Open Source Version Control

Thomas Keller

HTWK Leipzig

University Of Applied Sciences

Department Computer Science, Mathematics & Natural Sciences

Bachelor Thesis

in Media Computer Science

Open Source Version Control

Thomas Keller

Tutor:	Prof. Dr. Michael Frank
Co-tutor:	Prof. Dr. Klaus Hering
Date of Completion:	March 20th, 2006

I affirm that I have created this Bachelor Thesis independently and only with the use of the documented references.

Leipzig, in March 2006 .....

I would like to thank

My beloved girl (soon wife) Marlen,

who always supported me during the process of writing this work,

My tutor, Mr Prof Dr Michael Frank,

who always had a sympathetic ear for my concerns,

My good friend Chad Connolly from Delaware/USA,

who helped me a great deal to find and fix spelling issues in this work,

And finally my cousin Martin Fischer,

another guy who has helped me a lot with his positive criticism.

Thank you all!

Table of Contents

Preface

Source Code Management (*SCM*), also often referred to as *Software Configuration Management*, is the most important technique for software companies to manage their intellectual property. One of the key technologies in *SCM* is *Version Control* (*VC*). It offers a variety of possibilities to support the development cycle - e.g., by allowing the user to track and revert changes, save history information, manage multi-user access to the source code, and more.

There are many products available which offer *Version Control*. These come with tight integration in a certain development environment (e.g. *Visual SourceSafe*), or are advertised as very easy to set up, administrate and access (*Perforce*). But is it really necessary to spend money on a commercial solution? In small and medium-sized companies, there is likely no money for software which supports the development cycle; the price pressure for software engineering is high enough anyway. And, if a product's life-cycle ends, or if more than the initial five concurrent users need to get access to *Version Control*, additional licenses may need to be bought.

One approach to overcoming the vicious circle of licensing is to use *Open Source Software* (*OSS*). While many people believe that „something which does not cost any money cannot be good“ when it comes to Open Source products, this is actually wrong. For example, official statistics about web server usage shows that the Open Source web server *Apache* runs on about 70% of all webservers around the world[1]. Even business models depend on Open Source products, such as *MySQL AB*[2] *(*the Swedish database developer) or *Trolltech (*the company behind the *Qt Window Toolkit*[3]*)* demonstrate.

This thesis is dedicated to both *Open Source* and *Version Control* and is thus organized into two main sections:

The first section deals with *Open Source Software* in general and tries to give an insightful view as to how *Open Source Software* is driven, how it is developed and examines the differences between conventional software (CS) development and Open Source development.

The second section deals with *Version Control*, ranging from history, concepts, tools, and ends with day-to-day usage tips.

Both sections should give the reader the theoretical and practical knowledge that should help them to benefit from *Open Source Software* in general and also from one of the free *SCM* systems which are reviewed in this work.

1 http://news.netcraft.com/archives/web_server_survey.html
2 http://www.mysql.com
3 http://www.trolltech.com

Vorwort

Softwarekonfigurationsmanagement (*Software Configuration Management, SCM*) ist ohne Frage die wichtigste Technik für Software-Unternehmen, ihr intellektuelles Gut in Form von Bits und Bytes zu verwalten. Eine der Kerntechniken von *SCM* ist Versionskontrolle (*Version Control, VC*), welche eine Reihe von Möglichkeiten offenbart, den Entwicklungsprozess aktiv zu unterstützen, etwa indem (frühere) Änderungen verfolgt und eingesehen werden, mehrere Benutzer gleichzeitig auf die Codebasis selbst über Netzwerkgrenzen hinweg Zugriff erlangen können und vieles mehr.

Auf dem Markt gibt es eine Vielzahl von Produkten, die Techniken zur Versionskontrolle unterstützen. Diese sind dann entweder eng in eine bestimmte Entwicklungsumgebung integriert (zum Beispiel *Visual SourceSafe*), oder werden als besonders einfach aufsetzbar und administrierbar dargestellt (*Perforce*). Aber ist es wirklich notwendig, in eine kommerzielle Lösung Geld zu investieren, oder gibt es eventuell Alternativen? In kleineren und mittelständischen Unternehmen (KMUs) ist oft kein Geld für teure Lizenzen von Softwareprodukten, die lediglich den Entwicklungsprozess unterstützen sollen; der Preisdruck bei der Projektierung und Softwareentwicklung ist ohnehin schon hoch genug. Und selbst wenn investiert wird, und der Lebenszyklus des Produkts abläuft oder die Anzahl der erlaubten, gleichzeitigen Zugriffe auf das Produkt überschritten wird, ist der Unternehmer oft gezwungen, zusätzliche Lizenzen und unter Umständen sogar zusätzliche Hardware anzuschaffen, um die Entwicklung am Laufen zu halten.

Eine Möglichkeit, dem Teufelskreis der Relizensierung zu entkommen ist, *Open Source Software (OSS)* einzusetzen. Obwohl viele Menschen glauben, "etwas, das nichts kostet, kann nichts taugen", ist genau das Gegenteil der Fall. Es gibt für etliche Open Source Produkte mittlerweile Firmen, die kommerzielle Unterstützung anbieten, sodass auch dem professionellen Einsatz im Unternehmen nichts entgegensteht. Im Endeffekt wird dann zwar der Einsatz von Open Source nicht komplett kostenlos, aber im direkten Vergleich schlagen die Open-Source-Pendanten mancher Produkte meist die kommerziellen Lösungen.

Die Arbeit ist sowohl dem Thema *Open Source Software*, als auch dem Thema Versionskontrolle gewidmet und unterteilt sich daher in diese beiden Hauptsektionen.

Beide Sektionen sollten dem Leser das nötige theoretische und praktische Wissen vermitteln, um von *Open Source Software* zum einen, und zum anderen von einem der hier vorgestellten, freien Versionskontrollsysteme zu profitieren. Eventuell erlangt der Leser sogar eine neue Sicht auf freie Software und versucht die eigenen Bestrebungen bzw. die seiner Firma daran auszurichten.

Document Conventions

All technical terms are written recursive and marked with a superscript [G] - they are explained in the glossary in chapter .

Quotes are written in italics and wrapped with double quotes, like this: *"This is an example quote"*. References to quotes are either given before the quote by a bibliographic reference [ReferenceYYYY] or after the quote by a bibliographic reference or a footnote.

1 About Open Source Software

The term "Open Source" was introduced during the foundation of the *Open Source Initiative*[4] in 1998. At that time, it was only a different term for what people of the *Free Software Foundation*[5] were doing since the mid-80's: promoting free software. "Free" in the sense that an open license, not commercial, was applied to the software, and that everyone who wanted to use the software or make it better could just do that, because he or she had access to the source-code and the right to modify it.

But "Free" does not necessarily mean that the software is given away for free. Open Source Licenses allow the selling of *Open Source Software*, as long as the distribution always contains an easy way of accessing the sources. A good example of this is *GNU / Linux* (see chapter 1.1), an alternative operating system. Linux distributors like *Mandriva*[6] (formerly *Mandrake*) or *RedHat*[7] are basing their products almost entirely on Free Software, but sell it in packaged, tested and reviewed form for professional users and companies. They also offer services like training programs or support; sometimes they even enrich their products with special *proprietary software*[G].

Since all of these companies rely on Open Source, all of them are actively supporting the community. It typically happens that hired software engineers develop components in a community process, and if the specific component is deemed to be *matured*[G] and stable, it will be used in the commercial product later on. For RedHat, this community process happens in the *Fedora Linux Project*[8].

The concept behind Open Source does not just apply to software alone today. Generally, whenever license restrictions apply without consideration to the fact that a product or service could be improved upon or made available for everyone, people look for "free" or for "open" alternatives. One popular example is the *Wikipedia Project*[9] which provides a free encyclopedia. Another company put their recipe for a drink called *OpenCola*[10] under the *GNU Public License* and made it publicly available for everyone on their website.

According to [Perens2006], Free "Open" Software is characterized by ten attributes, which were introduced by *Bruce Perens* (one of the founders of the *Open Source Initiative*) in 1997. Primarily, these attributes have been specific to the *Debian Linux Project*[11], but Perens has removed these specific parts later on to define the *Open Source Definition*. The most important attributes are:

4 htp://www.opensource.org
5 http://www.fsf.org
6 http://wwwnew.mandriva.com/
7 http://www.redhat.com/
8 http://fedora.redhat.com/
9 http://www.wikipedia.org
10 http://www.colawp.com/colas/400/cola467_recipe.html
11 http://www.us.debian.org

Free Redistribution

"The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale."

Source Code

"The program must include source code, and must allow distribution in source code as well as compiled[G] form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost–preferably, downloading via the Internet without charge. [...]"

Derived Works

"The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software."

No Discrimination Against Persons or Groups or Against Fields of Endeavor

"The license must not discriminate against any person or group of persons. [...] The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research."

Distribution of License

"The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties." This means that rights which are granted in an Open Source license may not be overridden by additional licenses or *Non-Disclosure Agreements*[G] (*NDAs*).

License Must Not Be Specific to a Product or Restrict Other Software

"The rights attached to the program must not depend on the program's being part of a particular software distribution. [...]" otherwise the program alone cannot be distributed on its own. Also, *"The license must not place restrictions on other software that is distributed along with the licensed software. [...]"* which makes it possible e.g. to distribute *Open Source Software* and other licensed software on the same medium.

License Must Be Technology-Neutral

Finally, *"No provision of the license may be predicated on any individual technology or style of interface.",* which is directed against so-called "click-wrap" license agreements. These kind of

licenses are often displayed, for example, on websites before you are directed to your target. Since no such kind of interface is possible in many forms of distribution (e.g. FTP / CD-ROM distribution), this is prohibited.

More on specific licenses, which acknowledge these points, follows in chapter 1.2.

1.1 The Origins of and the Drive Behind Open Source

The roots of *Free Software* go back to the 1960s and 1970s, when *Richard Stallman* was working at the MIT Artificial Intelligence (AI) Lab [King1999]. At this time there existed a community of "Hackers" which shared the software they developed between each other, and Stallman was part of this community. The community was close-knit and small [Rasch2000], and it was a matter of course that if one made an improvement to a piece of software, he / she shared this improvement with the others. Sharing was one of the fundamentals for this community to work at all. Stallman later said: *"We didn't call our software 'free software', because that term did not yet exist; but that's what it was."* [King1999].

Illustration 1: Richard Matthew Stallmann - source: http://de.wikipedia.org/wiki/Bild:Richard_Matthew_Stallman.jpeg

The community broke up in the early 1980s when the MIT discontinued their *PDP-10 personal computer*, for which all the software had been written for. There was no way to port this software over to other systems, and as time went by, many of the original hackers of the AI lab had been hired by spin-offs of the MIT or other companies and worked on different projects. Stallman: *"We couldn't sustain ourselves. This was the hardcore of the free software hackers, and now it was gone."* [King1999].

The general fact that a program, proprietary designed and developed by a commercial company, could not be altered by the end user was not a big worry to the developers back in the day. They thought that they developed good software and made good money, so what could be wrong with that?

