the spiral model, incremental development and extreme programming. [Sommerville, 2001: 7-14]

Figure 3 helps make the difference between how the four concepts are used in this thesis clearer:

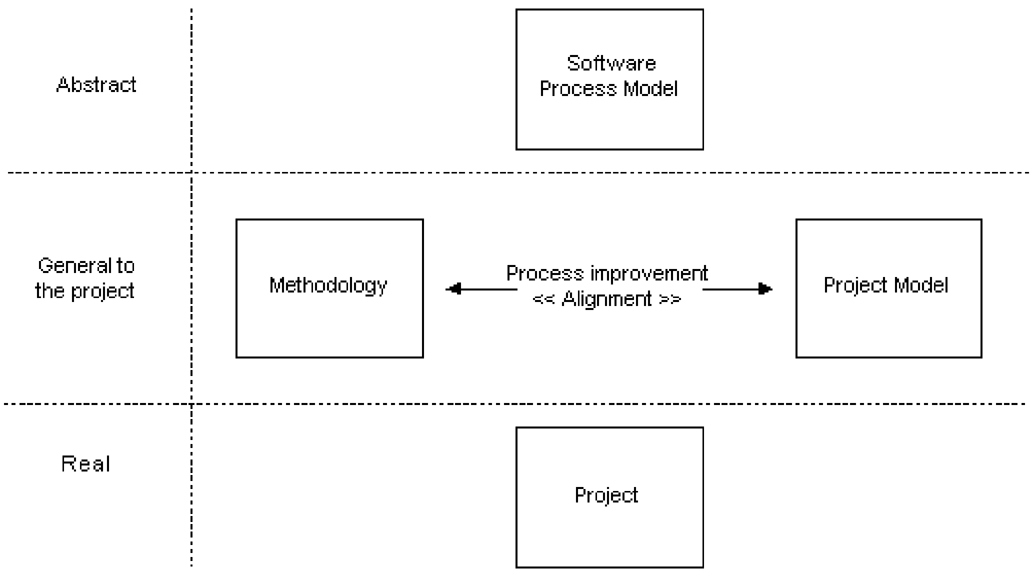


**Figure 3. The relationship between the four terms**

The software process model is an abstraction of how a methodology can be made. It can be compared to Plato's idea "horse" when the methodology is compared to the actual horse. The project model is a generalisation of how the project is right now, while the methodology is a plan of how the project should be. In terms of process improvement, the project model is the current status and the methodology is how the project should be run. Doing process improvement would thus be aligning the two.

By "Berkeley", the University of California at Berkeley is meant unless otherwise explicitly stated.

By "BSD", the Berkeley Software Distribution and its project at the Computer Science Research Group ("CSRG") is meant.

## 2.2. FreeBSD and Open Source

The FreeBSD Project is an open source project. In order to understand what this means, we need to understand where FreeBSD is coming from and what the open source culture brings. This section will first present a history of BSD development, and will then continue to investigate the open source community.

### 2.2.1. The History of BSD development

#### *2.2.1.1. Early Days - the Berkeley Computer Science Research Group (CSRG)*

UNIX, originally developed at Bell Labs in 1969, was rewritten in C, rather than the original assembler, in 1973 by Dennis Richie and Kenneth Thompson. Since its presentation, professors at Berkeley had taken a strong interest in UNIX. With their first running UNIX machine early 1974, strong links were built to the UNIX communities at Bell Labs. Old ties to batch processing systems made the UNIX running machine have to run a batch job system for 16 hours and UNIX for 8 every day. With the success of Berkeley's Ingres database written for UNIX and increased demand by students, the computer science department ordered a new machine to run UNIX only.

Together with the new UNIX machine, Bill Joy and Chuck Haley started as graduate students. They wrote a Pascal system and a number of utilities and editors such as 'em' and 'vi'. The Pascal system became popular among the computer science students because of its good error handling system. Joy and Haley took an interest in the kernel when installing a tape with 50 changes to the kernel supplied by Bell Labs. These changes together with the Pascal system and their utilities was put together to form the "Berkeley Software Distribution" by Joy early 1977. This feedback and changes together with new features such as extensive support for terminals led to the second version, known as 2BSD in 1978.

As the need for larger address spaces for user programs came, the computer science department ordered the newly announced VAX-11 in 1978. Since the department was used to UNIX, and UNIX with virtual memory support was not available for the VAX, Ozalp Babaoglu and Bill Joy ported the virtual memory support to V/32, the VAX UNIX version. Seeing that this architecture would obsolete the older computers in the department, Joy began porting the 2BSD software to the VAX. The new kernel enhancements and utilities led to 3BSD, the first VAX distribution from Berkeley.

With 3BSD, Berkeley had proved its ability to create and maintain a working system and was able to land an 18-month contract with DARPA to create the operating system that should run on multiple hardware platforms so that DARPA would have a unified platform at the operating system level. This provided funds to employ Laura Tong to do the project administration. She set up a distribution system that could ship a much larger number of copies: 150 for 4BSD and 400 for 4.1BSD. Being satisfied with the results, DARPA continued its funding and a steering committee was set up to guide the design work and ensure that the needs of the DARPA research community were met.

Apart from being a credible actor, the research community needed the CSRG at Berkeley and BSD. Bell Labs and its UNIX had long had the role as a clearing house for computer science research. Many researchers were interested in UNIX which made their research results often either UNIX specific or easily integratable into UNIX. But with Bell Labs' commercialisation of UNIX in 1979, Bell Labs could no longer play this role, and the CSRG soon stepped into this role. [McKusick, 1992]

The work to implement the requests required much redesign and implementation of TCP/IP based network services. This led to an internal intermediate release, called 4.1a. However, many people grew impatient waiting for 4.2BSD, and this led to many copies of the system. Based on the feedback from 4.1a, a proposal for the new system, called "4.2BSD System Manual", was made. It contained a concise description of the proposed user- and application programming interfaces.

When Joy left for Sun Microsystems in the late spring of 1982, Sam Leffler was given the responsibility to finish the project that had been promised to the DARPA community by Spring 1983. To conform to this deadline, the remaining projects were evaluated and strict priorities were set. April 1983, an intermediate release, 4.1c was released, and after reworking the I/O system and making the install process easier, 4.2BSD was released in August 1983. [McKusick, 1985]

As with 4BSD, many people were critical to 4.2BSD because of performance issues. As with 4.1BSD to 4BSD, 4.3BSD, released in June 1986, would resolve the issues causing the stir.

A requirement for using BSD was that a System V license was held. AT&T had increased their fees much and vendors who wanted to use the TCP/IP-based products that Berkeley had pioneered, lobbied CSRG to provide them under terms that did not require such a license. In response to this, Networking Release 1 (Net/1), consisting of the original networking code and supporting utilities, was released in June 1989 under a liberal license. While Berkeley charged a $1000 fee to provide a tape, the code could be redistributed by anyone as long as the copyrights in the source code was kept and the use of Berkeley's source was acknowledged in the documentation. This led many sites to provide public access to the source through the network, and this made the release very popular. Also, institutions regarded this as a way of supporting CSRGs research, and many hundred tapes were bought.

Inspired by the popularity by Net/1, Keith Bostic suggested that making a release that included more BSD code. However, Marshall Kirk McKusick and Mike Karels noted that this would require hundreds of utilities and libraries to be rewritten and the kernel to be closely examined and partly rewritten. Rather than giving up by the size of the effort, Bostic pioneered large-scale net-based development by approaching people to rewrite Unix utilities based only on their published descriptions. With his continued effort to solicit people, within 18 months almost all the utilities and libraries has been rewritten. Keeping McKusick and Karels to their promise, CSRG went through the kernel code file by file and removed code originating from 32/V. This effort culminated in the release of Network Release 2 that included a fully working system except for 6 files (out of 18.000) that had been deemed to difficult to rewrite.

As with Net/1, Net/2 became widely spread. Within 6 months, a version bootable on the Intel 386 Architecture had been made by Bill Jolitz who had rewritten the 6 files. This sparked communities that would lead to the major BSD open-source projects at the time of writing: NetBSD, OpenBSD, FreeBSD and Darwin.

Based on Jolitz six files, the company BSD Incorporated started shipping a commercially supported version. They marketed it as being a system as costing only 1% of what System V cost with source and binaries. Unix System Laboratories (USL), which was mostly owned by AT&T and owned the Unix trademark, quickly demanded they stop marketing their product as Unix. BSDI complied, but still unhappy about the competition, USL filed suit to stop BSDI from selling their product, claiming that it contained USL code and trade secrets. BSDI defended that they only used code publicly available from Berkeley and 6 other files, which led USL to include Berkeley and its Net/2 in the suit.

After six weeks of advisement, the judge ruled in January 1993 that only two of the complaints should be brought to court. These complaints should be heard in a state court rather than a federal court. Berkeley followed up with a counter-suit against USL for using code from Berkeley without stating so in their documentation. Soon after the filing in state court, USL was bought by Novell whose CEO, Ray Noorda, stated publicly that he would rather compete in the marketplace than in court. A settlement was reached in January 1994 that resulted in three files being removed and numerous minor changes and added copyright notices. With the settlement, USL promised to not sue any organisation that used the new release called 4.4BSD-Lite. [McKusick, 1999]

4.4BSD was the final release from CSRG. The three reasons were

- The time required to attain funding had increased dramatic, limiting the time the scientists could spend working on BSD. In 1994, computer corporations were not prioritising funding such research. Rather, they preferred in-house research of which results they had the sole ownership of. Since BSD was freely redistributable, the revenue from distributions sold was small.
- BSD became a victim of its own success in that the features pioneered by BSD were included in most commercial operating systems.
- Space constraints at Berkeley as well as the lack of funding limited the group from recruiting more people than the four involved. With 4.4BSD, the project had grown so large that four people could no longer architect and maintain it.

[McKusick, 1994]

#### *2.2.1.2. FreeBSD*

March 29th, 1992, Karl Lehenbauer published the FAQ for the 386BSD, which is by many recognised as the start to make the Berkeley Net/2 release available on the Intel architecture. Version 0.1 was made available in June and a community grew quickly. However, the author, Bill Jolitz, did not maintain the software much. This led Jordan K. Hubbard, Nate Williams and Rod Grimes to create the "Unofficial 386BSD Patchkit". The maintenance of this kit, however, came to a halt when Jolitz withdrew his sanction from the project. In response to this, the FreeBSD Project, name coined by David Greenman, was started.

The first FreeBSD distribution was released in December 1993, based on 4.3BSD-Lite but also included work from 386BSD and by the Free Software

Foundation [Kolstad, 1994]. After being updated to version 1.1 in May, 1994, the project was a success. However, at this time the lawsuit between AT&T and Berkeley had been settled, and the project had to switch to the 4.4BSD-Lite code base from Berkeley, leaving the encumbered 4.3BSD-Lite code behind. Because the Intel version of 4.4BSD was highly incomplete, it was not until November 1994 that the project could release version 2.0 [Hubbard, 1994].

Since version 2.1, FreeBSD has enjoyed a reputation as a stable, feature-full UNIX system for the Intel platform. Kirk McKusick describes the evolution of FreeBSD as a random path walk. This is because new features have been developed on the initiative from the developers rather than through decisions by the core team. The project has since its early days been split it two main branches. FreeBSD-STABLE is the branch that maintains the current working release that is offered to the public for production use. FreeBSD-CURRENT is the branch where the main development happens.

The FreeBSD Project was hosted and to a large degree sponsored by Walnut Creek CDROM during its initial years. In March 2000, BSDI merged with Walnut Creek CDROM to form a united front for the BSD operating systems. In April 2001, Wind River bought Walnut Creek CDROM to acquire the rights to BSD/OS, and decided in October 2001 to end its relationship with FreeBSD, a relationship it had got when acquiring Walnut Creek CDROM [Reed, 2001]. The FreeBSD trademark and related rights were transferred to FreeBSD Mall Inc in January 2002.

### 2.2.2. Open Source

Open source projects distinguish themselves from normal, commercial projects (also called closed source projects), in that the source code is distributed either alone or upon request, either for free or for a nominal fee to cover the distribution costs. With the internet, the distribution costs are often so amortised that most open source projects distribute both its binaries and source for free.

An important philosophy of the open source movement that comes from the Free Software Foundation and university projects such as BSD is that open source software should be self-hosting. This means that the operating system, the compilers and the utilities needed to make, compile and run the product should be open source. FreeBSD fits into this by being an open source operating system and providing most essential compilers and utilities, either their own or made by other open source projects.

[Hars et al, 2001] talks about "the open source development model", a term that in magazines and by many open source supporters is used in contrast to software process models. However, comparing these is meaningless. The open source development model refers to a significant detail in the project, that the source code will be available. The equivalent detail in commercial projects is that the project in some fashion contributes to the financial gain of the company that creates it. It makes as much sense to contrast software process models to the open source model as it makes to contrast them to the commercial model.

Other features common to open source projects is that they have internet based communities and the authors give the public the right to redistribute and modify the source code free of charge. [Jørgensen, 2001] discovered in his research that 43% of the developers had been paid to some extent for their involvement in their last project. In [Hars et al, 2001], 50% of the surveyed open source developers were paid to some extent. Although both surveys have a relatively small group of respondents, the fact that companies such as Sun, Ximian, IBM and Trolltech have contributed many projects to the open source community suggests that many developers are being paid for their time. The myth that participants do not get direct compensation for their work is therefore not necessarily correct.

However, a large part of the open source community does not get paid for their contribution. [Hars et al, 2001] has studies what motivates people to contribute to open source projects in general. They find that students and hobby programmers, the group that does not get paid for their involvement at all, the fun of having it as a hobby and expanding their skills, capabilities and knowledge are the most motivating factors. The gratification of such a hobby is explained by referring to Maslow's pyramid and the need for a stable, usually high evaluation of oneself. The increased human capital promises a future reward of being able to get a good job in a business which the participant is interested in.

According to [Raymond, 2000], all good programs come from scratching someone's itch. However, only a little more than a third of the respondents to [Hars et al, 2001] claimed that a personal need for the product was a main reason for participation.

The open source community prides itself on writing secure, well-reviewed software. [Raymond, 2000] quotes Linus Torvalds saying that "given enough eyes, all bugs are shallow". This is a paradigm in the community and a common belief that has many examples where problems have been discovered and fixed within hours of having been discovered. We will revisit this review in the verification techniques used by the FreeBSD Project to see if this holds true for this project.

## 2.3. Project structures

### 2.3.1. Software processes

As defined in our introduction, a software process model is a simplified description of the software development process. In being an abstraction, it can fit many software development processes whereas methodology usually only fits one project. Software process models help us both classify development efforts and serve as a basis for developing a methodology for new projects or projects whose methodologies are being re-engineered.

As outlined in our definition of a software process, it can consist of multiple sub-processes. A software process model is often presented as a structured set of sub-processes, so before we outline the most common software process models, a description of common sub-processes is in place.

#### *2.3.1.1. Requirements analysis*

During requirements analysis, the requirements are interpreted by the developer and an alignment of the understanding of the problem by the developers compared to the formal definition of the problem laid out in the requirements. This stage often needs the transformation of requirements written in a human language (for instance English) to a common technical and unambiguous notation, often one that can be interpreted by computers. Such models will not only align the understanding of the problem at hand between developers, but also determine whether the requirements are complete, contradictory or too ambiguous.

#### 2.3.1.2. Software design

Design consists of forming an overall architecture for the project and reducing parts of the model to sub-problems that are tractable and simple enough in their complexity to have the full scope of the sub-problem in a developers mind at a time. For this, technical figures, design document, interface descriptions, user interfaces and such are created. A very much used tool for the design stage that includes all of these is use cases.

#### 2.3.1.3. Implementation

The only phase that is certain to be within a successful development effort is the implementation. Even with the most sophisticated Computer Aided Software Engineering (CASE) tools used for design, there are always parts that have to be implemented in code.

#### 2.3.1.4. Verification

Verification is the verifying that the developed system does as designed, that the design completely covers the requirement and that the system developed thus completely covers the requirements. Also, testing that the program does not misbehave and behaves predictably is a part of this phase.

#### 2.3.1.5. Integration

Integration is the deployment of the system at the customer, seeing that it integrates in the environment of the customer.

#### 2.3.1.6. Maintenance

The software engineering skill that is probably least taught at university to computer science student is software maintenance. The absence of this process in many software process models shows the way it is regarded in the software industry. [Boehm, 1973] states in his article that almost 40% of the software effort in Great Britain in 1973 was maintenance, and he predicted the figure to rise.

According to [Canning, 1972], causes for initiating software maintenance are "program won't run," "program runs but produces wrong output," "business environment changes", and "enhancements and optimisation". [Swanson, 1976] classifies maintenance into corrective maintenance, adaptive maintenance and perfective maintenance.

Corrective maintenance is due to processing failures (i.e. abnormal termination, erroneous output in reports or files), performance failures (not meeting the performance criteria specified) and implementation failures (implementation not following coding standards, not adhering to design etc).

Adaptive maintenance happens due to environmental changes. Especially, change in data (i.e. a logical restructuring of the database used) or processing environments (new hardware or software architectures implemented) lead to adaptive maintenance.

Perfective maintenance is done to improve the maintainability of the code, processing inefficiency (i.e. due to poor use of the operator's time) and performance enhancement (increasing readability of reports, adhering to new data presentation standards).

In his article, [Swanson, 1976] writes about the importance of keeping a maintenance database. A maintenance database is a database that keeps track of all problem reports, preferably both submitted by users and developers. This database should cover all issues that are interesting for the project to track. Such items are usually the problem description and a detailed description of how it occurred, preferably with debugging information. But it can also include who is responsible for the bug, where in the code it was found, how long time it took to be resolved and so forth.

[Canning, 1972] has written on the "maintenance iceberg" that is due to bad coding practises. Since then, people have been taught what is considered better coding practises and more high level programming languages have become available, thus ideally making software more maintainable. However, the maintenance issue has not been reduced. But just as [Canning, 1972] dared not measure the iceberg in 1972, I have not found articles saying how large the iceberg is today and how it has evolved since 1972.

What is true is that the software engineering field has become more maintenance aware and authors like [Sommerville, 2001] have included it in the description of modern software process models. Indeed, many agile software process models are said to be maintenance centric, seeing every change as a maintenance change.

The world of open source is also aware of the maintenance issue. [Tatham, 1999] writes on reporting problems effectively, enabling the author of a program to find and resolve problems. The open source community has been good at using maintenance databases like GNATS and Bugzilla. Indeed, open source hosting sites, such as Sourceforge, often come with bug reporting, version tracking and user discussion systems allowing for ongoing maintenance in all active projects.

### 2.3.2. Common software process models

There exist many software process models. The most used models, that again are parents to many more refined models, are the stepwise, waterfall, evolutionary, spiral, transform, incremental and agile models. This section will briefly summarise these

#### *2.3.2.1. The stepwise model*

With the introduction of assemblers and compilers for higher level programming languages, it became possible to write large systems. Problems of testing and documenting the systems led to the introduction of the "stepwise model". This model has eight stages that software development needs to go through in a linear order. The model is heavily document-driven: Royce gives the example of 15.000 pages for a system consisting of 100.000 instructions. [Benington, 1987]

#### 2.3.2.2. The waterfall model

The "waterfall model" is a refinement of the stepwise model. The two primary enhancements are feedback loops between the stages and the building of a prototype. The feedback loops is the realization that the current stage is an evaluation of the work done at the previous stage, and findings in this stage can force us to go back to a previous stage to do changes. The "build it twice" philosophy of the waterfall model builds a prototype to explore difficulties such as technical uncertainties or to gain understanding of how the user wants the interface. The main problem with the waterfall model is the criteria of the completion of fully elaborated documents. While this criterion is good for products that can easily be formalised, such as compilers and spacecraft controllers, this does not work well for end-user applications. Since the uses and interfaces of end-user applications are often poorly understood by the development staff in the beginning of a project, there is no purpose to writing the specifications in great detail. [Royce, 1970]

#### 2.3.2.3. The evolutionary model

The "evolutionary model" assumes that there is software out there that with modifications can become the software the user wants. This approach matches well with situations where the user knows he or she needs a product but is not certain about how this product should be. This kind of development has been made less costly with the introduction of fourth-generation languages. There are mainly three problems with evolutionary programming:

1. Like the code-and-fix model, the code can easily turn into an unmanageable mess. This can lead to temporary fixes and features that the programmer assumes are nice but which are not used in the product.
2. The model assumes that the operational environment of the customer is flexible enough to follow the evolutionary path the process takes.
3. Like code-and-fix, the system can become poorly documented and structured, which will lead to problems when it has to be integrated with other applications or is going to be phased out and the data need to be transferred to another system.

[Boehm, 1988]

#### *2.3.2.4. The transform model*

The transform model approaches the problem of spaghetti code that easily results from the many of the before mentioned models. It assumes that formal specifications can be transformed into code. It is an iterative process where the first goal of the first iteration is to make a formal specification with the best initial understanding of the system. The following development iterations are spent changing the specifications based on the customers input and optimising the code by giving more detailed guidelines to the tool converting the specifications into code. When the program is deployed, further iterations are used to adjust the specifications based on operational experience. In this way the transform model bypasses the problem of poorly structured code that is hard to modify. While this sounds very promising, it has proved hard to build tools that will convert a specification into complete code for other than just small projects. [Saers, 2002]

#### *2.3.2.5. The incremental model*

The "incremental model" [Mills et al, 1980] gives the customer the ability to delay decisions until he has some experience with the system on which he can make the decision. The customer identifies which parts of the project are the most important to them, and these parts get developed in early increments. Each increment is a delivery that the use customer can put into production. Each increment is a small waterfall model, and this way specification, development, verification and validation can be processes that come over and over. The customer will get the necessary documents to keep an overview of the project status. The use of increments, the system being put into production and the customer overview minimise the risk of the project failing due to miscommunication. One major problem with the incremental model is that it is difficult to estimate how large an increment should be. Also, since the requirements do not come all at once, it is difficult to identify and build general structures that will make it easier to code the system.

#### *2.3.2.6. The spiral model*

The "spiral model", known from [Boehm, 1988], is an incremental model focused on risk. It is illustrated as a cycle going outwards where each cycle involves risk analysis, prototyping, simulation, design, validation, planning the next cycle and evaluating alternatives. The risk analysis, prototyping, simulation, validation, and alternative evaluation all work towards minimising risk of project failure, and the cycles are short enough that if the result of the analysis is that the project should be halted, then there is an opportunity to halt it at the end of every cycle. As with the evolutionary model, prototyping is used heavily, ending in an operational prototype given to the customer. [Saers, 2002]

#### *2.3.2.7. Agile models*

The most resent addition to our repertoire of models is "agile models". These models focus mainly on handling change and the most well known of these, extreme programming, defines change as the norm and stability as an abnormality. Extreme programming is a set of 13 practises that together reduce project risk through improving the response to change and improves productivity through making building software in teams more fun. It can be summarised as a series of many stepwise models where the time to go through the model is at most a couple of days, usually about a day. In practise, this means that design, implementation and verification are ongoing processes. [Saers, 2002]

### 2.3.3. Methodologies

Just like two people living on an island can live together fine, a society with many individuals needs rules and regulations in order to function well. While two people can use a technique or method to do a job, [Cockburn, 1998] argues that for 20 people to do the job together, a methodology is needed. With an increased number of people involved, the need for communication becomes greater. Two people talking together use little time to synchronise their work, but 20 people who have to talk to everyone to be synchronised will spend all their time talking. A methodology greatly reduces the amount of communication needed to be done in order for the project to work satisfactory.

A methodology is the realization of a software process model within a project. While the model is just that, a model of how a project can be structured, the methodology is how this project is organised with all the details involved.

A methodology is to a software process model what an object is to a class in object-oriented programming. That is, it is an implementation of the model followed. [Cockburn, 1998] identifies nine elements that are usually described in methodologies. The elements [Cockburn, 1998] lists are:

- *Roles* - A description of duties and responsibilities within a project. A role can typically be something asked for in a job description, but one person may have many roles within a project. I.e. requirements gatherer
- *Skills* - Applied knowledge that enables a person to fulfil the duties of a role. I.e. object-oriented design
- *Techniques* - Methods for doing a piece of work. This is what [Hohmann, 1997] calls process. I.e. use case modelling. When working to reach a

particular deliverable as a group, a more fitting word is process. I.e. requirements analysis

- *Teams* - How you group people and how you assign them to roles.
- *Tools* - What artefacts are used in your technique to do a piece of work. I.e. a compiler
- *Deliverables* - The output from the technique or process used. [Hohmann, 1997] calls this outcome.
- *Standards* - What is permitted and not permitted and what conventions should be followed. I.e. coding standards
- *Activities* - what meetings, reviews, milestones and other general activities the person must attend, generate or do. I.e. create delivery
- *Quality* - what rules, issues or concerns are to be tracked for each deliverable.

Methodology weight refers to how much is specified in the methodology. For instance, if coding standards are very thorough, this adds to the methodology weight and allows for less tolerance for variations than if more coding standards issues were unstated. This would make it more open and allow for higher variation tolerance. More thorough standards are more precise, and precision thus adds to methodology weight. [Cockburn, 1998]

Projects can have different criticality level. The levels of criticality are classified by the result of product failure. The four levels that [Cockburn, 1998] and others refer to are (1) nuisance, (2) loss of a recoverable sum of money, (3) irreversible loss of a large sum of money and (4) loss of life. Cockburn's 4th principle is that the more critical the product made by the project, the heavier methodology is called for. The additional weight is added as a result of greater precision and less tolerance for variations.

The methodology scope is the range of roles, activities and project deliverables the methodology attempts to cover. The deliverables follow the project's life cycle. [Cockburn, 1998]

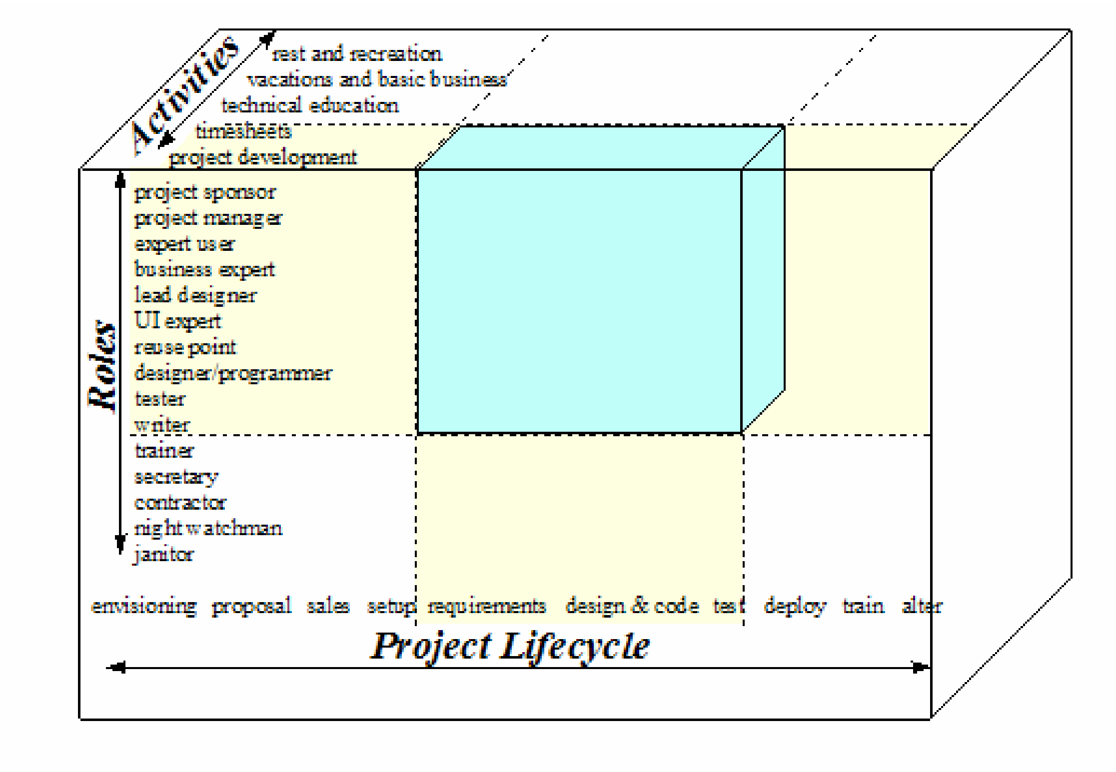


**Figure 4. Scope of a Methodology**

It is important not to confuse project activities with the core organisation's activities. While handling requests for vacation is a part of the core organisation's activities, planning how to handle people being unavailable is part of the project manager's activities.

## 2.3.4. Process development and evaluation

[Osterweil, 1987] writes in his article that software processes are software. In a follow-up paper, [Turk et al, 2000] shows that software process models are software too. Software processes are based on a process model and are key parts of a methodology and a project model. Thus, methodologies and project models can be considered to be software.

Osterweil notes that software processes are dynamic and therefore much harder to comprehend and reason about than static process descriptions. This thesis will look at the processes in FreeBSD as a snapshot in time, thus looking at them as static so that they can be analysed. In reality, discussing the processes contributes to their change or reenforcement.

Because the project model is considered software, we can use a software process model to create a project model. We will discuss the software process model used in Section 3.4.

Since software processes and software process models are considered software, they are victims of the same quality criteria as software. One of these to be particularly careful about is "rotting". Rotting is referred to when code becomes obsolete because identified problems are not fixed or the context in which the software is used evolves while the software is not updated. The same is true for methodologies and project models. If they are out of date, the information becomes unreliable. When parts of the information are unreliable, the users of the documents do not know which parts they can rely upon or not, making them not use it at all. This is actually worse than not having a methodology or project model as the trust in future, updated and maintained models is at stake. Examples of changes that happen are: new roles are created, roles merged, responsibilities broadened or split, techniques changed and communication lines altered. Changes in organisational goals can influence project methodologies even more radically.

[Osterweil, 1987] argues for formalising processes through programming them in a language where human tasks are represented as functions or procedures. In my experience, many people who have programming jobs do open source development on their spare time because it allows them to do the work they find interesting without the bureaucracy and demands they find at work. The processes described in the project model will therefore be descriptive, accompanied by a flow model, rather than normative as a programmed function implies.

[Osterweil, 1997] talks about process evaluation, saying that because process representation is often unavailable, process testing has to review the artefacts produced by the process. Although there are some guidelines available, this is the first written down project model for the FreeBSD Project. To evaluate it properly would require a study comparing what it says happens to what actually happens. This is outside the scope of the project. We have decided, like [Osterweil, 1997] suggests, to investigate the artefacts produced.

### 2.3.5. Brooks' Laws

Brooks' Law, "adding manpower to a late software project makes it later" has since it was first published in 1975 been treated in many studies, but with the republishing in 1995 and discussing these studies, the author finds it as true today as it was in 1975. [Brooks, 1995]

Brooks' Law is of particular interest to the open source community as it depends on new actors coming in and studying the source code. Netscape did a bold attempt to draw on the benefits of the open source community when they released the source code to their browser in January 1998. The company decided to use an open source model for the continued development of their browser. But from releasing a new browser many times a year, it would take more than two years until they launched their first Mozilla-based browser in November 2000. [Wilson, 2002] Projects such as OpenOffice, a continuation of Sun's StarOffice, have had a similar fate of becoming late.

[Brooks, 1995] also discusses the "Second System Effect" and argues that the second system built is likely to include all the frills and ideas that the architect did not have the time for during the implementation of the first system.

Many open source implementations stress design less as they are re-implementations of systems found in the closed source world. For example, Unix systems were well understood when the FreeBSD Project started and browsers well understood when the Mozilla project was initiated. While it is often not the second time the developers build such a system, they are very familiar with the products they are copying, and inclined to make "corrections" where they see them fit. This has led many open source products to look more complex and feature rich than the products they copy. Yet they use not their time for true innovation, and this allows the closed source world to gain a technical advantage, forcing the open source community to play catch-up.

## 2.4. Quality

[Boehm et al, 1976] assert that attention to quality can lead to significant saving in software life-cycle cost. However, there is no good, quantifiable definition and metric to say what quality is. This is often illustrated by the classic quote from [Pirsig, 1989]:

"Quality .. you know what it is, yet you don't know what it is. But that's self-contradictory. But some things are better than others, that is, they have more quality. But when you try to say what the quality is, apart from the things that have it, it all goes poof! There's nothing to talk about. But if you can't say what Quality is, how do you know ... Obviously some things are better than others ... but what's the betterness? ... So round and round you go, spinning mental wheels and nowhere finding anyplace to get traction. What the hell is Quality? What is it?"

In order to avoid going "poof", while not solving the broader issue of what quality is, we need to have a good definition of what we mean by quality. [Boehm et al, 1976] defines and classifies 23 metrics of quality. The definitions given in his paper such as maintainability, completeness, conciseness, structuredness and understandability are extremely hard to quantify. Therefore we have investigated two ways of measuring quality and chosen one for our studies.

### 2.4.1. Fuzz

The open source community often talks about quality in terms of execution speed, fault tolerance and features available. The articles that immediately seemed in line both with this and the history of operating system development were the "Fuzz" articles [Miller et al, 2000].

In the Fuzz articles, the authors made tools that went systematically through common tools in multiple operating systems and exposed them to random input data and checked if the programs either went into an endless loop or crashed. They did this first in 1990, then repeated the test in 1995 and 2000. In 2000 they found that they had crashed all tested Win32 programs, 15-43% of all tested programs on available, commercial UNIX systems and 9% of the tested programs on the open source UNIX system Linux.

They found that testing the developed product repeatedly also said something about the methodologies used. During the tests they supplied error reports and even bug fixes to the vendors of the operating systems. They were therefore surprised to see that many of the bugs reported in 1990 were still in place for the modern systems in 2000.

Although robustness is a good way of measuring the quality of a product, I found that investigating only this dimension and trying to use this dimension to say something about the quality of the project model was not persuading. It may be that FreeBSD is very robust, but that ignores important issues as implementing new features, having regular releases etc.

