AALBORG UNIVERSITY
Department of Computer Science

Technical Report

Configuration Management for Open Source Software

by

Ulf Asklund & Lars Bendix

R-01-5005 **January 2001**

DEPARTMENT OF COMPUTER SCIENCE
Fredrik Bajers Vej 7E ? DK-9220 Aalborg Ø ? Denmark
Phone +45 9635 8080 ? Telefax: +45 9815 9889
URL: http://www.cs.auc.dk

Configuration Management for Open Source Software

Ulf Asklund[1], Lars Bendix[2]

[1] Department of Computer Science, Lund Institute of Technology, Box 118, S-22100 Lund, Sweden, Email: Ulf.Asklund@cs.lth.se
[2] Department of Computer Science, Aalborg University, Fredrik Bajers Vej 7E, D-9220 Aalborg Øst, Denmark, Email: bendix@cs.auc.dk

Abstract

Any organisation that produces high quality software merits a closer analysis of their methods such that good techniques can be transferred to other organisations. Open Source Software projects is such a case. We make explicit their underlying process for handling change management and analyse to what extent their success can be attributed to good process, tools or people. Furthermore, we discuss to what degree lessons learned from OSS can be transferred to more traditional ways of developing software.

1. Introduction

It is beyond doubt that OSS projects produce software of high quality. This despite a seemingly anarchistic way of organising projects and having a set-up (many, distributed developers) that is usually considered difficult to handle within the field of configuration management. In this paper, we would like to investigate more closely what they actually do, and why they are so successful. We will describe their underlying change management process, thereby making it explicit, so it can be followed in case others want to start an OSS project. We will also analyse to what extent their success is due to a good process, good tools or simply to outstanding people participating in OSS projects. Based on this we discuss which lessons learned form OSS could be transferred to traditional ways of developing software. Change management is a part of configuration management that is keeping your configurations consistent under change.

This led us to establish two research questions that we wanted to investigate with relation to configuration management for Open Source Software:

Research question #1: How do they do it - can we make their implicit process *repeatable*? If a (commercial) company wants to start an OSS project or a project having the same characteristics (like many, distributed developers), what should they then look out for and how should they handle the configuration management task? This research question is dealt with in section 3 of this paper.

Research question #2: Can we learn something - *why* are they successful and does it *transfer*? What are they reasons for their success? What are they doing better than traditional projects? Are they putting restrictions on the project and thus obtaining a simpler process? Can traditional projects transfer some (or all) of the successes from OSS? These questions are treated in section 4 where we analyse the reasons for their success and section 5 where we discuss what lessons transfer from OSS projects to projects involving conventional software (CS).

In the following section, we will establish a framework for treating the configuration management process in general. Using this framework, we will describe, analyse and discuss similarities and differences between OSS and conventional projects. The section will, furthermore, allow people with no specific knowledge of configuration management to follow the remaining sections of the paper.

In the last section of this paper we draw our conclusions.

2. Traditional Configuration Management

Configuration Management is a discipline within software engineering with the aim to control and manage projects and to help developers synchronize their work with each other. Configuration management is part of the entire development life cycle and is also a very broad area with respect to the means of how to achieve its goals, which are obtained by defining methods and processes to obey, making plans to follow and by using configuration management tools that help developers and project leaders with their daily work. One definition of configuration management is:

Configuration management is the controlled way of leading and managing the development of and changes to combined systems and products during their entire life cycle.

There are, however, many definitions, all with a different focus. One reason for the differing definitions is that configuration management has two target groups with rather different needs: management and developers.

From a *management perspective* [Berlack,1992], [Leon,2000], configuration management directs and controls the development of a product by the identification of the product components and control of their continuous changes. The goal is to document the composition and status of a defined product and its components, as well as to publish this such that the correct working basis is being used and that the right product composition is being made. One example of a definition supporting this discipline is ISO 10 007 meaning that the major goal within configuration management is "to document and provide full visibility of the product's present configuration and on the status of achievement of its physical and functional requirements". This standard also states that configuration management consists of four activities, or areas of responsibility. These are (extracted from the standard):

?? *Configuration Identification*
Activities comprising determination of the product structure, selection of configuration items, documenting the configuration item's physical and functional characteristics including interfaces and subsequent changes, and allocating identification characters or numbers to the configuration items and their documents.

?? *Configuration Control*
Activities comprising the control of changes to a configuration item after formal establishment of its configuration documents. Control includes evaluation, co-ordination, approval or disapproval, and implementation of changes. Implementation of changes includes engineering changes and deviations, and waivers with impact on the configuration.

?? *Configuration Status Accounting*
Formalized recording and reporting of the established configuration documents, the status of proposed changes and the status of the implementation of approved changes. Status accounting should provide the information on all configurations and all deviations from the specified basic configurations. In this way the tracking of changes compared to the basic configuration is made possible.