Well, for Richard Stallman it was an issue. Going over to the proprietary software world and signing the NDAs was not an option, as he felt more comfortable with his fellow hackers. Quitting

completely and leaving the computer science was no option either, so he asked himself: *"[...]was there a program or programs [that] I could write, so as to make a community possible again?"* [King1999]

Out of this insight Stallman created the *GNU Project*[12] in September 1983. The overall aim of the GNU Project was (and still is) to create a free, full-featured operating system which could be used and modified by any computer user like it was back in the "old days". The recursive acronym "GNU" stands for "GNU is Not Unix" in this context [GNU2006]. Unix was a proprietary operating system at the time, proven stable and widely adopted, and *GNU* should introduce a free, but compatible version of it. To be able to work independently from any licensing issues on the project, Stallman quit his job at the *MIT* in January 1985 and started to work on *GNU.* Also, in October of the same year, Stallman founded the *Free Software Foundation*. Many developers joined his effort, mostly volunteers, but some had been paid by companies to develop needed parts.

The project was very successful over the years and created many so-called *user space programs*[G]; the *GNU Compiler Collection (GCC)*[13], or *Emacs*[14] (a popular text editor) as a couple of examples. Only one thing was missing over the years and made only small improvements: the *kernel*[G], called *GNU Hurd.* The work on Hurd started in 1990 and up until today (2006) no stable version of the kernel was published.

Illustration 2: Linus Torvalds, 2002 - source: http://en.wikipedia.org/wiki/Image:Linus_Torvalds.jpeg

In 1991 *Linus Torvalds* began his work on the *Linux Kernel*[15], and publicly announced this on an Internet newsgroup about the *Minix* (an operating system derived from Unix and developed by Andrew Tanenbaum): *"I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones..."* [Linux2006].

His approach differed in some technical ways in comparison to *GNU Hurd*, and also, *Linux* was not supposed to be Open Source right from the start. Not until 1992 was the *GNU GPLv2* applied to the Kernel and could have been called "Free Software" from then on.

Today the word "Linux" is resounded throughout the land as a free operating system, but to be

12 http://www.gnu.org
13 http://gcc.gnu.org
14 http://www.gnu.org/software/emacs/
15 http://kernel.org

correct, "Linux" is only the kernel of it. Many other programs and tools from the *GNU* project work together and make the operating system complete, so its more correct to speak of *GNU/Linux* when talking about it.

Having the terms "Open Source" and "Free Software" heard in many places, it might be interesting if there is actually a distinction between these two? This is a big controversy throughout the community. The OS movement's main goal seems to be "higher-quality software", while the FSF's main goal is that the software is "free", as in "freedom of speech". Stallman put it like this:

> *"Free Software is a political action which places the principle of freedom above everything else. It is completely different from Open Source, which is a purely practical way of getting software written, and doesn't raise the point that users deserve freedom. Open Source has no ideology."* [King1999]

Still, the term "Open Source" is more common today than "Free Software", and one cannot say that an Open Source developer or user is free of any political attitude. It is probably best to say that the Open Source movement has its roots in the Free Software Foundation and to acknowledge that Open Source is about more than getting a piece of software for free.

Proprietary, Closed-Source Software will always be the enemy of the real hardliners of the *FOSS,* the *Free and Open Source Software,* while others try to join the best of these two worlds.

1.2 Open Source Licenses

The Open Source community is big and diverse, and it is easy to say the same about the amount of Open Source licenses available. For a non-lawyer (and most developers have no judicial knowledge), it is hard to oversee the jungle of different licenses, and it is even more complicated to find a license which fits all the needs.

Still, the biggest problem the community has yet to face is the incompatibility between certain licenses. If a developer choses to include a *library*[G] which is, for example, licensed under the terms of the *Mozilla Public License* (*MPL*), and his / her project itself is licensed under the terms of the *GNU General Public License* (*GNU GPL*), he / she is not legally allowed to do so, because the *MPL* restricts some of the rights the *GPL* offers.

Furthermore, it seems that many licenses have been primarily created to credit a certain company, but do not bring any new aspects to the original license. A good example for this is the *Sun Public License* (*SPL*), which is used by *Sun Microsystems* for their Open Source products. The *SPL* is

nothing else than a slightly adapted "version" of the *Mozilla Public License* (*MPL*), used by the Mozilla Foundation and formerly set up by *Netscape,* in which the main difference is that the company credit "Netscape" has been replaced by "Sun". The website of the Open Source Initiative lists currently more than 50 licenses which have been approved by them and thus can be named "*OSI* certified", since they fulfill all ten aspects of the Open Source Definition[16]. Obviously, there have been announced plans to reduce this number by the OSI in early 2005, nothing has happened until today[17].

Most Open Source licenses have in common, that they deny any warranty to the end user; the software is provided "as is". This is an important fact if a company decides to use *Open Source Software* for critical processes. If the software fails and creates money loss or other damage, there is no one to blame, unless a specific agreement has been made with the distributor of the software, which may include additional warranty clauses (this is not forbidden or covered by most Open Source licenses). After that has been said, one can understand that *Free / Open Source Software* does not necessarily means that the software is gratis or runs free of charge. Still, *FOSS* is supposed to have a lower *TCO*[G] value than Proprietary Software[18].

The next chapters give a short overview over the most important Open Source licenses, their strengths and weaknesses, and the compatibility between each other. They are based on the comments of Jonas Kölker [Licenses2006] on different software licenses and explained further where applicable.

What kind of license should be chosen when a new project is started? This is hard to answer. One should use the one that fits the needs of the project perfectly, but still is somehow accepted and used in the community.

1.2.1 GNU General Public License (GNU GPL)

The *GNU GPL*, or *GPL* for short, is the most popular Open Source license and also the most used one[19]. The license exists in two versions; a third, renewed version is currently discussed in the community, and version 2, the most recent one as of today, is dated on June 1991. *Richard Stallman* is the original author of version 1 which was published in 1989 primarily for the use in the *GNU* project. For version 2 and version 3, Stallman worked and still works together with *Eben Moglen,*

16 As of 02/02/06, 58 licenses were listed: http://www.opensource.org/licenses/index.php
17 http://www.eweek.com/article2/0,1895,1858314,00.asp
18 A study by Soreon Research from 2004 shows that the intial costs of *Open Source Software* is about 25% lower compared to Proprietary Software, even if a certain amount of risk-costs are applied: http://www.soreon.de/site1/index.php/german/soreon_studien/kassensturz_open_source_und_propriet_re_software_im_vergleich_update_2004_95_seiten_31_abbildungen_und_tabellen/kassensturz_open_source_und_propriet_re_software_im_vergleich_update_2004__3
19 Based on SourceForge statistics, more than 50% of all listed projects are licensed under the terms of the GNU GPL, http://sourceforge.net/softwaremap/trove_list.php?form_cat=14, last viewed 02/04/06

Professor for law and history at the Columbia University.

Illustration 3: Eben Moglen - source: http://en.wikipedia.org/wiki/Image:Eben_Moglen.jpeg

Both, version 1 and version 2, are "forward compatible". This means that any program can be licensed under the terms of a newer version of the license as long as the following license header exists (here for version 2):

> *"This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version."*[20]

The *GPL* has a strong *Copyleft*, *"[...] a general method for making a program or other work free, and requiring all modified and extended versions of the program to be free as well."*[21]

A *Copyleft* license has to not only ensure that there are several "freedoms" applicable on the software, but especially that these freedoms cannot be removed from the software through redistribution (see "Derived Works" in the *Open Source Definition* in chapter 1).

Critics of the *GPL* often state its "viral" or "pervasive" attributes through to this Copyleft, since every software which includes *GPL* code has to be released under the *GPL* itself. However, this is the only way to ensure that code which was once Open Source cannot become Closed Source.

A popular method to circumvent the pervasion brought by the *GPL* is to release a software under two licenses, a commercial one which is applied for a fee and a free one like the *GPL*. This is then called *Dual Licensing*; derived works do not have to be put under *GPL* and any *Intellectual Property*[G] (*IP*) is safe. Examples for dual licensed software products are the products from *MySQL AB* and *Trolltech* mentioned in the preface.

1.2.2 GNU Lesser General Public License (GNU LGPL)

The first version of the *GNU LGPL* was introduced together with the *GNU GPL* version 2 by Stallman and Moglen in 1991. It was originally named *GNU Library General Public License,* because it primarily allowed (and allows) the inclusion of proprietary code in the form of software libraries. The license puts the same restrictions (such as the *GPL*) on the program itself, but does not apply these restrictions to other software which merely links with the program, and thus does not have such a strong *Copyleft* as the *GPL*.

20 http://www.gnu.org/licenses/gpl.html, section *"How to Apply These Terms to Your New Programs"*
21 http://www.gnu.org/copyleft/copyleft.html

With version 2.1, it got its new name: *GNU Lesser General Public License*[22]. The *GNU LGPL* is fully compatible with the *GNU GPL*, since it contains a passage[23] which allows an author of derived work to put the software under the *GPL*. This is irreversible, though.

1.2.3 BSD License

The *Berkeley Software Distribution License* is one of the most widely used licenses for free software. It is a very free license which allows the mixture between open and closed source, similar to the *GNU LGPL*, but without restrictions on the main program. Therefore, it puts software released under this license relatively closer to *Public Domain*[G] than the *GPL*.

Since 1999, a modified version of the *BSD License* and the *GPL* are compatible, thus one can mix code released under either license. The license text itself is *Public Domain* as well, and can be adapted and credited for own projects.[24]

1.2.4 Apache Software License

The *Apache Software License* is a free software license, which is primarily used by the *Apache Software Foundation* [25] (*ASF*) for all of their software products. Software released under this license can be used in Commercial and *Open Source Software* development, though it is not recommended to use it for new Open Source projects because of the missing *Copyleft* restrictions.

Despite the fact that the compatibility has been improved with other Copyleft licenses over time, a certain part makes this license still incompatible with the *GNU GPL* in the eyes of the *GNU* project:

> *"[The license] has certain patent termination cases that the GPL does not require. (We don't think those patent termination cases are inherently a bad idea, but nonetheless they are incompatible with the GNU GPL.)"*[26]

The *ASF* has their own determination about the compatibility of both licenses, though.[27]

22 In Feburary 1999, Richard Stallman wrote the article entitled "Why you shouldn't use the Library GPL for your next library", where he explains amongst other things why this name change happened: http://www.gnu.org/licenses/why-not-lgpl.html

23 http://www.gnu.org/copyleft/lgpl.html: *"3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library.[...]"*

24 http://en.wikipedia.org/wiki/BSD_License

25 http://apache.org

26 http://www.gnu.org/licenses/license-list.html

27 http://www.apache.org/licenses/GPL-compatibility.html

1.2.5 Mozilla Public License (MPL)

The *Mozilla Public License* is, like the Apache Software License, a license without strong Copyleft. It has some restrictions in it which make it incompatible with the *GNU GPL*. The first version was created by the lawyer *Mitchell Barker* when she worked for *Netscape Communications Corporation*.

The license is used for most products of the *Mozilla Foundation*[28], which also published version 1.1 of the license. Today Baker is a member of the Board of Directors in the foundation.

1.2.6 Other Open Licenses not primarily created for Software Licensing

The overwhelming success of Free Software has made other people looking for alternatives in general content licensing. Most Source Code licenses seem to be not applicable for that, so there are a variety of other licenses for specific purposes, ranging from artistic works, to fonts and documentation. A closer look at these licenses is out of scope of this work; an annotated list can be found in [Licenses2006].