### 2.4.2. The Goal/Question/Metric Approach (GQM)

GQM was first used for a set of projects in the NASA Goddard Space Flight Centre environment and a widely used technique today, for instance used in evaluating projects with the Capability Maturity Model (CMM) for finding conformance with its key process areas [Basili et.al, 1994].

GQM assumes that measurement is only useful if it is backed by a goal, and is thus a top-down approach to measuring quality. It is divided into a conceptual level, operational level and quantitative level.

The "conceptual level" states the goal. The goal is defined for an object, which can be a product, process or resource. An example of a goal for a lamp is that we should be able to behave indoors as if we were outdoors in daylight.

At the "operational level", questions are used to characterise the way the achievement of the goal is going to be assessed. A question characterises the quality of an object from a viewpoint, and it can therefore be favourable to have multiple questions to a goal. An example of a question to our example goal is whether people can read a book as fast indoors with the lamp turned on as they can outdoors.

At the "quantitative level" we find metrics for our questions. Metrics can be measured either objectively or subjectively, and multiple metrics can be associated with one question. An example of an objective metric to our question is measuring how many words per minute were read by the test subject indoors and outdoors. An example of a subjective metric is asking the test subjects to fill out a form about how they found reading indoors with the lamp turned on as opposed to reading outdoors.

This is the approach chosen to measure quality in this thesis as it allows the investigation of many dimensions of the FreeBSD Project. In order to use it, we must be careful to have reasons for the goals selected as they must not be simply something selected out of the interest of the author, but be goals of practical value to the project.

# Chapter 3

# Research Methods

The following chapter will describe the research process that has led to this thesis. In doing so, the different methods will be discussed in light of my experience through using them and literature concerning them.

The following figure describes best how I have been working:

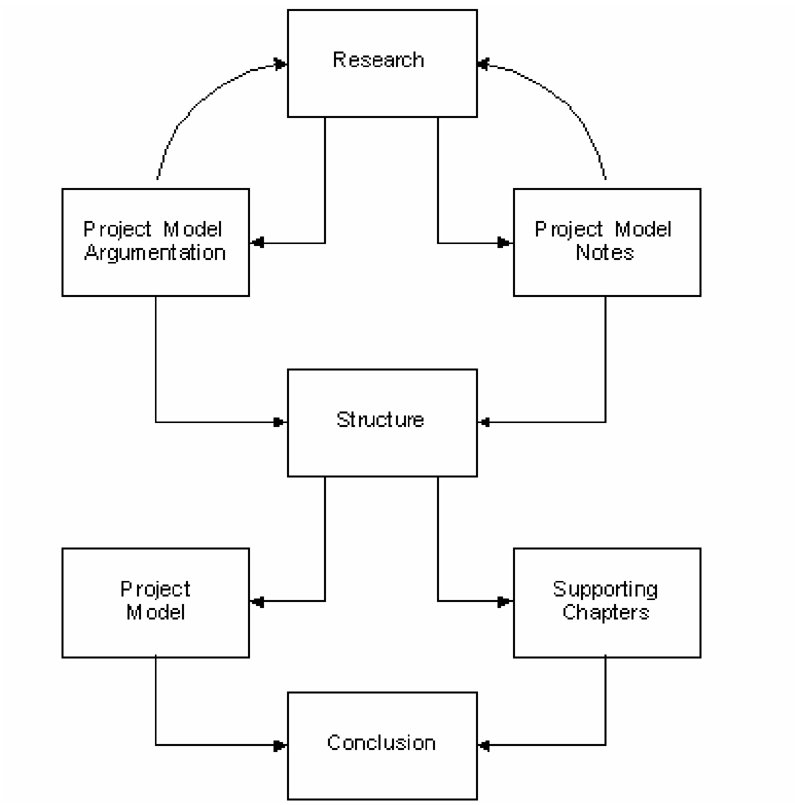


Figure 5. Research model

The research process was an iterative process of researching and taking notes. The notes consisted of factual notes and theories that needed to be explored, and argumentation to support my findings through literature. When this research process was complete, the notes were structured into a project model and supporting chapters, and the conclusions in each chapter were brought together in an overall conclusion chapter.

My research coincided with the release of FreeBSD 5.0. Many examples will be used from this release and the work concerning it. As it is the latest release by April 1st, 2003, its continued development should be using the proposed project model.

During the research, three methods of data-gathering have been used: literature studies, quantitative and qualitative studies.

## 3.1. Literature studies

The literature studies will especially concern the development of BSD at Berkeley, but also draw upon articles written on the issue of the other projects. Also, the main study of process models and their critique and classification will be through literature studies. Much of this literature will be presented in the theory chapter and used in the discussions in chapters 4-7 and 9.

## 3.2. Qualitative methods

My two main methods of gathering qualitative data are interviews and reading mailing lists.

For this research I have used two kinds of interviews: semi-structured interviews and non-structured interviews. Since I live in Norway and most the people I have interviewed live in other countries in Europe and in the US, my interview media have been telephone, email and internet chat (IRC). Because of the cost of phone-calls, these have been the most well planned interviews and thus been the most structured. During these interviews the interviewees have mainly been elaborating on the questions without going outside the topic. Email interviews have been quite structured so as to encourage a quick response. My hypothesis has been that long emails containing vague phrasing takes longer time to answer and the replies are therefore more likely to be postponed by the interviewee. The use of IRC has been for ad-hoc interviews with people interested in the questions I had to discuss.

My second qualitative method was examining mailing lists. Because very little has been written about the history and methodologies of FreeBSD, it has been necessary to browse through mailing lists for topics that have been discussed that are of particular interest for this research. The mailing lists inspected have been selected on recommendation from senior members of the projects that have been reading the lists on a day-to-day basis for years and know where interesting kinds of discussions have tended to take place. More recent mailings on the lists have been the basis for short, email-based interviews.

The challenge of qualitative analysis lies, according to [Patton, 2002] "in making sense of massive amounts of data[,] ... sifting trivia from significance ... and constructing a framework for communicating the essence of what the data reveal". [Patton, 2002: 432] While it is easy to mentally distinguish qualitative information gathering and analysis, the information retrieved through analysis affects the questions asked further in the interview and in following interviews. The analysis process starts in the brain as soon as some information has been received, and is thus an ongoing process even at the time of information gathering.

The problem with analysis beginning so soon is that first impressions have a stronger impact than later impressions and questions can be altered as to reenforce the first impressions rather than to question them.

### 3.2.1. Mailing lists

There has not been written many articles on the development of the FreeBSD Project when it comes to software engineering. As the largest source of how things have been done in the FreeBSD project is its mailing lists, these have been studied to understand how processes in the project have worked. As noted in section 8.6.4, FreeBSD has 68 mailing lists of which 62 are both publicly available and archived. Mailing lists has been the primary form of communication since the projects' inception.

One particular feature of the FreeBSD mailing lists is that they are archived so that they serve well as research material. But this trait can deter people from writing. In order to allow discussions that are not wanted to be archived, a closed mailing list for committers has been created in which they can discuss anything openly without the outside world gaining access to it. It is considered bad manners to share opinions voiced in this mailing list in public forums without public consent [Lambert, 2002].

However, the amount of correspondence on these lists is too much to possibly study within the time constraint of this research. It has therefore been necessary to narrow down the scope of the search to periods of time on a low number of lists where it is most likely that there will be found evidence of processes being in use. I have selected the lists through recommendations by my interviewees.

The cutdown has been made on the basis of interviews with senior developers. When searching for a topic, the lists they recommended have been queried for. Then a period of time of at least five days before and after the hit have been read as well as the thread the corresponding mail was a part of. When researching things with a known date, the same approach has been taken. The result is that many snapshots in time on selected lists have been chosen

The mailing lists that have been studied are:

- freebsd-stable
- freebsd-current
- freebsd-chat
- freebsd-hackers
- freebsd-announce
- freebsd-audit
- freebsd-qa

In the appendix is a set of mail threads that I have found of particular interest. As threads of mail can change topic as the thread evolves, the list has been ordered such that the first mail on the topic I have found of interest has been listed.

I have not been able to find any mailing list archives on the development of BSD at Berkeley, but I have deemed this unnecessary because of the coverage of the development process by Kirk McKusick's many articles on the topic.

## 3.3. Quantitative method

The quantitative research has been done through writing scripts that will gather statistical data from sets of semi-structured data such as commit logs, source code and problem reports.

Data analysis was done on the data materials that can be found in the FreeBSD project. This was done primarily to find answers to the questions posed in Section 3.5. Due to the active use of CVS, mailing lists and GNATS in the project, there was enough semi-structured data to be able to find at least answer estimates.

## 3.4. Project model development

This section will discuss the requirements for making a project model for the FreeBSD project and develop a project model that fits the FreeBSD project reasonably.

Even though I used an evolutionary model, my work can be grouped in traditional groups of requirements gathering and analysis, design, implementation and verification. This makes a seemingly chaotic process easier to grasp.

### 3.4.1. Software process model for developing the project model

As discussed in Section 2.3.4, project models can be considered software, and we can thus use the same tools we would use to develop software to develop a project model. In order to build a project model for the FreeBSD project, a software process model is needed.

I have chosen an evolutionary model as shown in the figure in the beginning of this chapter. The scope and purpose has been defined in the first chapter of this thesis. The top of the figure is research that leads to either notes or argumentation, and this leads back to further research. After my research has been completed, I have restructured my notes into the project model and supporting chapters.

In the development of the methodology, the current activities will be undertaken: analysis, design, implementation, verification and maintenance. Analysis will draw upon the research done to say one activity is done in one particular setting. Design will based on the analysis generalise how similar problems are being solved. Implementation will integrate the design into the methodology saying how similar problems should be solved. The methodology will be verified by comparing it to the evidence found in the analysis and discussing the methodology with active committers of the FreeBSD project. Maintenance will be the activity of keeping the project and methodology in sync once the project has taken the methodology into use. As an evolutionary model, the maintenance part will include analysis, design, implementation and verification. All the four other stages will produce output that will affect the other stages so that each process is a continuous, ongoing process.

### 3.4.2. Project model requirements

From Section 1.1, we find here are three main requirements for the project model:

- The project model should be of use to the FreeBSD Project
- The project model should be of a high academic quality
- The project model should give people not familiar with the FreeBSD Project a better understanding of the project

For the project model to be of use to the project, it should be founded upon details of what the current state of the project is so that it can be used for setting goals. It should also provide easy access to the people and groups listed. It should be easy to read and should not require significant knowledge in order to understand. It should be a document new members of the project can read in order to familiarise themselves with the project, and a document project members can refer to when working on other domains than they usually work with.

With high academic quality, we will mean that it should contain clear, precise definitions of words that are not necessarily precise and clear. It should be backed up with references and be correct about the facts. It should follow guidelines for project models to ensure it contains what project workers expect to find in a project model.

### 3.4.3. Project model design

In order to set goals, it is important to know the current project state. This is requires gathering data to generate descriptive statistics.

In order to make access easy, email addresses and names of the groups and people mentioned will be listed.

While the project model should be correct factually and have references, it should not be cluttered with references. This thesis will support the project model by providing more in-depth observations and references to literature.

As discussed by [Hohmann, 1997], every process has an outcome. In identifying the processes in the project, it is important to see the process' purpose, the activities, the associated roles and what the outcome of the process is.

### 3.4.4. Project model implementation

The implementation of the project model is available in Chapter 8.

### 3.4.5. Project model verification

In order to verify the project model, it is sent to the interviewees and senior project managers for proof-reading and commenting with the sanction of the FreeBSD Documentation Project. After being revised, it is sent to the project members for further comments. Finally it is incorporated into FreeBSD’s documentation.

## 3.5. Metrics and criteria

To be able to say something about how well this project is running, we need to be able to measure set criteria. Without measuring, we can only say something about how we think things are. By measuring, we say something about how things are.

We will use the Goal/Question/Metric (GQM) framework described in the theory chapter to do our measurements as described in Section 2.4.2.

To identify our goals we need to ask who are the ones defining goals. FreeBSD is a UNIX based system, and the original UNIX was developed by developers, for the developers, to make their jobs of creating applications easier. Most developers will claim a personal interest and that they benefit from developing FreeBSD. The developers are stakeholders in the way that they invest time and equipment in the development.

Another group of stakeholders is users. Users can be both organisations and individuals. Like the developers, they invest their time and equipment in FreeBSD, hoping to gain the benefits of its promised features.

Having identified the users and developers as stakeholders, we can now set up goals that, based on experience, at least one of the groups have. Defining goals based on experience, or domain knowledge as Gilb calls it, is in accordance with [Gilb, 2003: 12]. Together with the goals, we will set up what questions we need answered to determine if the goals have been met, and what metric we will use to answer the questions.

### 3.5.1. Product specific goals

The product specific goals come as a response to the users wishes. The users in the project can communicate directly to the developers through mailing lists.

Product specific goals have been selected by going through the mailing list archives and noting requests that have often been voiced.

#### *3.5.1.1. The OS should run interesting, new software*

Third party software that is intended to run on the system should not be unnecessarily encumbered, as such encumbrance can lead to it dropping support for the system. Such an act would lead users that require new releases of this software to prefer using systems that are supported.

Does much third party software run on the system?

To measure whether third party software runs on the system, we will look at how much software is officially said to be supported by the system, and how much of this that actually runs at the time of writing.

#### *3.5.1.2. New features should be developed*

As the technology evolves, so will the needs of the users and developers as their technological fantasy will continue to suggest new uses that they think should be possible. Some of these new uses require new features to be added. These should be developed, or users will move to systems that provide these features and the momentum of the project will decrease.

Are new features implemented? Do new features deprecate old features?

To measure if new features are implemented, we will investigate release announcements and interview some of the developers in the sub-projects that created the new features.

#### 3.5.1.3. New releases should be made available regularly

In order to stay visible at conventions, to give an impression of progress for people worried about version numbers and in order to supply new users with the most up-to-date version, releases should be made regularly.

How often are releases made?

The release history will be examined and how many days there are between the different releases will be examined.

### 3.5.2. Project specific goals

The project specific goals are administrative goals that emphasis on developer productivity rather than having to have developers perform administrative tasks. Most goals refer to managing commit privileges and distributing written code. When these goals fail, developers have to spend time making workarounds for having code added and find alternative means of distributing the updated code. These goals have been selected on the basis of interviews with developers.

#### 3.5.2.1. Active developers should be made committers

In order to stimulate to further recruitment to the project, commit privileges should be given to active contributors.

Is there a steady flow of new committers coming into the project?

This will be measured by monitoring how many committers have been added per month during the past year.

#### 3.5.2.2. Inactive committers should have their commit privileges revoked

An inactive committer, that is a committer that has not performed a commit during the past 18 months, should have his commit privileges revoked. Commit privileges are also a security risk as they can be used to cause harm to the code.

Dormant privileges can be picked up by intruders, and thus be a security problem for the project.

Are commit privileges regularly reviewed and revoked for inactive committers?

This will be measured by monitoring how many committers have their commit privilege removed and how regularly this is done.

#### *3.5.2.3. Developers should be able to add their code*

For the project to be able to maintain its momentum, it is imperative that it is easy for developers to add their code.

How many commits happen? How easy is it for new committers to make commits?

To determine how many commits happen, we will watch commit logs for a year and split them into fit categories. To determine how easy it is for new committers to make commits, we will watch commit statistics for new committers for a three month period after they receive their commit privileges.

#### *3.5.2.4. Updated code should be easily available for integration with running systems*

One of the main features of the FreeBSD project is its frequent releases. These releases are distributed by CVS, and users synchronise with these updates through a program called CVSup. This requires CVSup servers they can connect to, which in turn requires an infrastructure to handle the requests.

The CVSup system is by far the FreeBSD project's largest logistical system. In addition, the project provides CD-ROM images that are used to produce installation CD-ROMs, either by companies that sell them for a profit or by individual users.

How many CVSup servers are there available? How long do users have to wait in order to use a CVSup server? Are CD-ROMs and DVDs easily available to users?

CVSup servers will be counted over time. An access simulation will be performed in order to estimate the availability of the servers. Sales statistics will be gathered and aggregated to measure the availability of discs.

### 3.5.3. Process specific goals

Up until now, FreeBSD has been very little process oriented. It has good communications with its users through mailing lists, but mailing lists are not well suited for bug management. This is confirmed by [Swanson, 1976] who stresses the importance of a maintenance database. To handle this, they have had to create problem reporting routines for problems identified by users. The following goal is built upon the documents describing these, [FreeBSD, 2002C] [FreeBSD, 2002D]

#### *3.5.3.1. Problems should be identified and solved*

There will inevitably be problems in any software. Therefore there should be ways that those who have not written a particular piece of software but experience trouble with it can alert the author about the problem, and possibly submit a patch.

With most of FreeBSD's users not being developers, it is very likely that they will identify problems that the developers do not know about. It is therefore important to have a standard way of error reporting that simplifies the task of duplicating the error so that the author can find and fix it.

In addition, there is little use in problem reporting without the problems being reviewed and solved. Ideally they should be resolved quickly so that the report originator sees that it is useful to report errors and is thus be stimulated to send further reports about other errors.

Are problem reports sent to the author? If so, how are they sent to the author? Are problems reported given attention? Are the problems resolved? If so, in what time frame are they resolved? How many problems are being worked on? How many problems are being resolved? How long does it take from a problem report is received until it is assigned to a committer? How long does it take before a problem is given attention? How long does it take before the problem is resolved?

To measure whether problem reports are sent, we will look at the frequency of problem reporting. Through interviewing a sample set of committers we will determine if it is common to receive problem reports in other ways than the standard format. To measure whether problem reports are given attention, we will see how quick the status of the report is changed from its initial state. We will also check how many have reached the state "closed". To measure how long it takes before the problem is assigned to the correct committer, the submit date and the first time the responsibility for the report is changed will be measured. To measure how long it takes before the problem is given attention, we will check the first time the state was changed from "open". To measure how long it takes before the problem is resolved, the time the report is submitted will be compared to the time the report is closed.

## 3.6. Research Process

During the research process, I've used both quantitative and qualitative research methods. These are my main impressions of this information gathering, and a critique of this process

### 3.6.1. Impressions

The "quantitative" research has consisted of gathering data from CVS logs, Problem Reports and classifying news announcements. This information has been easily available through a net browser and the CVS tool, and has been very consistent. The only major inconsistencies have been when some of the machines responsible for handling the logs and reports have had an incorrect time. This has not been discovered for many entries and has not caused significant errors.

The "qualitative" research has consisted of email interviews, reading mailing list correspondence, internet chat and regular interviews.

I have found it difficult to get in contact with people in projects I did not know well. [Mann et. al, 2000: 28] have found that many regard contact with outsiders as threatening and time consuming. I have also tried enlisting altruistic support that is perceived common in the open source and academic community [Hars et al, 2001]. The success of a good response has been perceived as dependant on how much the person's interests stand to gain from the output of my research.

I have had problems with people being gone from discussions for long periods of time because of other issues in life such as vacations hindering them in replying.

It has been difficult to get people to use their time to answer questions. Also, many of the interviewees that did respond would stop responding to follow-up questions. Because of the distance between people and the low resources available for transportation and phone, I have only been able to interview two committers by phone and in person. The interviews have been interesting and the interviewees answered most of my questions.

Monitoring mailing lists has proved to be a very time-consuming task, but has helped provide a more coloured view of the project and its processes than I got from interviews. The two have thus been complementary. Because of time constraints and the vast amount of both daily incoming and archived messages, I have only been able to read snapshots in time from selected lists.

### 3.6.2. Critique

In my research, I have only been in contact with people that have access to the net and the time to work on the project and engage in discussions. I have not been in contact with people that have been marginalised from the project because of any of these issues. I have thus not found how the project model includes or excludes people that cannot have regular access to the net.

I have drawn upon [Mann et. al, 2000] for handling consent. When interviewing people, I always started the interview by presenting what I was researching and clearly stating that I was asking them questions for my research. After my research was completed, I emailed all my interviewees an email with the notes I'd taken and asked if they minded anything in my notes so that I could include them with my research data. I have thus been explicit about gathering consent from my interviewees. However, I do not have signatures or such, except for the occasional PGP-signed message. I have not explicitly sought consent when siting public mailing lists as people posting on such lists are aware that they are saying this in a public forum.

For the three last months of the thesis, the project model, most of the data and my thesis have been available to a larger and larger part of the community. The project model was presented to the community at large 1,5 months before the thesis was submitted and the thesis was presented two weeks before it was submitted. I have thus made sure to give people access to the data I have gathered about them.

During my research I have obeyed general netiquette as described in [Mann et. al, 2000: 59-62] and the rules regarding netiquette of the Department of Informatics at the University of Oslo.

# Chapter 4

# The FreeBSD Project's Organisational Structure

The following chapter will discuss the organisational structure of the FreeBSD Project. It will not include the community surrounding the project.

## 4.1. Contributors

A contributor is a person who has done at least one of the following:

- Developed software that FreeBSD is derived from or integrates
- Made a change to FreeBSD that has been committed into the repository
- Donated funds, hardware, entire servers and network connections
- Sent a Problem Report

[FreeBSD, 2002F] has a list of the entire list except for contributors sending problem reports. These make up 1467 contributors that are not committers.

By going through the problem reports database, noting down only unique names by manually removing all duplicates on the letter 'A', I got a factor of 0.8870 for which I had to multiply the number of report originators with. Removing active committers from the count, we get an estimate of 6538 contributors. [1]

Acknowledging that there can be an overlap between the two groups, my estimate is that there are approximately 5500 contributors to the FreeBSD Project.

[1] The scripts and output data used for this estimate are available in the file data/originators. More details in Appendix A

## 4.2. Committers

Since a contributor needs a committer to integrate his contribution into FreeBSD, this is an extra overhead to the project compared to if the contributor had commit privileges. When contributors start making many contributions, a committer will at some point recommend they get commit privileges to core. An example is [Kamp, 2002]: "On behalf of the FreeBSD project I have to warn you that if you persist in doing work of this kind over and over again, you will eventually be subjected to a commit bit. You have been warned"

As noted in Committer, a committer must have committed code during the past 12 months. However, as stated in the legislature [FreeBSD, 2002H], it is not until a committer has been inactive for 18 months that his commit bit is revoked. Thus, the amount of true committer is not equal to the amount of people that have commit privileges.

Using "commitscan.pl" (available in Chapter 11) and setting a cut-off at 12 commits, we find that there were 242 active committers in the year examined. Compared with the 277 committers who have made a commit that year, 34 were inactive committers. Of the total number of committers, the average was 545,5 commits, the median was 211,5 and standard deviation was 1073.

Every new committer is announced at FreeBSD News Flash. This file has therefore been scanned to get an overview of how many committers are added every month. The file CVSROOT/access in the CVS repository contains a list of all committers in the project. The CVS log has been analysed to see how many committers have been removed every month.

The following table shows us how many committers have been added and removed in the period from January 1st, 2000, to and including January 31st, 2003:

| Month | # Added | # Removed |
|---|---|---|
| January, 2003 | 4 | 2 |
| December, 2002 | 0 | 1 |
| November, 2002 | 1 | 0 |
| October, 2002 | 8 | 0 |
| September, 2002 | 2 | 3 |
| August, 2002 | 6 | 0 |
| July, 2002 | 2 | 0 |
| June, 2002 | 4 | 2 |
| May, 2002 | 2 | 1 |
| April, 2002 | 8 | 5 |
| March, 2002 | 5 | 0 |
| February, 2002 | 3 | 1 |
| January, 2002 | 0 | 0 |
| December, 2001 | 3 | 0 |
| November, 2001 | 10 | 2 |
| October, 2001 | 3 | 0 |
| September, 2001 | 0 | 0 |
| August, 2001 | 5 | 2 |
| July, 2001 | 5 | 0 |
| June, 2001 | 7 | 0 |
| May, 2001 | 1 | 0 |
| April, 2001 | 4 | 0 |
| March, 2001 | 6 | 0 |
| February, 2001 | 2 | 0 |
| January, 2001 | 3 | 0 |
| December, 2000 | 4 | 0 |
| November, 2000 | 7 | 0 |
| October, 2000 | 6 | 2 |

| Month | # Added | # Removed |
|---|---|---|
| September, 2000 | 1 | 0 |
| August, 2000 | 3 | 0 |
| July, 2000 | 10 | 3 |
| June, 2000 | 5 | 0 |
| May, 2000 | 2 | 0 |
| April, 2000 | 1 | 0 |
| March, 2000 | 2 | 0 |
| February, 2000 | 4 | 0 |
| January, 2000 | 4 | 0 |

Table 4.1. Added / Removed Committers

On the basis of this table, the following two figures illustrate how many new committers have been added per month and how many committers have been removed:

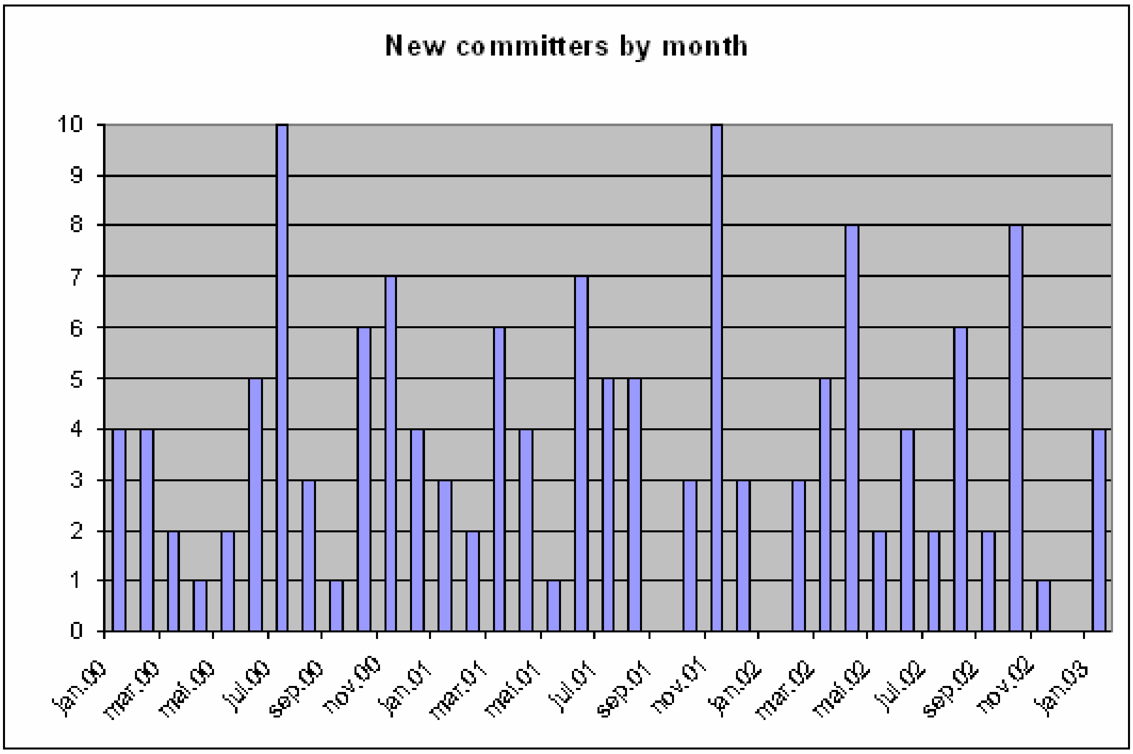


**Figure 6. New committers by month**

Figure 6 shows how many committers have been added to the FreeBSD Project in the period January 2000 to and including January 2003. It shows that there has been an average of about four committers added per month. There are occasional spikes and occasional lows, but I do not consider this abnormal.

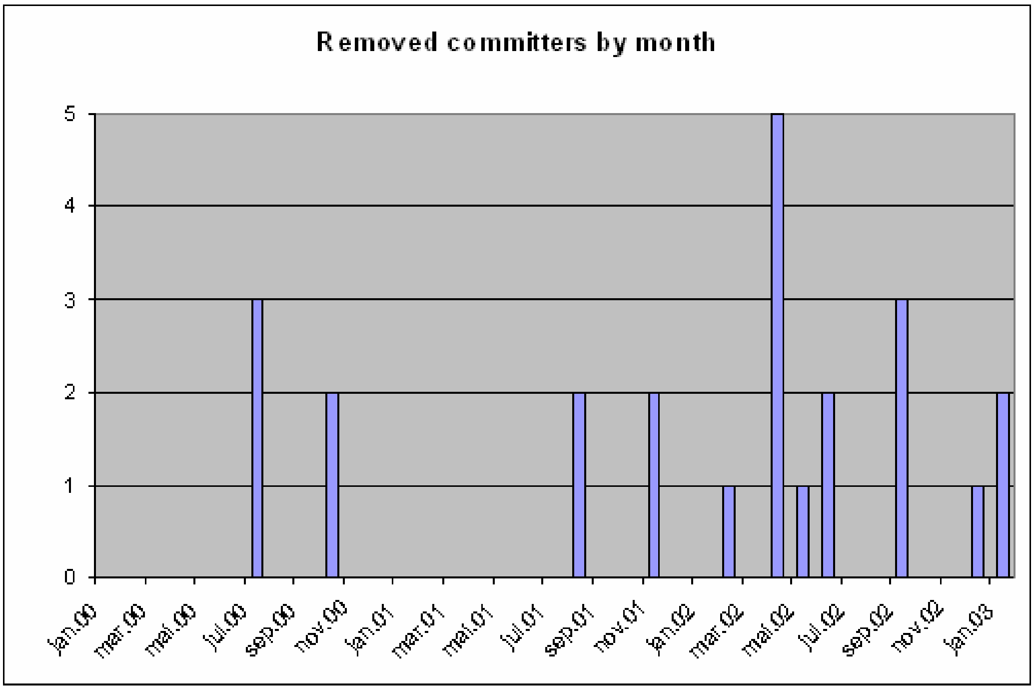


**Figure 7. Removed committers by month**

Figure 7 shows how many committers have had their commit privileges revoked in the period January 2000 to and including January 2003. It shows that committers are seldom removed. The spike of five committers removed was due to a clean-up of inactive committers. There have not been regular clean-ups, and most committers removed have removed themselves.

These figures show that the project adds many more committers than are removed. The adding of committers does not fluctuate very much, indicating that committers are added regularly. Committers do not seem to be removed on a regular basis. From this we understand that the project is in a state of steady growth.

## 4.3. The core team

The core team with the initial BSD had been the people employed at the Computer Science Research Group at Berkeley. When FreeBSD was started, this tradition was followed by the core group being those who started the project and those that were the most active committers. However, by year 2000 the people in the core group were no longer the most active committers. Some were even completely inactive. This led to the first core election in September 2000. [Lehey, 2002]

The role of core today is:

- Assigning commit privileges
- Intervene should a committer misbehave
- Mediate in disputes between committers

Core assigns commit privileges after a contributor has been recommended by a committer. This process is detailed further in Section 8.5.1.

Should one or more committers misbehave and constitute a threat to the code base or the project morale, core can suspend or completely remove the offenders commit privileges. This process is detailed further in section 8.5.8.

If disputes arise among developers, core can mediate between the two. An example is that during a dispute in the SMPng development February 2002, core decided to give maintainership to the most active SMPng developer and thus the authority on what is included and what is not. [FreeBSD, 2002J]

To make core's work more transparent so that developers can track how their requests are handled, core publishes monthly reports on the issues discussed and decisions made. This report is written by the Core Secretary.

A core election is held at least every second year. This process is detailed further in section 8.5.4.

## 4.4. Maintainers

Since everyone with commit privileges can make changes to any file, there have to be social structures in place to avoid people with conflicting views on how a piece of code should work from applying their changes as soon as the other party has made their. To avoid such conflicts, each part of the code should have a maintainer.

A maintainer can lay out rules for how changes to the code should be applied. Such notes are laid down in src/MAINTAINERS. The log of this file can then be used to find out who were committers before and why these rules have been put in place. One example is that the maintainer for the OpenSSH module has requested to review patches before they are committed.

The example of OpenSSH is interesting because OpenSSH is a project outside of FreeBSD, closely related to the OpenBSD Project. A patch should therefore be relayed to the OpenSSH project so that they can apply it to the product. If this is not done, the patch will need to be applied for every release of OpenSSH. Also, when the patch is given to the OpenSSH project, other systems that use OpenSSH can benefit from it. This is in line with the motivating factor that the developers want their code to be used.

Even if the primary policy is to send patches to the originating project, called the vendor, the vendor may not always accept the patch. If it is felt in the project that this patch is necessary even though the vendor does not accept it, it is kept and applied for every import. An example where this has been the case was that the GNU C compiler, GCC, for a period was without project activity. Even though there was no further development on GCC, the FreeBSD Project needed changes made. Thus a huge amount of patches were made that were applied on FreeBSD versions of GCC. The alternative would have to change to another compiler that had project activity, but much of the FreeBSD code was dependent on this compiler. Eventually the GCC project was resumed and the patches were accepted and integrated, and FreeBSD runs GCC today almost without any patches.

Assigned maintainers are identified in the Makefiles of the modules or collection of modules with a MAINTAINER variable. In addition to this, many maintainers have chosen to specify how they prefer changes to the code to be made, and these two lists are put together and make the assigned maintainers.

A total of 22 maintainers were found in Makefiles, where in two instances maintainership was split between two committers. A total of 16 maintainers were found in src/MAINTAINERS that were not in the Makefile maintainer list. Of these, two were aliases for group of maintainers and one module had two committers as its maintainer. The source tree examined for Makefiles was a snapshot from February 27th, while src/MAINTAINERS was updated February 17th.

Superimposing the data from src/MAINTAINERS on the data extracted from the module list, we find that 125 out of the FreeBSD's 716 modules have defined maintainers, leaving 591 modules without defined maintainers. The data results can be found in "modules.txt". (Modules are discussed further in Section 6.2.3.3)

It is worth noting that there are limitations with identifying modules the way chosen here. While the utilities and libraries, called the userland, is well modularized, much of the kernel is not, and the size of kernel modules varies much. However, because of the high number of modules found, this selection should be representative.

We have thus found that although it is beneficial to have maintainers for modules, only 18% of the modules have maintainers. From experience, many people have a tacit understanding of who the maintainers are, and it is a project wish that all modules should have maintainers. We must thus assume that there are de-facto maintainers that are not listed in the files investigated.