?? *Configuration Audit*
Examination to determine whether a configuration item conforms to its configuration documents. Functional configuration audit: *a formal evaluation to verify that a configuration item has achieved the performance characteristics and functions defined in its configuration document.* Physical configuration audit: *a formal evaluation to verify the conformity between the actual produced configuration item and the configuration according to the configuration documents.*

From a *developer perspective* [Babich,1986], [Dart,2000], configuration management maintains the product's current components, stores their history, offers a stable development environment and co-ordinates simultaneous changes in the product. Configuration management includes both the product (configuration) and the working mode (methods) and the goal is to make a group of developers as efficient as possible in their common work with the product. From the developer's point of view, much of this work may be considerably facilitated by the use of suitable tools in the daily work. The definition by [Babich,1986] stresses the fact that it is often a group of

developers that together shall develop and support a system: "Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team". A list of the tool aspects we regard to be most relevant is:

- ?? *Version Control - the possibility to store different versions and variants of a document and to subsequently be able to retrieve and compare them.*
- ?? *Build Management - mechanisms for collecting all the source modules of a system and generating the system, and for keeping the generated files up to date, preferably without doing any unnecessary work.*
- ?? *Configuration Selection - functionality to choose the versions of different documents or modules that constitutes a complete and consistent system.*
- ?? *Workspace Management - developers often want to work transparently with the configurations without being bothered with versioning or seeing the changes of others working on the same configuration.*
- ?? *Concurrency Control - manages the simultaneous access by several users, i.e. concurrent development, either by preventing it or by supporting it. Helps synchronizing the work of the developers.*
- ?? *Change Management - a system supporting the management of the collection of change requests, the generation of error reports, firm change requests, implementation of those changes, documentation of the problem and the solution, and when it is available.*
- ?? *Release Management - the identification and organisation of all documents and assets incorporated in a release. The build manager is responsible for providing the packed product with the correct configuration and features.*

Version Control

The possibility to store, recreate and register the historical development of an item (document or source code) is a fundamental characteristic of a version control system. Every stable issue of an item's content is termed a version. The tool has to minimize the storage space needed to keep all versions of an item. It has to impose a structure on how versions can develop from each other. And, finally, it has to keep track of all information about the different versions. Each single item of a given system will undergo many changes during the development and maintenance of that system. The set of changes that transform one version of an item into a new version is called a delta and represents the difference between the two versions. In addition to the actual changes, we are also interested in keeping other useful information about the change such as who did it, when, for what reason. This information is called the log entry for a version. Together with the delta, the log entry constitutes one history step in the development history of an item.

Versions of an item may be organized in a number of different ways. The structure that a version control tool imposes on the development of history steps is called a version graph and is basically linear. One version follows the other and a new version is always created from the end of the line. For simple development needs this model is sufficient even though limited. However, it cannot support maintenance (i.e., further development) of older versions, it does not handle parallel development (for instance, additional development and maintenance going on in parallel) and thus variants of the same item cannot be represented in this model. To solve that problem most tools allow branches to be created from older versions and thus support a tree model. In this model several branches can exist in parallel to reflect either maintenance of older versions, parallel work or variants. Other models support acyclic graphs; in this way, a version may have two or more predecessors, for example, in order to express that a bug fix in an old version is merged with the currently developed version.

Build Management

Build management handles the problems created by following the good programming advice to divide large programs into modules. These modules have to be put back together and compiled in order to create a running system. The number of modules or items in a system is increased by the fact that we might want to keep them as small as possible to avoid sharing conflicts when many people have to work together.

To build a system from its modules we need both a description of the structure of the system and information about how to derive object code from the source code modules. In early tools both the description of the dependencies and the information about how to compile items was given in a system model, which was used to derive object code and to link it together. This automates the build process and avoids errors from human intervention. Furthermore, build tools implement a minimal build as they only recompile source modules that

are out-of-date with respect to existing object code. This can lead to huge savings in compile time for big systems, if only a small part of the system has been changed.

Further improvements have been made to build tools to remove the limitations of the early ones. The most severe problem with the original build tool (make [Feldman,1979]) is that it keeps no data of its own. To decide whether an object module is up-to-date or not, it uses time stamps from the file system. This can be very unsafe especially in distributed environments where there is no global clock. Furthermore, it does not remember the way a given object module was created. This means that source modules don't get recompiled if such things as compilation options are changed or another version of the compiler is used, and this may create several problems. Additionally, the successors take advantage of the fact that today's computers are networked and support for compilation in heterogeneous environments is common as is the support for parallel compilation.

Configuration Selection

The build management we just described has no notion of versions of modules. Each module that has to go into the system is assumed to exist in only one version and the task managed by the build tool is only to compile and link these modules as efficiently as possible. This works quite well for projects without version control - and in the case of working in a workspace that has all the needed modules locally.