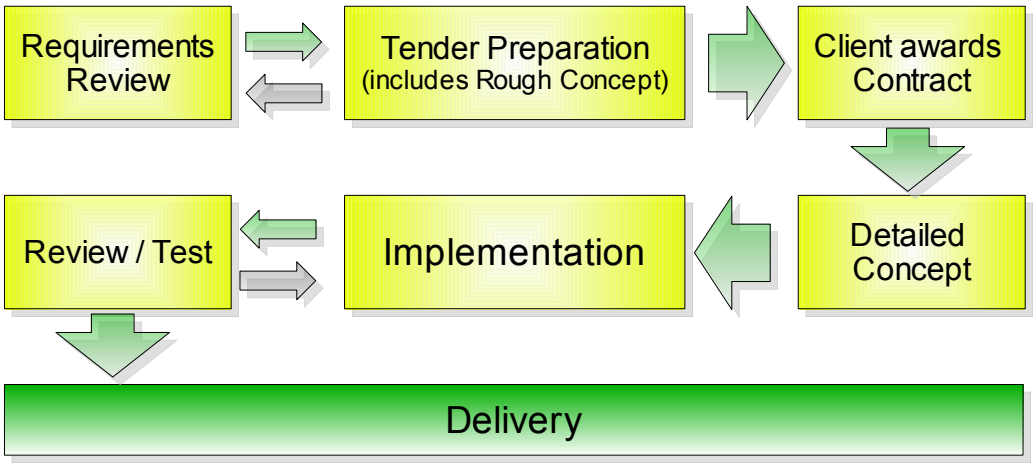
1.3 Developing the Open Source Way

Now an interesting question is how all this Open Source development actually happens. To answer this question, we need to have a closer look at the initial conditions of both Conventional Software Development (CSD), often referred to as closed-source or proprietary software development, and Open Source Development (OSD).

In CSD, we have two roles most of the time: a consumer / client role and a contractor / developer role. Often the client is not a specific company or person, but is rather described abstractly as "target group", if a general software product is developed for a broader market range. Another form of conventional software development is in-house development, where a department of a larger company develops software components which are used either as part of conventional products (e.g. software which runs on certain hardware, like car electronics, Video/ CD-Player, aso.), or drive the internal IT infrastructure, like software for insurances, banks or even public authorities.

The following graphic illustrates the whole process of CSD:

28 http://www.mozilla.org



Drawing 1: Software engineering process in CSD - source: own drawing

After the requirements have been reviewed, a rough concept is created and discussed. If all of the requirements have not been incorporated or certain functionalities do not operate as desired, this part is repeated until the customer gives his OK and awards a contract. Now a detailed concept is created and afterwards the implementation is started. Before the process finishes and the software is delivered, the implementation is tested and reviewed by the QA (Quality Assurance) team. If it does not pass all tests or certain requirements have not been implemented as described in the detailed concept, the process is pushed back to implementation until all of the all requirements are met.

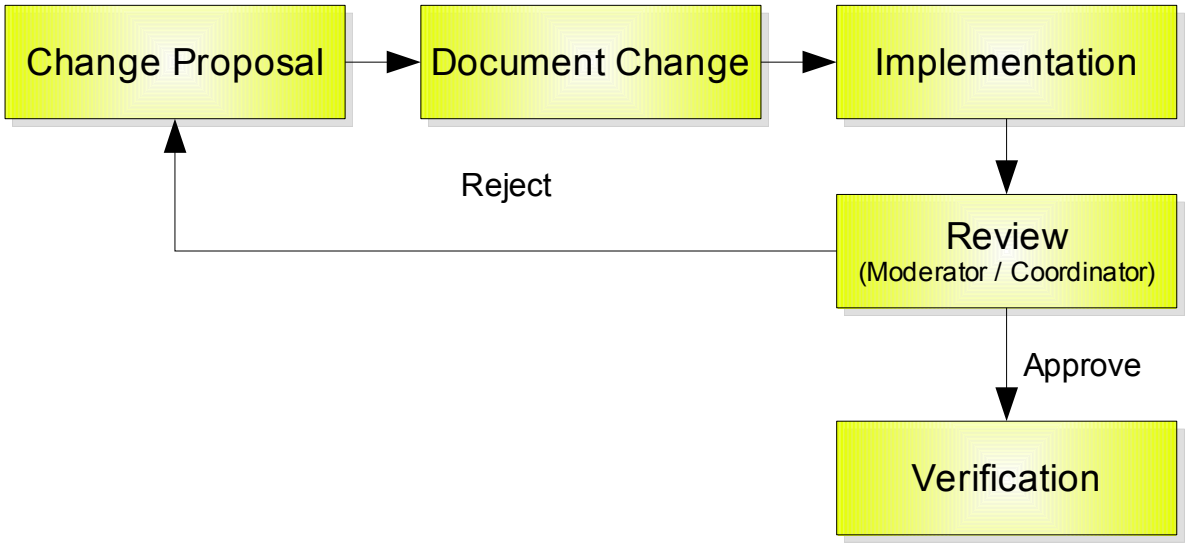
Now, in OSD there is no client role. One could say that client and contractor are the same: it is the developer himself who is asking for a certain software or functionality. There could be a couple of different reasons for him to start the development:

- There are no or no free alternatives of a particular software (typical desktop development),
- Current implementations lack certain features or do not run stably,
- The software is needed to make a certain hardware work (driver development),
- Personal or academical interest in a certain technique, environment or software,
- The fun of programming and the resulting personal reputation.

In many OS projects where there is no clear engineering process available, nobody is forced to create a detailed concept of what he is doing before he starts implementing. Also, there is no time, interest or even resources for bureaucratic overhead [Asklund2002]. If something is up for discussion and no compromise could be achieved, the "team leader" has the final say. The team leader is the original author of the software in most cases, or the maintainer of a certain area within a bigger software. *Linus Tovalds* is, for example, the ultimate authority for the *Linux Kernel* and he decides in the end

which extensions go into the kernel and which do not.

The following graphic takes a closer look at the *Change Management Process*[G] in OS projects. At first, a particular change proposal is put up for discussion. Then, a test implementation is created, usually by the developer or by the group who puts up the proposal. If the implementation is judged as "good enough" during a review, the implementation is taken over into the software. The review is done by team leaders; in bigger projects it is done by "coordinators" or "moderators". These people got their status because they are long-term contributors, so they have enough knowledge to evaluate the change submission. The procedure can be found in many OS projects, for example the *KDE* project, or, as already mentioned, the *Linux Kernel* development:



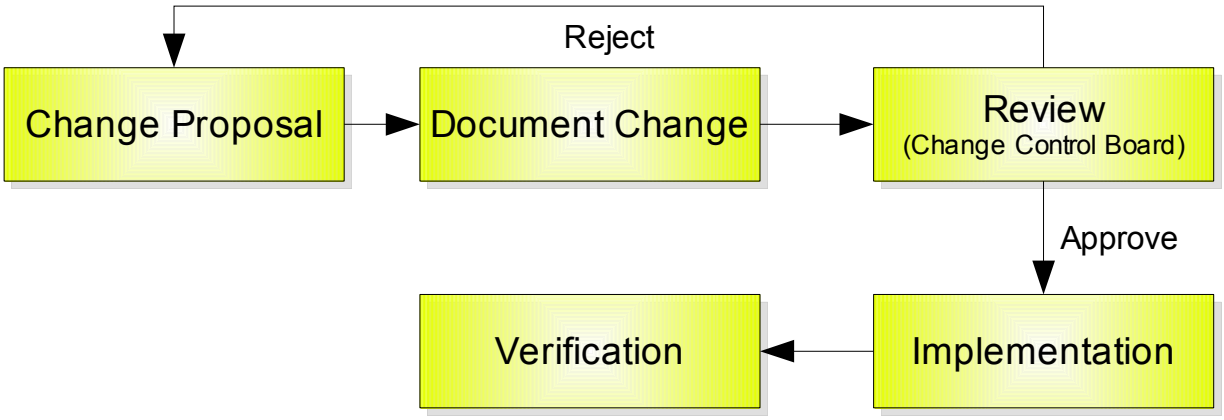
Drawing 2: Change Management Process in OSD - source: own drawing, based on [Asklund2002]

In CSD, the Change Management Process is quite different. Most traditional projects review change proposals using a *Change Control Board*[G] (*CCB*) and assign approved proposals to developers for implementation [Asklund2002] - there is no "free" choice of what a particular developer can do as it is in OS projects.

The *CCB* involves people from the Project Management, the Development folks, the QA team and other persons who are working in the operational area. In smaller projects only three people (the client, a developer and the project manager) are concerned with that, and in larger projects up to 30 people can be involved in the *CCB*.[29]

The following graphic illustrates Change Management in conventional software projects:

29 freely adapted from http://www.software-kompetenz.de/?7225



Drawing 3: Change Management Process in CSD - source: own drawing, based on [Asklund2002]

Many OS projects have very modest websites and use standard tools for project management, documentation and testing. Programs are often just available as source files which need to be configured and compiled on the clients computer. Real packaging is done by distributors which include the software within their distribution in binary and in source form (since this is required by most OS licenses). This helps the developers to concentrate on their most important task: develop software.

One major distinction between CSD and OSD is how communication takes place between team members. In general there are two forms of communication, *synchronous* communication (face-to-face or technologically supported instant communication via telephone, chat, etc.) and *asynchronous* communication (emailing, mailing lists, community boards, etc.).

In CSD, development happens locally most of the time (in "Collocated Teams" [Kotulla2002]), so synchronous communication takes place. Issues can be discussed almost instantly; if a problem arises during development (this could be of technical as well as of social nature), the problem can most likely be resolved within a few hours or a day.

In OS projects, the communication is asynchronous, and therefore good communication skills are even more important. Questions are answered with a huge latency, if the other developers are scattered around the world and live in different time zones. Therefore asking and answering precisely is a prerequisite in this regard.

[Kotulla2002] speaks of so-called "Virtual Teams" in this context: *"Because of the constantly high need to communicate, managing Virtual Teams is the hardest part. [...] A problem, which could have been solved by a local team within a few hours, could easily take almost a week in Virtual Teams."*

Many OS developers have a full-time job and are doing the work on their project in their spare time. Responsibilities are handled a bit slacker; obviously a free software author feels responsible for his program, he cannot and will not give any guaranty if any part of his program may do something what it should not do[30].

Since there is no financial or other relationship between the actual user and the developer(s) of an *Open Source Software*, harsh communication skills of one party will not work out at all; the user could get frustrated and switch the software, or the developer(s) could simply ignore the user's request for an improvement or a bug fix, if he / she does not stand to rules. However, politeness and thoughtfulness are needed for internal communication between developers as well, even if the opposite part is not right in his point of view: since the software is developed in an open manner, nobody can bar a developer to start a rival project based on the sources of the original one.

The next part will now give an overview which tools are used in OSD to overcome the lack of synchronous communication, and what else is needed to set up and to manage an Open Source project.