## 4.5. De-facto maintainers

De-facto maintainers are maintainers that have made 50% or more of the commits during the past 12 months.

These maintainers are found by scanning commit logs and making a hash table by file of who has made a commit to it. When grouping the files within a module, an overview is made of who has made how many commits to the file.

The modules examined for de-facto maintainers are the 591 orphaned modules found in the modules section. They are checked with "commitscan.pl" on a snapshot of the FreeBSD-CURRENT branch from February 27th, 2003.

Of the 591 modules that did not have any listed maintainers, 257 had committers that had made 50% or more of the commits in that module, and thus are counted as de-facto maintainers.

One interesting observation was that 118 modules had only one commit, and 344 modules had 4 or less commits. In order to see if there was a difference between modules with high activity and modules in general, it was measured how many of the modules with 10 or more commits had de-facto maintainers. A total of 63 out of 106 modules with 10 or more commits had de-facto maintainers.

43% of the maintainerless modules had de-facto maintainers, although only 25% of the modules with de-facto maintainers had 10 or more commits. Most de-facto maintainers have thus got this role through making few changes.

## 4.6. Sub-projects

FreeBSD contains many sub-projects. As part of the research, I have selected a few for closer investigation. The criteria for selecting these projects were that they were ongoing and relevant to the current releases at the time of the research: version 4.8 and version 5.0. Further more, they were selected on the responsiveness from the project leaders. The projects selected are: The FreeBSD Documentation Project, the Bluetooth implementation, and the OpenPAM implementation.

We will see that there are different motivations for starting sub-projects. The three found in our examples are initiated by the project, initiated by an individual and donated by a vendor in order to minimize maintenance work. They each have a different current state: being an acknowledged sub-project of FreeBSD, being fully integrated into FreeBSD and having ceased being a sub-project, and being an independent project but committing regularly to FreeBSD.

### 4.6.1. The FreeBSD Documentation Project

The FreeBSD documentation project was started January 6th, 1995, on an initiative of Jordan K. Hubbard, one of the FreeBSD project's founders [Hubbard, 1995A]. The main motivation for its inception was to gain a selling point: FreeBSD had so far been used mostly by its developers, while Linux, at that time a new contender, was gaining a lot of new users and it was felt that better documentation was what helped them.

The project was set up with a manager, four team leads and regular members [Hubbard, 1995B]. At the present, the project has grown to an estimate of 32 committers [Interview with Nik Clayton], which is roughly 10% of the FreeBSD committers. However, very few committers contribute to the documentation project alone.[2] Many of the contributors often contribute to other parts of the FreeBSD project as well. The members are structured in one project leader, several teams responsible for translations into different languages, and a documentation release engineering team [FreeBSD, 2003B].

The documentation release engineering team consists of all the team leaders as well as a member from FreeBSD's release engineering team. Its purpose is to have up to date documentation for each FreeBSD branch at release time and make sure documentation is correct for the different branches.[3]

More on what the Documentation project does can be found in section 8.7.2.

### 4.6.2. The Bluetooth implementation

Bluetooth is a standard for how wireless devices should form an ad hoc local network, including devices up to ten meters away. The standard details a stack is similar to the ISO network stacks.

The implementation of the Bluetooth stack started in a classic open source way: "one day I just thought that it would be nice thing to do, and since I had spare time I started to work on it" [Interview with Maksim Yevmenkin]. The project was founded upon the Bluetooth Specification Book version 1.1 and was developed for FreeBSD-CURRENT.

---

[2] According to "commitscan.pl" on the commits done between December 1st, 2001 and December 1st, 2002, there were four committers that only committed to the FreeBSD Documentation Project. Of these, only one was an active committer.

[3] An example of incorrectness is when a feature is modified from one version to another, and while the old version remains in the old state, the documentation for the new version is the only documentation available.

The first step was to go through the specification and try to make sense of it all and get the big picture. This step can be seen as requirements analysis. This led to a primary TODO-list.

The next step was to identify the different parts of the system and see how they should interface each other and FreeBSD. This step included drawing a layer graph of the stack and go into each node and find out what needed to interface it and how it should be interfaced. Answering these questions gave the overall interface and algorithm design. The specification also gave many guidelines as to how the implementation should be done to obtain standards conformance. Based on these guidelines and the answers found, the TODO-list was refined as a list of needed functionality.

The implementation consisted of many short cycles that took an item from a TODO list and implemented it. The items were prioritised in the following order:

- Bugs - The existence of bugs would hinder knowing that further work was correct implemented.
- Missing functionality - This is functionality required by the specification that had not yet been implemented.
- Features - Features are things that would be nice but which are optional in the specifications, requested by creative users or which the developer found interesting.

In order to verify the implementation, testing was used and at different stages in the process announcements were made giving people access to the code for testing and review.

For testing, a Bluetooth device simulator was created that would interface the code. After obtaining Bluetooth-enabled hardware, the hardware was used for ad hoc testing. "Sometimes I do some crazy things like pulling out the PC-CARD while [the] stack is active, connecting both devices to the same laptop and have them ping each other for 24 hours etc. to see what will happen" [Interview with Maksim Yevmenkin].

The development went through six months of snapshots. Very few comments were made on the actual code itself. People with Bluetooth devices would download the code and try it out and report back any errors. Interoperability was thus tested mainly through random combinations of devices the developer had and what devices the people trying out the code had.

During the development, the design changed. But as it was developed and designed by one person, this was not a problem and even expected. The design documents, consisting of pictures, explanatory text and proof of concept, were never released for review nor added to the project when the Bluetooth implementation was committed to FreeBSD-CURRENT. The tests were also not committed as they were deemed to be too special-case for the developer and of little or no use to anyone else.

Being committed, the Bluetooth project is now a part of FreeBSD and updates are committed directly into FreeBSD's repository. It has thus ceased to be a subproject.

### 4.6.3. The OpenPAM implementation

OpenPAM is a project that implements Pluggable Authentication Modules (PAM). It replaces LinuxPAM that was used up to FreeBSD version 4.8. Much like Kerberos, it allows for a single signin to cooperating computers in a network. However, rather than to dictate a technology that should be used, it allows for multiple authentication modules to interoperate.

PAM originates from Sun's Solaris, and exists in a standards draft from the Open Group called "X/Open Single Sign-On Service" or XSSO [Open Group, 1997].

The preliminary implementation of PAM into FreeBSD was the integration of LinuxPAM by Juniper Networks. Juniper donated its work to the FreeBSD project so that they would not have to maintain the code themselves. The continued integration of PAM was started on contract from Enitel A/S. Further funding was sought through Network Associates Laboratories (NAI) that had received $1.2 million contract from the US Navy's Space and Warfare Systems Command to develop security extensions for FreeBSD.

After some time of trying to fix bugs in LinuxPAM, it was deemed better by its maintainer to reimplement PAM. Leading up to this decision, LinuxPAM had undergone much testing with a set of headers based on the standard. These headers were now used to generate a code skeleton and were implemented during the next month.

The requirements for OpenPAM were following the XSSO standard, ensuring cross-platform compatibility (especially focusing on Solaris and Linux) and a few extensions deemed useful for many applications. During implementation, further requirements to extensions were discovered, and after the initial integration with FreeBSD, further feature suggestions were made by developers in the FreeBSD community.

There was done very little design. Solaris' implementation was studied carefully where the XSSO standard was unclear.

During implementation, most parts were either basic enough to be trivial to write or consisted of multiple basic parts put together. Much debug code was added to make bug tracking easy.

Verification consisted of writing tests, implementing the framework in other applications and code review. For testing, test modules and a test program were written. After this, OpenPAM was integrated with existing userland utilities that require authentication. During this implementation, further debugging code was added to make bug tracking simpler.

Review was sought during the initial three releases. The mailing lists "freebsd-audit" and "freebsd-arch" were noticed about the implementation and the project web site, but very little feedback was given.

After two weeks of availability and requests for review, OpenPAM was integrated with FreeBSD-CURRENT. After ten more days of availability in the project, it replaced LinuxPAM after a notice on the freebsd-current mailing list.

When the switch-over from LinuxPAM to OpenPAM had happened, people started responding. The responses, however, were very little constructive with regards to making the code better. Only a handful patches have been submitted from the code was integrated February 23, 2002, until the time of writing.

Although OpenPAM has been committed into FreeBSD, it still operates as an independent subproject to facilitate integration with other operating systems. When it is updated, the updates are committed to FreeBSD.

## 4.7. Summary

The following are my main findings from this chapter:

- Commit privileges are seldom revoked
- Committers are added regularly, and because commit privileges are seldom revoked, the project is in a state of steady growth.
- There are an estimate of 5500 contributors to the project
- Core has well defined tasks and authority, and is elected at least every two years.
- Just under 50% of the modules (382 of 716) have a designated or de-facto maintainer, most modules do not
- FreeBSD consists of many sub-projects. We have studied three quite different sub-projects.

# Chapter 5

# FreeBSD Project Administration

Every project has administrative parts, and this chapter will look closer at some parts of the FreeBSD administration. We will start by looking at what tools are used followed by how communication is handled. Then we will look at how problems are reported, and then how the product is distributed.

## 5.1. Tools

FreeBSD uses the Concurrent Version System (CVS) to handle changes to the project source repository with different versions (called branches). In addition, the project documentation, problem reports (which we will describe in Section 5.3) and the project web site is in the repository. CVS has been the tool of choice for versioning since when the project was still a patchset to 386BSD, version 386BSD-pk2.4.

Perforce is another version control system that is used in the project. The project has been donated a 300 user license to use the Perforce Server. The clients used are free. Perforce is developed using FreeBSD, and its developers want the developers to like it.

The main differences between CVS and Perforce are speed issues and that Perforce is significantly better at handling meta-data. As a result of the faster meta-data handling, Perforce handles branches better than CVS. Many sub-projects in FreeBSD, such as OpenPAM, use Perforce.

Many developers do not want to use CVS for their projects because of CVS' restrictions, and the use of Perforce allows them to be a much closer sub-project to FreeBSD than if they were an external project.

CVSup is a tool to distribute the sources of a particular FreeBSD branch and keeping them up to date. It uses CVS to supply only increments of the code, thus making this a fast way to keep the system source up to date without using much bandwidth. The user then recompiles his system and stays thus up to date with feature and security changes.

[Swanson, 1976] stresses the importance of a maintenance database to do effective maintenance. FreeBSD uses GNATS to organise maintenance issues. One of the programs GNATS consists of, send-pr, is used to generate error-reports that are both readable and editable to humans and the GNATS system. Finished reports are sent by email to a central GNATS database. This is maintained by the GNATS Administrator.

Several "TODO-lists" are kept for giving ideas to people wanting to contribute to the FreeBSD project but not knowing where to start. Poul-Henning Kamp's "Junior Kernel Hacker TODO List" is a set of suggestions of things that would be preferable to have in the kernel that make good starting points for people wanting to contribute [Kamp, 2003]. The BSD Driver list [Yancey, 2003] is a similar initiative, connecting people with unsupported hardware to people with time to write drivers for the hardware. Particularly it allows for the recruitment of testers to devices that have not got as much testing as the developer would like to give the code before committing.

Pretty Good Privacy is a public-key cryptographic system that allows the sharing of public keys for securing communication between two parties and identifying both parties. This identification can be used to check that documents appearing to be written by an author are in fact written by this author. An example of key handling is [Vidrine, 2003] that shows how the Security Officer uses PGP and had to issue a new key because the private key had been available to too many of its officers than what would satisfy some advisory-submitters.

We have thus seen that the FreeBSD Project uses a repertoire of tools that aid it in the project administration. Using a homogenous kind of tools allows for little conversion overhead and more time to do what people are in the project for: to create and maintain the FreeBSD operating system.

## 5.2. Mailing lists

The main tool for reducing communications overhead in the FreeBSD Project is mailing lists. A mailing list is where one email address archives and forwards messages sent by users to all its subscribers. Lists are forums for announcements, discussions and regular chat.

As of February 21st, 2003, there were 68 public mailing lists available. There are two closed lists, one list available for committers only and one list for core members only. In addition, several subprojects have their own mailing lists.

With an average of 1167 members per list[4], and guidelines on how to behave sent to all new members, the lists are a successful means of communication. In addition, many people read the lists through web interfaces that are indexable by search engines such as Google, so that users can easily find discussions where the problems they have have been discussed.

[4] A list of mailing lists was supplied from the computer running them that is available to committers only. The list can be found in the [Interview with Dag-Erling Smørgrav].

## 5.3. Problem reporting

With a complex system such as an operating system and a large amount of users, users are likely to discover problems the developers have not found. In order for problem reports to be handled well, a standardised way for receiving and handling them is required. Such a standard way is described in Section 8.5.7.

To exemplify this, I followed the process described in Section 8.5.7. I used the "send-pr" tool to write a problem report [FreeBSD, 2002D]. A mail was returned by the GNATS tool with ID "i386/45558" stating the problem report was available at http://www.freebsd.org/cgi/query-pr.cgi?pr=45558. The page linked contained the problem description entered and the report status within the project. When this thesis was submitted, the issue reported has not been solved.

One sample report, however, is not sufficient to determine if problem reports are given attention. In Section 3.5.3.1 we have set up measures to check if the problem reports are being given attention and resolved. The results of the checks are presented here.

To identify whether problems are identified and given attention, the GNATS database was examined with data up to February 26th, 12.00 CET. The total number of problem reports was found to be 48697.

To gain a list of the states used for this examination, the following command was used to produce a data file:

```
grep -R "^>State" * > ~/states.txt
```

The number of reports pr area was done through a

```
ls area/ | wc -l
```

where area is the name of the problem area in only lower case letters.

The following table lists the number of problem reports by area. The areas categories are ones that the project has found it right to group problem reports into.

| Area | # of reports | Percentage of total |
|---|---|---|
| Advocacy | 9 | 0.02 |
| Alpha | 95 | 0.20 |
| Bin | 4728 | 9.71 |
| Conf | 952 | 1.95 |
| Docs | 3465 | 7.12 |
| GNU | 339 | 0.70 |
| i386 | 1416 | 2.91 |
| ia64 | 4 | 0.01 |
| Java | 68 | 0.14 |
| Junk | 839 | 1.72 |
| Kern | 5233 | 10.75 |
| Misc | 2613 | 5.37 |
| Pending | 546 | 1.12 |
| Ports | 28250 | 58.01 |
| PowerPC | 0 | 0.00 |
| Sparc64 | 11 | 0.02 |
| Standards | 47 | 0.10 |
| WWW | 82 | 0.17 |

Table 5.1. Problem reports by area

In table 5.1 we see that most categories contain less than 1% of the total number of problem reports. Based on this table, a pie chart showing what area receives the most problem reports is in order:

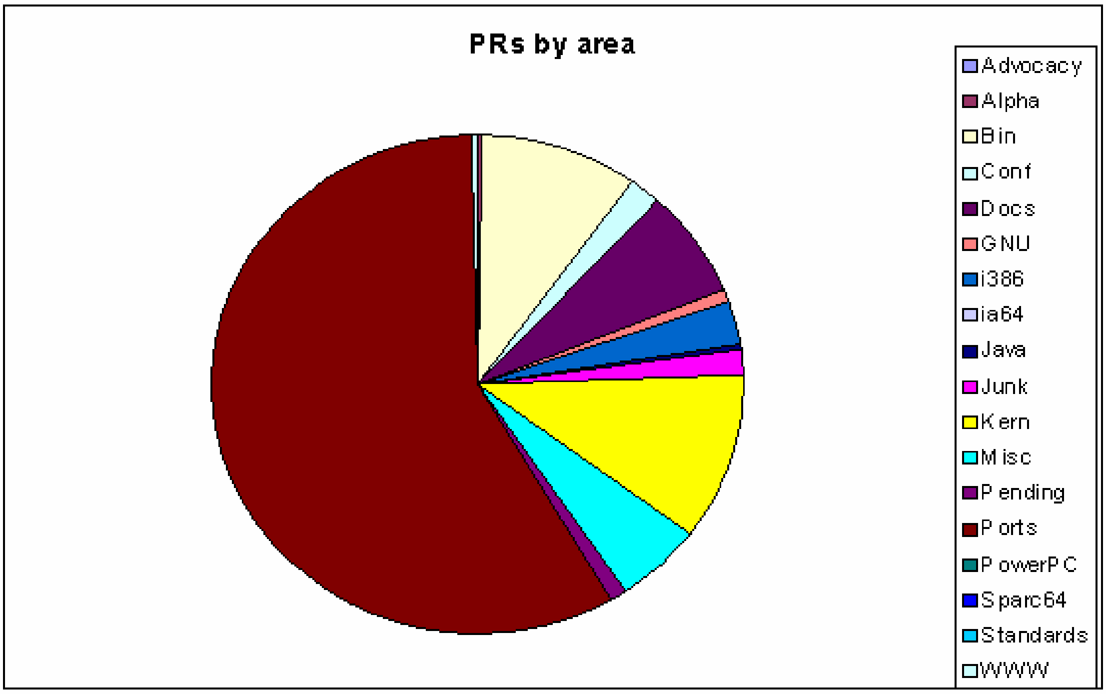


**Figure 8. Problem Reports by Area**

From figure 8 and table 5.1 shows the distribution of the 48697 problem reports filed in the project lifetime. We see that the areas that receive the most reports are ports, kern, bin and docs. It is interesting to see that this is the same way the project organisation is structured. Bin is the userland. The remaining categories have a fairly small amount of problem reports, the largest of the remaining categories being general categories as misc and problems associated with the i386 hardware architecture.

In order to see if problem reports are given attention, we check their current status. The following table summaries how many problem reports have what status. Reports which have not been worked on are "open", then they are "analyzed", possibly sent for "feedback" or "patched". Finally, when the problem is resolved, state is changed to "closed". If during problem resolution the report originator becomes unavailable and the developer working on solving the problem cannot continue without additional information from the report originator, the report is marked "suspended".

| State | # of PRs |
|---|---|
| Closed | 44485 |
| Open | 3572 |
| Analyzed | 122 |
| Patched | 63 |
| Feedback | 358 |
| Suspended | 97 |

Table 5.2. Problem report state

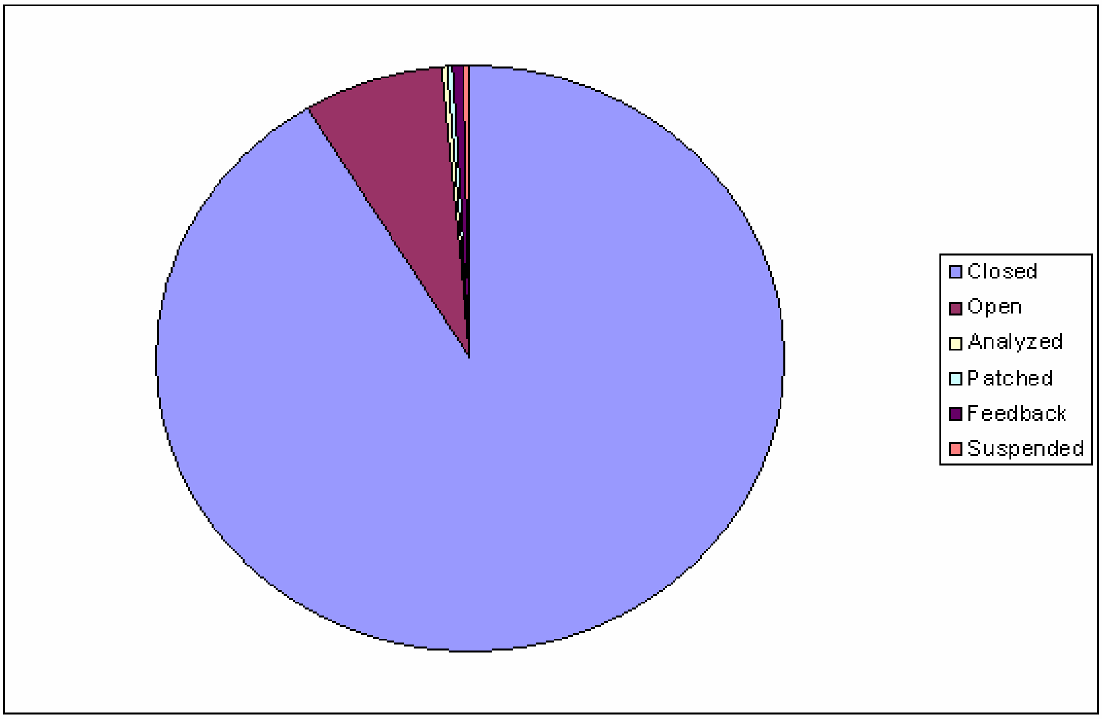


**Figure 9. Problem Report State**

Figure 5.2 shows what state the 48697 problem reports filed in the project lifetime are in. As described over the report goes through the states open, analysed, patched, feedback and closed, or is suspended should communication problems with the report originator occur.

From the figure we see that the majority of the reports that have been received are closed. We can also see that there are far more open reports than reports that are currently being worked upon (543). This indicates that there is not capacity enough for reports to be worked on when they arrive and that reports queue up.

To measure how long it takes from the report is filed until it is given attention, the problem reports submitted until February 26th are analysed. Every report has an "Arrival-Date" field that is automatically added by GNATS when the report is received. If a state is changed, a "State-Changed-When" field is added to the report. To see when the report is first given attention, we examine the first time a State-Changed-When field appears.

The data material contains too many entries to be listed here, and references to the data can be found in the appendix. The average time was 76.82 days with a median of 6 days and standard deviation 184.19. More than 60% of the reports had their state changed within 15 days. Not included are the 5376 reports that have not yet gotten any attention. The following figure illustrates the distribution of samples ordered by time taken for the report to get attention. Note that because of software limits in the graph-creating program, the number of samples had to be halved.

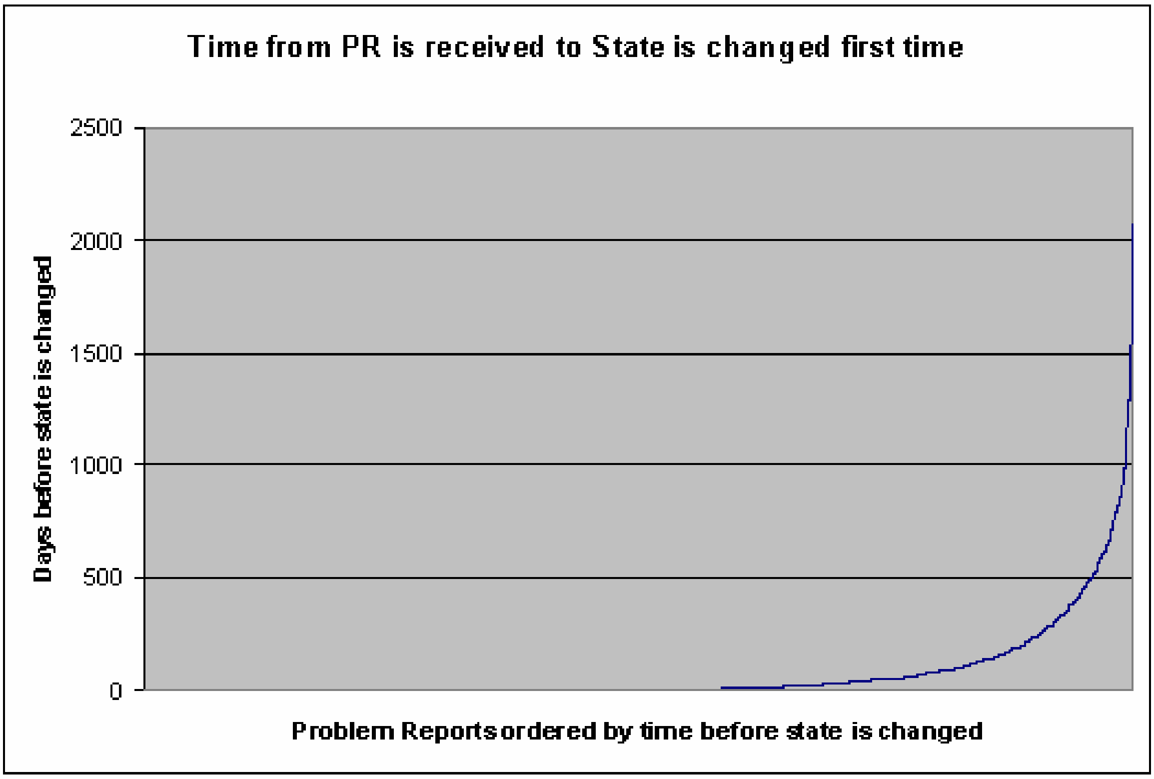


**Figure 10. Time from Problem Report is received until State is changed first time**

Figure 10 shows the problem reports sampled ordered by how many days it takes until a problem report's state is changed the first time. The further to the right a sample is, the more time elapsed before it was given attention.

From this figure, we see that most reports receive attention very quickly, but those in the 4th quartile can use a very long time to receive attention. Thus the general case is very good, but there are many cases are significantly worse than the general case.

To measure how long it takes from a report is received to responsibility is assigned, we measure the field "Arrival-Date" to the first occurrence of "Responsible-Changed-When". This field is added every time a report is assigned to someone.

The data material contains too many entries to be listed here, and references to the data can be found in the appendix. The average time was 29.96 days with median 0 days and standard deviation 119.96. Not included are the 22311 reports, or 45.8% of the total reports, that do not yet have any responsible person or group assigned to it. The following figure illustrates the distribution of samples ordered by time taken for responsibility to be assigned.

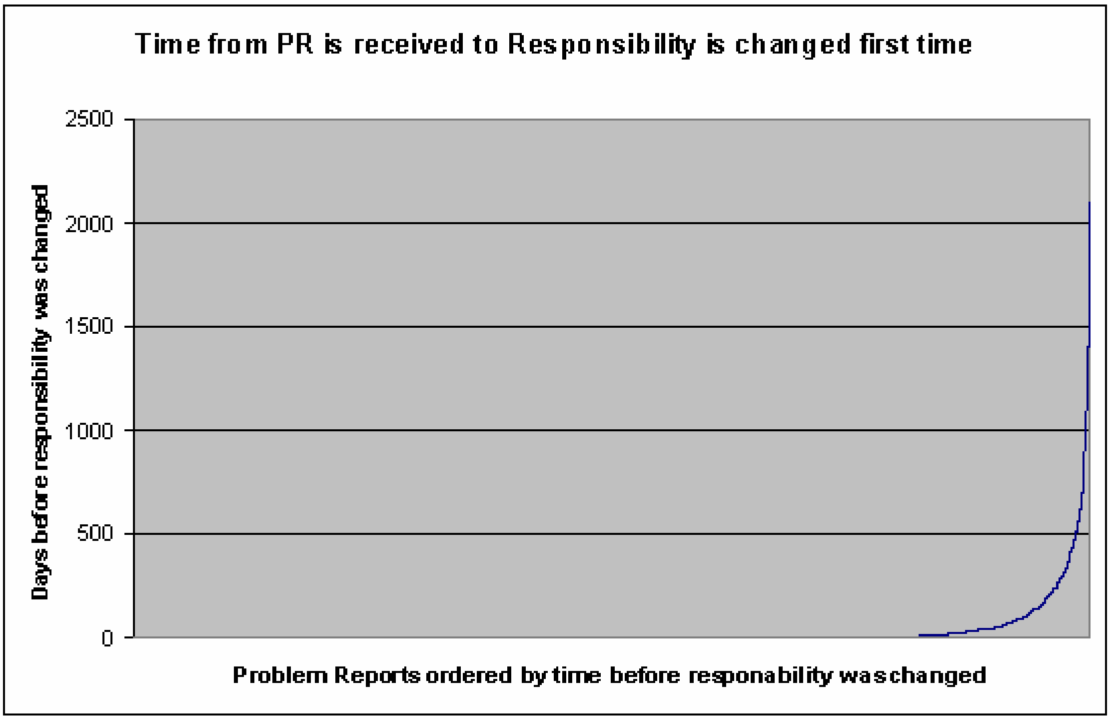


**Figure 11. Time from Problem Report is received until Responsibility is changed first time**

Figure 11 shows the problem reports sampled ordered by how many days it takes until a committer is given the responsibility of a problem report. The further to the right a sample is, the more time elapsed before responsibility is assigned.

From the figure, we see that just as with the time it takes for the report to be analysed, in the general case it takes very little time to assign an issue to someone. However, a good portion of reports remain unassigned for a long time.

Therefore it is interesting to analyse old reports with regards to if there is a connection between unassigned reports and unchanged logs.

A script has been created that matches reports that have not yet been analysed with reports that do not have someone responsible for it. To ensure that no recently submitted reports are included, we limit the check to reports that are older than December 1st, 2002. The result is that 1770 of the 3735 reports that did not have anyone assigned to it had not been changed.

This means that 1770 reports seem to be lost. But it also means that even though no-one has been assigned to the report, someone has been working on the remaining 1965 reports.

## 5.4. Distribution

FreeBSD is distributed in three major ways: through CD-ROMs and DVDs made by third parties in retail outlets, as CD-ROM and disc images that are downloaded from FreeBSD servers and burned to CDs or put onto discs by users, and through CVSup.

I have not been able to determine how available CD-ROMs, DVDs and images are because many companies do not want to share their sales statistics. Collecting logs from distribution sites has also proved difficult. However, the official CVSup servers are handled centrally by the CVSup Mirror Site Coordinator.

As noted in Section 3.5.2.4, we need to know how many servers are available for users and how available these really are. To see how many servers there are today, we check the file doc/en_US.ISO8859-1/books/handbook/mirrors/chapter.sgml and find 128 CVSup mirrors by February 26th, 2002. While this seems like a large number to choose from, this is only a snapshot in time. To measure how many CVSup mirrors have been added or removed over time, the CVS logs of the before-mentioned file have been analysed to and including January 2003.

| Month | # Added | # Removed |
|---|---|---|
| January, 2003 | 1 | 1 |
| December, 2002 | 0 | 0 |
| November, 2002 | 0 | 0 |
| October, 2002 | 1 | 0 |
| September, 2002 | 2 | 0 |
| August, 2002 | 1 | 0 |
| July, 2002 | 0 | 0 |
| June, 2002 | 0 | 0 |
| May, 2002 | 2 | 0 |
| April, 2002 | 5 | 1 |
| March, 2002 | 1 | 0 |
| February, 2002 | 2 | 0 |
| January, 2002 | 1 | 1 |
| December, 2001 | 1 | 1 |
| November, 2001 | 3 | 1 |
| October, 2001 | 0 | 0 |
| September, 2001 | 3 | 0 |
| August, 2001 | 0 | 1 |
| July, 2001 | 1 | 0 |
| June, 2001 | 2 | 0 |
| May, 2001 | 1 | 0 |
| April, 2001 | 3 | 0 |
| March, 2001 | 1 | 0 |
| February, 2001 | 0 | 0 |
| January, 2001 | 1 | 0 |
| December, 2000 | 4 | 0 |
| November, 2000 | 4 | 0 |

| Month | # Added | # Removed |
|---|---|---|
| October, 2000 | 0 | 0 |
| September, 2000 | 3 | 0 |
| August, 2000 | 3 | 1 |
| July, 2000 | 2 | 0 |
| June, 2000 | 1 | 0 |
| May, 2000 | 0 | 0 |
| April, 2000 | 4 | 0 |
| March, 2000 | 2 | 0 |
| February, 2000 | 1 | 0 |
| January, 2000 | 2 | 1 |

Table 5.3. Added / Removed CVSup servers

Table 5.3 shows the number of CVSup servers added and removed in the period January 2000 to and including January 2003. While the number of servers added fluctuates, that table shows that no more than one server is removed per month. The following figure illustrates CVSup mirror servers added:

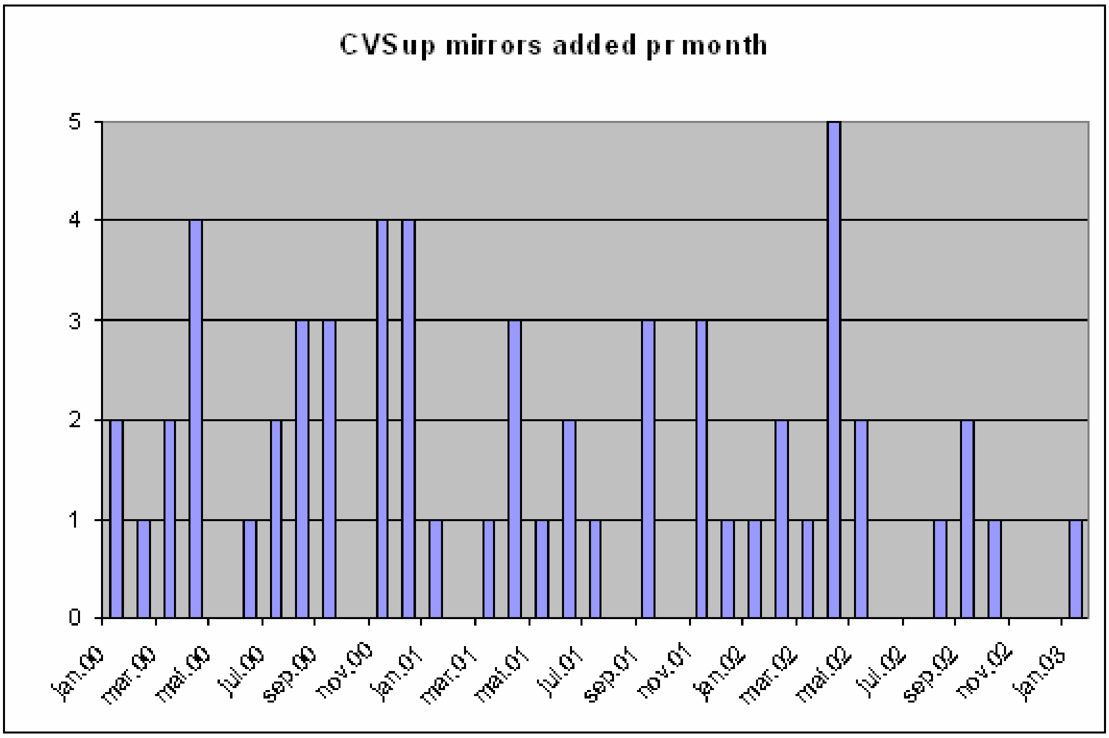


**Figure 12. CVSup mirrors added per month**

Figure 12 shows the number of CVSup server added in the period January 2000 to and including January 2003. We see that it varies somewhat, contains one spike and many months with no servers added at all.