If we add versions to the modules this has to be reflected in the system model. For each node in the graph describing the dependencies, we now have a version group instead of a single file. This is much the same situation we have when we look at the repository and want to check some (or all) modules out into our workspace. This means that the build tool must be extended with the ability to select from the repository. Unfortunately, not all queries to the repository are unambiguous. In the case of a human being checking something out, this is not a problem as he can refine the query to become unambiguous. However, a tool does not have that capacity. This means that we have to be very careful about how we write our selections. Most tools have opted for the rather unsafe solution to automatically (without user intervention) solve all ambiguities by choosing the latest version. However, once the selection has been made we are back to normal build management.

In all situations, it is desirable to ensure that there is a consistent selection and configuration, in terms of the inclusion of versions with connected modifications. A useful technique for the specification of a configuration supported by several systems is to offer a rule-based selection mechanism. A configuration is called a partially bound (sometimes "generic") configuration, if the exact versions that are included can vary in time. A configuration where all selections have been resolved is called a bound configuration and is particularly suitable for deliveries, as the versions of all files included are fixed and therefore it can be guaranteed that the system can be recreated. Certain bound configurations can form a baseline, i.e. are a basis for further development with formal change management. In the same way that the development of individual files can be considered to be a version history, so can a corresponding development of configurations. A facility for naming versions ("tagging") can be used to manage the selection of bound configurations in that all files are tagged with the same name, e.g. "Release 2.3". Consistent naming may also be used to represent logical changes, i.e. changes arising from a change request and result in the modification of several files.

Workspace Management

The different versions of the documents in a project are kept in a repository by the version control tool. Because these versions have to be immutable, developers cannot be allowed to work directly within this repository. They have to take out a copy of the document, modify it, and add the modified copy to the repository. The fact that developers copy out files to their own private area also means that they are able to work in isolation from other people's changes.

The workspace management must provide functionality to create a workspace from the repository. In the simple case this only consists in being able to copy out a single file, however, more often an entire bound configuration is copied out from the repository to the workspace. This means that the developer has all the necessary documents and modules for the system at his disposal locally. He does not have to decide what should be kept globally in the repository and what he needs locally for carrying out his changes. Furthermore, he is isolated from other people's changes to the repository - and other people are isolated from his changes. This means that he is in complete control of his world and knows exactly what has changed and why.

When the developer has finished carrying out his modifications, he needs to add the changed documents and modules to the repository. This operation in the simple case of a single file consists in adding it to the repository using functionality in the version control tool. However, when he has a complete bound configuration in his workspace there are probably some files that have remained unchanged and therefore do not need to be added to the repository. The workspace manager can discover which files have changed and make sure that all of these - and only these - are added to the repository.

Concurrency Control

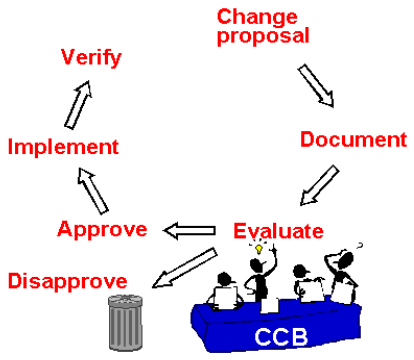
When we want to allow several developers to work on the same system at the same time, we must also provide mechanisms to synchronize their work. The problem that can occur is that more than one developer in his workspace makes a change to the same document or module. If this situation is not detected - or avoided - the second developer will overwrite the first developer's change when he adds his workspace to the repository.

Situations where changes are in conflict can be avoided by using a mechanism that locks a file in the repository when it is copied out to the workspace. This is, however, a serious obstacle to people working in parallel. The mechanism can be improved by locking only files that are copied out with the intention of being changed, but this on the other hand forces the developer to decide prematurely on what he wants to change. This mechanism is often used if only single files are copied out to the workspace or where it is very important to avoid conflicting changes.

In the case where we use bound configurations to create workspaces, we cannot use a locking mechanism for concurrency control, as this would allow only one developer as a time to work on anything related to a system. The approach followed in this case is to optimistically assume that no one will change the file - and if the assumption does not hold, to detect it and help the developers resolve the conflict. There are two ways to resolve conflicting changes, i.e. to merge the two changes. In one the responsibility is on the repository and if a merge cannot be made automatically, a branch is created to hold the second change added to the repository. The other places the responsibility on the second developer, who in case of an unsuccessful merge has to manually resolve the problem.

Change Management

The reasons for changes are multiple and complex and handles all changes in a system, both perfective, corrective and adaptive changes. Change management includes tools and processes, which support the organization and track the changes from the origin of the change to the approval of the actually implemented source code. Changes should only be carried out as the result of an approved change request to have full traceability.