1.3.1 What is Needed to Organize an Open Source Project?

It is relatively easy to start on a new Open Source project. Usually, somebody develops a little piece of software on his own, and if he finds it stable enough, he shares the source code with others. This can happen through mailing lists or newsgroups (*.announce), for example, or by announcing the software on portals, like *FreshMeat*[31]. Usually a free software license is applied on the code so that it can be freely modified by others. If the software is found useful by others as well, they download it and give feedback to the original developer. More seasoned users may even send in *patches*[G], small chunks of code, to expand the functionality or fix problems they discovered inside the code. Out of this little micro cosmos many Open Source projects have been established in the past.

The more users download the program, use it and give feedback, and the more developers join the development, it becomes hard to organize everything by hand. At this time several well-known tools come into play to ease the interplay between end users and developers. A *Version Control* server is set up so that every developer has access to the code base while being in sync with the code changes from other developers. A mailing list is created to allow easy email communication between

30 Most Open Source licenses discussed earlier do consider this fact as well. See chapter 1.2 for more details.
31 http://www.freshmeat.net

all team members. An issue tracking system is set up to organize the way that *bug*[G] reports, feature and help requests are handled. Finally a *Wiki*[32] is eventually set up to speed up the creation of documentation, FAQs and general websites of the project.

While there are many software packages available for each of the mentioned tasks (commercial software as well as *Open Source Software*), certain packages have been evolved to standard packages effectively. Most of the software is web-based, so there is probably no restriction on what platforms it can run. On the other hand, binary components are also often available on platforms like Microsoft Windows, on which it is not easy to compile software from sources without bigger knowledge or even proprietary software packages. Still, it is recommended that a development system set up with these components runs a *Linux / Unix* operating system, because there are generally more web resources available which help on administrative tasks.

Version Control is done by *CVS*[33], the *Concurrent Versions System*, which is slightly replaced by its successor *Subversion (SVN)*[34]*;* we take a closer look at both systems in the second section.

Email communication is mostly setup with *GNU's Mailman*[35], which has been developed by Barry A. Warsaw for almost ten years. Mailman is easy to configure through its matured web interface.

The de facto standard for issue tracking systems is the free *Bugzilla*[36], developed by the *Mozilla Foundation*, the creator of many other popular free software products like the *Firefox* browser or the *Thunderbird* email client. Since it is used for all of these products, it is under heavy development. Their list of installations[37] looks like a big who is who in both Open Source (*KDE*, *Linux Kernel*, *Eclipse*, etc) and Closed Source (*NASA*, *Netscape*, *VMWare*, *Siemens*, etc) software development.

For *Wiki* systems, there has not yet evolved a standard software, only a common way of using these kind of systems. The first *Wiki* was *WikiWikiWeb*[38], developed by Ward Cunningham in 1995. Its syntax has many things in common with modern wiki software like *MediaWiki*[39], the system which drives the *Wikipedia* project and many others. For almost any platform, there exists a *Wiki* implementation, and most of them are free software as well.

32 A Wiki is a dynamic website, which allows to add and edit contents almost in realtime - without any interfering workflows - directly in place. It is probably the fasted way of collaborative web publishing today; the name "Wiki" comes from Hawaiian "wiki wiki" which means "fast".
33 http://ximbiot.com/cvs/
34 http://subversion.tigris.org
35 http://www.gnu.org/software/mailman/index.html
36 http://www.bugzilla.org
37 http://www.bugzilla.org/installation-list/
38 http://c2.com/cgi/wiki
39 http://www.mediawiki.org/wiki/MediaWiki

One now has the choice to either set up all of these services by hand or to use a common platform where everything is provided in an integrated environment. The most popular platform for this purpose is more than likely *SourceForge*[40].

SourceForge is a member of the *Open Source Technology Group (OSTG)*[41], which was formerly known as *Open Source Development Network (OSDN)*. The group provides services around *Open Source Software*, primarily through its different portals which serve commercial and non-commercial purposes. The popular news portal *SlashDot*[42] and the previously mentioned site *FreshMeat* are members of these portals. SourceForge is the biggest portal for *Open Source Software*, currently listing more than 100.000 projects and almost 1.3 million users.

To host a project on SourceForge, one needs to register it and to put it under an Open Source license. After the registration has been approved, one can instantly start with the development by accessing the supplied *CVS* / *Subversion* server, and can setup a web page on the provided disk space. Other services include a user forum, a simple bug tracking software and more. The whole service is totally free of charge.

Having all of the needed tools available does not make a good Open Source project though. There are a couple of "soft skills" which should make the project a real success:

1. Accept patches - nobody's code is perfect, allow others to improve it.
2. Do regular *releases*[G] - show the users somebody is working on the project.
3. Write a good documentation - without a good documentation, it is likely that nobody will join the efforts if the code itself is not self-describing, and well, most code is not.

The next part explains why it could be interesting for proprietary software companies to use and maybe even participate in Open Source development in one way or the other.

40 http://sourceforge.net
41 http://www.ostg.com
42 http://slashdot.org

1.3.2 Benefits of Open Source and Possible Business Models

Open Source Software is already used today by many companies and, without a doubt, plays a more important role than ever; in fact, Open Source products exist in many areas which do not even have a commercial competitor anymore, or the market share of this competitor is negligible. For example, if one looks for a Java Enterprise *IDE* (*Integrated Development Environment*) you pretty much have the choice between *Eclipse*[43], developed under an OS license by the Eclipse Foundation (which includes industrial leaders like IBM, Borland SuSE and others) since 2001[44], and *NetBeans*[45], which became Open Source in 2000 and whose main sponsor is still the company of the once commercial product, *Sun Microsystems*[46]. So for development-specific tasks *OSS* is already the way to go for many companies which are not bound to a mainly proprietary platform like .Net[47].

On the other hand, *Open Source Software* is also a political issue. Many European countries have plans to adopt or already have adopted OSS for their public body, and drop proprietary solutions like *Windows* or *Office* in favor of *Linux* and *OpenOffice.org*. A popular example is the *LiMux*[48] project: the city of Munich (Germany) replaces specific software on Desktop PCs at the end of their normal life cycle with *Open Source Software*, while the interoperability between open and proprietary solutions is ensured with additional software and tools. Europe therefore promotes *OSS* greatly; for example, in public tenders Open Source solutions are preferred in comparison to proprietary ones. One of the reasons for this fact is planning reliability. There is no "end of lifetime" for *OSS* solutions, since the software can always be adapted and recompiled to run on newer hardware or to support new features - with minimal effort. Also, specifications of file formats and protocols or general documentation are freely available and not proprietary licensed, which makes the building of custom solutions based on Open Source easier and less expensive.

As already mentioned, Open Source does not necessarily mean "free of charge", so one can earn money through *Open Source Software*, though other commercial aspects come into play. *Eric Raymond* describes different business models which are based on Open Source in chapter *"Indirect Sale-Value Models"* of his essay *"The Magic Cauldron"* [Raymond2000]:

43 http://www.eclipse.org
44 Information taken from http://www.eclipse.org/org/
45 http://www.netbeans.org
46 taken from http://www.netbeans.org/about/index.html
47 .Net is an open standard, though free (good) alternatives to Microsofts Visual Studio .Net are still rare. The community tries to bring .Net to Linux with the *mono* project (http://www.mono-project.com), but still has only partial support for Microsoft-specific classes like System.Windows.Forms. It is more likely that one can compile a mono source package under Visual Studio .Net without compile errors than vice versa.
48 http://www.muenchen.de/Rathaus/dir/limux/english/147197/index.html

1. Sell Services, Not the Product Itself (*"Give Away the Recipe, Open a Restaurant"*)

This business model is adapted by *RedHat, MySQL AB, MarchHare* (the company behind *CVSNT* which will be later discussed) and many others: the main product is put under Open Source, while the company behind the product sells support contracts, certification services and more. If a closed-source software product has a small market share, opening its source would boost the market acceptance. Raymond says that *"[a product is] likely to generate more value as a market-builder than as a secret tool"* if the manufacturer understands, that the *"true core asset is actually the brains and skills of the people"*[49] working for him.

Companies also could use that model and try to support (sponsor) already existing Open Source projects. The developers could receive a contract which allows them to spend a certain amount of their work time on the Open Source project, while the company could offer services around the software or create a product of it. Since the main development happens in-house, the company would be able to quickly react on support / feature requests and bug reports.

2. Concentrate on Innovation (*"Widget Frosting"*)

Raymond targets this model on hardware manufacturers which need to supply drivers and UI tools for their products usually for no additional charge. Obviously, there is the fear that by opening the specifications, certain important things about how the hardware operates are revealed; however, this is no longer true. Most products have a very short market window of six to twelve months today, so it would take a competitor much too long to get to know how your hardware is operates.

Raymond quotes the former KGB chief Oleg Kalugin about this issue, who got this insight even earlier: *"For instance, when we stole IBMs in our blueprints, or some other electronic areas which the West made great strides in and we were behind, it would take years to implement the results of our intelligence efforts. By that time, in five or seven years, the West would go forward, and we would have to steal again and again, and we'd fall behind more and more."*[50]

Another benefit of opening driver development would be that the company could just concentrate on innovation of the hardware product, rather then putting manpower in developing drivers and software, while these efforts would be dropped anyways after some time for economical reasons when the product's life-cycle is over.

3. *"Accessorizing"*

By selling accessories for *Open Source Software* (this could be everything from mugs and T-shirts to books and professional documentation) revenue can be created. Raymond mentions *"O'Reilly &*

49 http://www.catb.org/~esr/writings/cathedral-bazaar/magic-cauldron/ar01s09.html
50 http://www.catb.org/~esr/writings/cathedral-bazaar/magic-cauldron/ar01s18.html

Associates Inc., publishers of many excellent reference volumes on open-source software" [51] as an example for this business model.

There are other models imaginable which include the concentration / commercialization of the content or brand of the Open Source product. However, these models have only been applied by a very few companies as of now.

Doing closed-source development is mainly argued with protecting competitive advantages, or to hide confidential aspects of a business plan. While the second "reason" is no real reason, but leads to the fact that the proprietary application is badly designed, the first one needs more examination. Raymond says, that *"[the] real question is whether your gain from spreading the development load exceeds your loss due to increased competition from the [competitor which uses your disclosed source]"* and many people seem to underestimate the *"functional advantage of recruiting more development help"*[52].

One has to say though that not every software product, which is developed in a closed manner, qualifies itself to be "opened". If there are already very popular Open Source projects which provide the same functionality and the market niche is very small, it may be hard to attract developers from the community and to keep the project vital. However, it is not a problem to step back. For example subsequent versions of a software could be developed as closed source again[53], but one has to acknowledge, that older versions cannot be relicensed afterwards if they've been once licensed under the terms of an Open Source license. A market analysis should therefore always be done before the decision to open a particular software takes place, in order to prevent running into the above mentioned problems.