We see from the data that there has been a steady growth of CVSup servers. According to the CVSup Mirror Site Coordinator there should rarely be any waiting time for the official servers. To measure this, a script was created to query each CVSup server regularly.

Doing a

```
more cvsup.result.010303.txt | grep "Cannot connect" | uniq | wc -l
```

gave 11 servers that were unavailable, and upon inspecting the resulting file further, one server refused connection two times and another one time (of a maximum of five retries) due to too many concurrent users. Two of the servers failed with an "Inactivity timeout" during the synchronisation.

There were no prior statistics on how available the servers were. This test has therefore been done four times. The following table shows the date the test started and how the different servers responded during the test:

| Date | Successes | Cannot Connect | Queues | Inactivity error |
|---|---|---|---|---|
| March 1st, 2003 | 113 | 11 | 2 | 2 |
| April 1st, 2003 | 117 | 6 | 3 | 2 |
| April 4th, 2003 | 118 | 7 | 2 | 1 |
| April 8th, 2003 | 112 | 9 | 5 | 2 |

Table 5.4. Availability of CVSup servers

Table 5.4 shows successes and three categories of errors for four samples of CVSup server availability. The table shows that queues vary and are in general low. The inactivity errors reported were all from the same servers, and thus remained quite constant. While the total number changed a bit from day to day, the total number of servers that were down at least once during the surveys was 14. Looking further through the data, we find that many of the servers were down permanently during this period, thus possibly being permanently down and due for removal. However, for the main part the servers are available and have no queues, so we must conclude that they are available.

## 5.5. Summary

In this chapter we have looked at several administrative parts of the FreeBSD Project. From what we have seen we can draw the following conclusions:

- Several tools are used to support administrative functions and ensure consistency between related items (such as code changes and problem reports).
- Mailing lists are used as the main form of communication within the project
- Problem reports are kept in a standardised, machine- and human-readable format.
- Until February 26th, 2003, 48697 problem reports had been filed. Of these, 44485 reports were closed.
- Most problems are resolved quickly. More than 60% received attention within the first 15 days But when problems are not resolved immediately, they may take very long to solve.
- Most problems are assigned to someone in the project when they arrive. Unassigned problems may take long before being assigned.
- Members of the project work with problem reports that are not assigned to anyone.
- FreeBSD is distributed by CD, DVD and CVSup. The data for number of CDs and DVDs sold or burnt are unavailable. CVSup is a tool for distributing updated source code through the net. There are 128 official CVSup servers to which people can synchronise their systems, and the number of available servers is growing.
- 8-13% of the CVSup servers are at the sampled times unavailable.

# Chapter 6
# The FreeBSD development model

In this chapter we will explore how development is done within the FreeBSD Project. We start by looking at the development through a software process model, and classify the development into traditional software development stages. Then we will look at how releases are done and what standards are being followed. Finally, we look at how third party software is integrated in FreeBSD systems.

## 6.1. Evolutionary software process model

The UNIX development effort was, already before the CSRG became involved, using an evolutionary model. The sharing of ideas led to both evolution of better and worse tools and features, and the ones that proved beneficial to the community became de-facto standards while others remained curiosities. Based on this, we must also believe that there have been parts that were superior, but have not gained the community's awareness or approval and has faded away even in being superior.

FreeBSD follows this tradition of using an evolutionary software process model. It implements this through parallel change-integration. [Jørgensen, 2001] proposes the following model for the integration of changes into the project:

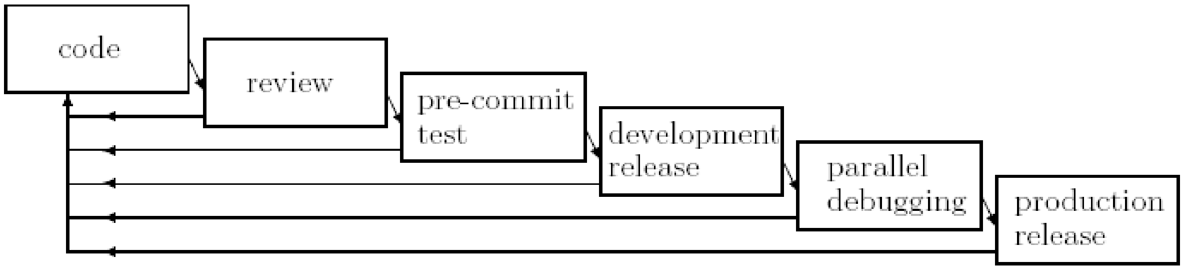


**Figure 13. Jørgensen's model for change integration**

This figure shows that after coding a change, review is sought from the developer's peers. After being reviewed, the change is tested, and when deemed well tested it is integrated into the development release, called FreeBSD-CURRENT. In -CURRENT it undergoes further scrutiny to find problems, and when deemed mature enough, it is merged into the production release, called FreeBSD-STABLE. A failure at any of these stages may lead to further coding and a restart of the cycle.

During the year December 1st, 2001, to December 1st, 2003, 132148 changes were integrated into -CURRENT. When a change is integrated, it is called a "commit". This means that there are many such processes ongoing at the same time, and this massively parallel development and integration can be illustrated as the rolling wheel that is the evolutionary model.

A property with the evolutionary model is that frequent releases ensures that developers get early feedback from the customer on what features are not wanted so that they do not spend long developing unwanted features. This is also true for the FreeBSD Project. No commits may break the system, as the system should be able to build and be used at any point in time. This makes the releases very frequent, and is in line with [Raymond, 2000] who states that developers should release early and release often. 132148 commits mean in practise 132148 available releases during one year. This enormous availability makes it possible for developers and users to review new development features quickly, thus enabling issues regarding unwanted or controversial features to come to the developer attention quickly.

[Jørgensen, 2001] notes that the commits are for the most part maintenance changes. The commits are classified according to the classes defined by [Swanson, 1976] and shows how they apply to FreeBSD. These are

- "Adaptive changes" - the creations of new device drivers and ports
- "Corrective changes" - security fixes
- "Perfective changes" - improved performance and addition of minor features

The release of FreeBSD 5.0 includes many new features that have been developed that are too radical changes to fit into these categories. Therefore, the following model is proposed for development of new features:

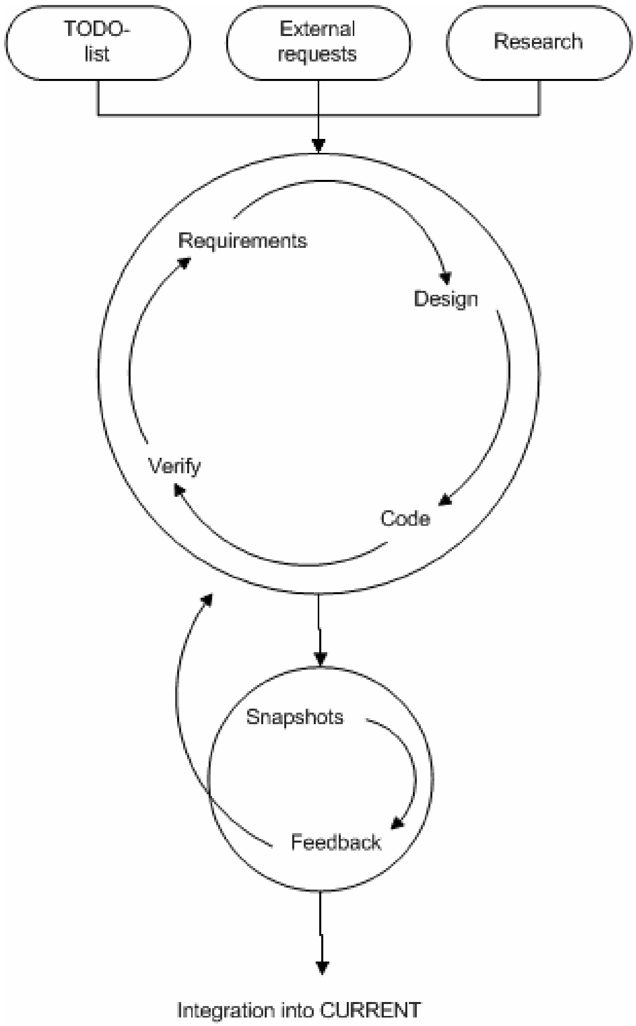


**Figure 14. My model for development of new features**

The development of a new feature or a major redesign of an existing feature is spawned by either a wish by the developer or the community (several TODO-lists are kept, both on the FreeBSD project level and at subproject level), by demand of a company that is paying to have a feature made, or as a part of a research project. This leads to a development effort using traditional software engineering models and techniques. At milestones the project may decide to release snapshots that people are free to try and give feedback on. A special case of such snapshots are when the code is deemed mature enough to be integrated into -CURRENT. This lets the code be more widely tested, and does not restrict the continued development from the project in any other way than that fixes to problems introduced by the project are expected to be handled and integrated swiftly into -CURRENT.

## 6.2. Traditional development stages

This section will frame the practises of development within the FreeBSD Project within the traditional development stages: requirements, design, implementation, verification and deprecation.

### 6.2.1. Requirements

Developers have their own development methods. [Jørgensen, 2001] has a figure [Section 6.1] describing how each individual's own work is integrated into the project. The main part of this figure is "Code". The method each developer has is for the coding or the making of their contribution. During my interviews, not one developer has described the way they work with coding the same as the next developer. The development models they used ranged from the traditional waterfall model to agile methods.

As each developer up until the delivered code has his own method, there is no standard way of gathering requirements and analysing them and prioritising among them. For the Bluetooth stack implementation discussed in Section 4.6.2, the primary requirements were to implement the "Bluetooth Specification Book" version 1.1, and a TODO list was kept of bugs, missing functionality and suggested features. For the similar projects, such as the implementation of smart card support, standard documents (ISO/IEC 7816) were also the primary source for requirements. All the people I have interviewed have kept TODO lists.

Maintenance work does also gather requirements. As noted by [Jørgensen, 2001], FreeBSD is maintenance-centric. Requirements are gathered by the maintainers through feedback the maintainers get through Problem Reports. These Problem Reports, held in the GNATS maintenance database, serve as a TODO-list together with the maintainer's wishes and ideas.

### 6.2.2. Design

As noted in the requirements section, design notes are not kept systematically. There are neither design standards nor requirements to how design should be done. Therefore, the outputs of the design processes are lost.

As noted by many authors on open source, many of the successful open source projects have reimplemented something that has been developed before. Much of the design is already done and tacitly understood by the people involved in the project. When implementing standards, much design work has also already been done. Some standards, such as the Bluetooth specification, even include implementation guidelines. Many people in the FreeBSD Project and the open source world view such standards and previous implementations as design enough.

The only recognised design documents is [McKusick et.al, 1997]. People wanting an overview of the kernel, either for understanding it out of interest or for contributing to its development are referred to this book. The reason for this is that the design is regarded to be relatively stable after more than 20 years of operating system development. FreeBSD has very much stuck to the file structure and interfaces of BSD 4.4, although much of the code has been altered or replaced and many bits have been added. This book provides an introduction and understanding of how the kernel is organised and what the various parts do and how they interface one-another. An update of this book is in the works that will describe FreeBSD 5.0, but for the first ten years of the project, this book has provided the stability of design.

Developers who are starting to work in a new area of the code should be careful to study the commit logs of the files in that area. The commit logs will often explain why a certain design has been chosen should there be multiple options or choices made that are different from what would be expected. This ensures that developers do not make the same mistakes as have been done before or reimplement disputed or unwanted code.

### 6.2.3. Implementation

#### *6.2.3.1. Committing*

The FreeBSD development is based on initiative. A developer makes something he wants to share with the community. If he has commit privileges, he commits his change to the FreeBSD-CURRENT branch. If not, he contacts a committer involved in that area of the code, who reviews the change and commits it if he approves it.

According to two interviewees, many committers do not check old commit logs before applying changes to a section of code. The result is that errors that had been done before happen again and that interface design that was left for a better design is reintroduced. Other results are that people do not know who the maintainer is of the module or who made the last commits and thus do not know neither their wishes nor the history of the file.

#### *6.2.3.2. Code modifications*

As discussed in Section 3.5.2.3, we examined commit logs from the timeframe December 1st, 2001, to December 1st, 2002. There were 132,148 commits in this period according to commitscan.pl (more on this utility in Appendix A). From this we must conclude that committers can easily add code, and do in fact do so.

To determine how easy it is for new committers to commit code modifications, we examined the results for all committers who got their commit privilege in August and September 2002. As this is not an investigation into how many commits individuals make, we will only list the date of becoming a committer and the number of commits made by that person by December 1st, 2002.

To be able to compare this with the other data, we need to extrapolate for the entire year. This is done by dividing the days between the committer joined and

December 1st, 2002, by 365 and dividing the number of commits made by this factor.

| Date of becoming a committer | Commits | Extrapolated total |
|---|---|---|
| August 7th, 2002 | 119 | 374.44 |
| August 8th, 2002 | 5 | 15.87 |
| August 14th, 2002 | 122 | 408.53 |
| August 15th, 2002 | 119 | 402.18 |
| August 20th, 2002 | 39 | 138.20 |
| August 21st, 2002 | 965 | 3453.19 |
| September 1st, 2002 | 65 | 260.71 |
| September 10th, 2002 | 52 | 231.46 |

Table 6.1. New Committers commit-rate

For the real data, this gives us a mean of 186 commits, median of 92 commits and standard deviation of 318. For the extrapolated data, this gives us a mean of 661 commits, a median of 318 commits and a standard deviation of 1137. From this we must conclude that new committers can also easily add code.

#### *6.2.3.3. Modules*

A module is parts of code that logically belong together and interface one another far more than they, as a logical unit, interface other parts of the code.

FreeBSD is divided into modules that are available in a source code tree for each branch. The branch we have chosen to analyse is the development branch, -CURRENT. The tree is connected with Makefiles that describe how the underlying system is being built. A Makefile may include multiple Makefiles that lie further down in the tree, and with a few exceptions, each leaf node can be called a module. The exceptions are where one Makefile will control multiple modules, but these modules have been identified through listing modules known to their respective configuration utilities.

A total of 716 modules have been identified using the script findmodules.pl (more on this utility in Appendix A). We must thus conclude that FreeBSD is divided into many modules.

### 6.2.4. Verification

Open source projects have in particular been credited for being well verified, in particular through review. The prime example of this is Eric Raymond's quote of Linus Torvalds: "Given enough eyeballs, all bugs are shallow". [Raymond, 2000] The FreeBSD Project uses testing and review to verify its product.

#### *6.2.4.1. Tests*

As noted in the requirements section, test cases and test suites are not gathered in a common test suite. Up until version 5.0, FreeBSD did not come with a test suite at all. A testing framework with a minimal test suite is available now, but very few have started submitting tests for their code, and this is not felt as a thing that should be done rather than a thing that would be nice.

One example of why tests were used is that the developer of the test framework wanted to rewrite the "xargs" utility. To ensure that no functionality changed as compared to the original version, she wrote a number of tests to which she compared the new version. Had the utility behaved differently, the consequences may be that many user programs relying on the utility would break.

There are very few tests written today compared with the number of modules in the system. As one committer put it, "we test FreeBSD on the user's system". While it is not recommended, many people run the development version, FreeBSD-CURRENT. Feedback from these gives valuable input as to how stable the code added is, giving the developer information as to whether it is safe to include this code in FreeBSD-STABLE.

Another testing mechanism is tinderbox builds. This consists of individuals having dedicated computers to rebuild the system on different platforms every day. When these tests fail for an architecture, a mail is sent to the mailing list for that particular branch. These tests usually find problems in software that compiles fine on one architecture but not on another that the developer did not test his code upon.

In conclusion, a test framework has recently become available, but it is little used at the moment. Most testing happens on the user's hardware and on dedicated tinderboxes.

#### *6.2.4.2. Review*

The Committers Guide [FreeBSD, 2001] says "Discuss any significant change before committing" and in conclusion of its elaboration says "When in doubt, ask for review". For this to work, one would expect to find a good review culture. But, according to my interviewees, changes involving little code are more likely to get review than large commits. Complex code or large portions of code are less likely to be reviewed. This is supported by the findings in [Jørgensen, 2001].

In contrast to my interviewees experiences, [Jørgensen, 2001] found that 86% of the surveyed committers had received feedback on their latest request for review (before committing the code). The article does however not distinguish between comment on the rationale for a commit or review of the actual code. This difference was seen in the case of the OpenPAM implementation discussed in Section 4.6.3.

Review is important not only before integration, but especially when a change has been committed to -CURRENT. Every commit sends a commit log mail to the developers. These commit logs makes peer review easy. Commit logs contain an explanation of what has been done, why it has been done and references to the modifications and files affected. Thus the rationale for the change is stored and it is easy to get to the actual code for review.

As with large commits that do not get reviewed properly, committers are more likely to get comments on the basis of their rationale for making the change, and not on the code itself. The code committed must in most instances be read together with the entire file, which constitutes as much reading as with a larger commit if the reviewer does not already know the code. Thus, code committed does only really get verified by the compilers and through running on the systems of other developers and users who run development releases.

During implementation of Bluetooth, the developer added "thank you"-notes to those peers that had pointed out flaws in his code. In his experience, this encouraged peer reviews. During the implementation of OpenPAM, the developer had to actively request review multiple times in order to get any. There came little review until he committed OpenPAM, and most of the comments were on the basis of his rationale for integrating it rather than the code itself.

We must thus conclude that most review happens on the basis of the commit log. The commit log helps other developers be on top of what happens in the project and express their opinion on that. Little review happens on the basis of the code itself, especially for larger pieces of code.

### 6.2.5. Deprecation

An important aspect of maintenance is removing old code. This is done in the FreeBSD project by mailing a message to the developer community putting it out for vote.

For instance, in the development of FreeBSD 5.0, Ruslan Ermilov suggested renaming the library libexpat to libbsdxml because the XML code, originally based upon expat, deviated so much from the original code that leaving it with the original name, which originally was done to credit the expat library, was misleading. [Ermilov, 2003]

Although deprecated code should stay with FreeBSD for at least one major version after the decision to deprecate it has been made, there are exceptions to this rule. One example of this is Poul-Henning Kamp's argument [Kamp, 2003] for deprecating the previous systems for the new disk handling system GEOM and device handling system DEVFS. [FreeBSD, 2001: 9.4]

## 6.3. Release Engineering

Release engineering is the process of making an official release of a branch. This section will give details that lead up to the release engineering process found in the project model, section 8.5.9.

During the release engineering process, the release engineering team uses Perforce for the work even though the project uses CVS. This allows the project to move on even when the release engineering team requires a stability period that is to long for the project to wait. This is especially the case with major releases. The use of perforce also means that they have more version options available without having to clutter the CVS tree much. Rather than setting a tag for every milestone (such as a release candidate) in the release engineering process, they can set only a final tag when the release is made. In this way, they avoid CVS' poor branch handling.

### 6.3.1. Major releases

Major releases (.0 releases) are releases of the development branch. They are likely to include radical technological changes whereas minor releases (.X releases) are not. Jordan K. Hubbard wrote that "a ".0 release" translates to [...] New, interesting, sort of usable, buggy in the extreme." [Hubbard, 1994] Therefore it is necessary for more testing and bug fixing before releasing a major release than a minor release.

Examples of such new radical changes is that version 2.0 was based on a newer code base from Berkeley than 1.0 was, 3.0 was in transition of changing the binary format, 4.0 included support for IPv6 networking along with security features such as OpenSSL and OpenSSH and the possibility to jail applications, and 5.0 has a new disk handling system and has a brand new SMP handling system. 6.0 is planned to among other features include kernel thread handling for userland threads.

### 6.3.2. Minor releases

Minor releases are usually releases of the production branch (-STABLE). Sometimes they may be of the development release if the development branch has been through a stabilisation process following up to a major release and this is the subsequent release.

### 6.3.3. Release Candidates

A "release candidate" is a version aimed at early adopters that would like to try out a new version. It has been through stabilisation, but it is not yet tested enough to be considered an actual release. Even though features that are integrated into -STABLE have been through much testing in -CURRENT, the production branch (-STABLE) should not be considered an ongoing release candidate. Changes to release candidates require the explicit approval of the Release Engineering team, and release candidates are used for much more widespread testing. Also, release candidates are used to see that bugs reported in one release candidate are gone in the next.

An example of the benefits of having separate release candidates is that before an upcoming release, many committers will merge code from the -CURRENT to the -STABLE branch. This is possibly the most unstable time for the -STABLE branch and thus a bad time to upgrade systems in a production environment. Those following the -STABLE branch are assumed to understand this and should be careful when release engineering periods begin to evaluate if they really would like to upgrade their systems at that time.

### 6.3.4. Security Branches

A security branch is a branch that is identical with a release, but has security patches applied to it. Each node along the branch is called a patchlevel. People following a security branch are ensured that nothing else will change in their system, so that no programs will change behaviour. This is particularly useful in a production environment as server needs a higher degree of predictability yet must be updated with the latest security patches at all times.

As mentioned, before a code freeze for an upcoming release starts, many committers will merge those parts of their code they deem stable from the -CURRENT to the -STABLE branch. Because of the inconsistencies that can occur during such a mass-merge, this is not a recommended period of time to upgrade the system. However, not upgrading may be unacceptable as some changes such as security fixes are necessary to have installed as soon as possible in production environments. Security branches have solved this problem, and administrators can track a security branch rather than the -STABLE branch.

### 6.3.5. Release frequency

As mentioned in Section 3.5.1.3, new releases should be made regularly. Release history also makes it easier to see whether the development model and the release engineering process work.

The following table shows the release dates for every version and how long time it was since the previous release, measured in days.

| Version | Release date | Number of days since last release |
|---|---|---|
| 1.0 | 1.11.1993 | - |
| 1.1 | 6.5.1994 | 186 |
| 1.1.5 | 29.6.1994 | 54 |
| 1.1.5.1 | 5.7.1994 | 6 |
| 2.0 | 22.11.1994 | 140 |
| 2.0.5 | 10.7.1995 | 230 |
| 2.1 | 19.11.1995 | 132 |
| 2.1.5 | 15.7.1996 | 239 |
| 2.1.6 | 16.11.1996 | 124 |

| Version | Release date | Number of days since last release |
|---|---|---|
| 2.1.6.1 | 26.11.1996 | 10 |
| 2.1.7 | 20.2.1997 | 86 |
| 2.2 | 16.3.1997 | 24 |
| 2.1.7.1 | 19.3.1997 | 3 |
| 2.2.1 | 25.4.1997 | 37 |
| 2.2.2 | 16.5.1997 | 21 |
| 2.2.5 | 22.8.1997 | 98 |
| 2.2.6 | 25.3.1998 | 215 |
| 2.2.7 | 22.07.1998 | 119 |
| 3.0 | 16.10.1998 | 86 |
| 2.2.8 | 29.11.1998 | 44 |
| 3.1 | 15.2.1999 | 78 |
| 3.2 | 17.5.1999 | 91 |
| 3.3 | 17.9.1999 | 123 |
| 3.4 | 20.12.1999 | 94 |
| 4.0 | 14.3.2000 | 85 |
| 3.5 | 24.6.2000 | 102 |
| 4.1 | 27.7.2000 | 33 |
| 4.1.1 | 27.9.2000 | 62 |
| 4.2 | 21.11.2000 | 55 |
| 4.3 | 20.4.2001 | 150 |
| 4.4 | 20.9.2001 | 153 |
| 4.5 | 29.1.2002 | 131 |
| 4.6 | 15.7.2002 | 167 |
| 4.6.2 | 15.8.2002 | 31 |
| 4.7 | 10.10.2002 | 56 |
| 5.0 | 19.1.2003 | 101 |

Table 6.2. FreeBSD release dates

Table 6.2 shows the releases of FreeBSD since its first release in November 1993 to and including its latest release before April 1st, being in January 2003. It shows how many days have passed between each release as a measure to how regular the releases are and provides the version number of the release.

For the project lifetime, this gives an average of 96,2 days between every release with a standard deviation of 62,9 days. If we look at minor releases, not including security releases since January 1st, 2000, a release is made every 107,4 days, with a standard deviation of 41,7 days.

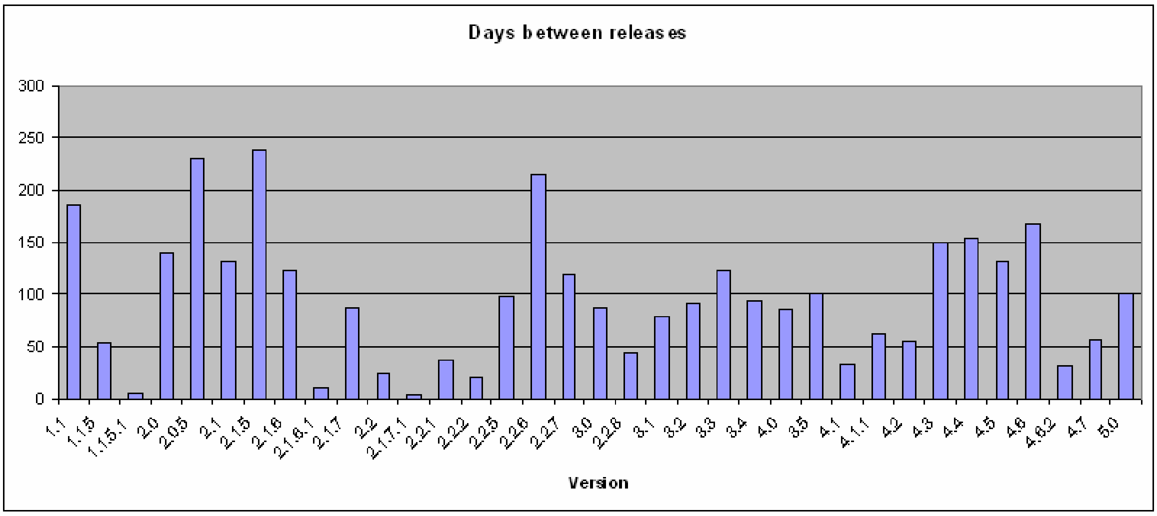


**Figure 15. Days between releases**

Figure 15 shows the amount of days passed between each release of FreeBSD in the time period November 1993 to January 2003. This shows how regular the releases are. The closer the bars are to the neighbouring bars, the more regular are the releases.

Because the only condition for when a FreeBSD release is made is the time, it would seem that a FreeBSD release is no more than a snapshot of the evolving project in time. However, because of the release engineering process used, a version is indeed a well tested and trustworthy product. The user community has made it clear that it does not tolerate reduced security as a result of rushing to meet a self-imposed release deadline. A result of this was that the release of FreeBSD 4.6.2 followed shortly after 4.6 to solve numerous issues.

Putting the evolutionary model, the new feature development model and the version tree together, we get the following model. At the top we have the new feature development that is followed by the integration model proposed by [Jørgensen, 2001]. This model is used for maintenance changes as well. The third last step in Jørgensen's model is the commit living in the -CURRENT branch. Then it is merged from current into the -STABLE branch. A release along the -STABLE branch marks the beginning of a security branch. Finally, older versions are not supported any more.

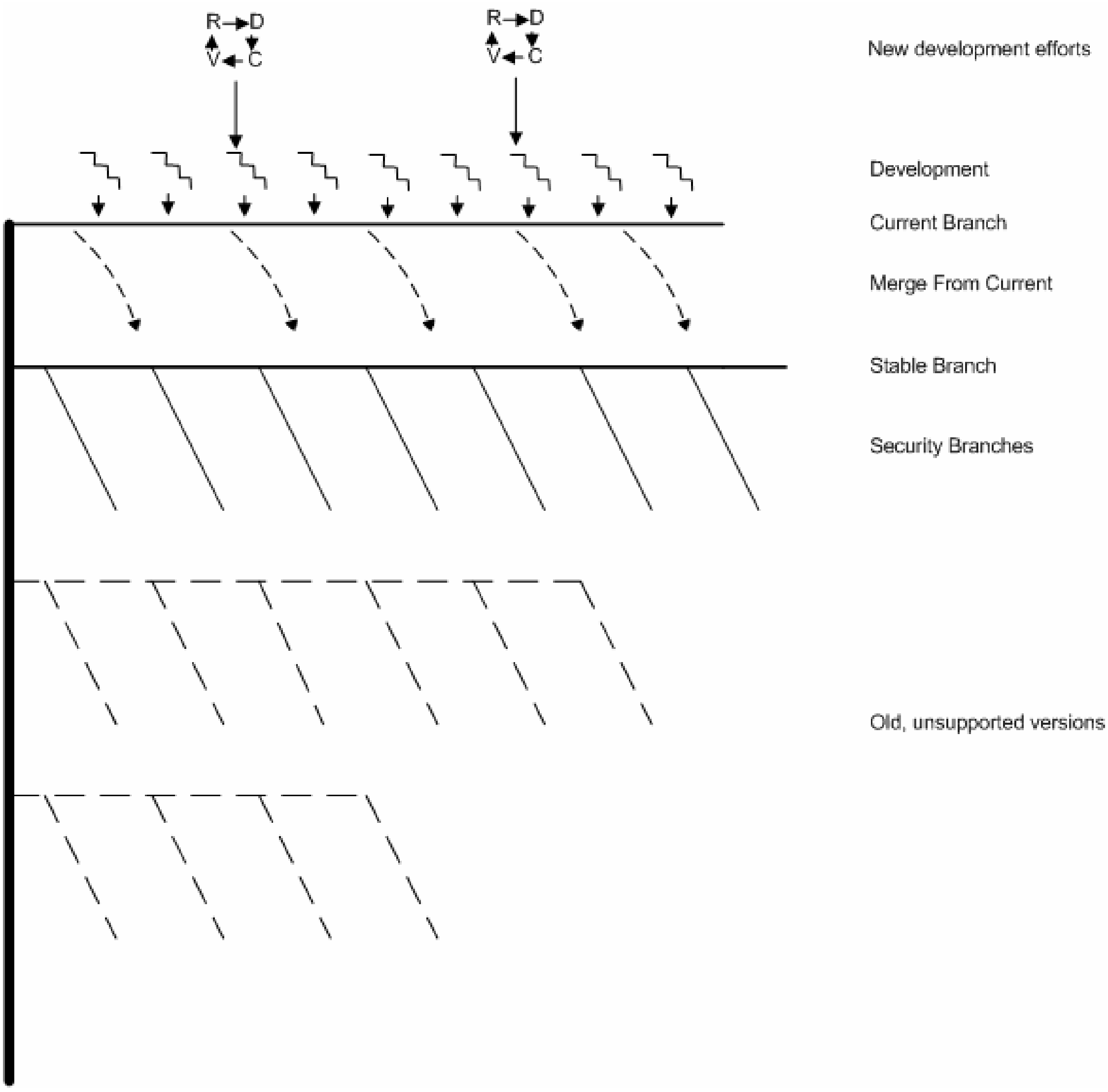


**Figure 16. The overall development model**

In conclusion, the FreeBSD Project provides multiple versions of FreeBSD intended for audiences with different priorities between the newest features and rock solid stability and security. This means that the newest minor release or update along a security branch is already old in terms of being well tested when it is released.

We have also seen that new versions of FreeBSD are released on a regular basis and that the release model fits in with the overall development model.

## 6.4. Standards

People wanting an overview of the kernel, either for understanding it out of interest or for contributing to its development are referred to [McKusick et.al, 1997]. The reason for this is that the design is regarded to be relatively stable after more than 20 years of operating system development. And this book provides an introduction and understanding of how the kernel is organised and what the various parts do and how they interface one-another. An update of this book is in the works that will describe FreeBSD 5.0, but for the first seven years of the project, this book has provided the stability of design. The implementations, on the other hand, have changed to meet changing needs.

Other than this book, there is no target list or list of implemented standards. There is no way for users to be certain that features that have a standards specification are standard compliant other than to test them themselves. New standard implementations are announced with the announcement of a new FreeBSD release.

Having said that, the project does implement many standards, especially for supporting applications (i.e. POSIX, ANSI C) and communications with other networked systems (i.e. SSH, TCP/IP, NFS)

## 6.5. Number of ports

A port is the meta-data required to fetch the source code for a third party application and build it to run on the system. It serves as the basis for package generation, and many users use the ports system to track their installed third party software.

In the root of the ports tree, there is a file called INDEX containing one line of information for every port. Counting the lines of this file tells us that February 18th, 9:45 PST, there were 8153 ports available.

A port can have one of either two states: working or not working. A port that is working can be compiled into a package, whereas a non-working port can be in one of three states: that the source code was unavailable, that a dependency was a non-working port or that the port failed to build correctly. A port author can mark a port as being broken. In this case, the port build will automatically fail without doing any build attempts. It needs not be marked the same for all versions of FreeBSD.

Bill Fenner has made a survey regarding how many ports do not have their source code available. He has also made available package building logs. The survey data used are based on a CVS snapshot from February 18th 9:45 PST, while the package build is based on a CVS snapshot from February 16th, 17:22 PST.

Because ports is not branched, the same amount of ports and ports with unfetchable sources will be available to both systems. However, build results may vary. Also, build results may vary between platforms the system is built for. The two platforms supported by both 4-STABLE and 5-CURRENT are "i386" and "alpha".

| FreeBSD Branch | # of Ports | # build errors | # without source | resulting # of port breaks | # marked broken |
|---|---|---|---|---|---|
| 4-STABLE (i386) | 8153 | 419 | 502 | 499 | 55 |
| 4-STABLE (alpha) | 8153 | 1894 | 502 | 18358 | 2 |
| 5-CURRENT (i386) | 8153 | 600 | 502 | 908 | 69 |
| 5-CURRENT (alpha) | 8153 | 746 | 502 | 1182 | 71 |

Table 6.3. Ports status

Table 6.3 shows a breakdown of ports compiled for two different branches at two different platforms. The second column is constant, being the number of ports sampled. The third column is how many builds failed among these ports. The fourth number is constant, saying how many source files could not be found. The fourth column is how many ports fail because they depend on this port. The final column is how many of the ports were marked as broken for this branch on this platform.