Change Request (CR) Process

When a change is initiated, change requests are created to track the change until it is resolved and closed. The support organisation receives the change request, taking direct action and solving the problem if possible. If they cannot address the issue, the request is passed to the next instance. The change control board (CCB) analyses the

change request and decides which action is to be taken. If the change is approved, the change request is filed to the developer responsible for implementing the change. When the developer has performed the change its status becomes "implemented" and a test is performed. When the subsequent new release is to be built, the change control board decides which changes are to be included and the customer receives a patch including documentation of all the changes made.
Various tools are used to collect data during the process of tracking a change request. It is important to keep traceability between the change request and the actual implementation - in both directions. Change management data can also be used to provide valuable metrics about the progress of project execution. From this data it can be seen which changes have been introduced between two releases (a set of change requests). It is also possible to check the response time between the initiation of the change request and its implementation and acceptance.

Release Management

Software released to users must consist of documents and modules that have been approved as fit for their intended use. Usually this requires that they have been completely approved by the change control board. The release procedures must identify precisely which documents and modules should go into the release and in which versions. Usually there are two types of releases: internal and external. Internal releases (also called baselines) are used for development use to create a stable configuration of the system from which further changes can be made and integrated. External releases are intended for customers. Internal releases are made more often than external releases and are usually subject to much less rigor in their creation.

3. Managing Configurations in Open Source Projects

The description of OSS projects in this section is based on papers like [Raymond,2000], [Mockus et al,2000], [Feller et al,2000] and on interviews with key people from three OSS projects, KDE [KDE,2000], Mozilla [Mozilla,2000] and Linux [Linux,2000]. We use the framework from the previous section to describe how OSS projects handle different aspects of configuration management. We describe the general model and point out where a project differs from this model.

Version management

The tool CVS [Berliner,1990] is used for version control in most projects. It satisfies all basic requirements for version control. All versions are kept and deltas are used to minimize space consumption. It can handle branches, information about versions can be given in the change log, and it is possible to assign symbolic tags to versions. Usually write access to the CVS repository is generously granted such that several hundred developers can add new versions to it. It is also possible to submit regular patches that are then added to the repository by the moderators.

Linux, however, makes an exception, as they use no tool at all for version control. They simply put the code of each version in a separate directory, and apply contributions and patches to a "latest" directory. Contributions can only be applied to the repository by the moderators and there is no version history, as they violate the version control principle of immutability. When a new release is created, the latest is duplicated into a new release directory to conserve it even if development continues, which means that releases are immutable. There are only two branches in the Linux kernel development - one stable release branch and one development branch.

In the projects using CVS, versions are almost never used to revert to an older version. Instead versions are used as a history trail, describing how a file has developed by reading the log comments and by comparing versions using the diffing functionality. Most projects are targeted towards several platforms with great differences. They seem to handle the variants by either separating code into different files or directories, or by using conditional compilation. This way all variants can exist in the same branch. Changes apply either to the whole project, or platform specific code. In Linux architectural differences are handled in modules, so there is no need for variants.

Build Management

The fact that a local workspace containing all necessary files can be created, makes build management easier. The system model is included in these files and all projects use make or similar as their build tool. It is not time consuming to rebuild a project in the local workspace after a change, but the initial compilation is a complete build with no timesavings. This could have been obtained if object code had been included in the creation of the workspace. The drawback would have been a much slower creation of the workspace.

Configuration Selection

Almost always the latest version of all files is used to build a configuration, i.e. the selection is trivial. Since only one release is maintained (the latest) there is no need to return to other configurations. Only in the case where you want to install an older release are they retrieved. In the cases where one stable and one development release are maintained concurrently they are run as two separate projects. Branches are seldom used to provide concurrent work, i.e. no configuration selection is needed for that reason either. A new bound configuration is created for new releases by tagging the versions of the configuration.

In some projects the sheer amount of configurations possible because of variants poses a problem to the developers. Some changes break configurations, and feedback is needed to fix the problem. The obvious solution is to try to limit to a set of secure configurations, instead of all possible combinations.

Workspace Management

The version control tool used (CVS) gives optimal support for the special characteristics of OSS projects. It supports the concept of a project, which makes it a single operation to create a workspace, to synchronize your workspace with the repository (i.e. the changes of others), and to add your changes to the repository. However, the most significant feature of CVS is that it can operate in client-server mode, thus freeing the developer for thinking about file transport over the Internet. Furthermore, there is no need to be connected at all times. It is possible to create you workspace, disconnect and carry out all your changes off-line, and re-connect again only when you synchronize and add to the repository.

The cases where developers do not have write access to the CVS repository - and Linux - make an exception. In these cases a regular patch has to be created manually and sent to a moderator or co-ordinator that must then apply it to the repository.