Sometimes opening a software is the only way to attack a monopoly of another software vendor. In 1998, *Netscape Communications* decided to open the sources of their Internet suite *Netscape Communicator*, which included the browser *Netscape Navigator*. This was once a commercial product in the early 90's and very popular, until *Microsoft* decided to give their own browser *Internet Explorer* away for free (but bundled with their commercial operating system *Windows)*. On the date that *Netscape* opened the Sources for the *Communicator*, the so-called browser wars were over: the *Internet Explorer* won as it had more than 90% market share. With *Internet Explorer 6*, *Microsoft*

51 http://www.catb.org/~esr/writings/cathedral-bazaar/magic-cauldron/ar01s09.html
52 http://www.catb.org/~esr/writings/cathedral-bazaar/magic-cauldron/ar01s06.html
53 *Borland* released their Database Management System *InterBase 6* under a free license in 2000. After their management changed, it was decided to release further versions (beginning with version 6.5) again as closed source and to stop the support and development of the free software project which managed the publically released sources. Soon after, the *Firebird* project (http://firebirdsql.org) took up the sources for version 6 and continued the development on their own (from: http://en.wikipedia.org/wiki/Interbase).

decided to stop development as it seemed to be "feature complete" in 2001. Why invest more time into a product if you are already the market leader?

Well, the community thought that there was a lot of room for improvements and after many development cycles they released the final version of the *Firefox* browser in November 2004, which was based on the original sources of *Netscape*'s *Communicator*. Since then, it continuously gained more market share (more than 10% according to several web statistics[54]), but moreover, it made the market leader *Microsoft* resume the development of *Internet Explorer* in 2005.

In return, *Netscape* uses the sources as base for their *Netscape* browser suite. This is still given away for free, but no longer developed in-house, therefore it costs *Netscape* almost nothing. By applying customizations to the software, they are able to bind the user tighter to their Internet portal, which marks their new real business.

Now, what I wanted to explain with this example is the following: the *Mozilla Corporation*, the company behind Firefox, is of course far from being as capitalized as *Microsoft*, but they could create and successfully introduce a product into a market which many people have said is saturated.

Does anybody believe this would have been possible if the product would have been developed closed-source? Simply put, no. The huge community that *Mozilla* has and which was needed for this task could not be replaced by paid manpower from a startup company. And even if it was replaced, the final product would have cost hundreds, if not thousands of Euros[55]. This especially applies to *Netscape*, which would not be able to support its portal business with a customized software, if they would still have to pay for the development of this "tool", while their business orientation has completely shifted.

54 The statistics differ quite a bit, generally more technically oriented websites have even a higher amount of users browsing with Firefox (up to 40%) than general purpose websites.

55 The startup company would always lose against a competitor, which "sponsors" the development by his other activities, allowing him to give away his product for free.

2 Version Control

In chapter 1.3.1, *Version Control* was one of the issues which were needed for organizing distributed development in Open Source projects. It may not be clear yet why a specific software is really needed for this task. In fact, it is assumed that most development, especially closed-source development, is still done without taking *Version Control* into account at all (obviously there are no clear figures).

The previously mentioned *Linux Kernel* project, and in particular *Linus Torvalds*, refused to use any kind of software for *Version Control* until 2002, due to the fact that there was no such software available which would have supported the way he and the other kernel developers engineered the kernel[56].

Manual *Version Control* should not be taken into account anymore as of today. A VC tool should be something which has minimal effect on the way a developer works; the tool just should make all the complicated work in the background, so a developer can simply concentrate on his main task: develop software.

However, the problem seems to be that nobody is teaching computer science students in university the use of *Version Control,* and if they finish their studies and enter the workforce, the employers often do not do that either [Sink2004]. So, many people just do not know what problems could be addressed by using *Version Control* and thus make the development a lot easier.

The following list shows some of the main benefits, but since *Version Control* systems vary greatly on the implementation side, it cannot be guaranteed that all points are equally available or possible for all systems:

56 Patches for the Linux source code were sent to *Torvalds* by email and he then decided if he included the patch in the mainline kernel or not. As the Linux project grew bigger, this kind of "handish" *Version Control* was no longer suitable, because Torvalds became the "bottle-neck" in the process of accepting patches from other developers. Between 2002 and 2005 the commercial solution *BitKeeper* (http://www.bitkeeper.com) displaced the original procedure, after an initial call from *BitMover's* CEO *Larry McVoy*, the company behind BitKeeper (http://lwn.net/1999/features/BitKeeper.php3). *BitKeeper* seem to fit *Torvald's* needs perfectly, since it enabled multiple repositories (distributed model), but in April 2005 *BitMover* dropped their license of the free version, effectively only allowing a few major projects and developer to continue the use of their product . *Torvalds* looked for a free alternative of *BitKeeper*, and since he could not find any good, he decided to write his own Source Control software and called it *git* (http://www.kernel.org/pub/software/scm/git/). *git* completly replaced *BitKeeper* very fast and released its first stable version in December 2005. *Junio Hamano* is the new maintainer of *git* today.

1. One Central Place for all the Code

Like a file server, a *Version Control* system offers a place where all the files of your projects reside and are accessible for all team members. Ideally, this place can be reached through *VPN's*[G] and *proxies*[G], so distributed development is made easier.

2. History & People Tracking

With *Version Control,* one has the possibility to look back on older versions of a software, revert changes that have once be made and identify each line of code by author.

3. Management of different Development Lines

If one needs to manage different lines of development, e.g. different product versions or versions for different operating systems, a *Version Control* system organizes these different lines for the developers, and allows them to *merge*[G] code parts from one line to another (e.g. if a bug has been fixed which exists in several "versions").

4. Parallel Development

Many *Version Control* systems support parallel development. This means that two or more developers can work on the same code at the same time without waiting for the others to finish first. Again, changes are later merged together (mostly) automatically by the system with only minimal action by the developers.

5. Low Network Usage And Small File Repositories

Most *Version Control* systems do not send the complete files over the network, but only *deltas*[G], which are then used to reconstruct the particular version at the client's side. Deltas have been possible for text files almost since the beginning of *Version Control*, recently newer VC tools provide delta handling also for binary files. This saves bandwidth and makes the work with such systems faster if one deals with many and / or large files of several megabytes.

Furthermore, new versions are also only stored as deltas of previous versions in the VC *repository*[G], so the back end is not bloated up by many file versions compared to normal file servers.

Besides all these benefits, *Version Control* is also a necessary part to achieve *ISO 9000*[G] conformance, especially for (document) change tracking and auditing procedures. If a company likes to get certified e.g. after *ISO 9001:2000*, there is no way of getting around VC.

There are many solutions available for *Version Control*, both proprietary and Open Source. This

section will concentrate on the most popular Open Source solutions, though, and will give no further references to strictly proprietary solutions like *Perforce* or *Visual SourceSafe*. Even if one concentrates only on Free Software, the information provided in the next chapters probably are not applicable on all available systems, since there are several dozens alone for *GNU / Linux*.

Furthermore, *Version Control* has many names, such as *Revision Control, Source Control* or *Document Control;* sometimes it is even called *(Software) Configuration Management* ([S]CM) [Negulescu2003]. There have to be made a few distinctions though, since especially SCM covers more than *Version Control* alone. The following chapter presents a classification of *Version Control*.

2.1 Classification

Reidar Conradi and *Bernhard Westfechtel*, both long-standing researchers in the field of Software Configuration Management (SCM), classify the tasks of SCM into two main disciplines in their work "Version Models for Software Configuration Management" [Conradi1996]:

1. Management Support

In this discipline SCM is concerned with product and component identification as well as controlling of product changes, e.g. to enable strict procedures for audit, review and status accounting.

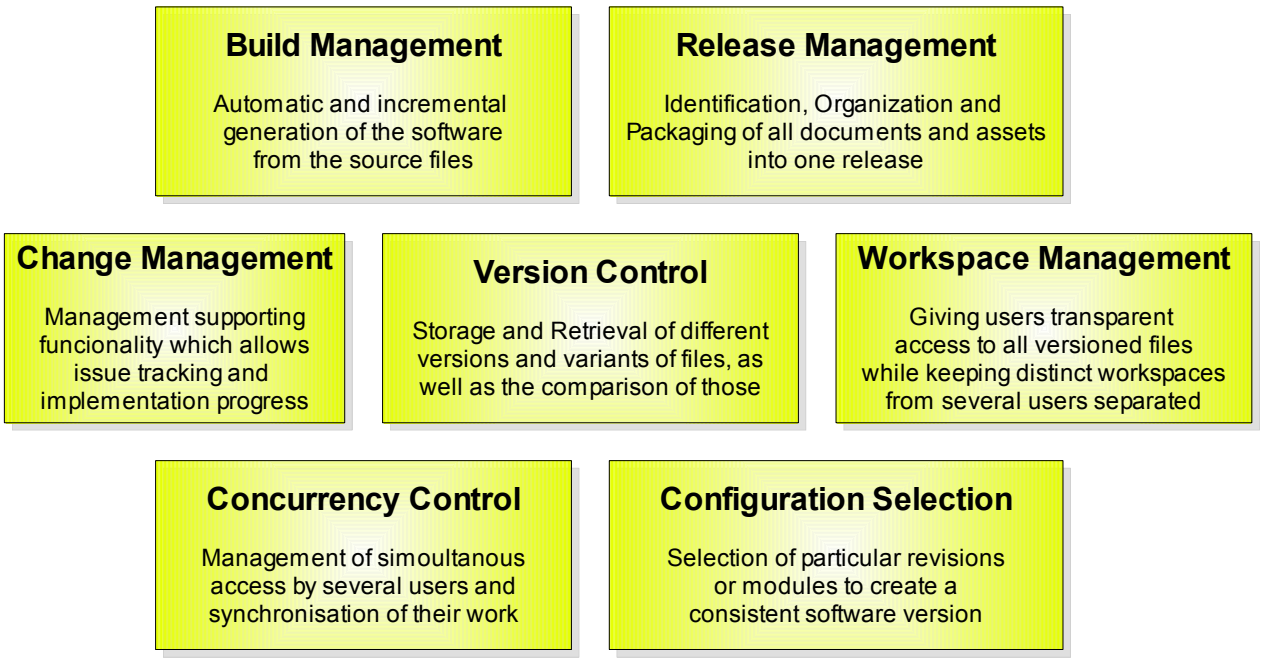
2. Development Support

"SCM provides functions which assist developers in performing coordinated changes to software products", such as helping them to accurately record *"the composition of versioned software products"*, maintain *"consistency between inter-dependent components"*, reconstruct *"previously recorded software configurations"*, build *"derived objects[G]"* from the source and finally construct *"new configurations based on the description of their properties."* [Conradi1996]

Conradi's and Westfechtel's closer examination aims the second discipline, where SCM is mainly considered a development support discipline. They do not use the term "Version Control" explicitly, but rather talk of *Version Models*, which *"[define] the objects to be versioned, version identification and constructing of new versions"*[57]. I will have a closer look at some of their concepts in chapter 2.3.

57 [Conradi1996], page 3

However, to get an easier introduction of what SCM and in particular VC is about, I am applying the developer perspective even more and follow the classification by [Asklund2002]: here, the seven most important aspects of SCM are:



Drawing 4: Seven aspects of Software Configuration Management - source: own drawing, after [Asklund2002]

Many *Version Control* tools can do more than just pure version control. They often also support *Workspace Management*, *Concurrency Control*, some even implement a fully transparent *Configuration Selection*, like *ICE (Incremental Configuration Environment)*, which implements the concept of *Version Sets* described by Andreas Zeller[58]. Other aspects like *Change Management* is the subject of issue tracking software like the previously mentioned *BugZilla* or projecting software like *dotProject*[59] and others. Popular tools for Build and Release Management are *make*[60] or its Java-complement *ant*[61].