We can note from the table that ports marked as broken is different on different architectures and versions. This is because some ports come with binaries that are dependent on an architecture, or that some ports require features that are not available in both versions of FreeBSD.

From the table we find that a number of ports need to be revised in order to work correctly. The amount of breaks is far larger than expected breaks. Also, the amount of breaks on the alpha architecture is higher than the i386 architecture. This is probably due to the far greater part of the community using the i386 architecture and thus a lack of port testing. The amount of breaks is a result of not enough people working on updating the ports, and there is certainly room for improvement here.

There is probably an error in the calculation of resulting ports breaks, as the actual number of port breaks is lower than the build errors. Especially the number for "4-STABLE (alpha)" is much higher than the real number of build errors. Still it is important to track the number of ports breaking because of other ports, and as this measure is meant to be a base line, it should be tracked. It is only by tracking and reporting them that we can expect the numbers to be corrected and improved.

## 6.6. Summary

In this chapter we have found a development model for FreeBSD, both in terms of maintenance and development of new features.

- [Jørgensen, 2001]'s model for integration of changes makes together with FreeBSD's release version model and my proposed model for development of new features an overall model of FreeBSD's evolutionary software process model.
- Although most changes are maintenance changes, we have seen that some changes do not fit into the change list of [Swanson, 1976], and based on experience a descriptive model for implementing new features has been proposed.
- There is no standard way of gathering requirements. However, TODO lists are widely used, and the problem reports database is a standard source of user requirements.
- Apart from [McKusick et.al, 1997], there is no explicit design for FreeBSD. The project does not maintain a repository of design artefacts.
- Committers can, and do, add code. During one year, 132,148 code changes were integrated into the code repository.
- New committers can, and do, add code just as easily as old committers
- FreeBSD consists of over 700 modules.
- A test framework is available, but it is little used at the moment. Most testing happens on the user's hardware and on dedicated tinderboxes.
- Most review happens on the basis of the commit log. The commit log helps other developers be on top of what happens in the project and express their opinion on that. Little review happens on the basis of the code itself, especially for larger pieces of code.
- Deprecated code should be in FreeBSD at least one major version after it has been decided through a vote to deprecate it. However, there have been exceptions from this rule.

- The project provides multiple versions of FreeBSD intended for audiences with different priorities between the newest features and rock solid stability and security. This means that the newest minor release or updates along a security branch are already old in terms of being well tested when they are released.
- New versions of FreeBSD are released on a regular basis and the release model fits in with the evolutionary software process model, the change integration model and the new feature development model.
- Apart from a book describing the 4.4BSD kernel, there are little standards the project has committed to follow. Having said that, the project does implement many standards.
- FreeBSD has over 8000 ports of third party software available. A small portion of these are not working or dependent on non-working ports, and some ports that work on one architecture do not work on another.

# Chapter 7

# Issues spanning over the project model

This chapter will discuss five issues that go across the three previous chapters. In discussing these, we bind together the aspects discussed in the previous chapters, so that we can present the project model in the next chapter.

## 7.1. Bikeshedding

The project members have noted how Parkinson's Law about dynamic management applies to the FreeBSD Project. Parkinson shows that if you show a board of directors a plan to build an atomic power plant, you will get little advice on how to do it as building an atomic power plant is a very complex undertaking. Because it is so complicated, the board members, rather than try to grasp it, will assume that someone else has already checked the details. If you on the other hand show them a plan to build a bike shed, you can be sure that someone will seize the opportunity to show his insights to how it should be done, thereby showing that he is doing his job as board member. This is because building a bike shed is a task most people can do in a few days. [FreeBSD, 2003C 16.19]

The equivalent problem in FreeBSD is that many people will comment on small undertakings and how they think it should be done and what should be corrected when it is done. Larger undertakings, on the other hand, get little feedback and the author is assumed to know what he is doing. This is especially true in reviewing commit logs, as discussed in Section 6.2.4.2.

Bikeshedding is also something that inhibits design. When people submit or ask to discuss their design, many people are likely to be voicing their opinions and often not really contributing to the project. To avoid this, the developers prefer to do the work first and then submit it for integration and rather let people with opinions make the changes their opinions require themselves. This means that the first person to submit code in an area has an effect much like the anchor effect: a weight is put down in the area he wants the discussion to be heading towards. This is in line with a widely held belief in the open source community that "talk is cheap, code counts".

## 7.2. Missing Guidelines

While the integrating of minor changes and new modules that resemble old modules is well understood and documented, doing large changes in FreeBSD has few guidelines. Ideally, the code should never be broken, especially for the -STABLE branch. Breaks are also seldom accepted in the -CURRENT branch.

As mentioned in [Jørgensen, 2001], large undertakings like implementing fine grained symmetric multiprocessing (SMP) are missing guidelines. Doing this implementation broke many of the existing paradigms of how the kernel worked, and would break much of the kernel code that again would interfere with userland applications. A similar break came with version 3.1 that brought the new ELF binary format rather than the AOUT format used in previous versions. Since 2001, when Jorgensen wrote his article, I have found no evidence of guidelines for similar undertakings, indicating that we must expect similar breaks and delays due to the arising organisational difficulties.

Jorgensen also mentions in his article the lack of guidelines for assigning people to work on a change. This is backed up by [Lehey, 2002]: "The FreeBSD project is a volunteer organization, so the core team does not have a mandate to tell anybody to do any- thing." An example of an issue that would require assignment of people is to non-trivial bugs that are neither very easy to find nor cause a big stir when they are fixed.

The project maintains TODO lists and problem reports ordered by severity. [FreeBSD, 2001] even states that the maintainer is responsible for fixing reported problems. However, with more than 50% of the modules being without a maintainer, there is no obligation on any project member to fix the reported problems.

## 7.3. Project model reenforcement

In the FreeBSD project, the project model is based upon a set of guidelines that are referred to when someone asks how to do some well-documented process. Developers are expected to follow these, and the connection between them and what is not written is understood by the community. If people do not adhere to this tacitly understood project model, they will be reminded of this by the other developers. However, it is common for people to ask first if they are unsure of how to perform a task. These practises reenforce the project model.

For instance, it is written in the handbook with regards to mailing lists that proper behaviour is to look if someone has asked questions on your topic before. On the high-volume list "freebsd-current", a question regarding trimming the userland was asked. Within half an hour, the sender had been noticed that this question had been discussed many times and was encouraged to read through those discussions. [Hatfield, 2003]

## 7.4. Versioning

A much used model for a projects lifespan is that of initiation, scaling up, production and termination. This model has the timeline as X-axis and the resources available to the project on the Y-axis. [IN331, 2002]

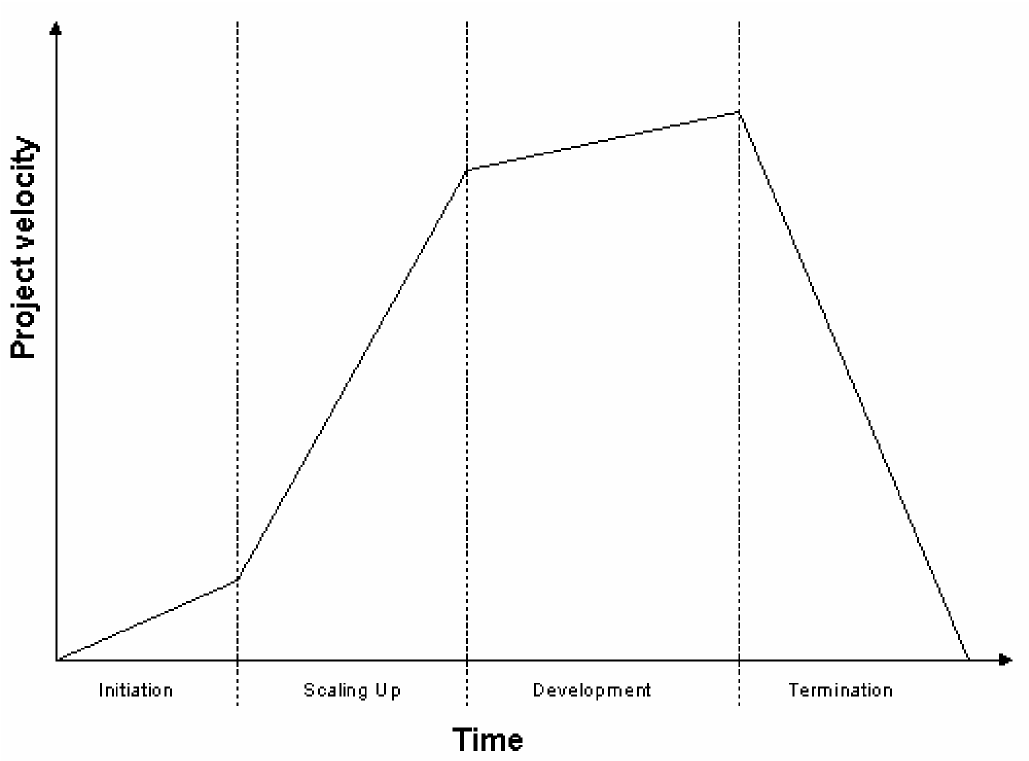


**Figure 17. Model of a release's life**

FreeBSD releases are made of one of the two branches, -STABLE or -CURRENT. The latest available release, as well as -STABLE are the versions that receive the most attention from the users with respect to problem reports and feature suggestions, while -CURRENT is the version getting the most attention from the developers. The user feedback triggers development, so the model is very much true for FreeBSD releases as well.

After a release has been in production a while, a new release engineering process is initiated. After a release has been followed by a new release, less maintenance work (with the exception of security updates) is applied to it, and development on this version is effectively terminated. This gives the following model for succeeding versions where the X-axis is the time line. The previous version is marked in green, the current version is marked in black and the next version is marked in cyan.

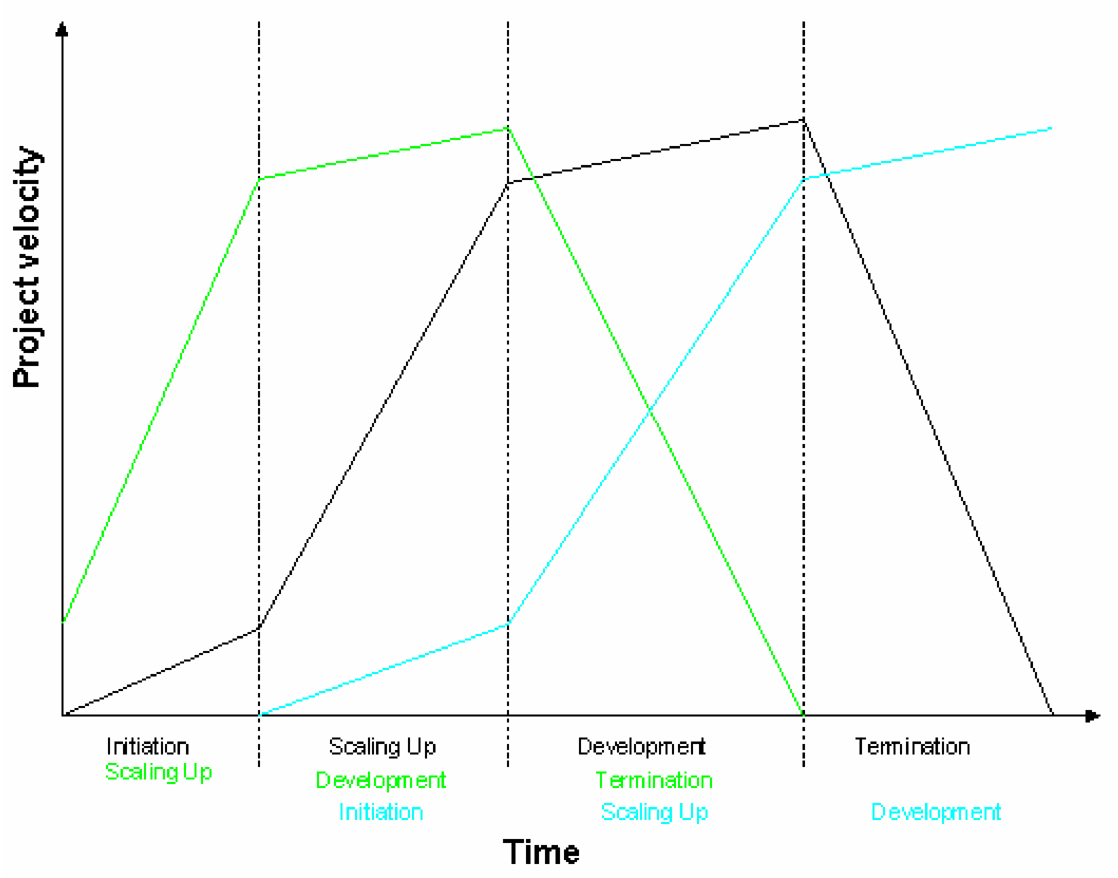


**Figure 18. Model of how multiple releases' lives fit together**

For FreeBSD, the initiation is determining that a new version is due, and the release engineering process to make a release. Such a process consists of a feature-freeze, code-freeze, the development of release candidates and the final release. This work is coordinated by the Release Engineering team. As the release is created, the user community gets gradually involved, and the resources available to it are scaling up. Being widely used without any reports of serious failures generates trust, the user base expands and the release is in production. The termination of the release is initiated by new releases being created. The only changes that will be applied to the release at this stage are the fixing of serious security issues should they occur.

As soon as the code freeze is over, recently developed changes are integrated into the branch they refer to. For the first release of major releases, much development will still happen when the release is scaling up. But when it has reached production level, the development version, FreeBSD-CURRENT, will scale up.

In addition to the continued development, such branching is good for the project in that some people are better at initiating ideas and implementing fresh technologies than maintaining code and refining it for public use. Such a branching makes sure they do not get bored with the project when the code they added comes into production. Also, it allows people who are better at seeing things through to take ownership of parts of the code and modifying it to satisfy the users both in terms of use and to changing requirements.

## 7.5. Maintenance

Just as we can measure a quality of the FreeBSD Project specifying goals, making questions that are relevant to the goal and metrics to measure the questions with, so can we do with the FreeBSD product. If we should set up a goal such as "FreeBSD should be free of problems", a requirement by most production sites, the perceived quality of a FreeBSD version will degrade as problems are identified and reported. We take the fixes and enhancements from the previous version for granted after a while and focus on the current problems. To avoid the product slowly "rotting", we need to maintain the code.

[Swanson, 1976] lists a number of maintenance types. Among these he lists adaptive and perfective maintenance. Adaptive is fitting already existing software into a new environment and perfective maintenance is for eliminating inefficiencies. But the border between maintenance and development of new features is not clear. For instance, the integration of LinuxPAM was an adaptive maintenance for LinuxPAM. But when LinuxPAM during this integration was deemed astandard and too full of bugs and was thrown out for the newly implemented OpenPAM, was this perfective maintenance or new development? The implementation of the Bluetooth stack, was that new development or adaptive maintenance of the network code?

We find that the distinction between maintenance and new development is not always clear. [Jørgensen, 2001] notes that FreeBSD is a maintenance centric project. Maintenance over time can also change the product into a new product, and it is perhaps this that unlocks the contradiction to [Canning, 1972]. Canning states that "a frequent reason for development programmers changing jobs is too much maintenance". If this was the case, then committers would part with project quickly. Instead, we have found in Section 4.2 that many more committers join the project than lose their commit privileges, and in Section 6.1 that there is a very high amount of changes committed every year.

[Swanson, 1976] notes that measuring maintenance performance is only meaningful in the context of the organisational structure on which it is based. Thus the maintenance data collected in this thesis with regards to FreeBSD's maintenance database, GNATS, is not directly comparable with other projects. Measuring it should therefore first and foremost be done for tracking the data over time while doing process improvement.

## 7.6. Summary

In this chapter we have found that

- Bikeshedding inhibits design and explains the earlier finding that small changes get the most feedback when understanding it to be that the best understood changes get the most feedback.
- The FreeBSD Project lacks guidelines for large undertakings that do not build on older modules or the adding of modules. An example of such changes is structural changes. The project also lacks ways of assigning people to problems.
- The up until now tacitly understood project model is reenforced through people referring to it.
- FreeBSD is released in versions where an overall velocity is kept in the project, but individual versions follow the model of initiating, scaling up, production and termination.
- Maintenance and its relation to new development is a complex issue. The FreeBSD project, a maintenance centric project, contradicts the hypothesis that much maintenance work is a motivation for leaving the project through constantly getting more committers and having a high volume of changes per year.

Having gone more in-depth into three aspects of the project model and going through five issues that are related to more of these aspects in this chapter, we have now a better understanding of how the project works. The following chapter will provide a project model that draws upon the findings we have done in the four previous chapters. It is meant to work as a stand-alone document, and will thus repeat the previous chapters somewhat.

# Chapter 8

# My Project Model

A project model is a means to reduce the communications overhead in a project. As shown by [Brooks, 1995], increasing the number of project participants increases the communication in the project exponentionally. FreeBSD has during the past few years increased both its mass of active users and committers, and the communication in the project has risen accordingly. This project model will serve to reduce this overhead by providing an up-to-date description of the project.

During the Core elections in 2002, Mark Murray stated "I am opposed to a long rule-book, as that satisfies lawyer-tendencies, and is counter to the technocentricity that the project so badly needs" [FreeBSD, 2002B]. This project model is not meant to be a tool to justify creating impositions for developers, but as a tool to facilitate coordination. It is meant as a description of the project, with an overview of how the different processes are executed. It is an introduction to how the FreeBSD project works.

The FreeBSD project model will be described as of April 1st, 2003. It is based on the Niels Jørgensen's paper [Jørgensen, 2001], FreeBSD's official documents, discussions on FreeBSD mailing lists and interviews with developers.

After providing definitions of terms used, this document will outline the organisational structure (including role descriptions and communication lines), discuss the methodology model and after presenting the tools used for process control, it will present the defined processes. Finally it will outline major sub-projects of the FreeBSD project.

[FreeBSD, 2002A, Section 1.2 and 1.3] give the vision and the architectural guidelines for the project. The vision is "To produce the best UNIX-like operating system package possible, with due respect to the original software tools ideology as well as usability, performance and stability." The architectural guidelines help determine whether a problem that someone wants to be solved is within the scope of the project

## 8.1. Definitions

### 8.1.1. Activity

An "activity" is an element of work performed during the course of a project [PMI, 2000]. It has an output and leads towards an outcome. Such an output can either be an input to another activity or a part of the process' delivery.

### 8.1.2. Process

A "process" is a series of activities that lead towards a particular outcome. A process can consist of one or more sub-processes. An example of a process is software design.

### 8.1.3. Hat

A "hat" is synonymous with role. A hat has certain responsibilities in a process and for the process outcome. The hat executes activities. It is well defined what issues the hat should be contacted about by the project members and people outside the project.

### 8.1.4. Outcome

An "outcome" is the final output of the process. This is synonymous with deliverable, which is defined as "any measurable, tangible, verifiable outcome, result or item that must be produced to complete a project or part of a project. Often used more narrowly in reference to an external deliverable, which is a deliverable that is subject to approval by the project sponsor or customer" by [PMI, 2000]. Examples of outcomes are a piece of software, a decision made or a report written.

### 8.1.5. FreeBSD

When saying "FreeBSD" we will mean the BSD derivative UNIX-like operating system FreeBSD, whereas when saying "the FreeBSD Project" we will mean the project organisation.

## 8.2. Organisational structure

While no-one takes ownership of FreeBSD, the FreeBSD organisation is divided into core, committers and contributors and is part of the FreeBSD community that lives around it.

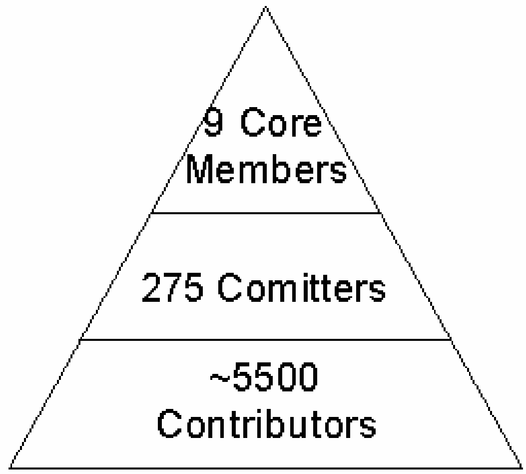


**Figure 19. The FreeBSD Project's structure**

Number of committers has been determined by going through CVS logs from December 1st, 2001 to December 1st, 2002 and contributors by going through the list of contributions and problem reports.

The main resource in the FreeBSD community is its developers: the committers and contributors. It is with their contributions that the project can move forward. Regular developers are referred to as contributors. As by January 1st, 2003, there are an estimated 5500 contributors on the project.

Committers are developers with the privilege of being able to commit changes. These are usually the most active developers who are willing to spend their time not only integrating their own code but integrating code submitted by the developers who do not have this privilege. They are also the developers who elect the core team, and they have access to closed discussions.

The project can be grouped into four distinct separate parts, and most developers will during their involvement in the FreeBSD project only be involved with one of these parts. The four parts are kernel development, userland development, ports and documentation. When referring to the base system, both kernel and userland is meant.

This split changes our triangle to look like this:

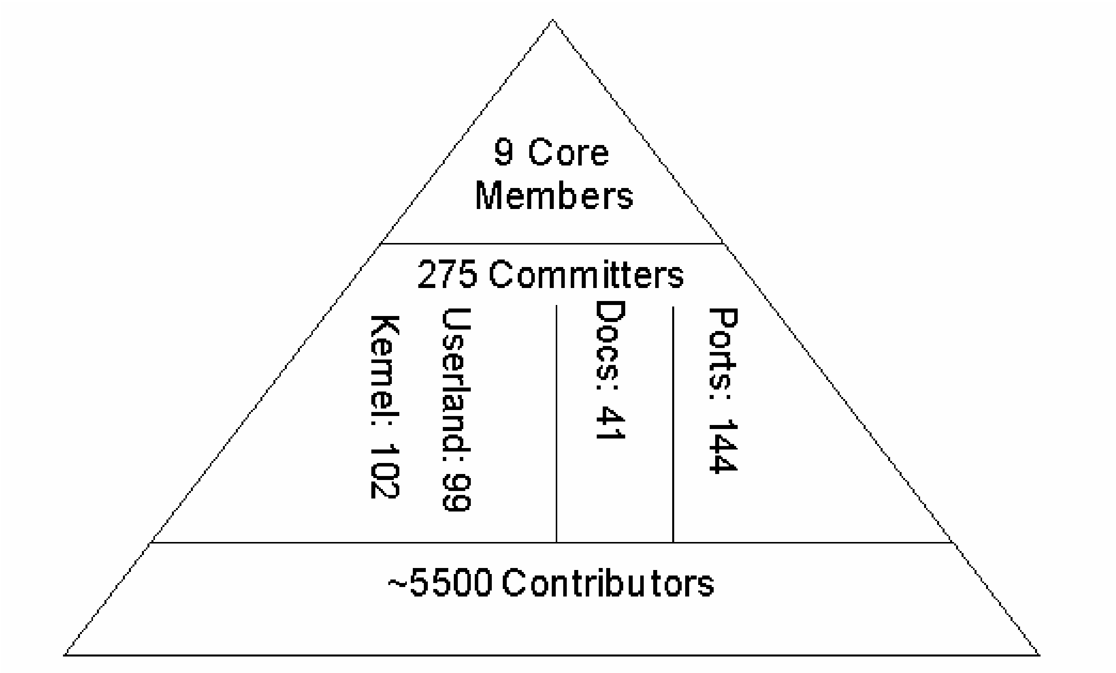


**Figure 20. The FreeBSD Project's structure with committers in categories**

Number of committers per area has been determined by going through CVS logs from December 1st, 2001 to December 1st, 2002. Note that many committers work in multiple areas, making the total number higher than the real number of committers. The total number of committers at that time was 275.

Committers fall into three groups: committers who are only concerned with one area of the project (for instance file systems), committers who are involved only with one sub-project and committers who commit to different parts of the code, including sub-projects. Because some committers work on different parts, the total number in the committers section of the triangle is higher than in the above triangle.

The kernel is the main building block of FreeBSD. While the userland applications are protected against faults in other userland applications, the entire system is vulnerable to errors in the kernel. This, combined with the vast amount of dependencies in the kernel and that it is not easy to see all the consequences of a kernel change, demands developers with a relative full understanding of the kernel. Multiple development efforts in the kernel also require a closer coordination than userland applications do.

The core utilities, known as userland, provide the interface that identifies FreeBSD: user interface, shared libraries and external interfaces to connecting clients. Currently, 99 people are involved in userland development and maintenance, many being maintainers for their own part of the code. Maintainership will be discussed in the Maintainership section.

Documentation is handled by The FreeBSD Documentation Project and includes all documents surrounding the FreeBSD project, including the web pages. There are currently 41 people involved in the FreeBSD Documentation Project.

Ports is the collection of meta-data that is needed to make software packages build correctly on FreeBSD. An example of a port is the port for the web-browser Mozilla. It contains information about where to fetch the source, what patches to apply and how, and how the package should be installed on the system. This allows automated tools to fetch, build and install the package. As of this writing, there are more than 7800 ports available[5], ranging from web servers to games, programming languages and most of the application types that are in use on modern computers. Ports will be discussed further in the section The Ports Subproject.

[5] Statistics are generated by counting the number of entries in the file ports/INDEX by January 1st, 2003.

## 8.3. Methodology model

### 8.3.1. Development model

There is no defined model for how people write code in FreeBSD. However, Niels Jorgensen has suggested a model of how written code is integrated into the project.

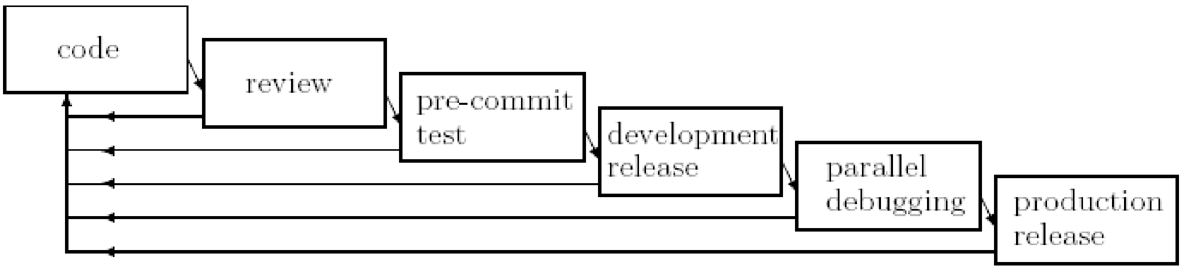


**Figure 21. Jørgensen's model for change integration**

The "development release" is the FreeBSD-CURRENT ("-CURRENT") branch and the "production release" is the FreeBSD-STABLE branch ("-STABLE") [Jørgensen, 2001].

This is a model for one change, and shows that after coding, developers seek community review and try integrating it with their own systems. After integrating the change into the development release, called FreeBSD-CURRENT, it is tested by many users and developers in the FreeBSD community. After it has gone through enough testing, it is merged into the production release, called FreeBSD-STABLE. Unless each stage is finished successfully, the developer needs to go back and make modifications in the code and restart the process. To integrate a change with either -CURRENT or -STABLE is called making a commit.

Jørgensen found that most FreeBSD developers work individually, meaning that this model is used in parallel by many developers on the different ongoing development efforts. A developer can also be working on multiple changes, so that while he is waiting for review or people to test one or more of his changes, he may be writing another change.

As each commit represents an increment, this is a massively incremental model. The commits are in fact so frequent that during one year[6], 132148 commits were made, making a daily average of 360 commits.

Within the "code" bracket in Jørgensen's figure, each programmer has his own working style and follows his own development models. The bracket could very well have been called "development" as it includes requirements gathering and analysis, system and detailed design, implementation and verification. However, the only output from these stages is the source code or system documentation.

From a stepwise model's perspective (such as the waterfall model), the other brackets can be seen as further verification and system integration. This system integration is also important to see if a change is accepted by the community. Up until the code is committed, the developer is free to choose how much to communicate about it to the rest of the project. In order for -CURRENT to work as a buffer (so that bright ideas that had some undiscovered drawbacks can be backed out) the minimum time a commit should be in -CURRENT before merging it to -STABLE is 3 days. Such a merge is referred to as an MFC (Merge From Current).

It is important to notice the word "change". Most commits do not contain radical new features, but are maintenance updates.

The only exceptions from this model are security fixes and changes to features that are deprecated in the -CURRENT branch. In these cases, changes can be committed directly to the -STABLE branch.

[6] The period from December 1st, 2001 to December 3rd, 2002 was examined to find this number.

In addition to many people working on the project, there are many related projects to the FreeBSD Project. These are either projects developing brand new features, sub-projects or projects whose outcome is incorporated into FreeBSD[7]. These projects fit into the FreeBSD Project just like regular development efforts: they produce code that is integrated with the FreeBSD Project. However, some of them (like Ports and Documentation) have the privilege of being applicable to both branches or commit directly to both -CURRENT and -STABLE.

There are no standards to how design should be done, nor is design collected in a centralised repository. The main design is that of 4.4BSD[8]. As design is a part of the "Code" bracket in Jørgensen's model, it is up to every developer or sub-project how this should be done. Even if the design should be stored in a central repository, the output from the design stages would be of limited use as the differences of methodologies would make them poorly if at all interoperable. For the overall design of the project, the project relies on the sub-projects to negotiate fit interfaces between each other rather than to dictate interfacing.

### 8.3.2. Release branches

The releases of FreeBSD are best illustrated by a tree with many branches where each major branch represents a major version. Minor versions are represented by branches of the major branches.

In the following release tree, arrows that follow one-another in a particular direction represent a branch. Boxes with full lines and diamonds represent official releases. Boxes with dotted lines represent the development branch at that time.

---

[7] For instance, the development of the Bluetooth stack started as a sub-project until it was deemed stable enough to be merged into the -CURRENT branch. Now it is a part of the core FreeBSD system.

[8] According to Kirk McKusick, after 20 years of developing UNIX operating systems, the interfaces are for the most part figured out. There is therefore no need for much design. However, new applications of the system and new hardware lead to some implementations being more beneficial than those that used to be preferred. One example is the introduction of web browsing that made the normal TCP/IP connection a short burst of data rather than a steady stream over a longer period of time.

Security branches are represented by ovals. Diamonds differ from boxes in that they represent a fork, meaning a place where a branch splits into two branches where one of the branches becomes a sub-branch. For example, at 4.0-RELEASE the 4.0-CURRENT branch split into 4-STABLE and 5.0-CURRENT. At 4.5-RELEASE, the branch forked off a security release called RELENG_4_5.

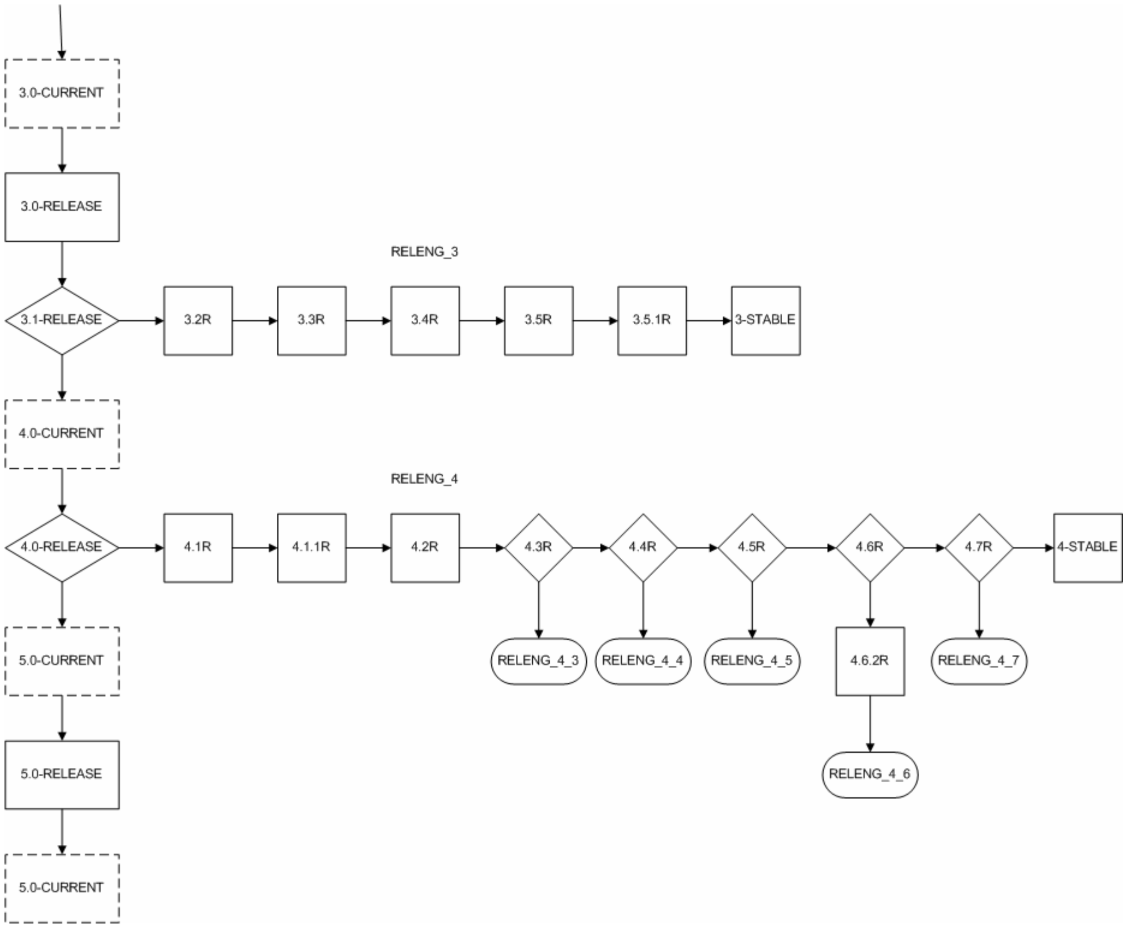


**Figure 22. The FreeBSD release tree**

The latest -CURRENT version is always referred to as -CURRENT, while the latest -STABLE release is always referred to as -STABLE. In this figure, -STABLE refers to 4-STABLE while -CURRENT refers to 5.0-CURRENT following 5.0-RELEASE. [FreeBSD, 2002E]

A "major release" is always made from the -CURRENT branch. However, the -CURRENT branch does not need to fork at that point in time, but can focus on stabilising. An example of this is that following 3.0-RELEASE, 3.1-RELEASE was also a continuation of the -CURRENT-branch, and -CURRENT did not become a true development branch until this version was released and the 3-STABLE branch was forked. When -CURRENT returns to becoming a development branch, it can only be followed by a major release. 5-STABLE is predicted to be forked off 5.0-CURRENT at around 5.1-RELEASE or 5.2-RELEASE. It is not until 5-STABLE is forked that the development branch will be branded 6.0-CURRENT.