Concurrency control

CVS uses optimistic concurrency control. The possibility to lock files is not used when they are copied out from the repository. CVS can detect when changes have been made in parallel to the same file and forces to last developer to add his change to resolve the conflict. In most cases it is sufficient to use the update operation to have the first changes automatically merged into the workspace of the second developer. He can then make sure that the previously made changes do not break his change. Only very rarely does he have to intervene manually because CVS cannot perform the merge automatically. Despite the rapid development and numerous developers with write access to the repository, update conflicts occur only very seldom. When they do happen, the contributors in question communicate directly to solve the problem. Mailing lists and newsgroups are used to provide awareness and reduce the risk of creating conflicts that will be hard to merge.

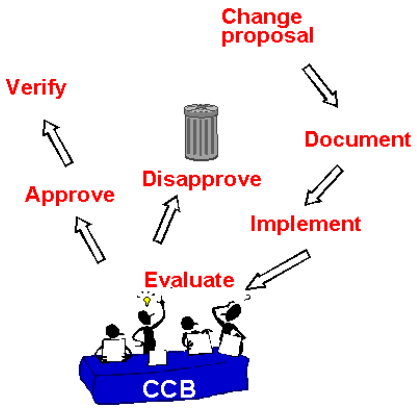
In Linux contributions are sequentialized as the moderator goes through the contributions he receives. If a later contribution conflicts with an earlier change, it is sent back and the contributor is asked to resubmit his patch. Only in cases where the conflict is easily resolved, the moderator carries out the necessary modifications.

Change management

The change management process is where OSS projects differ the most from CS projects. As described in section 2, traditional projects evaluate change proposals and only approved proposals are assigned to developers. In OSS the evaluation of change proposals is not explicit, if it is there at all. Anyone can propose a

change and most often changes are not even proposed before a change is submitted directly. Change proposals might be prioritised implicitly or explicitly, but an OSS project cannot assign tasks to developers - everyone works on what he chooses.

Two slightly different processes exist depending on whether contributions have to be sent to a moderator or if you can apply your changes directly to the repository through your write access. In both cases, however, it is the same overall process that is followed. An idea for a change is conceived, it is implemented and tested, it is submitted as a patch or applied directly on the repository, and finally the implementation (and sometimes the change idea itself) is evaluated through testing, review and discussion. The final evaluation may result in the patch being rejected by a moderator or a change to the repository being reverted by a co-ordinator. Usually write access to the repository is given only to trusted developers, so cases where a change to the repository is reverted are rare.



Change process for OSS

Linux, which is the prime example of an OSS project with moderators, gets patches submitted which are then worked through by the moderator. Patches are reviewed in multiple steps before testing, and only then inserted in the repository. Contributions that are found ill-designed or have not so sound ideas are rejected at reading time. If the idea is good but the code is bad, the contribution usually undergoes a few iterations of review before testing. If a contribution is rejected, there is sometimes feedback to the author.

Some projects, like Mozilla, have a mixed approach of module owners and direct write access to developers. The modules owner has the right to reject patches. In most cases, any of the developers with write access to the repository can make changes in most places of the code. There is no fixed policy; rather each module owner sets the contribution and change policy for his module. Sometimes, the module owners can pose a bottleneck to the processing of contributions. In project that work exclusively through co-ordination only, it seems like most changes are accepted immediately, and very few, if any, are rejected. The few patches that are received are handled by the co-ordinators.

Most change management problems seem to be caught at the review stage. Contributions are often tested via code reviewing and special run time tests. However, formal testing is not always used. Sometimes when a change has been made, developers simply use the new code. Developers that have submitted many good patches are more trusted, and their contributions make their way into the repository quicker. Accepted contributions show up immediately in the repository.

Even though wish lists and lists of bugs are kept, bugs and change proposals seem to be fixed in a somewhat arbitrary fashion. Changes are kept track of using detailed lists, in order for willing users to test new features. Mail and newsgroups are used to communicate wish lists, bugs, and changes and to discuss the general development of the project.

Release Management

None of the OSS projects release software in the traditional sense. Releases are all what we called internal releases in section 2 and people either have to do the rest of the job themselves or rely on a commercial

company wrapping an internal release up and turning it into a software packet. As a consequence, none of the OSS projects use fixed release dates and labelling and release is mostly arbitrary. There is a process, though, for internal releases. When an internal release is getting nearer, the development branch enters a freeze stage. Initially, the soft freeze stage means that new features, which break compatibility are discouraged, but not forbidden. They then move to a hard freeze, which in practice means that any contribution that will change an interface is forbidden. Only bug fixes are allowed.

The Mozilla project uses a time-based release schedule. This means that development proceeds until a certain date, and at that date, a release is labelled (called milestones in Mozilla terms). The milestone is then used to see what has been achieved. Consequently, features and achievements are not planned into milestones; the milestones only work as a feedback tool.