Still, there is *no* single Open Source tool for all SCM needs available, and unfortunately there are not even clear standards or interfaces defined which allow easy connectivity between these different tools, or even between different *Version Control* software[62].

58 http://www.infosun.fmi.uni-passau.de/st/papers/zeller-phd/zeller-phd.pdf
59 http://dotproject.net
60 http://www.gnu.org/software/make/
61 http://ant.apache.org
62 In 1999, Perforce introduced with *RevML* (http://public.perforce.com/public/revml/index.html) an open XML dialect which primarily serves the purpose of giving an data interchange format between different VC software. This format does not seem to be heavily used though, probably because of performance reasons.

2.2 History

The first system for *Version Control* was the *Source Code Control System (SCCS)* created in the early 1970's by *Marc J. Rochkind* at the AT&T Bell Labs. It was proprietary software which was shipped with some Unix distributions of AT&T. SCCS was built directly for developers and allowed multiple users to work on the same system[63]. A GNU implementation of SCCS was soon created and called *CSSC (Compatibly Stupid Source Control)*[64] - its main use has been the conversion of SCCS repositories to newer repository formats like CVS or Subversion, though.

The *Revision Control System (RCS)*[65], which was implemented by *Walter F. Tichy* in 1982 during his time at the Department of Computer Science at Purdue University, soon replaced SCCS. It is completly file-based and uses the GNU program *diff* to create deltas, which are then stored in a special file format. In 1991, RCS was licensed under the GNU GPL and adopted as GNU package[66]. RCS has a more user-friendly approach than SCCS. The aim was that anybody, not only software developers, could use the software to version their files.

In the mid-1980's *Dick Grune,* lecturer of Computer Science at the *Vrije Universiteit in Amsterdam*, looked for a way to co-operate with his students, while they all had quite different schedules. He created a set of shell scripts which worked on top of *RCS* and called it *CVS*[67], Concurrent Versions System, *"for the obvious reason that it allowed us to commit versions independently."*[68] Later, this functionality (lock-less *Version Control*) would make *CVS* one of the most important software innovations of the year 1986[69]. The other main achievement of *CVS* is its network capabilities.

Beginning in 1989 *Brian Berliner* took the sources from Grune and reimplemented them in the *C* programming language. The *CVS* which we know today was born and started its success story in the 90's. *CVS* is no longer dependent on *RCS*, but still uses its file format (with slight extensions).

From that time, several other Version Control systems popped up, many of which had the aim to replace *CVS* because of its disabilities (no *atomic commits*[G], bad handling of binary files[70], and more).

63 http://www.uvm.edu/~ashawley/rcs/manual/html/ch05s02.html and http://en.wikipedia.org/wiki/SCCS
64 http://cssc.sourceforge.net/index.shtml
65 http://www.gnu.org/software/rcs/rcs.html
66 http://www.uvm.edu/~ashawley/rcs/manual/html/ch05s02.html
67 http://ximbiot.com/cvs/wiki/index.php?title=Main_Page
68 http://www.cs.vu.nl/~dick/CVS.html#History
69 http://www.dwheeler.com/innovation/innovation.html
70 The original CVS stores and retrieves a binary file not through deltas like it does with a text file, but always as the complete file. This leads to heavy network usage and big repository sizes when handling many binary files with this system.

Some followed the common *centralized model* of CVS like *CVSNT*[71], others introduced the newer *distributed model*, like *GNU Arch*[72] or *monotone*[73] (chapter 2.3.3 explains both models in detail). The official successor of CVS is *Subversion*, though. It uses the centralized model as well and introduces many other features. I will have a closer look on Subversion in chapter 2.4.2.

2.3 Architectures and Concepts

Many different approaches have been made over the last couple of years to improve or expand certain functionality of the original *Version Control* systems like *SCCS*, *RCS* and *CVS*. Some of these approaches will even demand a completely different point of view on *Version Control* in this regard. The following chapter will present these approaches, gives examples of systems which implement them and tries to examine in which use cases these tools are applicable.

2.3.1 Product Space and Version Space

The versioned object base, which is all objects that are *versioned*[G], consists of Product Space and Version Space.

The Product Space *"describes the structure of a software product without taking versioning into account"*[74], in other words, it describes a single state of the product. Single software objects, such as components, can have composition or dependency relationships to other objects within this structure. A typical composition relationship would be "all files inside a directory", where each file has a composition relationship with the directory it resides in. The composition graph is a single directed graph which root node defines the software product.

Dependency relationships are orthogonal to composition relationships and denote dependencies between different objects. E.g. when one thinks of a single source code file, the inclusion of header files via `#include` denote the dependency of the current module to other modules.

Each software object has at least one object identifier. An external identifier is most likely chosen by the developer (in file-based systems the full filename) and may not be unique, while the internal, system-generated identifier is unique.

71 http://www.cvsnt.org
72 http://www.gnu.org/software/gnu-arch/
73 http://venge.net/monotone/
74 [Conradi1996], page 5

Software objects also may have different representations, for example an *XML* data structure may have a textual representation (*.xml file) or a binary tree representation if the structure is already parsed. One distinguishes here also between *source elements* and *derived elements*, where derived elements can be created automatically from the source elements, and only the source elements are subject to the *Version Control* system. Still, it is the task of the tool how to handle any kind of representation, for example to find the differences between two versions. Textual differences on non-structured sources (simple text) can be easily found and visualized by a general *diff* command, which is implemented in most VC systems and works line-based. Using such a command on clearly structured sources like XML fails, because the strict line-based approach does not take the structure, the *grammar* in which the data are organized, into account.

Most of the available VC systems do not distinguish between different representations of software objects or even apply custom commands based on their type. This task is left to external tools, which need to be installed and configured separately.

The Version Model, the main aspect of the Version Space, *"defines the items to be versioned, the common properties shared by all versions of an item, and the deltas"*[75], which denote the differences between them. According to [Perry2005], the Version Space can be distinguished between

a) The Logical Version Space

This space consists of all possible versions (built out of *revisions*^{G} and *variants*^{G} of every versioned object of the software). It is often not useful to take this space into account, since it contains many incomplete, or even not working versions.

b) The Practical Version Space

This space consists of all committed versions. One can assume that the developer has assured that this version is complete and working before he / she added it.

Depending on the *Version Control* tool it might or might not be possible to retrieve all logical, possible versions, but only certain, previously added version sets. This has the advantage that software objects such as files are not mixed between different versions in a checked out copy and therefore do not create a not working version.

75 [Conradi1996], page 8

Furthermore, one distinguishs in the Version Space between *extensional* and *intensional* versioning. Extensional versioning means that the versioned item is a container for a sequence of revisions in the back end of the VC system. Each single revision is explicit, which means that it was created and checked out before, and has a unique number assigned to it. This versioning scheme is implemented in the most VC tools.

Intensional versioning identifies a valid version by a variety of attributes on each versioned file. For example, in a software project there could be developed a GUI component, which has specific revisions for one operating system like *Microsoft Windows* and another, like *Unix / Linux*. Another component could be a database, where *Oracle* and *IBM DB2* are supported. To select a valid working version, a condition is created, such as "give me the GUI component for Microsoft Windows". This is a similar approach as the *preprocessor* directives of the programming language *C*, only that it is language independent. Of course the Logical Version Space is much larger here, since many possible versions make no sense or do not work at all (e.g. selecting a database which is not available on a certain platform). The only system which is known to implement this type of versioning is the previously mentioned *ICE*, unfortunately this system is no longer available.

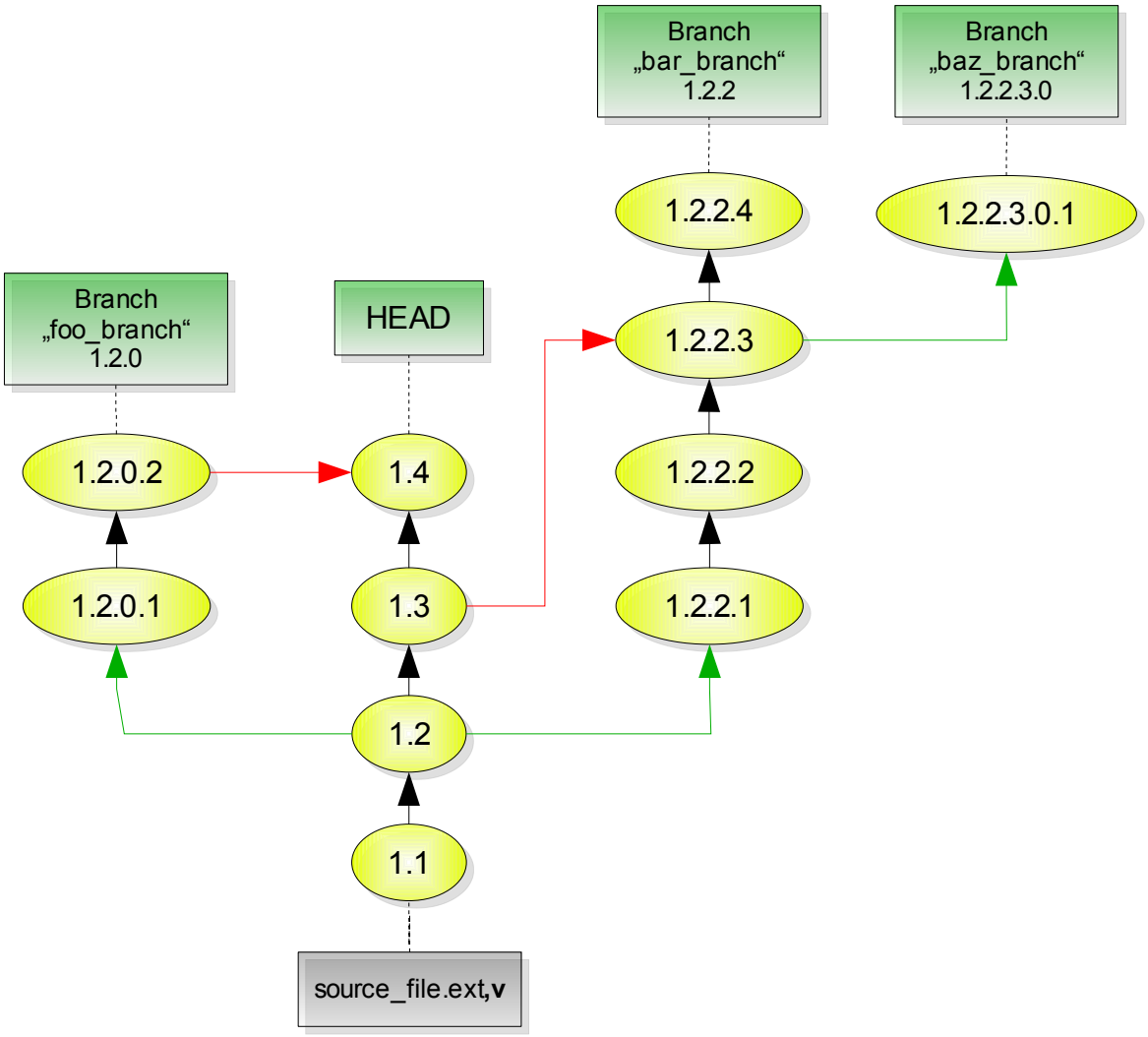
2.3.2 Database vs File System based Repository

Many Versioning Systems store each file as a single file in the back end repository. The folder structure of the versioned software is then mapped one by one to the folder structure in the back end. Examples for those systems are all tools which are based on *RCS* and the *RCS* file format[76], e.g. *RCS* obviously, *CVS* and *CVSNT*. Another system which uses a file system approach, which is different from *RCS* though, is *git*. In the following part the *RCS* file format is discussed.