A "minor release" is made from the -CURRENT branch following a major release, or from the -STABLE branch.

Following and including, 4.3-RELEASE[9], when a minor release has been made, it becomes a "security branch". This is meant for organisations that do not want to follow the -STABLE branch and the potential new/changed features it offers, but instead require an absolutely stable environment, only updating to implement security updates[10].

Each update to a security branch is called a "patchlevel". For every security enhancement that is done, the patchlevel number is increased, making it easy for people tracking the branch to see what security enhancements they have implemented. In cases where there have been especially serious security flaws, an entire new release can be made from a security branch. An example of this is 4.6.2-RELEASE.

---

[9] The first release this actually happened for was 4.5-RELEASE, but security branches were at the same time created for 4.3-RELEASE and 4.4-RELEASE.

[10] There is a terminology overlap with respect to the word "stable", which leads to some confusion. The -STABLE branch is still a development branch, whose goal is to be useful for most people. If it is never acceptable for a system to get changes that are not announced at the time it is deployed, that system should run a security branch.

### 8.3.3. Model summary

To summarise, the development model of FreeBSD can be seen as the following tree:

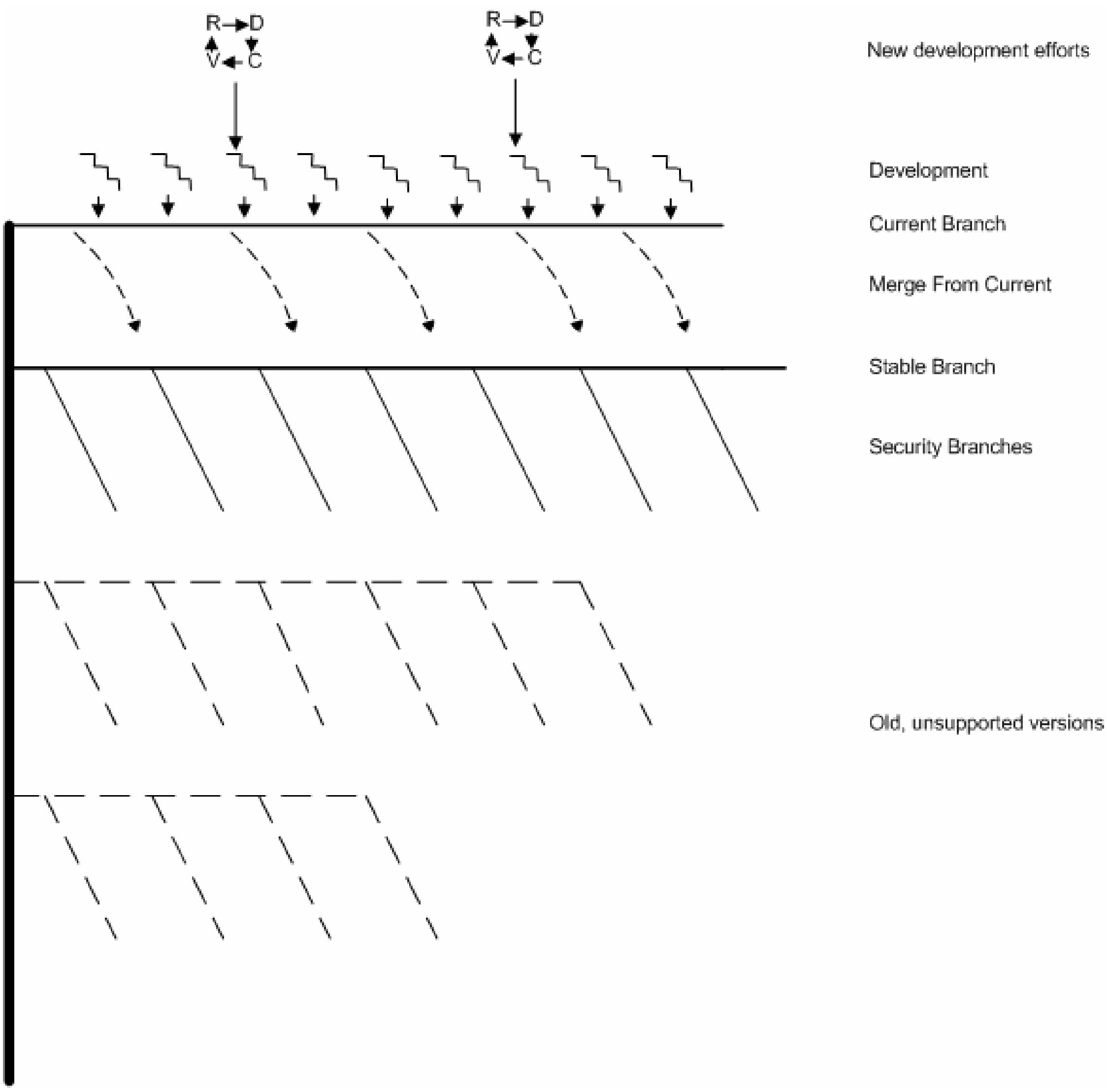


**Figure 23. The tree of the FreeBSD development with ongoing development efforts and continuous integration.**

The tree symbolises the release versions with major versions spawning new main branches and minor versions being versions of the main branch. The top branch is the -CURRENT branch where all new development is integrated, and the -STABLE branch is the branch directly below it.

Clouds of development efforts hang over the project where developers use the development models they see fit. The product of their work is then integrated into -CURRENT where it undergoes parallel debugging and is finally merged from -CURRENT into -STABLE. Security fixes are merged from -STABLE to the security branches.

## 8.4. Hats

Many committers have a special area of responsibility. These roles are called hats [Losh, 2002]. These hats can be either project roles, such as public relations officer, or maintainer for a certain area of the code. Because this is a project where people give voluntarily of their spare time, people with assigned hats are not always available. They must therefore appoint a deputy that can perform the hat's role in his or her absence. The other option is to have the role held by a group.

Many of these hats are not formalised. Formalised hats have a charter stating the exact purpose of the hat along with its privileges and responsibilities. The writing of such charters is a new part of the project, and has thus yet to be completed for all hats. These hat descriptions are not such a formalisation, rather a summary of the role with links to the charter where available and contact addresses,

### 8.4.1. General Hats

#### *8.4.1.1. Contributor*

A Contributor contributes to the FreeBSD project either as a developer, as an author, by sending problem reports, or in other ways contributing to the progress of the project. A contributor has no special privileges in the FreeBSD project. [FreeBSD, 2002F]

#### *8.4.1.2. Committer*

A committer is a developer who has the required privileges to add his code or documentation to the repository. A committer has made a commit within the past 12 months. [FreeBSD, 2000A] An active committer is a committer who has made an average of one commit per month during that time.

It is worth noting that there are no technical barriers to prevent someone, once having gained commit privileges, to make commits in parts of the source the committer did not specifically get permission to modify. However, when wanting to make modifications to parts a committer has not been involved in before, he/she should read the logs to see what has happened in this area before, and also read the MAINTAINER file to see if the maintainer of this part has any special requests on how changes in the code should be made Also, since The Ports Subproject is allowed to give commit privileges without approval from core, a committer who has gained his privileges through contributing to the ports sub-project should be careful and have his changes approved before committing anything outside the ports tree.

#### *8.4.1.3. Core Team*

The core team is elected by the committers from the pool of committers and serves as the board of directors of the FreeBSD project. It promotes active contributors to committers, assigns people to well-defined hats, and is the final arbiter of decisions involving which way the project should be heading. As by January 1st, 2003, core consisted of 9 members. Elections are held every two years.

#### *8.4.1.4. Maintainership*

Maintainership means that that person is responsible for what is allowed to go into that area of the code and has the final say should disagreements over the code occur. This involves proactive work aimed at stimulating contributions and reactive work in reviewing commits.

With the FreeBSD source comes the MAINTAINERS file that contains a one-line summary of how each maintainer would like contributions to be made. Having this notice and contact information enables developers to focus on the development effort rather than being stuck in a slow correspondence should the maintainer be unavailable for some time.

If the maintainer is unavailable for an unreasonably long period of time, and other people do a significant amount of work, maintainership may be switched without the maintainer's approval. This is based on the stance that maintainership should be demonstrated, not declared.

Maintainership of a particular piece of code is a hat that is not held as a group.

## 8.4.2. Official Hats

The official hats in the FreeBSD Project are hats that are more or less formalised and mainly administrative roles. They have the authority and responsibility for their area. The following illustration shows the responsibility lines. After this follows a description of each hat, including whom it is held by.

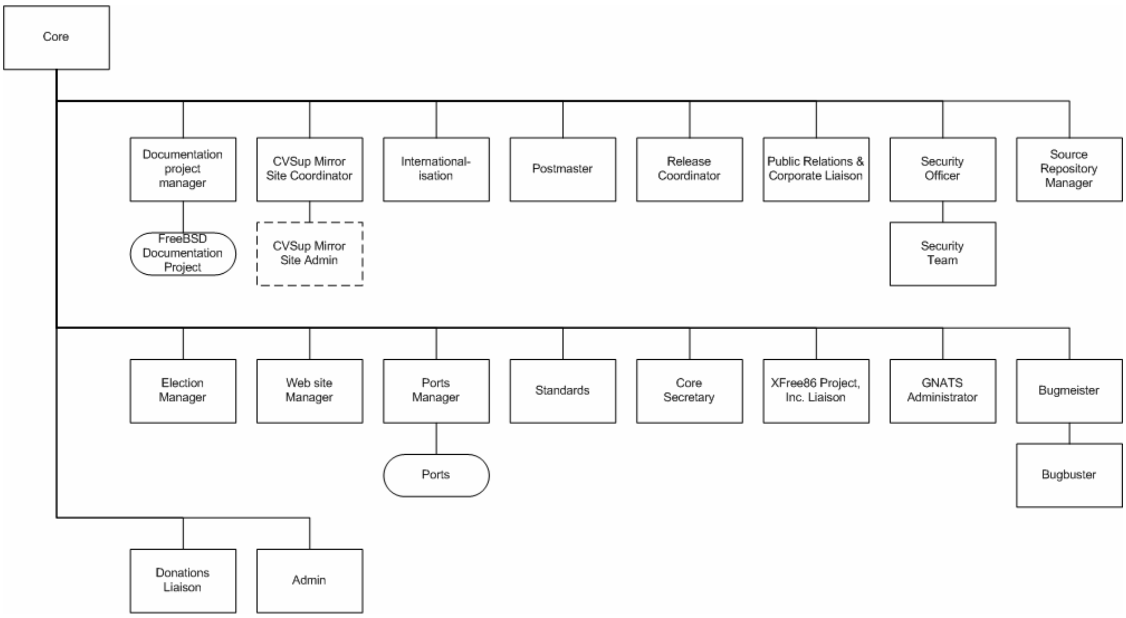


**Figure 24. Overview of official hats**

All boxes consist of groups of committers, except for the dotted boxes where the holders are not necessarily committers. The flattened circles are sub-projects and consist of both committers and non-committers of the main project.

#### 8.4.2.1. Documentation project manager

The FreeBSD Documentation Project architect is responsible for defining and following up documentation goals for the committers in the Documentation project.

Hat held by: The DocEng team <doceng@FreeBSD.org>. The DocEng Charter.

#### 8.4.2.2. CVSup Mirror Site Coordinator

The CVSup Mirror Site Coordinator coordinates all the CVSup Mirror Site Admins to ensure that they are distributing current versions of the software, that they have the capacity to update themselves when major updates are in progress, and making it easy for the general public to find their closest CVSup mirror.

Hat currently held by: John Polstra <jdp@FreeBSD.org>.

#### 8.4.2.3. Internationalisation

The Internationalisation hat is responsible for coordinating the localisation efforts of the FreeBSD kernel and userland utilities. The translation efforts are coordinated by The FreeBSD Documentation Project. The Internationalisation hat should suggest and promote standards and guidelines for writing and maintaining the software in a fashion that makes it easier to translate.

Hat currently available.

#### 8.4.2.4. Postmaster

The Postmaster is responsible for mail being correctly delivered to the committers' email address. He is also responsible for ensuring that the mailing lists work and should take measures against possible disruptions of mail such as having troll-, spam- and virus-filters.

Hat currently held by: Jonathan M. Bresler <jmb@FreeBSD.org>.

#### 8.4.2.5. Release Coordination

The responsibilities of the Release Engineering Team are

- Setting, publishing and following a release schedule for official releases
- Documenting and formalising release engineering procedures
- Creation and maintenance of code branches
- Coordinating with the Ports and Documentation teams to have an updated set of packages and documentation released with the new releases
- Coordinating with the Security team so that pending releases are not affected by recently disclosed vulnerabilities.

Further information about the development process is available in the release engineering section.

Hat held by: the Release Engineering team <re@FreeBSD.org>, currently headed by Murray Stokely <murray@FreeBSD.org>. The Release Engineering Charter.

#### 8.4.2.6. Public Relations & Corporate Liaison

The Public Relations & Corporate Liaison's responsibilities are:

- Making press statements when happenings that are important to the FreeBSD Project happen.
- Being the official contact person for corporations that are working close with the FreeBSD Project.
- Take steps to promote FreeBSD within both the Open Source community and the corporate world.
- Handle the "freebsd-advocacy" mailing list.

This hat is currently not occupied.

#### 8.4.2.7. Security Officer

The Security Officer's main responsibility is to coordinate information exchange with others in the security community and in the FreeBSD project. The Security Officer is also responsible for taking action when security problems are reported and promoting proactive development behaviour when it comes to security.

Because of the fear that information about vulnerabilities may leak out to people with malicious intent before a patch is available, only the Security Officer, consisting of an officer, a deputy and two Core team members, receive sensitive information about security issues. However, to create or implement a patch, the Security Officer has the Security Officer Team <security-team@FreeBSD.org> to help do the work.

Hat held by: the Security Officer <security-officer@FreeBSD.org>, currently headed by Jacques Vidrine <nectar@FreeBSD.org>. The Security Officer and The Security Officer Team's charter.

#### 8.4.2.8. Source Repository Manager

The Source Repository Manager is the only one who is allowed to directly modify the repository without using the CVS tool. It is his/her responsibility to ensure that technical problems that arise in the repository are resolved quickly. The source repository manager has the authority to back out commits if this is necessary to resolve a CVS technical problem.

Hat held by: the Source Repository Manager <cvs@FreeBSD.org>, currently headed by Peter Wemm <peter@FreeBSD.org>.

#### 8.4.2.9. Election Manager

The Election Manager is responsible for the Core election process. The manager is responsible for running and maintaining the election system, and is the final authority should minor unforseen events happen in the election process. Major unforseen events have to be discussed with the Core team

Hat held only during elections.

#### 8.4.2.10. Web site Management

The Web site Management hat is responsible for coordinating the rollout of updated web pages on mirrors around the world, for the overall structure of the primary web site and the system it is running upon. The management needs to coordinate the content with The FreeBSD Documentation Project and acts as maintainer for the "www" tree.

Hat held by: the FreeBSD Webmasters <www@FreeBSD.org>.

#### 8.4.2.11. Ports Manager

The Ports Manager acts as a liaison between The Ports Subproject and the core project, and all requests from the project should go to the ports manager.

Hat held by: the Ports Management Team <portmgr@FreeBSD.org>, currently headed by Satoshi Asami <asami@FreeBSD.org>.

#### *8.4.2.12. Standards*

The Standards hat is responsible for ensuring that FreeBSD complies with the standards it is committed to , keeping up to date on the development of these standards and notifying FreeBSD developers of important changes that allows them to take a proactive role and decrease the time between a standards update and FreeBSD's compliancy.

Hat currently held by: Garrett Wollman <wollman@FreeBSD.org>.

#### *8.4.2.13. Core Secretary*

The Core Secretary's main responsibility is to write drafts to and publish the final Core Reports. The secretary also keeps the core agenda, thus ensuring that no balls are dropped unresolved.

Hat currently held by: Wilko Bulte <wilko@FreeBSD.org>.

#### *8.4.2.14. XFree86 Project, Inc. Liaison*

The XFree86 Project liaison relays information from the XFree86 Project to the right people in the FreeBSD Project and visa versa. This enables the projects to be aligned without everyone in both projects staying up-to-date on the other project.

Hat currently held by: Rich Murphey <rich@FreeBSD.org>.

#### *8.4.2.15. GNATS Administrator*

The GNATS Administrator is responsible for ensuring that the maintenance database is in working order, that the entries are correctly categorised and that there are no invalid entries.

Hat currently held by: Steve Price <steve@FreeBSD.org>.

#### *8.4.2.16. Bugmeister*

The Bugmeister is person in charge of the problem report group.

Hat currently held by: Giorgos Keramidas <keramida@FreeBSD.org>.

#### *8.4.2.17. Donations Liaison Officer*

The task of the donations liaison officer is to match the developers with needs with people or organisations willing to make a donation. The Donations Liaison Charter is available here

Hat held by: the Donations Liaison Office <donations@FreeBSD.org>, currently headed by Michael W. Lucas <mwlucas@FreeBSD.org>.

#### *8.4.2.18. Admin*

(Also called "FreeBSD Cluster Admin")

The admin team consists are the people responsible for administrating the computers that the project relies on for its distributed work and communication to be synchronised. It consists mainly of those people who have physical access to the servers.

Hat held by: the Admin team <admin@FreeBSD.org>, currently headed by Mark Murray <markm@FreeBSD.org>

### 8.4.3. Process dependent hats

#### *8.4.3.1. Report originator*

The person originally responsible for filing a Problem Report.

#### *8.4.3.2. Bugbuster*

A person who will either find the right person to solve the problem, or close the problem report if it is a duplicate or otherwise not an interesting one.

#### *8.4.3.3. Mentor*

A mentor is a committer who takes it upon him/her to initialise a new committer to the project, both in terms of ensuring the new committers setup is valid, that the new committer knows the available tools required in his/her work and that the new committer knows what is expected of him/her in terms of behaviour.

#### *8.4.3.4. Vendor*

The person(s) or organisation whom external code comes from and who patches are sent to.

#### *8.4.3.5. Reviewers*

People on the mailing list where the request for review is posted.

#### *8.4.3.6. CVSup Mirror Site Admin*

A CVSup Mirror Site Admin has accesses to a server that he/she uses to mirror the CVS repository. The admin works with the CVSup Mirror Site Coordinator to ensure the site remains up-to-date and is following the general policy of official mirror sites.

## 8.5. Processes

The following section will describe the defined project processes. Issues that are not handled by these processes happen on an ad-hoc basis based on what has been customary to do in similar cases.

### 8.5.1. Adding new and removing old committers

The Core team has the responsibility of giving and removing commit privileges to contributors. This can only be done through a vote on the core mailing list. The ports and documentation sub-projects can give commit privileges to people working on these projects, but have to date not removed such privileges.

Normally a contributor is recommended to core by a committer. For contributors or outsiders to contact core asking to be a committer is not well thought of and is usually rejected.

If the area of particular interest for the developer potentially overlaps with other committers' area of maintainership, the opinion of those maintainers is sought. However, it is frequently this committer that recommends the developer.

When a contributor is given committer status, he is assigned a mentor. The committer who recommended the new committer will, in the general case, take it upon himself to be the new committers mentor.

When a contributor is given his commit bit, a PGP-signed email is sent from either Core Secretary, Ports Manager or nik@freebsd.org to both admins@freebsd.org, the assigned mentor, the new committer and core confirming the approval of a new account. The mentor then gathers a password line, SSH 2 public key and PGP key from the new committer and sends them to Admin. When the new account is created, the mentor activates the commit bit and guides the new committer through the rest of the initial process.

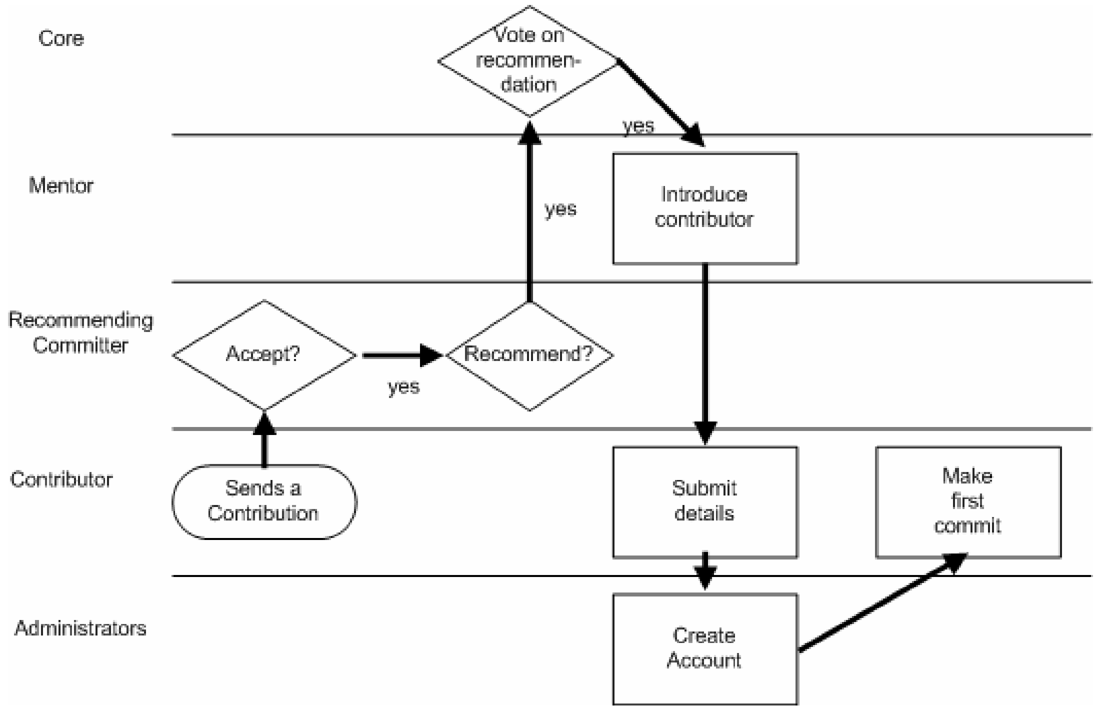


**Figure 25. Process summary: adding a new committer**

When a contributor sends a piece of code, the receiving committer may choose to recommend that the contributor is given commit privileges. If he recommends this to core, they will vote on this recommendation. If they vote in favour, a mentor is assigned the new committer and the new committer has to email his details to the administrators for an account to be created. After this, the new committer is all set to make his first commit. By tradition, this is by adding his name to the committers list.

Recall that a committer is considered to be someone who has committed code during the past 12 months. However, it is not until after 18 months of inactivity have passed that commit privileges are eligible to be revoked. [FreeBSD, 2002H] There are, however, no automatic procedures for doing this. For reactions concerning commit privileges not triggered by time, see section 8.5.8.

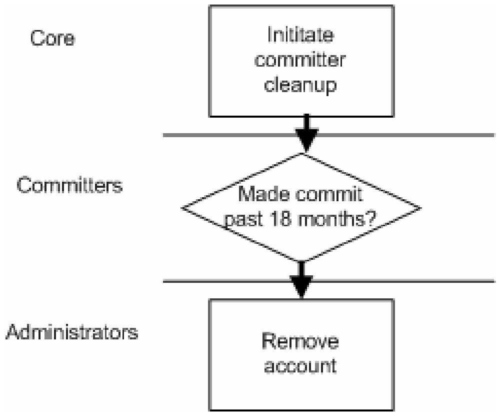


**Figure 26. Process summary: removing a committer**

When Core decides to clean up the committers list, they check who has not made a commit for the past 18 months. Committers who have not done so have their commit bits revoked.

It is also possible for committers to request that their commit bit be retired if for some reason they are no longer going to be actively committing to the project. In this case, it can also be restored at a later time by core, should the committer ask.

Roles in this process:

- Core team
- Contributor
- Committer
- Maintainership
- Mentor

[FreeBSD, 2000A] [FreeBSD, 2002H] [FreeBSD, 2002I]

## 8.5.2. Adding/Removing an official CVSup Mirror

A CVSup mirror is a replica of the official CVSup master that contains all the up-to-date source code for all the branches in the FreeBSD project, ports and documentation.

Adding an official CVSup mirror starts with the potential CVSup Mirror Site Admin installing the "cvsup-mirror" package. Having done this and updated the source code with a mirror site, he now runs a fairly recent unofficial CVSup mirror.

Deciding he has a stable environment, the processing power, the network capacity and the storage capacity to run an official mirror, he mails the CVSup Mirror Site Coordinator who decides whether the mirror should become an official mirror or not.

In making this decision, the CVSup Mirror Site Coordinator has to determine whether that geographical area needs another mirror site, if the mirror administrator has the skills to run it reliably, if the network bandwidth is adequate and if the master server has the capacity to server another mirror.

If CVSup Mirror Site Coordinator decides that the mirror should become an official mirror, he obtains an authentication key from the mirror admin that he installs so the mirror admin can update the mirror from the master server.

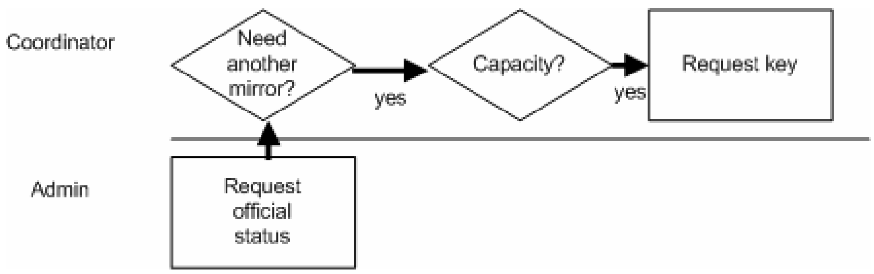


**Figure 27. Process summary: adding a CVSup mirror**

When a CVSup mirror administrator of an unofficial mirror offers to become an official mirror site, the CVSup coordinator decides if another mirror is needed and if there is sufficient capacity to accommodate it. If so, an authorisation key is requested and the mirror is given access to the main distribution site and added to the list of official mirrors.

Roles involved in this process:

- CVSup
- SSH 2

Tools used in this process:

- CVSup Mirror Site Coordinator
- CVSup Mirror Site Admin

### 8.5.3. Committing code

The committing of new or modified code is one of the most frequent processes in the FreeBSD project and will usually happen many times a day. Committing of code can only be done by a "committer". Committers commit either code they have written, code submitted to them or code submitted through a problem report.

When code is written by the developer that is non-trivial, he should seek a code review from the community. This is done by sending mail to the relevant list asking for review. Before submitting the code for review, he should ensure it compiles correctly with the entire tree and that all relevant tests run. This is called "pre-commit test". When contributed code is received, it should be reviewed by the committer and tested the same way.

When a change is committed to a part of the source that has been contributed from an outside Vendor, the maintainer should ensure that the patch is contributed back to the vendor. This is in line with the open source philosophy and makes it easier to stay in sync with outside projects as the patches do not have to be reapplied every time a new release is made.

After the code has been available for review and no further changes are necessary, the code is committed into the development branch, -CURRENT. If the change applies for the -STABLE branch or the other branches as well, a "Merge From Current" ("MFC") countdown is set by the committer. After the number of days the committer chose when setting the MFC have passed, an email will automatically be sent to the committer reminding him to commit it to the -STABLE branch (and possibly security branches as well). Only security critical changes should be merged to security branches.

Delaying the commit to -STABLE and other branches allows for "parallel debugging" where the committed code is tested on a wide range of configurations. This makes changes to -STABLE to contain fewer faults and thus giving the branch its name.

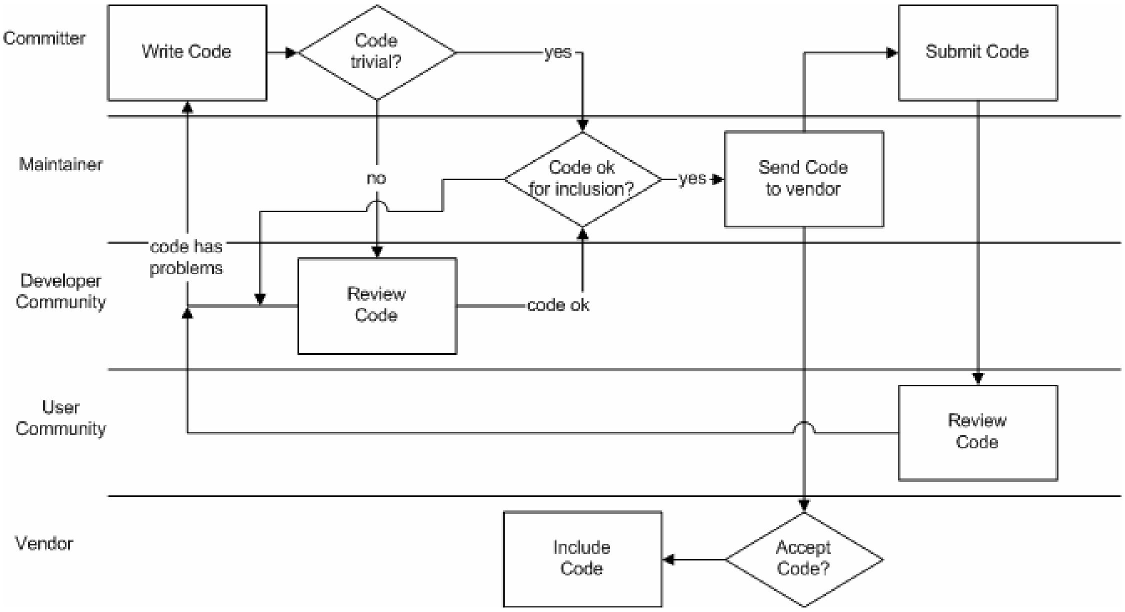


**Figure 28. Process summary: A committer commits code**

When a committer has written a piece of code and wants to commit it, he first needs to determine if it is trivial enough to go in without prior review or if it should first be reviewed by the developer community. If the code is trivial or has been reviewed and the committer is not the maintainer, he should consult the maintainer before proceeding. If the code is contributed by an outside vendor, the maintainer should create a patch that is sent back to the vendor. The code is then committed and the deployed by the users. Should they find problems with the code, this will be reported and the committer can go back to writing a patch. If a vendor is affected, he can choose to implement or ignore the patch.

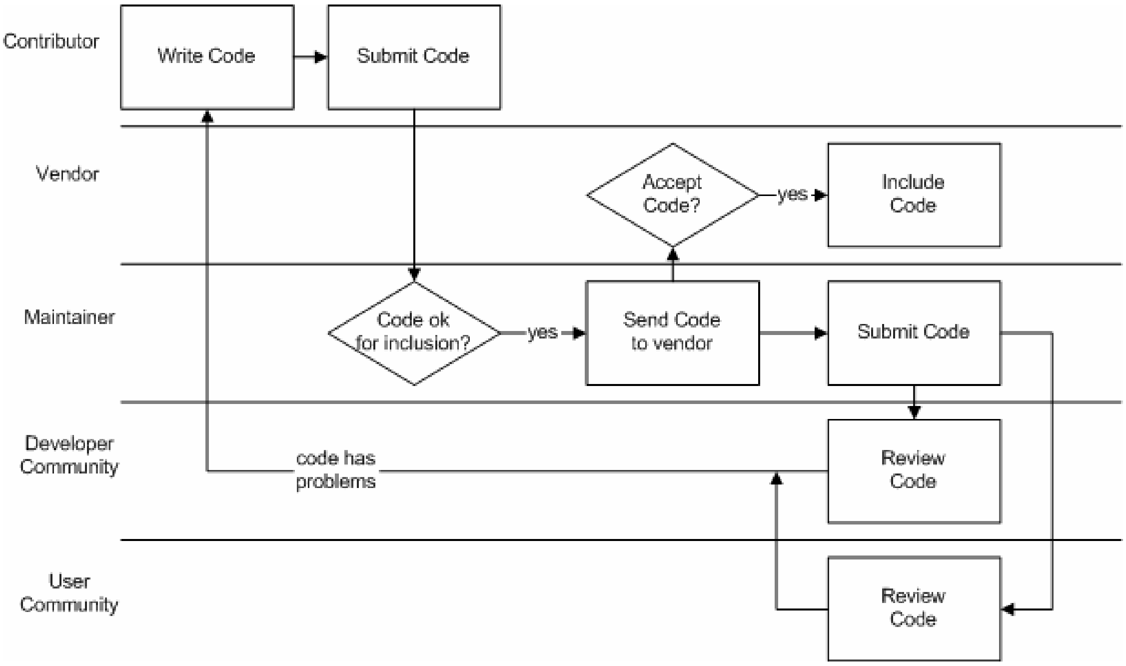


**Figure 29. Process summary: A contributor commits code**

The difference when a contributor makes a code contribution is that he submits the code through the send-pr program. This report is picked up by the maintainer who reviews the code and commits it.