4. Important CM factors in OSS success

In this section we will analyse some of the CM factors in an OSS project. We have divided the analysis into three categories: tool support, process, and people. For each category we discuss what we have found to be the most important properties in an OSS project. The aim is to make these important properties explicit, explained, and possible to copy for new OSS projects. I.e. we want to make successful projects repeatable (cmp. CMM level 2).

4.1 Tools

As in all software projects a set of tools are used. In a typical OSS project CVS is used as the CM tool, together with standard tools as mail, web browser, and newsreaders. We will not discuss compilers and such tools here.

For the CM tool it is important that it has one single server against many clients, which means that servers not need to be synchronized, just client workspaces. The implementation of server synchronization often (always?) relies on branching or concurrent work sets, which are not used in OSS projects by other reasons, see below. Best, of course, is if the tool is free of cost, but if a commercial tool is used the server should have floating licenses, and it should be "free" to install the clients and to create a workspace that you can work on off-line.

Since all developers have to learn the tool by themselves it must be simple to use (and to administer) and it should not enforce any particular process. The CVS, which is often used, does, however, support (enforce) the long transaction CM model to co-ordinate concurrent changes. A tool that supports this model and makes it easy to update a workspace from the server, including the actual transportation of the files that need to be updated, is perfect for distributed development, especially when the clients are off-line most of the time. Many tools use one model when the client is on-line and another (secondary) model when off-line which makes it more complicated for the developer. Moreover the off-line mode will be treated as an exception rather than the primary work model it really is.

The clients must also exist on many platforms; at least if the application developed should work on many platforms.

All versions should, of course, be stored in the server and it should be easy to browse through the history of the project and specific files, and to see the changes made between two versions. This facility is used by developers to learn about a project and to see what has happened to it since last time they were active. It is also possible that some bad submissions sometimes reach the code base, which may lead to difficulties building the system. In these cases the tool should provide support to back (redraw) the entire transaction containing the bugs, i.e. not only some files that first have to be detected.

Finally, it must be easy to create bound configurations, baselines, which are made quite often in most OSS projects.

Another very important property of the set of tools used is awareness. If this is not entirely supported by the CM tool itself, it must be provided by other tools, e.g. using the web, mail, or news.

4.2 Process

It is important that the process is simple and easy to follow. The personal return of investment of following the process is important! A too rigid process may increase the personal investment without increasing the return to the developer him/herself. It is often better to encourage a correct behaviour by providing a better personal return of investment than to enforce some process due to management requirements. A good example is the long transaction model, which encourages frequent commits leading to less merge conflicts and increased awareness. If you do not follow the process of long transactions you yourself are punished by having to do a more complicated merge. Frequent commits also mean short iterations, which generally seems to be a good strategy [Beck,1999]. For example, it is easy to make baselines/releases, since there are few long projects going on that must be waited for.

However, some projects also enforce a special code style, e.g. some naming rules, indentation, etc. Not "sound ideas" may be rejected, even though they work technically. The reason to add this complexity to the process is to make the code more easy to read and understand, thus to increase awareness, which is very important. Despite a lot of discussions via mail and news, the code is still the most important. Nicely written code and understandable commit comments, makes collective ownership work. Not only the creator of a piece of code can test and modify it, but everyone that are interested in its functionality.

Open (in OSS) means it is easy for all developers to identify the weakest link in the process, which puts social pressure on each developer to follow the process and guidelines provided for the project, e.g. to write understandable comments. From this, the risk to pollute the common repository is reduced.

Also the change management process must be appropriate for the task. An effective way to reduce the complexity of change management is to not maintain old releases. Instead all development, both bug fixes and new requirements, are committed directly to main. Otherwise several branches have to be maintained which now can be avoided. Change management is, however, probably the weakest part of the OSS process, and it is possible that a stronger support than current 'wish lists' had been cost (time) effective.

Awareness through discussions is also very central to the process. Usually you rely on both formal and informal communication, but OSS projects does not have face to face meetings, so even informal communication has to be electronic (and therefore seen/listened to by the whole group - as is the formal communication as long as it takes place in the newsgroup). Examples of formal communication are CVS comments (log), wish lists, bug reports, release documentation, and comments in the code.

The combination of self assigned tasks, a light process, stimulating discussions, direct communication, and group awareness is important to keep skilled and motivated developers. They often find it fun and stimulating to discuss technical solutions with other skilled developers, especially when it develops fast and gives a lot of personal return of investment.

4.3 People

The most important people in an OSS project are the moderators/coordinators. He/she protects the code base. Bad developers may slow the progress down, but cannot destroy the code. Bad moderators can allow the code to be corrupted gradually. Moderators should NOT write/contribute code themselves or try to improve bad contributions - otherwise they could soon end up being bottlenecks. Such bottleneck does not only delay the awareness and usability of the application developed, it also breaks some of the advantages of the long transaction model. One important property of the model is that the developer always commits a tested (and working) configuration, if needed after several iterations of 'update-merge-test' within the private workspace before a successful commit. The developer can, however, only update and test from the common repository and can not access the submissions not yet processed by the moderator. If these contain modifications not consistent with the new submission the moderator sends back the submission and the developer has to update and re-send it.