In the *RCS* file format single revisions are stored as deltas from the base, the initial revision. Each one gets its own decimal number (`2*n` digits). *Branches*[G], also called *offsprings*, are labeled internally by adding another decimal number to the original revision number from which the branch was created (`2*n+1` digits). The *RCS* format is therefore an example of extensional versioning discussed in the previous chapter.

Merging changes between branches happens simply by applying the changes (all deltas) for each file between two tags or two revisions, from one branch to the other (in the following drawing merge points are visualized by red arrows):

76 RCS files are recognizable by the additional ",v" at the end of the particular filename, e.g. "file.ext**,v"**.



Drawing 5: RCS version graph - source: own drawing

A graph of an *RCS* file therefore has a two-level organization which looks like a *directed acyclic graph* (*DAG*), though it differs from it in two things: there is only one root element (revision 1.1) and the applied merges are not joins; a branch will exist further even after it has been merged into another. Branch points mark *offspring relationships* (green arrows), while the sequential improvement within a branch is created by *successor relationships* (black arrows).

HEAD[G], often also referred to as *Trunk*[G], denotes the main development line for the file and is a special branch. Implementation-wise, branches are special *tags*[G] which are "tacked" to the revision for which the branch was created. They are also called "sticky" tags. Subsequent revisions on the branch (deltas to previous revisions) are then identified by their revision number.

The *RCS* file format is specified[77], so all systems using *RCS* files in the back end should be interchangeable. However, this is not true for all, since the format evolved slightly over the time. For

77 http://www.die.net/doc/linux/man/man5/rcsfile.5.html

example, *CVSNT* adds additional fields to the original format, which renders it useless for *CVS* and *RCS*. Therefore an upgrade path might only be from *RCS* over *CVS* to *CVSNT*, not vice versa.

The main drawbacks of file system based approaches, and in particular of the *RCS* format, is that they cannot be used to version directories or meta data, make problems when a file is renamed[78] and solely scale with the underlying file system in terms of performance. Real time statistical analyzes are hardly possible this way.

Another problem special to the *RCS* format is that since only single files are versioned, the only way to get a working set of multiple files is by applying tags on them. If a file was not tagged because the developer forgot to do so, the version is broken.

The other approach is using a relational or XML *database* (DB) back end for the repository. This approach is used by many different VC tools, for example *monotone* or *Subversion*, though there has no standard format evolved yet. An upgrade path from other, mostly file-based formats to the newer systems is provided by some tools, though only popular systems like *CVS* are supported here and there is no way going back to the old format in many cases. Switching the used *Version Control* tool amongst a critical development phase is never a good idea, since there is the risk of loosing the history or even the current data if another tool does not what it should.

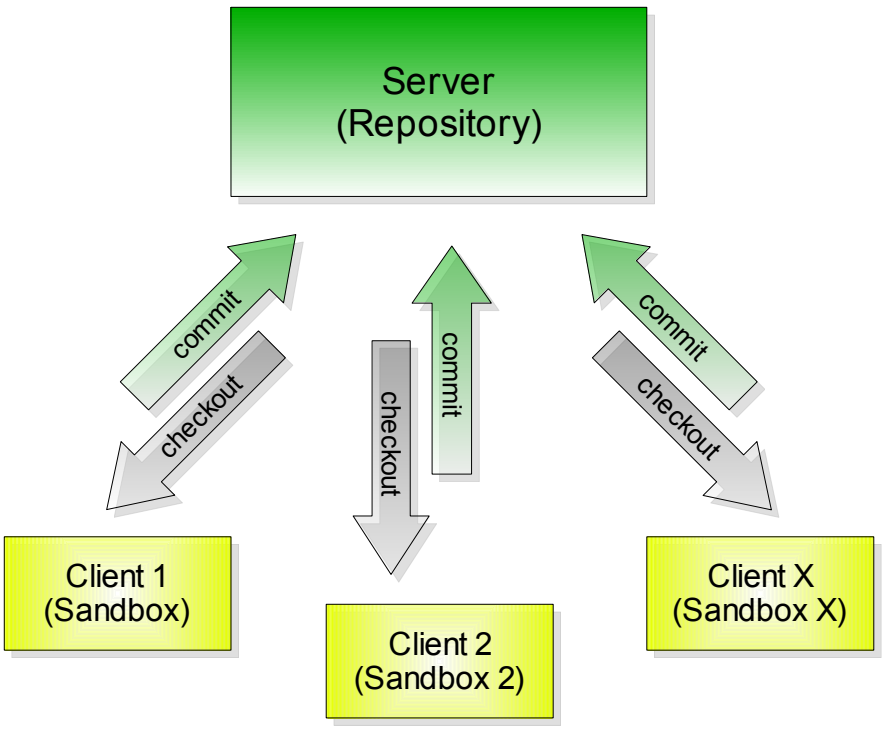
The main advantages of a database back end are the speed of accessing repository contents and to have a separate layer to which other tools can connect to. Also, the database back end could be distributed on other physical machines for performance reasons and thus scale with the number of clients. Obviously there are still more problems to solve here, like the ability to make hot backups or run multiple (failure) instances which need to synchronize between each other for really large teams.

2.3.3 Centralized vs Distributed Version Control

The question of what distinguishes the centralized from the distributed approach has not so much to do with the actual way networking is done between each participant, but more with the manner the project organization happens inside the team. I will now take a closer look at both approaches and try to outline advantages and disadvantages.

78 CVSNT includes experimental rename support in newer versions, the main problem is however that the RCS format uses the file's name/ path as identifier, thus when a file is renamed, the file has to be identified through another mechanism. What makes it this complicated is that it must not destroy older versions in the repository by renaming files in current version.

In VC tools which follow a centralized approach (the most popular here is probably CVS), there exists a single server which holds the code repository, the versioned object base. This single server can be accessed by multiple clients that retrieve their local copy via a *checkout*[G] over the network. The checked-out files then resides in the developer's *sandbox*[G], or working copy, in which he can make local changes. As soon as he finishes his work he *commits*[G] his changes back to the repository and shares them with the other developers[79]. The following drawing illustrates this process:



Drawing 6: Centralized Version Control - source: own drawing

The biggest advantage of this approach, having one central development repository which contains the mainline of the development, is also a disadvantage: if the repository server fails for some reason or the network is down, the developers cannot share their code with other users any longer, so distributed development stops. Also, most known centralized repository approaches cannot be mirrored, so the number of clients which are able to do concurrent checkouts, commits and other functions, directly scales with the underlying hardware and network connection of the server machine. Some *Version Control* systems like *CVSNT* tend to use big amounts of RAM to build files from the saved deltas e.g. for a checkout. Especially if one handles binary files with these systems, up to twice

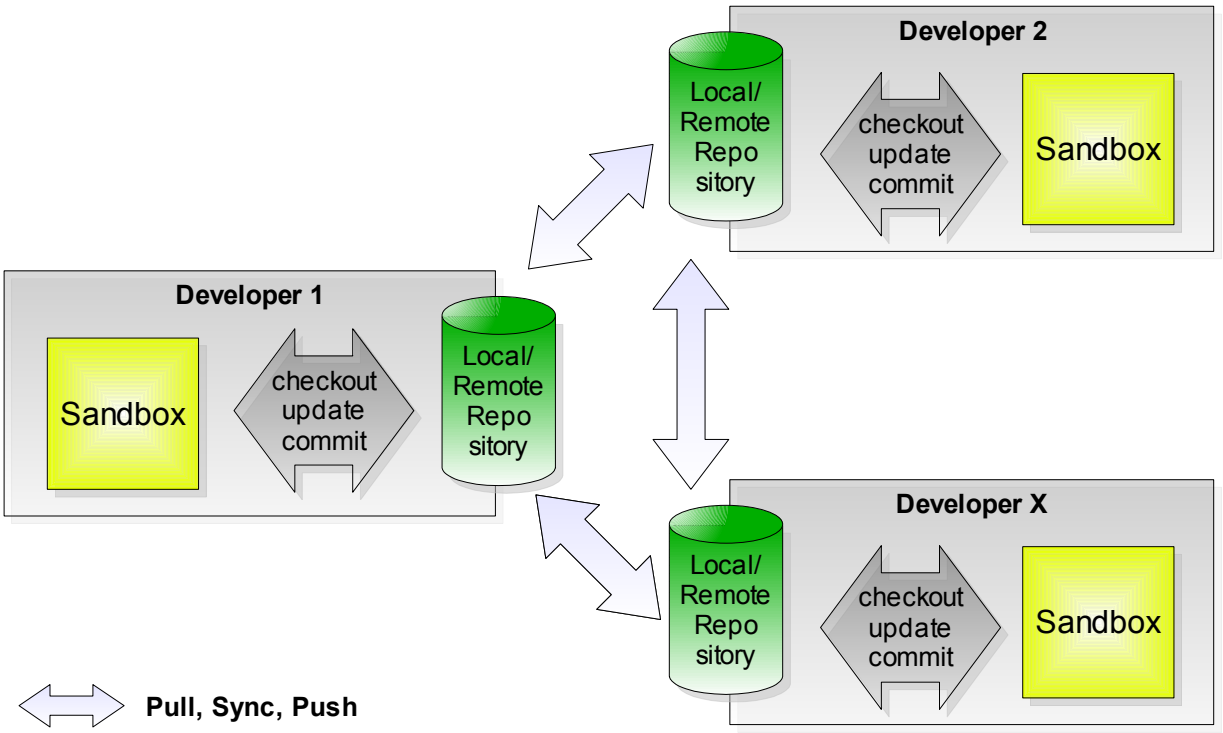
79 Other developers do not retrieve the changes automatically; instead they need to trigger an update action which is similar to the checkout action, but only transfers the differences between the new version and the version in the user's sandbox to the user. If updates would be automatically be pushed into other user's sandboxes they could create *conflicts*[G] with local changes if the work happens concurrently (which is the case in most VC tools).

as much memory as the original file size is, may be needed to perform these actions.

Another possible problem with the centralized approach can especially be found in large development teams, and even more in big Open Source projects. Here it is not suitable to give every developer write access to the code base, since thoughtless or inexperienced users may commit broken code accidentally or even intensional. Still, OS projects especially are willing to accept patches for bugs or improvements and also like to lower the barrier of how code can be contributed by 3rd parties, thus attracting more developers to work for the project. The *Apache Project* solved this issue by giving a group of "trusted" people write access, where each of them is responsible for a certain aspect / module of the project. Those "module owners" then receive and examine 3rd party *patches*, and then decide if they take them into the main repository or reject the submission. However, the way these *patches* are submitted is not generalized in any way and mostly happens via email (for example in the *Linux Kernel Project*). A patch is in this case always uncoupled from the original code base, version and context and the individual developer needs to take care how it can be included into the mainline development.