Hats included in this process are:

- Committer
- Contributor
- Vendor
- Reviewer

[FreeBSD, 2001] [Jørgensen, 2001]

### 8.5.4. Core election

Core elections are held at least every two years[11]. Nine core members are elected. New elections are held if the number of core members drops below seven. New elections can also be held should at least 1/3 of the active committers demand this.

When an election is to take place, core announces this at least 6 weeks in advance, and appoints an election manager to run the elections.

Only committers can be elected into core. The candidates need to submit their candidacy at least one week before the election starts, but can refine their statements until the voting starts. They are presented in the candidates list. When writing their election statements, the candidates must answer a few standard questions submitted by the election manager.

During elections, the rule that a committer must have committed during the 12 past months is followed strictly. Only these committers are eligible to vote.

When voting, the committer may vote once in support of up to nine nominees. The voting is done over a period of four weeks with reminders being posted on "developers" mailing list that is available to all committers.

The election results are released one week after the election ends, and the new core team takes office one week after the results have been posted.

Should there be a voting tie, this will be resolved by the new, unambiguously elected core members.

Votes and candidate statements are archived, but the archives are not publicly available.

[11] The first Core election was held September 2000

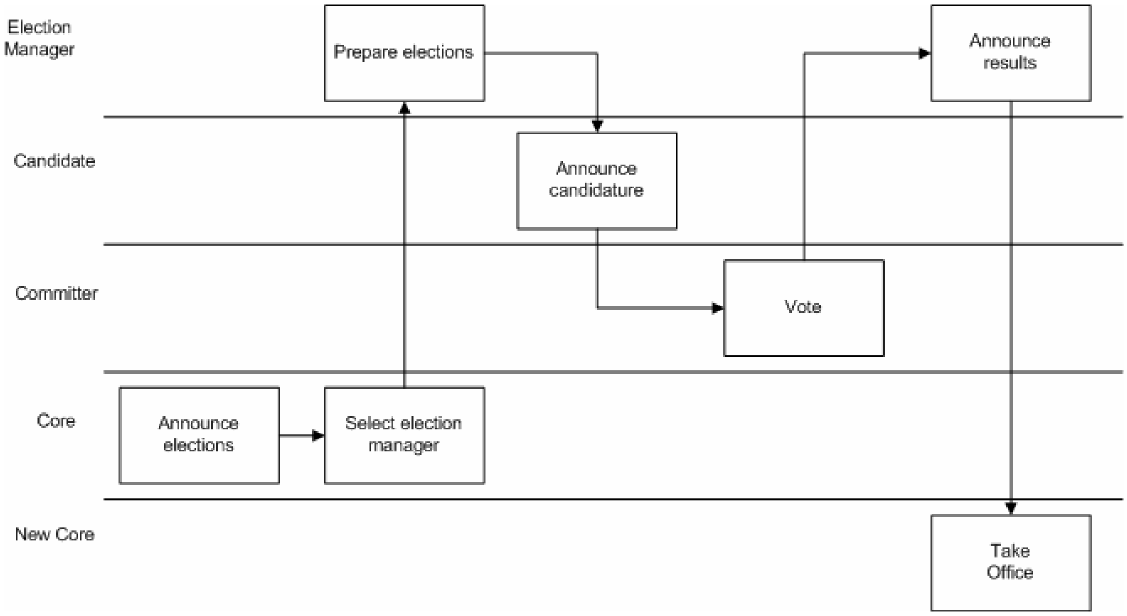


**Figure 30. Process summary: Core elections**

Core announces the election and selects an election manager. He prepares the elections, and when ready, candidates can announce their candidacies through submitting their statements. The committers then vote. After the vote is over, the election results are announced and the new core team takes office.

Hats in core elections are:

- Core team
- Committer
- Election Manager

[FreeBSD, 2000A] [FreeBSD, 2002B] [FreeBSD, 2002G]

### 8.5.5. Development of new features

Within the project there are sub-projects that are working on new features. These projects are generally done by one person [Jørgensen, 2001] . Every project is free to organise development as it sees fit. However, when the project is merged to the -CURRENT branch it must follow the project guidelines. When the code has been well tested in the -CURRENT branch and deemed stable enough and relevant to the -STABLE branch, it is merged to the -STABLE branch.

The requirements of the project are given by developer wishes, requests from the community in terms of direct requests by mail, Problem Reports, commercial funding for the development of features, or contributions by the scientific community. The wishes that come within the responsibility of a developer are given to that developer who prioritises his time between the request and his wishes. A common way to do this is maintain a TODO-list maintained by the project. Items that do not come within someone's responsibility are collected on TODO-lists unless someone volunteers to take the responsibility. All requests, their distribution and follow-up are handled by the GNATS tool.

Requirements analysis happens in two ways. The requests that come in are discussed on mailing lists, both within the main project and in the sub-project that the request belongs to or is spawned by the request. Furthermore, individual developers on the sub-project will evaluate the feasibility of the requests and determine the prioritisation between them. Other than archives of the discussions that have taken place, no outcome is created by this phase that is merged into the main project.

As the requests are prioritised by the individual developers on the basis of doing what they find interesting, necessary or are funded to do, there is no overall strategy or priority of what requests to regard as requirements and following up their correct implementation. However, most developers have some shared vision of what issues are more important, and they can ask for guidelines from the release engineering team and technical review board.

The verification phase of the project is two-fold. Before committing code to the current-branch, developers request their code to be reviewed by their peers. This review is for the most part done by functional testing, but also code review is important. When the code is committed to the branch, a broader functional testing will happen, that may trigger further code review and debugging should the code not behave as expected. This second verification form may be regarded as structural verification. Although the sub-projects themselves may write formal tests such as unit tests, these are usually not collected by the main project and are usually removed before the code is committed to the current branch[12].

[12] More and more tests are however performed when building the system ("make world"). These tests are however a very new addition and no systematic framework for these tests have yet been created.

### 8.5.6. Maintenance

It is an advantage to the project to for each area of the source have at least one person that knows this area well. Some parts of the code have designated maintainers. Others have de-facto maintainers, and some parts of the system do not have maintainers. The maintainer is usually a person from the sub-project that wrote and integrated the code, or someone who has ported it from the platform it was written for[13]. The maintainer's job is to make sure the code is in sync with the project the code comes from if it is contributed code, and apply patches submitted by the community or write fixes to issues that are discovered.

The main bulk of work that is put into the FreeBSD project is maintenance. [Jørgensen, 2001] has made a figure showing the life cycle of changes.

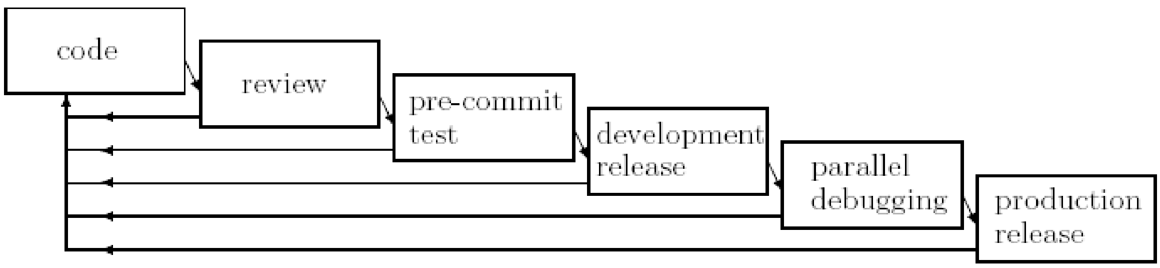


**Figure 31. Jørgensen's model for change integration**

Here "development release" refers to the -CURRENT branch while "production release" refers to the -STABLE branch. The "pre-commit test" is the functional testing by peer developers when asked to do so or trying out the code to determine the status of the sub-project. "Parallel debugging" is the functional testing that can trigger more review, and debugging when the code is included in the -CURRENT branch.

[13] sendmail and named are examples of code that has been merged from other platforms.

As of this writing, there were 275 committers in the project. When they commit a change to a branch, that constitutes a new release. It is very common for users in the community to track a particular branch. The immediate existence of a new release makes the changes widely available right away and allows for rapid feedback from the community. This also gives the community the response time they expect on issues that are of importance to them. This makes the community more engaged, and thus allows for more and better feedback that again spurs more maintenance and ultimately should create a better product.

Before making changes to code in parts of the tree that has a history unknown to the committer, the committer is required to read the commit logs to see why certain features are implemented the way they are in order not to make mistakes that have previously either been thought through or resolved.

## 8.5.7. Problem reporting

FreeBSD comes with a problem reporting tool called "send-pr" that is a part of the GNATS package. All users and developers are encouraged to use this tool for reporting problems in software they do not maintain. Problems include bug reports, feature requests, features that should be enhanced and notices of new versions of external software that is included in the project.

Problem reports are sent to an email address where it is inserted into the GNATS maintenance database. A Bugbuster classifies the problem and sends it to the correct group or maintainer within the project. After someone has taken responsibility for the report, the report is being analysed. This analysis includes verifying the problem and thinking out a solution for the problem. Often feedback is required from the report originator or even from the FreeBSD community. Once a patch for the problem is made, the originator may be asked to try it out. Finally, the working patch is integrated into the project, and documented if applicable. It there goes through the regular maintenance cycle as described in section maintenance. These are the states a problem report can be in: open, analyzed, feedback, patched, suspended and closed. The suspended state is for when further progress is not possible due to the lack of information or for when the task would require so much work that nobody is working on it at the moment.

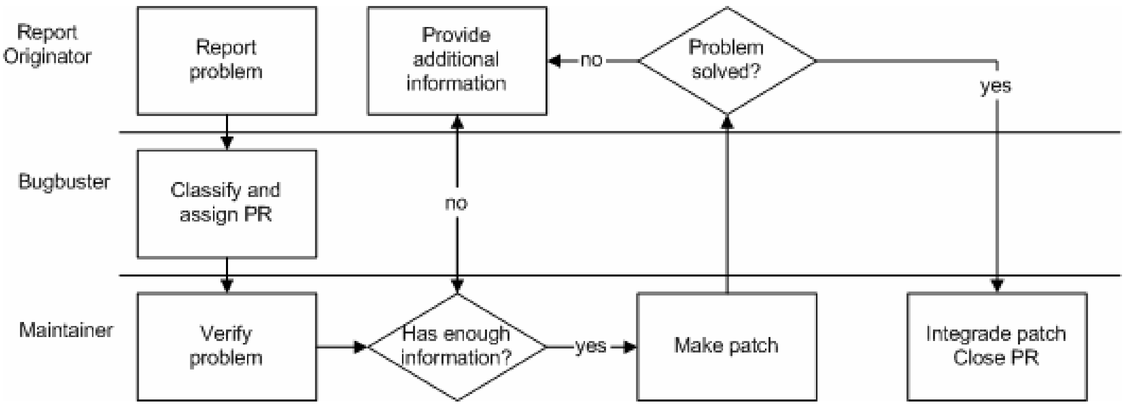


**Figure 32. Process summary: problem reporting**

A problem is reported by the report originator. It is then classified by a bugbuster and handed to the correct maintainer. He verifies the problem and discusses the problem with the originator until he has enough information to create a working patch. This patch is then committed and the problem report is closed.

The roles included in this process are:

- Report originator
- Maintainership
- Bugbuster

[FreeBSD, 2002C]. [FreeBSD, 2002D]

### 8.5.8. Reacting to misbehaviour

[FreeBSD, 2001] has a number of rules that committers should follow. However, it happens that these rules are broken. The following rules exist in order to be able to react to misbehaviour. They specify what actions will result in how long a suspension of the committer's commit privileges.

- Committing during code freezes without the approval of the Release Engineering team - 2 days
- Committing to a security branch without approval - 2 days
- Commit wars - 5 days to all participating parties
- Impolite or inappropriate behaviour - 5 days

[Lehey, 2002]

For the suspensions to be efficient, any single core member can implement a suspension before discussing it on the "core" mailing list. Repeat offenders can, with a 2/3 vote by core, receive harsher penalties, including permanent removal of commit privileges. (However, the latter is always viewed as a last resort, due to its inherent tendency to create controversy). All suspensions are posted to the "developers" mailing list, a list available to committers only.

It is important that you cannot be suspended for making technical errors. All penalties come from breaking social etiquette.

Hats involved in this process:

- Core team
- Committer

### 8.5.9. Release engineering

The FreeBSD project has a Release Engineering team with a principal release engineer that is responsible for creating releases of FreeBSD that can be brought out to the user community via the net or sold in retail outlets. Since FreeBSD is available on multiple platforms and releases for the different architectures are made available at the same time, the team has one person in charge of each architecture. Also, there are roles in the team responsible for coordinating quality assurance efforts, building a package set and for having an updated set of documents. When referring to the release engineer, a representative for the release engineering team is meant.

When a release is coming, the FreeBSD project changes shape somewhat. A release schedule is made containing feature- and code-freezes, release of interim releases and the final release. A feature-freeze means no new features are allowed to be committed to the branch without the release engineers' explicit consent. Code-freeze means no changes to the code (like bugs-fixes) are allowed to be committed without the release engineer’s explicit consent. This feature- and code-freeze is known as stabilising. During the release process, the release engineer has the full authority to revert to older versions of code and thus "back out" changes should he find that the changes are not suitable to be included in the release.

There are three different kinds of releases:

- .0 releases are the first release of a major version. These are branched of the -CURRENT branch and have a significantly longer release engineering cycle due to the unstable nature of the -CURRENT branch
- .X releases are releases of the -STABLE branch. They are scheduled to come out every 4 months.
- .X.Y releases are security releases that follow the .X branch. These come out only when sufficient security fixes have been merged since the last

release on that branch. New features are rarely included, and the security team is far more involved in these than in regular releases.

For releases of the -STABLE-branch, the release process starts 45 days before the anticipated release date. During the first phase, the first 15 days, the developers merge what changes they have had in -CURRENT that they want to have in the release to the release branch. When this period is over, the code enters a 15 day code freeze in which only bug fixes, documentation updates, security-related fixes and minor device driver changes are allowed. These changes must be approved by the release engineer in advance. At the beginning of the last 15 day period a release candidate is created for widespread testing. Updates are less likely to be allowed during this period, except for important bug fixes and security updates. In this final period, all releases are considered release candidates. At the end of the release process, a release is created with the new version number, including binary distributions on web sites and the creation of CD-ROM images. However, the release isn't considered "really released" until a PGP-signed message stating exactly that, is sent to the mailing list freebsd-announce; anything labelled as a "release" before that may well be in-process and subject to change before the PGP-signed message is sent[14].

The releases of the -CURRENT-branch (that is, all releases that end with ".0") are very similar, but with twice as long timeframe. It starts 8 weeks prior to the release with announcement of the release time line. Two weeks into the release process, the feature freeze is initiated and performance tweaks should be kept to a minimum. Four weeks prior to the release, an official beta version is made available. Two weeks prior to release, the code is officially branched into a new version. This version is given release candidate status, and as with the release engineering of -STABLE, the code freeze of the release candidate is hardened. However, development on the main development branch can continue. Other than these differences, the release engineering processes are alike.

[14] Many commercial vendors use these images to create CD-ROMs that are sold in retail outlets.

.0 releases go into their own branch and are aimed mainly at early adopters. It is not until .1 versions are released that the branch becomes -STABLE and -CURRENT targets the next major version.

Most releases are made when a given date that has been deemed a long enough time since the previous release comes. A target is set for having major releases every 18 months and minor releases every 4 months. The user community has made it very clear that security and stability cannot be sacrificed by self-imposed deadlines and target release dates. For slips of time not to become to long with regards to security and stability issues, extra discipline is required when committing changes to -STABLE.

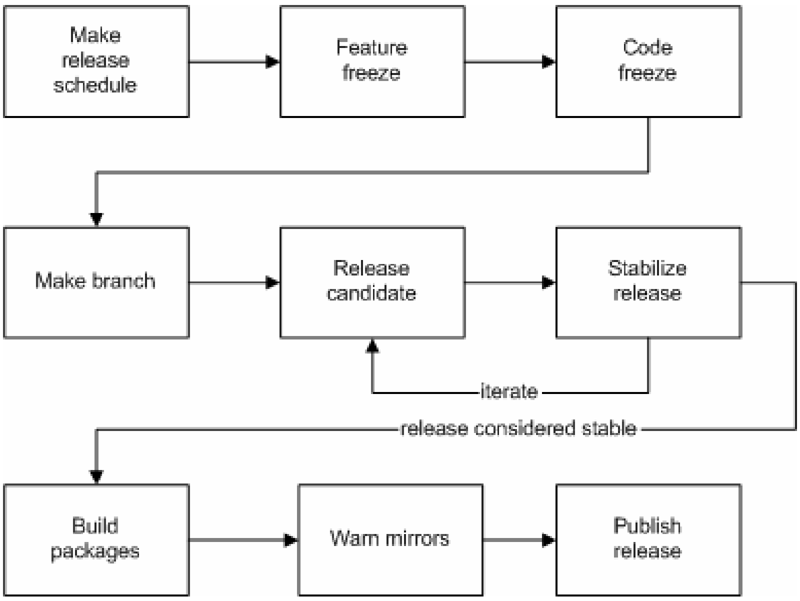


**Figure 33. Process summary: release engineering**

These are the stages in the release engineering process. Multiple release candidates may be created until the release is deemed stable enough to be released.

[FreeBSD, 2002E]

## 8.6. Tools

The major support tools for supporting the development process are CVS, CVSup, Perforce, GNATS, Mailman and OpenSSH. Except for CVSup, these are externally developed tools. These tools are commonly used in the open source world.

### 8.6.1. Concurrent Versions System (CVS)

Concurrent Versions System or simply "CVS" is a system to handle multiple versions of text files and tracking who committed what changes and why. A project lives within a "repository" and different versions are considered different "branches".

### 8.6.2. CVSup

CVSup is a software package for distributing and updating collections of files across a network. It is consists of a client program, cvsup, and a server program, cvsupd. The package is tailored specifically for distributing CVS repositories, and by taking advantage of CVS' properties, it performs updates much faster than traditional systems.

### 8.6.3. GNATS

GNATS is a maintenance database consisting of a set of tools to track bugs at a central site. It supports the bug tracking process for sending and handling bugs as well as querying and updating the database and editing bug reports. The project uses one of its many client interfaces, "send-pr", to send "Problem Reports" by email to the projects central GNATS server. The committers have also web and command-line clients available.

### 8.6.4. Mailman

Mailman is a program that automates the management of mailing lists. The FreeBSD Project uses it to run 17 general lists, 45 technical lists and 6 limited lists. It is also used for many mailing lists set up and used by other people and projects in the FreeBSD community. General lists are lists for the general public, technical lists are mainly for the development of specific areas of interest, and closed lists are for internal communication not intended for the general public. The majority of all the communication in the project goes through these 68 lists [FreeBSD, 2003A, Appendix C].

### 8.6.5. Perforce

Perforce is a commercial software configuration management system developed by Perforce Systems that is available on over 50 operating systems. It is a collection of clients built around the Perforce server that contains the central file repository and tracks the operations done upon it. The clients are both clients for accessing the repository and administration of its configuration.

### 8.6.6. Pretty Good Privacy

Pretty Good Privacy, better known as PGP, is a cryptosystem using a public key architecture to allow people to digitally sign and/or encrypt information in order to ensure secure communication between two parties. A signature is used when sending information out many recipients, enabling them to verify that the information has not been tampered with before they received it. In the FreeBSD Project this is the primary means of ensuring that information has been written by the person who claims to have written it, and not altered in transit.

### 8.6.7. Secure Shell

Secure Shell is a standard for securely logging into a remote system and for executing commands on the remote system. It allows other connections, called tunnels, to be established and protected between the two involved systems. This standard exists in two primary versions, and only version two is used for the FreeBSD Project. The most common implementation of the standard is OpenSSH that is a part of the project's main distribution. Since its source is updated more often than FreeBSD releases, the latest version is also available in the ports tree.

## 8.7. Sub-projects

Sub-projects are formed to reduce the amount of communication needed to coordinate the group of developers. When a problem area is sufficiently isolated, most communication would be within the group focusing on the problem, requiring less communication with the groups they communicate with than were the group not isolated.

### 8.7.1. The Ports Subproject

A "port" is a set of meta-data and patches that are needed to fetch, compile and install correctly an external piece of software on a FreeBSD system. The amount of ports has grown at a tremendous rate, as shown by the following figure.

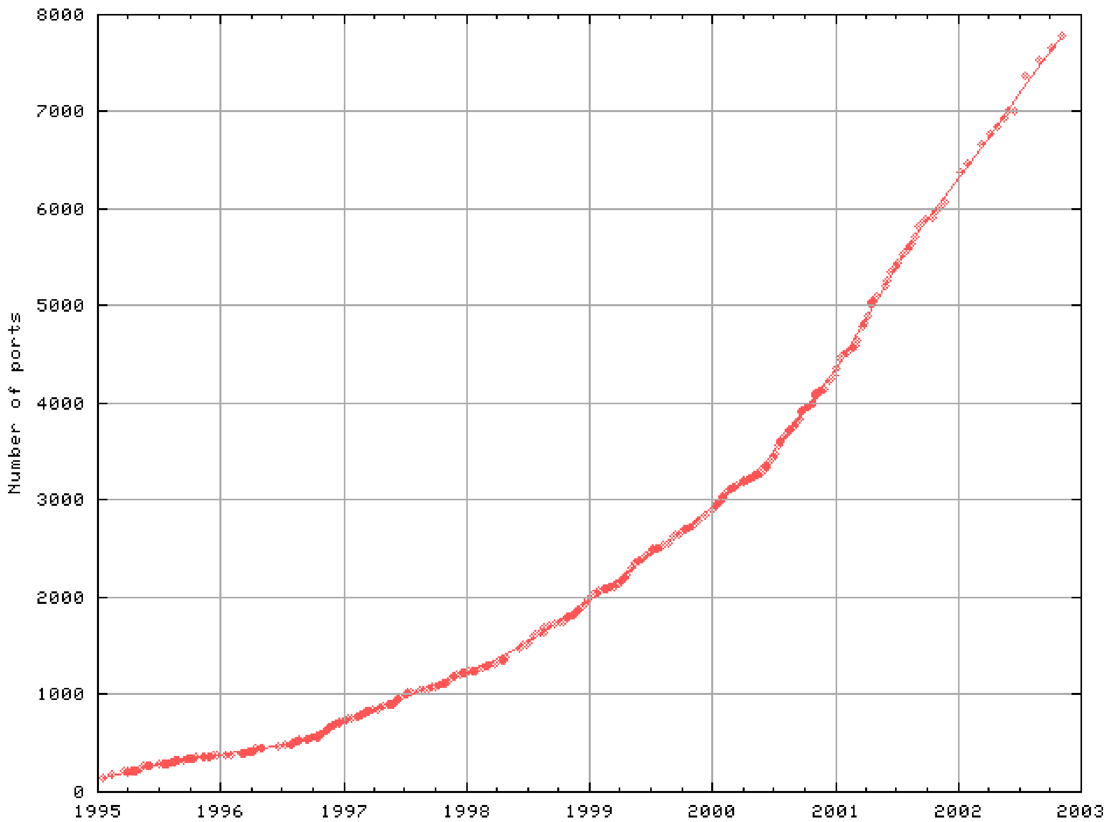


**Figure 34. Number of ports added between 1996 and 2003**

Figure 8.16 is taken from the FreeBSD web site. It shows the number of ports available to FreeBSD in the period 1995 to 2003. It looks like the curve has first grown exponentially, and then since the middle of 2001 grown linearly.

As the external software described by the port often is under continued development, the amount of work required to maintain the ports is already large, and increasing. This has led to the ports part of the FreeBSD project gaining a more empowered structure, and is more and more becoming a sub-project of the FreeBSD project.

Ports has its own core team with the Ports Manager as its leader, and this team can appoint committers without FreeBSD Core's approval. Unlike in the FreeBSD Project, where a lot of maintenance frequently is rewarded with a commit bit, the ports sub-project contains many active maintainers that are not committers.

Unlike the main project, the ports tree is not branched. Every release of FreeBSD follows the current ports collection and has thus available updated information on where to find programs and how to build them. This, however, means that a port that makes dependencies on the system may need to have variations depending on what version of FreeBSD it runs on.

With an unbranched ports repository it is not possible to guarantee that any port will run on anything other than -CURRENT and -STABLE, in particular older, minor releases. There is neither the infrastructure nor volunteer time needed to guarantee this.

For efficiency of communication, teams depending on Ports, such as the release engineering team, have their own ports liaisons.

### 8.7.2. The FreeBSD Documentation Project

The FreeBSD Documentation project was started January 1995. From the initial group of a project leader, four team leaders and 16 members, they are now a total of 44 committers. The documentation mailing list has just below 300 members, indicating that there is quite a large community around it.

The goal of the Documentation project is to provide good and useful documentation of the FreeBSD project, thus making it easier for new users to get familiar with the system and detailing advanced features for the users.

The main tasks in the Documentation project are to work on current projects in the "FreeBSD Documentation Set", and translate the documentation to other languages.

Like the FreeBSD Project, documentation is split in the same branches. This is done so that there is always an updated version of the documentation for each version. Only documentation errors are corrected in the security branches.

Like the ports sub-project, the Documentation project can appoint documentation committers without FreeBSD Core's approval. [FreeBSD, 2003B].

The Documentation project has a primer . This is used both to introduce new project members to the standard tools and syntaxes and acts as a reference when working on the project.

# Chapter 9

# Project Model Evaluation

In this chapter we will look at the project model and the evidence gathered, and see if the quality goals set up in the methods chapter have been met. Then we will classify the project model in accordance with what other authors have written on the matter, and finally comment on what is missing from the project model.

## 9.1. Have the quality goals set been met?

In Section 3.5 we found three categories of goals that we set out to investigate. During the past chapters, we have answered the questions derived from these goals by doing measurement in accordance with the metrics we have chosen. Now we will summarize our findings on these goals.

### 9.1.1. Product specific goals

#### *9.1.1.1. The OS should run interesting, new software*

As seen in section 8.7.1 and Section 6.5, there has been a steady growth of ports. This means that more software has steadily become available to FreeBSD. At the same time, this number has become so large that the amount of people required to keep every port up-to-date has become very large, and many ports are erroneous. As members use their own spare time to fix ports, it is likely that they will fix ports they are interested in themselves before fixing less interesting ports. As many in the community take particular interest in new software, it is more likely that the OS will run new, interesting software than old, uninteresting software.

#### 9.1.1.2. New features should be developed

As seen in Section 6.1, 132148 changes have been integrated into FreeBSD in the course of a year. At the start of the period investigated, -STABLE contained 716 modules. At the end, there were 723 modules in -CURRENT. According to an interviewee , most new development is targeted at -CURRENT and merged to -STABLE only if deemed stable enough and merging is feasible. This is thus a suitable way to measure change in the number of modules. Although some modules may be added due to maintenance, it indicates that new features are being developed.

#### 9.1.1.3. New releases should be made available regularly

As seen in Section 6.3.5, we have found that in general it takes between 33 and 160 days between each release of FreeBSD. The past three years, a minor release has been made every 3-4 months. This is considered regularly.

### 9.1.2. Project specific goals

#### 9.1.2.1. Active developers should be made committers

As seen in Section 4.2, there has been a steady increase in committers in the timeframe January 2000 - January 2003. This indicates that contributors are regularly made committers. This is backed up by Section 6.2.3.2 that shows that many new committers were indeed active in the period following them getting their commit privilege. This indicates that they have probably been active in the time leading up to receiving the commit privilege.

#### 9.1.2.2. Inactive committers should have their privileges revoked

As seen in Section 4.2, commit privileges are revoked much more seldom than they are granted. In general, the commit privileges were either revoked because the committer requested it, or during cleanup sweeps. These sweeps have been manually and infrequent.

#### 9.1.2.3. Developers should be able to add their code

In Section 6.2.3.2 we concluded that committers, both old and new, can easily and do add code.

#### 9.1.2.4. Updated code should be easily available for integration with running systems

In Section 5.4 we found that there are three major ways of distributing FreeBSD: discs sold, releases downloaded via the web and through CVSup. CVSup updates the FreeBSD code already existing in a system, and most of the 128 servers are at any time available to the users. We can therefore conclude that updated code is easily available for integration with running systems.

### 9.1.3. Process specific goals

#### 9.1.3.1. Problems should be identified and solved

In Section 5.3 we found that problems were identified through the creation and sending of problem reports. We found that the majority of problem reports had been assigned to a committer, and that the majority were handled so that they were considered closed. However, there were problem reports that had used more than two years to be either assigned to someone or to be handled at all. By sampling them we found that these problems were typically more complex problems. We must therefore conclude that although the problem solving works well in the general case, the issue of handling more complex problems should be given attention.

## 9.2. Project model classification

In Section 2.3.3 we used the framework of [Cockburn, 1998] to discuss methodology size. In this section we will discuss the project model as if it were a methodology. We will first classify it on the attributes defined in [Smevold, 2001], and then discuss it in the through Cockburn's findings.

### 9.2.1. Methodology attributes

"Methodology size" is a measure of how many control elements a methodology contains. Control elements are roles, activities, outcomes, standards and quality requirements described in the methodology. In Section 8.4 we find 24 roles. Section 8.5 lists 9 processes with a total of 59 activities. Both activities and processes have deliverables, and none of the activities or processes in the project model have more than one delivery. Thus there are 68 deliveries. The tools used in the process model give the standard of how the outcomes will be. Thus there are 7 standards followed. There are no quality requirements. Quality is upheld through review.

"Precision" is how precise the methodology is described. In Section 2.3.4 we decided rather than to formalise the processes completely, to describe the processes so that they can be executed as the people involved see fit rather than to use the project model as a straight jacket. This is in line with the research aim. The project model is thus of low precision.

"Tolerance" is a measure of how much deviation is allowed from the standards defined in the methodology. There is no defined tolerance except for that the tools have to understand the data. Many of the tools have been designed to be very tolerant, so with that respect there is a high tolerance on data variation.

The "specific density" of the methodology is a measure that can be understood as precision divided by tolerance. High specific density means high precision and low tolerance, while low specific density is the opposite. Since tolerance in this project model has a 1:1 relationship to precision, we find that the project model is of medium density.

[Smevold, 2001] describes "methodology weight" as methodology size multiplied with specific density, without detailing further the metrics used for this. The project model weight consists thus of the two factors methodology size, given by the vector [24,9,59,68,7], and the specific density, medium. While it probably gives little meaning comparing this attribute with other methodologies, it is an attribute that can be used to track the project model over time.

"Project size" is the number of people the methodology needs to coordinate. The project model uses most time at coordinating the 275 committers in the project, but also describes how the contributors and users can interface the project.

"Problem size" says how many elements there are in the problem domain and how complex these are to solve. The project model has not discussed this.

### 9.2.2. Methodology principles

Cockburn claims in his principle two that "A larger methodology is needed when more people are involved". We found the methodology size in the previous section. He divides methodologies into "little-m" and "big-M" methodologies where little-m are referring to methodologies as described in popular object oriented methodology books, and big-M are methodologies as used in large corporations. In such terms, this methodology is closer to a little-m methodology.

Being most similar to a little-m methodology is surprising as we saw in Section 1.2 and Section 8.2 that the number of people in the project has grown and the problem size broadened. This should indicate that a medium weight methodology is needed.

As noted in the introduction when discussing motivations, more people coming to the project means more opinions of what the problem domain is. This adds on both axis in the model seen in Section 1.2, and thus gives us an increased methodology size.

This is also in line with Brooks' Law (discussed in Section 2.3.5) as Cockburn's third principle says that "A relatively small increase in methodology size or specific density adds a relatively large amount to the cost of the project". [Cockburn, 1998] Brooks' discusses his law with regard to an increase in cost [Brooks, 1995: 274-275]. He finds that although sometimes cost is increased without an increase in time, but in the general case his law holds true. This is very true in the FreeBSD Project where close to all costs is developer time. An increase in people leads to an increase in methodology size that costs more developer time and makes the project later.

[Cockburn, 1998] divides methodologies into four classes based upon the problem size and criticality of the system developed. In his figure, however, he allows for many other attributes as well. A reduced figure, taking only these two variables into account is as follows

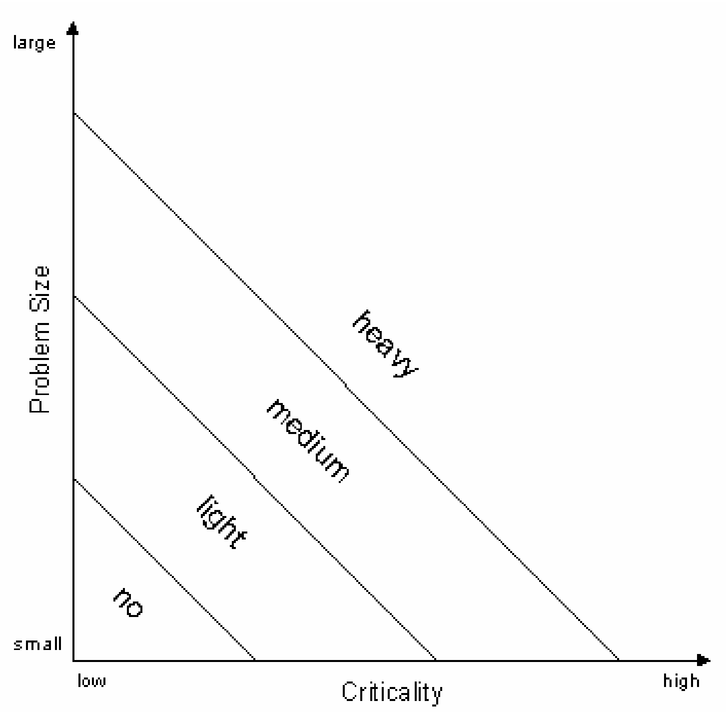


**Figure 35. Methodology weight needed**

Figure 9.1 shows the relation between what criticality and problem size the project faces and what kind of methodology it should have. For instance, development of pace makers has high criticality, because should the pace maker stop, the patients heart risks stopping. This warrants a heavy. Similarly, development of a Mars exploration vehicle has a large problem size, and thus should have a heavy methodology.