For all developers it is important to care about their reputation of being 'good developers'. If this social pressure works as a motivation factor no one wants to submit bad patches, which ensures good quality. Also the moderator is under a similar pressure, since it is always possible to clone the project with a new, more popular, moderator. A dialogue between the developers and the moderator is thus good for both parts.

All involved in OSS should also like to discuss their work and want to share their knowledge to other people. Even though most development is done off-line by single developers, it is really a teamwork that needs a lot of communication.

5. Transfer from OSS/CM to CS/CM and vice versa

In previous section we highlighted some important CM factors of OSS development with the aim to make them explicit to be able to repeat successful OSS projects. Some of these factors can be used also within CS (conventional software) projects, at least after some adjustments, others cannot. How CM is managed within OSS is, however, no 'silver bullet' solving all kinds of problems and there may (even) be some lessons learned from CS to OSS as well. In this section we will focus on the transfer of knowledge and best practices between OSS and CS, mostly from OSS to CS but also vice versa.

A general advice is to *avoid unnecessary branches*. In CS branches are used primarily for three reasons: (1) shorter parallel sessions of individual developers, (2) larger projects, and (3) maintenance of old releases. Most CS projects use branches together with the CM model checkout/checkin with locking and/or the composition model [Feiler,1991]. If instead the *long transaction model* is used the need for branches to synchronize shorter parallel development is reduced since each workspace takes the role of the temporary branch in which development can be made in isolation and where update and merge can be performed. Thus branches due to (1) and can be avoided. In some cases larger projects must exist and branches may then be a good strategy. In many cases, however, it is possible to *divide the large project to smaller increments* which then can be implemented following the general rule of short iterations, each followed by build and test (successfully proposed in both Daily build [McConnell,1996] and XP [Beck,1999]). Such a strategy also makes it easier to create frequent baselines, and we increase the group awareness. (No big bang integration needed towards release since all development is made in short increments. Even so, *'feature freeze*' is used near major releases to reduce the number of bugs in newly developed code.) In this way we do not need to branch that often due to (2) either. To maintain many releases increase the complexity of change management, but may unfortunately be needed due to market requirements. I.e. reason (3) still remains for these cases.

It is important to *protect your code base*. Traditional CS puts a lot of effort to classify and to give priority to change requests and to decide whether they should be implemented or not, but does not protect the code base for bad implementations (as reflected in the figures depicting the change processes). A role similar to a *moderator or co-ordinator* may be a good idea.

Use *one CM model that works for clients both on-line and off-line*, especially if the development is geographically distributed or there are other reasons for developers to (also) work off-line. If the model only works on-line, or hardly work off-line, there is a large risk that developers 'cheat' and not follow the model/process at all, which often is much worse than having a more light-weight process that is followed. It is also important that the tool really support the model used.

Let *developers communicate directly with each other*. If all communication goes through a deep hierarchy of management it will be too slow and not that effective/stimulating. In a traditional project 50% of a developer's time is spent communicating, 30% working alone, and 20% unproductive "work". I.e. it is very important to support communication and *awareness.*

Practice *collective ownership*. It is important though that communication and awareness is supported to let the developers together solve the synchronization needed to avoid complicated merge conflicts and misunderstandings. Do not mix up *modularisation* and *toolbox architecture* with ownership. It is important to have an architecture that allows concurrent work without creating merge conflicts and that makes it easy for a developer to add-on functionality (e.g. a driver to a specific hardware), but there is no need to have access restrictions on such modules.

Important to the success of OSS is self assigned tasks and the fact that the developers almost always also are users and testers of the system. This is unfortunately hard to copy to CS development, but if possible a project should strive towards a similar process. Instead of automatically assign each developer their tasks based on some document management system they could assign their own tasks from a set of tasks. If possible they could use the developed application. If this is not possible extensive testing should be performed, e.g. in the style of XP.

Most OSS projects lack the control and visibility of the 'requirements' and change requests implemented and the ones still on the wish list. In CS this is often managed by separate tools and considered one of the most important activities of CM. Most OSS projects had probably benefited from an updated wish list and a *better traceability between a change request (wish) and the actual change* made to the code.

In CS it is also important to set the correct priority on all requirements, depending on severity, how much it costs to implement, importance to different markets, etc. This is harder to do in OSS since all tasks are self assigned, i.e. each developer makes their own priorities independent of how other users/developers set their priority.