What is interesting about the FreeBSD project is that it lands in two categories of criticality: the -STABLE branch has a high criticality due to its use in production environments, and the -CURRENT branch has a low criticality as it is intended for hobby developers. However, both products are so intertwined that making a separate methodology for each does not make sense. The compromise made has been to have a high precision and low tolerance with regard to the release engineering process.

It is common in many organizations to have a written down or online methodology that is consulted for how sub-processes should be done. Some methodologies, such as Rational's "Unified Process" are designed to be consulted for each step. [Smevold, 2001] found that methodologies were likely to be used more actively in the project work if they were available on the project members' computers. In accordance with this, the project model is now available for both project members and people outside the project as a stand-alone, web-published document.

[Smevold, 2001] classifies methodologies in monumental and agile methodologies. (See Section 2.3.2.7 for agile models). The project model for the FreeBSD Project resembles the monumental methodologies when it comes to administration and project structure, but with its highly incremental development model, it resembles agile methodologies more when it comes to development.

## 9.3. Critique of Project Model

This section will discuss the weaker issues of the project model.

### 9.3.1. Project Model debugging

Since methodologies are software and are developed like software, they need to be debugged. How this should be done is not well defined. [Cockburn, 2000] suggests reflection sessions every 4-8 weeks asking "What did we learn?" and "What can we do better?" This may work well for project with clearly defined increments, but how is this done in a project with no set increments, such as the FreeBSD project? It may be done through an ongoing discussion on a mailing list, but this will rarely include the entire project group such as it would in the reflection sessions suggested by Cockburn.

[Kitchenham et al, 1997] discusses evaluation of project models and proposes the DESMET evaluation methodology. It uses formal experiments, case studies and surveys as means to evaluate a project model. Just like a methodology needs a methodology to support its creation, it needs to be evaluated. DESMET has successfully been applied to evaluate its own methodology. While this would probably be a very scientific correct way of evaluating it, it has a very high cost in the amount of time used, and I cannot say that the benefits it can offer would be worth the time invested.

Formalising the project model in accordance with [Osterweil, 1987] would make the project model maintainable through regular debugging just like any piece of software. The trade-off, however, is that the project members would feel more restricted in how they should do their work. Since many committers have expressed that they are in the project because it is fun, reducing the amount of enjoyment in the project would be counter-productive. However, tasks that are not development related, should be automated as much as possible so that the committers can do what they do the best and enjoy the most: develop software.

### 9.3.2. Few standards

Except for [McKusick et.al, 1997], [FreeBSD, 2001B] and the standard set of tools, there are not many standard templates for how things should look.

The project does not commit to many standards either. With every release comes a list of hardware supported and a list of what has been implemented since last time. As discussed in Section 6.4, there are no lists saying what standards are implemented and to what degree different standards are implemented. Also, the project does not maintain a list of standards it aims to implement and when it aims to have them implemented. From this, it seems the project does not commit to having standards implemented, and if standards are indeed implemented, this is good but not guaranteed to be true in the next version.

This is quite the opposite impression that the FreeBSD project wants to give. It has indeed implemented many standards. For example, [Open Group, 1997] was implemented through the OpenPAM project. There is also an active standards mailing list. FreeBSD is also reputed to be very standard compliant. Thus the impression the project model and its outcomes gives regarding standards is wrong. The model should be updated with a target list containing what standards are going to be implemented to what extent.

### 9.3.3. What have we found about quality?

As we saw in the first section of this chapter, we have created a set of quality goals, problemised them through questions and made them measurable by defining metrics. In this way, we have said something about quality instead of "going poof" as discussed in Section 2.4.

Is this enough? We have seen in this chapter that the problem domain increases with more people in the project, and people join the project regularly. Clearly, there are and will be more project goals.

Goals should be handled by the project. First, goals should be identified. These can be identified through a tracking system where people submit goals important to them. Then these goals could be rated among the project members. Also, it'd be beneficial if the users could rate the goals to see the difference between the two groups. Then a set of goals could be extracted from the list to form official project goals. These would probably be the best set of goals to use in measuring the project model quality over time.

At the moment, quality goals are ensured only through peer review and mailing list discussions. Project participants have been upset when their idea of quality has been reduced. This has led to a number of disagreements that core has had to mediate in. Also, it means that those who are most articulate about their opinions get heard the most.

But the way goals are defined today is in line with Section 7.1. Rather than to discuss the goals, the ones who implement their personal goals as code get the actual say in what quality is.

The discussion between quality goals and standards is closely linked. If a set of goals are identified, a list of targeted standards is easy to make. But if goals remain vague ideas that are only clear once they are implemented, such a list is probably not in order.

### 9.3.4. What are the outcomes?

There are three main outcomes of the development processes: code, documentation and releases. Code and documentation are seen as source code that is used to build releases. Source code is the only outcome from the development processes that is harvested. Is this beneficial?

The different developers and sub-projects use each their own development method or methodology. The only level they interface one-another on is source-code and by mailing lists and private email. The software engineering community has long investigated the benefits of cooperating at design level and making design repositories for projects that consist of many subprojects. The same is true for requirements, verification documents and tests. However, to cooperate on these levels means less freedom for the individual developer which again may limit their willingness to contribute. Even so, this possibility should be investigated further.

### 9.3.5. Processes for testing missing

At the moment, testing in FreeBSD happens very random. This makes tracking errors harder as it requires very experienced users in order to find an error.

In order to structure testing more, there should be a central way for interested testers to list their hardware. This would allow device driver developers to contact interested testers and notify them about upcoming drivers for testing. Such a database could also give an indication of what popular hardware lacks driver support.

Juli Mallet made a test environment she could use to test the reimplementation of xargs. When PAM was reimplemented, no such testing was done. Rather, a test set of headers was used to compile applications with to see if they integrated well with LinuxPAM, and when they did not and the maintenance of LinuxPAM became too cumbersome, a reimplementation was done. What kinds of testing are most beneficial while at the same time being easy to maintain should be investigated. A test set allowing and automating multiple kinds of testing (regression testing, unit testing, structural testing) would be beneficial to aid continued testing.

### 9.3.6. Who are the users?

The project model focuses on "what does a developer do?". That already implies that we know who the developers are, which we answer in the project model. What is not answered is "who are the users?" [Lehey, 2002] notes that Core's focus is not on the users, and the culture is in my experience very committer-centric.

Is this a bad thing? After all, the committers are real power-users of FreeBSD. Some committers are employed by major companies that need the expertise on FreeBSD or need features implemented. We found in Section 4.1 that by far most contributors are contributors through sending problem reports. There are also very many users active on mailing lists (see Section 5.2). This indicates that there are many users that want to be included and have opinions, needs and take their time to contribute in one way or another.

Other than problem reports and mailing lists, users make also up a lot of the community around the project. Examples of this are online communities, providing among other things services such as peer support and how-to articles (see FreeBSD China Community, The FreeBSD Diary, Defcon1 and BSDVault), FreeBSD User Groups (such as Norwegian BSD User Group, Home Unix User Group and Daibou *BSD Users Group), News sites (such as Dæmonnews, BSD Today and BSDForums.org) and commercial services (such as FreeBSD Services, BSD Mall and BSD Consulting).

With such a diverse community, a committer-centric environment is probably not the most favourable environment. Ways to include the users more in the project should be investigated. The software engineering community has written much on such issues, and inspiration could be sought from this community.

### 9.3.7. Unclear roles

The FreeBSD Foundation is a non-profit organisation dedicated to supporting the FreeBSD Project. It does this in terms of fund raising, represent the FreeBSD Project in legal contracts or other arrangements that require a recognized legal entity.

The FreeBSD Foundation interfaces the FreeBSD Project in that one member is a member of the core team. As the core team is elected every two years, this will not necessarily continue to be the case.

The FreeBSD Foundation is not included in the project model as it is not considered a part of the project, but a useful, outside entity.

In November 2002, a technical review board was established for a trial period of three months. Its aim is to make decisions with regards to what technical direction the project is going to take. It handles issues like what technical solutions are the wisest and in the case of multiple solutions, which one to choose on technical grounds.

The FreeBSD Foundation does already much administrative work and is an important legal entity. With the Technical Review Board doing the technical work of Core, Core remains as a conflict resolving and commit privilege assigning entity. An alternative organisation, in line with the division between development processes and administrative processes, would be to have the foundation to handle administrative issues and the review board to handle development issues. The legal person in charge of the foundation could be a secretary for the foundation with the foundation's board being elected like Core is today. The technical review board could be elected in the same way, at the same time. This could bring back the way the fun to the FreeBSD Project that the two members that resigned from the first elected core were missing.

The project model has thus a problem of how the FreeBSD Foundation should be integrated, and in that process needs to discuss the management roles in the FreeBSD Project. Alternatives, of which I have outlined one, should be investigated.

## 9.4. Summary

The main findings in this chapter are:

- The FreeBSD Project scores well on most of the defined quality goals
- We have classified the project model in accordance with [Smevold, 2001] and [Cockburn, 1998]
- There are issues regarding the project organisation that needs to be addressed

# Chapter 10

# Conclusion

In this thesis, we have the following three products:

- A descriptive project model for the FreeBSD Project
- A set of quality metrics for the FreeBSD project model
- A comparison of the project model to the quality metrics

The project model has been verified by many community members as factually correct, and contains most of what a project model should. It does not address some issues as discussed in the evaluation because these issues are not resolve by the community.

The metrics found support quality goals set for the project. These goals are some of the project goals, but as a clear list of goals is not available, these may not be the most representative goals. The metrics were well suitable to apply to the project model and have given new information on the quality of the project.

This chapter will revisit the findings done and conclude from them. Then it will suggest issues for further research.

## 10.1. My findings

In chapter 4-7 we have made a list of findings about the FreeBSD Project. These findings are as follows:

### 10.1.1. Project organisational findings

- Commit privileges are seldom revoked.
- Committers are added regularly, and because commit privileges are seldom revoked, the project is in a state of steady growth.
- There are an estimate of 5500 contributors to the project
- Core has well defined tasks and authority, and is elected at least every two years.
- Just under 50% of the modules (382 of 716) have a designated or de-facto maintainer, most modules do not
- FreeBSD consists of many sub-projects. We have studied three, quite different, sub-projects.

### 10.1.2. Administrative findings

- Several tools are used to support administrative functions and ensure consistency between related items (such as code changes and problem reports).
- Mailing lists are used as the main form of communication within the project
- Problem reports are kept in a standardised, machine- and human-readable format.
- Until February 26th, 2003, 48697 problem reports had been filed. Of these, 44485 reports were closed.
- Most problems are resolved quickly. More than 60% received attention within the first 15 days But when problems are not resolved immediately, they may take very long to solve.

- Most problems are assigned to someone in the project when they arrive. Unassigned problems may take long before being assigned.
- Members of the project work with problem reports that are not assigned to anyone.
- FreeBSD is distributed by CD, DVD and CVSup. The data for number of CDs and DVDs sold or burnt are unavailable. CVSup is a tool for distributing updated source code through the net. There are 128 official CVSup servers to which people can synchronise their systems, and the number of available servers is growing.
- 8-13% of the CVSup servers are at the sampled times unavailable.

### 10.1.3. Development-related findings

- [Jørgensen, 2001]'s model for integration of changes makes together with FreeBSD's release version model and my proposed model for development of new features an overall model of FreeBSD's evolutionary software process model.
- Although most changes are maintenance changes, we have seen that some changes do not fit into the change list of [Swanson, 1976], and based on experience a descriptive model for implementing new features has been proposed.
- There is no standard way of gathering requirements. However, TODO lists are widely used, and the problem reports database is a standard source of user requirements.
- Apart from [McKusick et.al, 1997], there is no explicit design for FreeBSD. The project does not maintain a repository of design artefacts.
- Committers can, and do, add code. During one year, 132,148 code changes were integrated into the code repository.
- New committers can, and do, add code just as easily as old committers
- FreeBSD consists of over 700 modules.

- A test framework is available, but it is little used at the moment. Most testing happens on the users' hardware and on dedicated tinderboxes.
- Most review happens on the basis of the commit log. The commit log helps other developers be on top of what happens in the project and express their opinion on that. Little review happens on the basis of the code itself, especially for larger pieces of code.
- Deprecated code should be in FreeBSD at least one major version after it has been decided through a vote to deprecate it. However, there have been exceptions from this rule.
- The project provides multiple versions of FreeBSD intended for audiences with different priorities between the newest features and rock solid stability and security. This means that the newest minor release or updates along a security branch are already old in terms of being well tested when they are released.
- New versions of FreeBSD are released on a regular basis and the release model fits in with the evolutionary software process model, the change integration model and the new feature development model.
- Apart from a book describing the 4.4BSD kernel, there are little standards the project has committed to follow. Having said that, the project does implement many standards.
- FreeBSD has over 8000 ports of third party software available. A small portion of these are not working or dependent on non-working ports, and some ports that work on one architecture do not work on another.

### 10.1.4. Findings from issues that span over the project model

- Bikeshedding inhibits design and explains the earlier finding that small changes get the most feedback when understanding it to be that the best understood changes get the most feedback.
- The FreeBSD Project lacks guidelines for large undertakings that do not build on older modules or the adding of modules. An example of such

changes is structural changes. The project also lacks ways of assigning people to problems.

- The up until now tacitly understood project model is reenforced through people referring to it.
- FreeBSD is released in versions where an overall velocity is kept in the project, but individual versions follow the model of initiating, scaling up, production and termination.
- Maintenance and its relation to new development is a complex issue. The FreeBSD project, a maintenance centric project, contradicts the hypothesis that much maintenance work is a motivation for leaving the project through constantly getting more committers and having a high volume of changes per year.

### 10.1.5. Evaluation findings

- The FreeBSD Project scores well on most of the defined quality goals
- We have classified the project model in accordance with [Smevold, 2001] and [Cockburn, 1998]
- There are issues regarding the project organisation that needs to be addressed

### 10.1.6. Conclusion summary

The most interesting findings are the following:

- The FreeBSD Project scores well on most of the defined quality goals
- There are issues regarding the project organisation that needs to be addressed

The first finding is a summary of many findings and shows that the FreeBSD Project is a healthy project. The second requires a continued debate about how the project should be run to resolve the issues found.

The conclusion is therefore that the research aim has been met, and that during the research, many findings regarding the FreeBSD Project has been made.

This research has provided a detailed insight into the FreeBSD Project lays the ground for a study between this and other open source projects to gain a better understanding of the open source community so we can use this knowledge to improve software development efforts.

## 10.2. Further research

The Project Model has been donated to the FreeBSD Documentation Project where it will be kept up-to-date with regards to how the project evolves. It will probably to at some point be representative enough and both well read and discussed enough that it can be considered a methodology for the project. Discussions regarding it are expected to contribute directly to the process improvement work in the project. How the project model changes into a methodology, and if it does not, why, are subjects that should be investigated further. From this we can learn about the interactions of the open source community and the established software engineering community.

So far, Linux has been the most researched open source project. Very few other projects have been studied, but this thesis allows for a comparison between the FreeBSD Project and the papers written on Linux and future papers written on other open source projects. Cross-case studies between multiple open source and closed source projects can be done to find out how we can improve software development efforts.

The FreeBSD Project and many open source project interface sub-projects only at code-level and do not collect outcomes from other development stages such as design documents. Whether it is beneficial to interface other projects at more levels than only through code, and what these benefits might be, should be investigated.

FreeBSD has many expert programmers. Many of these have realized the value of design patterns. This reduces the need for detailed design as more information can be communicated with less by referring to the design pattern. How much is it used? How efficient is it? Does its use encourage newcomers to learn about them? Does it alienate anyone? These are questions that can shed light on the use of patterns in the open source community.

This project model has little interaction with the end users. However, end users are major stakeholders in terms of time and equipment. How can this be handled? How should the end users influence the project?

# Appendix A
# Data collected

The data collected in my research, including the tools and interviews, is available at http://niklas.saers.com/thesis. This section will go through the different parts

"commitscan/commitscan.pl" is a utility to count the amount of commits done for a committer that categorises the commits in kernel, userland, documentation and web commits. The categories are represented with Y, U, D and W respectively. A committer will be marked with a capital letter for the category he has committed the most in, and with a lower case letter for each of the other categories that he made at least 12 commits to. The utility allows setting cut-offs and can show who has made how many commits by module.

"findmodules/findmodules.pl" is a utility that goes through the source code and makes an estimate of how many modules there are. It does this by looking for Makefiles that usually bind together a module. However, there are places in the kernel where a Makefile will bind together more modules, and there are a few places where there are multiple Makefiles to a module, so the number is a close estimate not a precise number.

"PRs/reduce.php" is a script that goes through the data retrieved by going through the Problem Reports and halving the amount of samples.

"contributors/originators.sh" gathers information on who has contributed a problem report. It serves as an example of how all analysed fields are collected from the GNATS database. It is followed by "contriubtors/originators.php" that counts the number of samples starting on A in both the original and edited contributors list.

"cvsup/cvsup.php" goes through a set of CVSup servers and tries to complete a CVS update. It notes down all output for manual processing later.

"findmodules/modules-orphans.php" goes through the modules that do not have an official maintainer and check if they have a de-facto maintainer.

"timediff/timediff.py" makes a list of when a problem report arrived, when the problem reports state was changed and how long time it took. "timediff/timediff_responsible.py" does the same, but notes down how long it takes before responsibility for the problem report is assigned. The equivalent PHP files launch the utility for each problem report listed in the GNATS database.

All the resulting raw data are in the same subdirectory as the utilities. Processed data are available as Excel spreadsheets in the excel subdirectory.

# Appendix B
# Mailing list and mail references

In the absence of good standards for referring to online material such as archived mailing lists, the first part of this chapter will serve as references to the relevant mailing lists. The second part will serve as references to email interviews and interview notes.

## B.1. Mailing lists

As threads of mail can change topic as the thread evolves, the list has been ordered such that the first mail on the topic I have found of interest has been listed.

[current-041002a]

List: freebsd-current@freebsd.org
Author: Poul-Henning Kamp
Subject: Re: Junior Kernel Hacker page updated...
Date: Fri, October 4th, 2002 22:34:22 +0200

[current-041002b]

List: freebsd-current@freebsd.org
Author: Ruslan Ermilov
Subject: Vote: lib/libexpat -> lib/libbsdxml
Date: Fri, October 4th, 2002 17:56:29 +0300

Follow-ups:

Daniel Flickinger, Fri, October 4th, 2002 19:36:59 +0000 (GMT)

Garance A Drosihn, Fri, October 4th, 2002 17:15:24 -0400

Poul-Henning Kamp, Fri, October 4th, 2002 23:21:47 +0200

[arch-150103]

List: freebsd-arch@freebsd.org

Author: Poul Henning Kamp

Date: January 15th, 2003

link

[news-37193]

Newsgroup: comp.os.386bsd.announce

Author: Jordan K. Hubbard

Subject: FreeBSD 2.0 - a status report

Date: October 26th, 1994

link

[chat-020517]

List: freebsd-chat@freebsd.org

Author: Terry Lambert

Subject: Re: FreeBSD core team questions

Date: Fri May 17 12:55:05 2002

[hackers-950106]

List: freebsd-hackers@freebsd.org

Author: Jordan K. Hubbard

Subject: Desperately Seeking Doc hackers!

Date: Fri Jan  6 01:52:26 1995

[hackers-950110]

List: freebsd-hackers@freebsd.org

Author: Jordan K. Hubbard

Subject: Doc project status

Date: Tue Jan 10 04:24:35 1995

[announce-27014]

Newsgroup: comp.os.386bsd.announce

Author: Rob Kolstad

Subject: USL vs. BSDI Lawsuit Settled

Date: February 6th, 1994 17:51:58 -0800

link

List: freebsd-security@freebsd.org

Author: Jacques A. Vidrine

Subject: Updated FreeBSD Security Officer PGP Key

Date: Tue, January 7th, 2003 11:47:11 -0600

link

[current-20030305]

List: freebsd-current@freebsd.org

Author: Jim Hatfield

Subject: Plea for base system trim

Date: Wed, 5 Mar 2003 09:54:13 -0000

Follow-ups:

Doug Barton, Wed, 5 Mar 2003 02:14:16 -0800 (PST)

## B.2. Email interviews

[int-maksim]

Interviewee: Maksim Yevmenkin

Period: September 13th - October 21st, 2002

link

[int-nik]

Interviewee: Nik Clayton

Period: November 27th - December 7th, 2002

link

[int-des]

Interviewee: Dag-Erling Smørgrav

Period: November 21st, 2002 - March 11th, 2003

# Bibliography


[Basili et.al, 1994] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Copyright © 1994 Encyclopedia of Software Engineering. John Wiley & Sons. *The Goal Question Metric Approach*. 528-532.

[Benington, 1987] Herbert D. Benington. Copyright © 1987 Computer Society Press of the IEEE. . Computer Society Press of the IEEE. *9th Conference on Software Engineering*. Production of Large Computer Programs. 299-310.

[Boehm, 1973] Barry W. Boehm. Copyright © September 1973. *The High Cost of Software*.

[Boehm et al, 1976] Barry W. Boehm, J. R. Brown, and M. Lipow. Copyright © 1976 TRW Systems and Energy Group. *Quantitative Evaluation of Software Quality*.

[Boehm, 1988] Barry W. Boehm. Copyright © May, 1988 IEEE Computer. IEEE Computer. *A Spiral Model of Software Development and Enhancement*.

[Brooks, 1995] Frederick P. Brooks. Copyright © 1975, 1995 Pearson Education Limited. 0201835959. Addison-Wesley Pub Co. *The Mythical Man-Month*. Essays on Software Engineering, Anniversary Edition (2nd Edition).

[Budde et al, 1992] R Budde, K Kautz, K Kuhlenkamp, and H Ziillighoven. Copyright © 1992 Springer Verlag. 3-540-54352-x. Springer Verlag. *Prototyping*. an Approach to Evolutionary System Development.

[Canning, 1972] Canning. Copyright © 1972 EDP Analyzer. Canning Publications. *That Maintenance Iceberg*. 1-14. 10.

[Cockburn, 1998] Alistair Cockburn. Copyright © 1998 Central Bank of Norway. Central Bank of Norway. Oslo Norway . *The Methodology Space*. http://alistair.cockburn.us/crystal/articles/ms/methodologyspace.htm

[Cockburn, 2000] Alistair Cockburn. Copyright © 2000 Central Bank of Norway. *Just-In-Time Methodology Construction*. http://alistair.cockburn.us/crystal/articles/jmc/justintimemethodologyconstruction.html

[Daemon News, 2001] Copyright © 2001 Daemon News. *Does Microsoft run BSD code?* http://www.daemonnews.org/200108/dadvocate.html

[Daemon News, 2002] Copyright © 2002 Daemon News. *Jordan Hubbard resigns from FreeBSD core*. http://daily.daemonnews.org/view_story.php3?story_id=2837

[FreeBSD, 2001B] Copyright © 2001 The FreeBSD Project. *FreeBSD Kernel Developer's Manual*. style(9) - Kernel source style guide.

[FreeBSD, 2003C] Copyright © 2003 The FreeBSD Documentation Project. *Frequently Asked Questions for FreeBSD 2.X, 3.X and 4.X*. Miscellaneous Questions. http://www.freebsd.org/doc/en_US.ISO8859-1/books/faq/misc.html

[FreeBSD, 2002J] Copyright © 2002 The FreeBSD Project. *FreeBSD core status report*. Month: February 2002.

[FreeBSD, 2000A] Copyright © 2002 The FreeBSD Project. *Core Bylaws*. http://www.freebsd.org/internal/bylaws.html

[FreeBSD, 2002A] Copyright © 2002 The FreeBSD Documentation Project. *FreeBSD Developer's Handbook*. http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/

[FreeBSD, 2002B] Copyright © 2002 The FreeBSD Project. *Core team election 2002*.
http://election.uk.freebsd.org/candidates.html

[FreeBSD, 2002C] Dag-Erling Smørgrav and Hiten Pandya. Copyright © 2002 The FreeBSD Documentation Project. The FreeBSD Documentation Project. *Problem Report Handling Guidelines*.
http://www.freebsd.org/doc/en/articles/pr-guidelines/article.html

[FreeBSD, 2002D] Dag-Erling Smørgrav. Copyright © 2002 The FreeBSD Documentation Project. The FreeBSD Documentation Project. *Writing FreeBSD Problem Reports*.

http://www.freebsd.org/doc/en/articles/problem-reports/article.html

[FreeBSD, 2001] Copyright © 2001 The FreeBSD Documentation Project. The FreeBSD Documentation Project. *Committers Guide*.
http://www.freebsd.org/doc/en/articles/committers-guide/article.html

[FreeBSD, 2002E] Murray Stokely. Copyright © 2002 The FreeBSD Documentation Project. The FreeBSD Documentation Project. *FreeBSD Release Engineering*.
http://www.freebsd.org/doc/en_US.ISO8859-1/articles/releng/article.html

[FreeBSD, 2003A] The FreeBSD Documentation Project. *FreeBSD Handbook*.
http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook

[FreeBSD, 2002F] Copyright © 2002 The FreeBSD Documentation Project. The FreeBSD Documentation Project. *Contributors to FreeBSD*.
http://www.freebsd.org/doc/en_US.ISO8859-1/articles/contributors/article.html

[FreeBSD, 2002G] Copyright © 2002 The FreeBSD Project. The FreeBSD Project. *Core team elections 2002*.

http://election.uk.freebsd.org

[FreeBSD, 2002H] Copyright © 2002 The FreeBSD Project. The FreeBSD Project. *Commit Bit Expiration Policy*. 2002/04/06 15:35:30.

http://www.freebsd.org/internal/expire-bits.html

[FreeBSD, 2002I] Copyright © 2002 The FreeBSD Project. The FreeBSD Project. *New Account Creation Procedure*. 2002/08/19 17:11:27.
http://www.freebsd.org/internal/new-account.html

[FreeBSD, 2003B] Copyright © 2002 The FreeBSD Documentation Project. The FreeBSD Documentation Project. *FreeBSD DocEng Team Charter*. 2003/03/16 12:17.
http://www.freebsd.org/internal/doceng.html

[Giddens, 1997] Anthony Giddens. Copyright © 1990, 1997 the Board of Trustees of the Leland Stanford Junior University. 82-530-1920-3. Pax Forlag. *Modernitetens Konsekvenser*.

[Gilb, 2003] Tom Gilb. Copyright © 2003. *Quantifying System Qualities*.

[Hars et al, 2001] Alexander Hars and Shasong Ou. Copyright © 2001 Marshall School of Business at University of Southern California. Working for Free?. Motivations of Participating in Open Source Projects.

[Hohmann, 1997] Luke Hohmann. Copyright © 1997 Prentice Hall. 0 13 236613 4. Prentice Hall. *Journey of the Software Professional*. A sociology of Software Development.

[Hubbard, 1997] Copyright © 1997. Jordan K. Hubbard. *A Brief History of FreeBSD*.
http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/history.html

[IN331, 2002] Geir Amsjø. Copyright © 2002 University of Oslo, Department of Informatics. *IN 331*. Produkt og prosessforbedring i systemutvikling. http://www.ifi.uio.no/in331/foiler/

[Joy, 2000] Bill Joy. Copyright © 1994, 2000 Wired. . Wired. *Why the future doesn't need us*. . 1-18.

[Jørgensen, 2001] Niels Jørgensen. Copyright © 2001. *Putting it All in the Trunk*. Incremental Software Development in the FreeBSD Open Source Project. http://www.dat.ruc.dk/~nielsj/research/papers/freebsd.pdf

[Kamp, 2003] Poul-Henning Kamp. Copyright © 2003. *Poul-Hennings JKH (Junior Kernel Hacker) TODO list*. http://people.freebsd.org/~phk/TODO/

[Kitchenham et al, 1997] Barbara Kitchenham, Stephen Linkman, and David Law. Copyright © 1997 Computing & control engineering journal. Computing & control engineering journal. *DESMET*. a methodology for evaluating software engineering methods and tools. 8 #3. 120-126.

[Laird et.al, 2001] Cameron Laird and Kathryn Soraiz. Copyright © 2001 DefCon1. *The story on FreeBSD*. http://www2.defcon1.org/html/Software_Articles/boot_floppies/Daylight/mail-spool/fbsd-story.html

[Lehey, 2002] Greg Lehey. Copyright © 2002 Greg Lehey. Greg Lehey. *Two years in the trenches*. The evolution of a software project. http://www.lemis.com/grog/In-the-trenches.pdf

[Losh, 2002] Warner Losh. Copyright © 2002 The FreeBSD Documentation Project. *Working with Hats*. http://www.freebsd.org/doc/en/articles/hats/article.html

[Mann et. al, 2000] Chris Mann and Fiona Stewart. Copyright © 2000 Biddles Ltd. 0 7619 66 26 9. 0 7619 66 27 7. Biddles Ltd. Surrey . *Internet Communication and Qualitative Research*. A Handbook for Researching Online.

[McKusick, 1985] Marshall Kirk McKusick. Copyright © 1985 UNIX Review. UNIX Review. *A Berkeley Odyssey*. Ten years of BSD history. 31-42.

[McKusick, 1992] Marshall Kirk McKusick. Copyright © 1992 UNIX Review. UNIX Review. *UNIX Variants*. BSD. 46-47.

[McKusick, 1994] Marshall Kirk McKusick. Copyright © 1994 UNIX Review. UNIX Review. *What's New in 4.4BSD*. The final release of BSD UNIX shows a surprising host of new features and enhancements. 51-56.

[McKusick et.al, 1997] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. Copyright © 1996, 1997 Addison-Wesley Publishing Company, Inc. 0-201-54979-4. Addison-Wesley Publishing Company, Inc. *The Design and Implementation of the 4.4BSD Operating System*.

[McKusick, 1999] Marshall Kirk McKusick. Copyright © 1999 O'Reilly. 1-56592-582-3. O'Reilly. *Open Sources: Voices from the Open Source Revolution - Twenty Years of Berkeley Unix*. From AT&T-Owned to Freely Redistributable.

[Miller et al, 2000] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekanada Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Copyright © 1976 Computer Sciences Department, University of Wisconsin. *Fuzz revisited*. A Re-examination of the Reliability of UNIX Utilities and Services.

[Mills et al, 1980] Harlan D. Mills, Donald O'Neill, Richard C. Linger, Michael Dyer, and Robert E. Quinnan. IBM Systems Journal. Copyright © 1980 IBM Systems Journal. *The management of software engineering*. 19 #4. 414-465.

[Open Group, 1997] Copyright © 1997 Open Group. 1-85912-144-6. *X/Open Single Sign-On Service (XSSO)*. Pluggable Authentication.

[Osterweil, 1987] Leon J. Osterweil. Copyright © 1987 University of Colorado Boulder. University of Colorado Boulder, Colorado, USA. *Software processes are software too*.

[Osterweil, 1997] Leon J. Osterweil. Copyright © 1997 University of Massachusetts, Dept. of Computer Science. University of Massachusetts, Dept. of Computer Science, Amherst, MA 01003, USA. *Software processes are software too, Revisited*. An Inviterd Talk on the Most Influential Paper of ICSE 9.

[Patton, 2002] Micahel Quinn Patton. Copyright © 1990, 2002 Sage Publications, Inc. . Sage Publications, Inc. *Qualitative Research & Evaluation Methods*. 3rd. edition. 432-534.

[Pirsig, 1989] Robert M. Pirsig. Copyright © 1974, 1989 The Bodley Head. 0-09-978640-0. Black Swan. London England . *Zen and the art of motorcycle maintenance*. an inquiry into values.

[PMI, 2000] Project Management Institute. Copyright © 1996, 2000 Project Management Institute. 1-880410-23-0. Project Management Institute. Newtown Square Pennsylvania USA . *PMBOK Guide*. A Guide to the Project Management Body of Knowledge, 2000 Edition.

[Raymond, 2000] Eric Steven Raymond. Copyright © 2000 Eric Raymond. *The Cathedral and the Bazaar*.

[Reed, 2001] Jeremy C. Reed. Copyright © 2001 BSD Today. *Wind River and FreeBSD's relationship ending*. http://www.bsdtoday.com/2001/October/News572.html

[Royce, 1970] Dr. Winston W. Royce. Copyright © 1970. *Managing the Development of Large Software Systems*.

[Saers, 2002] Niklas Johannes Saers. Copyright © March, 2002 Niklas Saers. *Process models for modern software development*. a comparision of eXtreme Programming and the Spiral Model.

[Scherrer et.al, 1992] Deborah Scherrer and Vance Vaughan. Copyright © 1992 UNIX Review. UNIX Review. *UNIX Variants*. Mach. 32-38.

[Smevold, 2001] Terje Smevold. Copyright © 2001. Department of Informatics, University of Oslo. Oslo Norway . *Prosesstøttende intranett*. Metodikkweb og prosjektweb.

[Sommerville, 2001] Ian Sommerville. Copyright © 1982, 1984, 1989, 1992, 1995, 2001 Pearson Education Limited. 0 201 39815 X. Pearson Education Limited. Edinburgh Gate Harlow Essex CM20 2JE England . *Software Engineering*. 6th Edition.

[Swanson, 1976] E. Burton Swanson. Copyright © 1976 IEEE. IEEE - International Conference on Software Engineering. *The Dimensions of Maintenance*. IEEE Catalog No. 76CH1125-4C.

[Tatham, 1999] Simon Tatham. Copyright © 1999 OpenContent. *How to Report Bugs Effectively*.
http://www.chiark.greenend.org.uk/~sgtatham/bugs.html

[Turk et al, 2000] Daniel Turk and Vijay Vaishnavi. Copyright © 2000 Idea Group Publishing. Idea Group Publishing, Hershey. *Software process models are software too*. a domain class model for software process models.

[Wilson, 2002] Brian Wilson. Copyright © 2002. *Browser Timelines*. Netscape Navigator.
http://www.blooberry.com/indexdot/history/netscape.htm

[Yancey, 2003] Kelly Yancey. Copyright © 1999-2003. *BSD Driver list*.
http://www.posi.net/freebsd/drivers/