6. Conclusions

In this paper, we have examined the configuration management (CM) aspects of OSS projects with thee goals in mind: (1) to make the CM process within an OSS project explicit and understood and thus make it possible to copy its properties correctly when starting new OSS projects, (2) to analyse the reasons for the apparent success of the OSS CM process and (3) to discuss which parts of this process could be transferred with success to conventional software development and how.

We note that OSS development is can be considered as 'Individual development in groups'. In OSS, the developer has a direct personal interest or gain from his contributions to the project, which normally is not the case in traditional (commercial) development projects. As a consequence of this there is a minimum of administrative overhead and management. Instead all steps in the process (that exists and actually is followed) have a clear return of investment for the developer himself.

The configuration management process in OSS has the following characteristics that you have to pay attention to if you want to manage an OSS project of your own:

- ?? there is a very high degree of awareness and developers communicate directly with each other.
- ?? many clients are working distributed and off-line against one single server. Workspaces are coordinated following the long transactions CM model.
- ?? the task is simplified through not maintaining old releases and thus avoids branching.
- ?? the process encourages many small and quick increments on the main line. This results in early testing, high awareness, many baselines that are usable - the goal is to use the product.
- ?? the moderator/coordinator protects the code base from the entry of bad contributions. This is an important role and he has to be able and quick to reduce the risk of becoming a bottleneck in the change process.

It may be possible to transfer the lessons learned from the following parts of the OSS configuration management process to conventional development projects:

- ?? OSS development is most similar to the maintenance phase of CS development. Before adopting the OSS process, a toolbox architecture should be designed.
- ?? practice collective ownership and let the developers communicate directly with each other.
- ?? protect the code base from bad implementation, not only from implementing lower priority change requests.
- ?? use the long transaction model and work directly on the main development line.
- ?? develop small increments and commit often to increase awareness and enable early testing.
- ?? consider the knowledge of your developers as an intellectual capital that can create better awareness if shared.

Some of these lessons have already been followed, as they are in line with some of the recommendations in [Wingerd et al,1998]. In our opinion, the CM process in OSS shows great potential. Especially if augmented with another change control board to eliminate also upstream defects earlier, which is one of the four major problems that [McConnell,1999] sees with the overall OSS process.

It is outside the scope of this article to attempt an evaluation of how efficient OSS development really is. Future work could, however, be to take a look at the following topics specifically related to the configuration management process: how long does it take to respond to a bug report, how much double/useless work is there

in submissions, how many contributions are lost because of bad change management, can the OSS model (for CM) be used for the start-up phase too?

References

[Berlack,1992]: H. Ronald Berlack: Software Configuration Management, John Wiley & Sons, 1992.

[Leon,2000]: Alexis Leon. A Guide to Software Configuration Management. Artech House Computer Library, 2000.

[Babich,1986]: Wayne A. Babich: Software Configuration Management - Coordination for Team Productivity, Addison-Wesley Publishing Company, 1986.

[Dart,2000]: Susan Dart: Configuration Management - The Missing Link in Web Engineering, Artech House, 2000.

[Feldman,1979]: Stuart I. Feldman: Make - A program for Maintaining Computer Programs, Software - Practice and Experience, April 1979.

[Raymond,2000]: Eric S. Raymond. The Cathedral and the Bazaar, http://www.tuxedo.org/~esr/writings/cathedral-bazaar/

[Mockus et al,2000]: Audris Mockus, Roy T. Fielding, James Herbsleb: A Case Study of Open Source Software Development: The Apache Server, in Proceedings of the International Conference on Software Engineering, Limerick, Ireland, June 4-11, 2000.

[Feller et al,2000]: Joseph Feller and Brian Fitzgerald. A Framework Analysis of the Open Source Development Paradigm, 2000.

[KDE,2000]: The K Desktop Environment. 2000. http://www.kde.org.

[Mozilla,2000]: The Mozilla Project. 2000. http://mozilla.org

[Linux,2000]: The Linux Project. 2000. http://www.linuxworld.com.

[Berliner,1990]: Brian Berliner: CVSII - parallelizing software development, in Proceedings of USENIX Winter 1990, Washington D.C.

[Beck,1999]: Kent Beck. Extreme Programming explained: embrace change. Addison-Wesley. 1999.

[Feiler,1991]: Peter H. Feiler: Configuration Management Models in Commercial Environments, CMU/SEI-91-TR-7, Carnegie-Mellon University/Software Engineering Institute, March 1991.

[McConnell,1996]: Steve McConnell: Daily Build and Smoke Test, IEEE Software, July 1996.

[Wingerd et al,1998]: Laura Wingerd, Christopher Seiwald: High-Level Best Practices in Software Configuration Management, in Proceedings of the Eight Symposium on System Configuration Management, Brussels, Belgium, July 20-21, 1998 (LNCS 1439).

[McConnell,1999]: Steve McConnell: Open Source Methodology - Ready for Prime Time?, IEEE Software, July/August 1999.