The distributed approach solves this problem. Here every developer can have his own repository based on the original one. One could speak of a branch or even a "fork", a split of the project base which is now under the direction of the user. This leads to the fact that multiple HEAD's of the project exist, since there is (technically) no central server which always contains the most recent and up-to-date version of the software. Instead each user has its very own local repository, from which he checks out his sandbox and also commits changes back to. If he likes to share his changes with other users or likes to retrieve other user's changes, he synchronizes his local repository with a remote one.

The next drawing illustrates the procedure for the *Version Control* software *monotone*:



Drawing 7: Distributed Version Control: architecture of *monotone* - source: own drawing

The biggest issue when using *Distributed Version Control* is integration, however. The hierarchy in which changes are merged "upwards" (like in a pyramid) is often not mapped by the tool.

The Linux Kernel project and their tool *git* have introduced a signing process: external contributions are merged into local repositories by a group of trusted people first, of which everybody digitally signs the submission with his / her very own name. The more people do *sign off* a contribution, the more the contribution gets trusted and therefore will later be included in the official repository of the branch owners (e.g. this is Linus Torvalds for Kernel version 2.6).

However, there is some general criticism about *Distributed Version Control*. *Greg Hudson,* a free software developer, outlined four limitations of this „pyramid“ approach in *"Why Bitkeeper Isn't Right For Free Software*"[80] amongst other things[81]:

1. Limited Development Speed

Hudson refers here to the problem that only the one branch owner on top of the pyramid examines and merges a huge set of patches and *"even with the best tools, a single integrator can only achieve a certain level of throughput."*[82]

80 BitKeeper is a commercial *Version Control* Tool which works after the distributed approach (see footnote 56 on page 23 for more information)
81 http://web.mit.edu/ghudson/thoughts/bitkeeper.whynot, last changed: March 2003
82 Line 53f in http://web.mit.edu/ghudson/thoughts/bitkeeper.whynot

2. Single Point of Failure

Furthermore, if this single *integrator* fails for some reason, e.g. because he *"suffers an accident, goes on vacation, or simply burns out, development is disrupted until a new integrator can be selected and comes up to speed."*[83]

3. Opinionated Maintainers

One maintainer / *branch* owner decides what goes into the mainline development and what not, and according to Hudson *"it is a rare individual who is always right."*[84]

4. Limited Filtering

Only a few people do *review* (and actually have the right to accept or reject) an external contribution at a lower level of the pyramid scheme, and such a review might be *"cursory or nonexistent"*[85] depending on if they are under heavy workload or even do their work biased and not open-minded.

For Hudson, these four limitations lead to *„slow and unpredictable release schedules, poor stability of release branches, and a lack of important standards"*[86], which is primarily targeted against the *Linux Kernel* project in first instance, but what could popup in any similar maintained project as well.

There are commercial solutions available that try to take the best of the two approaches, namely *WANdisco*[87], which is available for *CVS*, *CVSNT* and *Subversion*. They add distributed („multi-site") features to these VC tools that allow the synchronization of different server repositories which may reside anywhere in the world; in case of *WANdisco* this even happens in real-time and without a central transaction manager (which would be a single point of failure). However, all these tools can only assist, but not replace a good project management.

83 Line 57ff in http://web.mit.edu/ghudson/thoughts/bitkeeper.whynot
84 Line 61f in http://web.mit.edu/ghudson/thoughts/bitkeeper.whynot
85 Line 69 in http://web.mit.edu/ghudson/thoughts/bitkeeper.whynot
86 Line 74ff in http://web.mit.edu/ghudson/thoughts/bitkeeper.whynot
87 http://www.wandisco.com

2.4 Tool Shootout – Feature Comparison of Popular OS Versioning Software

This section will present three Open Source *Version Control* tools: *CVS* and its successor *CVSNT*, *Subversion* and *monotone*. Each system has its up- and downsides. To classify important functionality which every modern Versioning System should be capable of and support, I will use parts of the taxonomy introduced by [Fish2006] and expand it where needed. This taxonomy will be used in the review for every system. Chapter 2.4.4 concludes this section by giving a summary and a final feature matrix containing all gathered aspects.

a) Technical Aspects:

- **Support for Atomic Operations**
  Are operations, such as commit, atomic to the repository or do unfinished operations leave the repository in an intermediate state?
- **File and Directory Renaming and Copying**
  Is it possible to rename (move) and copy files and directories while keeping the history of these objects intact?
- **Replication Support**
  Is the system capable of replicating and also synchronizing repositories which have been replicated?
- **Permissions on the Repository**
  How fine-grained can access be given to the repository, if permission settings are possible at all?
- **History Tracking**
  Does the system offer a way to track changes, by file or even by line?
- **Meta Data Support**
  Is it possible to version additional meta data for each versioned object?

b) Status and Deployment

- **Activity**
  Is the software actively developed? Do other companies push the development?
- **Deployment**
  For what platforms does the software exist? Is it easy to deploy it? Are there upgrade paths from other VC software?
- **Networking**
  How does the system integrate into an existing networking infrastructure? Can the traffic be secured?
- **Ease of Use**
  Is the software through its command set easy to use? Do graphical clients exist, and if yes, for which platforms are they available?

c) Documentation and Support

- **Documentation and Help**
  How well is the documentation on the system maintained? Do other help resources exist?
- **Professional Support**
  Is it possible to get professional (paid) support on the system? What is offered and how much does it cost?

2.4.1 Matured and Well-known: CVS / CVSNT

CVS is the most known *Version Control* system available today; the latest stable version is 1.11.21. It is still the de facto standard when it comes to *Version Control*, and many other tools try to be as compatible as possible with the offered command set and operating methods of this tool. The history of *CVS* has already been discussed in chapter 2.2, since this system has been an important part of the overall history of *Version Control*. However, *CVS* is no longer actively developed in favor of its successor *Subversion*. *CVS* is licensed under the terms of the *GNU General Public License*.

CVS uses the centralized approach, which means that there exists usually one repository which is accessed by multiple clients. The network traffic is done by a proprietary, relatively insecure protocol, but can be *tunneled*[G] over the *SSH* (*Secure SHell*) protocol which allows different types of encryption.

The back end repository uses files to store versions of objects (the file format is based on the previously discussed *RCS* file format), obviously „objects“ are only files and not directories or meta data in this case: there is no versioning support for these other objects in this system. This is also one of the biggest problems with *CVS*, because the tool sees directories merely as paths in which versioned files exist in a single state. Therefore, file renaming and moving is not possible since the system is unable to store or reproduce the directory structure of a certain version.

The *CVS* client works on the checked-out version, called the *sandbox*. Since *CVS* works per file (and each file gets its own revision number) it is not necessarily the case that all contained files compose a valid working version. It is possible to update to an invalid version by retrieving an older revision of just a single file, or by selecting revisions by date, rather than by revision number. In the latter case revisions of files are selected which may be completely incompatible because they mark different states of development.

As compensation *CVS* introduces the concept of tags to mark different revisions of different files as coherent, still, such a tag is never automatically set on all files in the repository, but only on the selected ones. Special *branch tags* allow offsprings from certain versions for parallel development. The symbolic tag *HEAD* marks the mainline development and is automatically set in all files of the repository.

Since *CVS* does not support atomic commits, it is not very failsafe. If a repository action is canceled before it finishes, the repository contains partially committed and uncommitted changes. Obviously this does not happen that often, the easiest way to fix such broken commits is to update and then commit the missing files again.

Permissions can be set only repository-wide. A user has either no rights, read rights or read / write rights. There is no way of setting permissions on certain modules or even branches. Altering permissions and other repository settings is accomplished by a special, versioned directory called *CVSROOT* which is automatically established when a new repository is created. This directory contains several files which can be used to administrate the repository and which are automatically checked out in the repository once they have been committed[88].

CVS is used in many places and many developers are familiar with it, therefore there are a lot of help ressources available. The documentation on the system is very good as well, obviously one should consult other resources before one starts to learn the basics, otherwise one gets confused with the used terminology if it is the first time one deals with such a system. Commercial support can be retrieved

88 It is out of scope to explain all administrative files; the interested reader can find more information here: http://ximbiot.com/cvs/manual/cvs-1.11.21/cvs_18.html#SEC158

by *Ximbiot*[89], and starts at approx. $18.000 / year, which is limited to a maximum of 20 support issues and 12 users. There are no additional warranties supplied by this contract, though, so the license terms of *CVS* are not touched.

The original *CVS* is natively only available on Unix platforms (*Linux*, *BSD*, *MacOS X*) and is normally already included in most *Linux* distributions as well. For the Windows platform *CVSNT* has emerged[90].

CVSNT is a fork of the original code base of *CVS* and is actively developed by *Tony Hoyle* since 1999 as an Open Source project[91]. I am reviewing version 2.5.03 build 2151 of *CVSNT*, like *CVS* also released under the terms of the *GNU General Public License.*

The primary objective of this project was to bring *CVS* to the *Windows* platform while providing the highest possible compatibility to the older system. *CVSNT* however was also extended to „fix" some problems *CVS* had, thus it was not only released for *Windows*, but for a variety of other platforms (including *Linux*) as well. Today it supports delta versioning of binary files (while *CVS* stores the complete file on each commit), includes *ACLs* (Access Control Lists) to apply fine-grained permissions on repository files, is fully Unicode-aware, introduces *smart branch points*[92], which ease the use of branches in *CVS* a lot, integrates as system service under *Windows*, and many other functions[93]. Still, even *CVSNT* has some limitations because it is still based on *CVS*: atomic commits are not possible, replication has to be accomplished with external tools, renaming support is at beta-stage, and the versioning of meta data or even directories is not possible at all as of today.

One can get professional support from *March Hare Software Ltd*[94]. *March Hare* sells licenses and services for their product *CVS Suite,* which is based on *CVSNT*, and adds additional features to it (integration into 3rd party issue tracking, workspace management, auditing software and more). Licenses for 10 concurrent users including one year telephone support, upgrades, prioritized feature requests and security alerts start from about 2900 Euro[95].

89 http://ximbiot.com/CVS_support.html
90 One can use Cygwin (http://www.cygwin.com/) to run the original *CVS* as native application under Windows, though the integration and speed is not the same as of *CVSNT*.
91 http://cvsnt.org/wiki/HistoryPage
92 The original *CVS* needed several normal tags applied on different branches to know which changes should be merged into another branch and which not. Particular for subsequent merges this was very difficult, since one could get easily conflicts by merging changes into another branch which have already been applied there earlier. *CVSNT* sets internal tags, branchpoints, automatically when a merge happens, and therefore recognizes on the next merge what should be applied and what is already applied.
93 http://cvsnt.org/wiki/CvsntAdvantages
94 http://www.march-hare.com
95 http://march-hare.com/cvsnt/techspecs/en.asp

To ease the daily use with the *Version Control* system and hide the sometimes complex command set from the user, there is a variety of *Graphical User Interfaces* (*GUIs*) available for many different platforms. *IDEs* like the previously mentioned *Eclipse* even include native *CVS* support without the need of external binaries. A particular well-designed multi-platform *GUI* is *LinCVS*[96], which exists in a free version and a "XXL" version with additional features. The latter version is free for non-commercial use, otherwise the pricing starts at about 79 Euro per license.

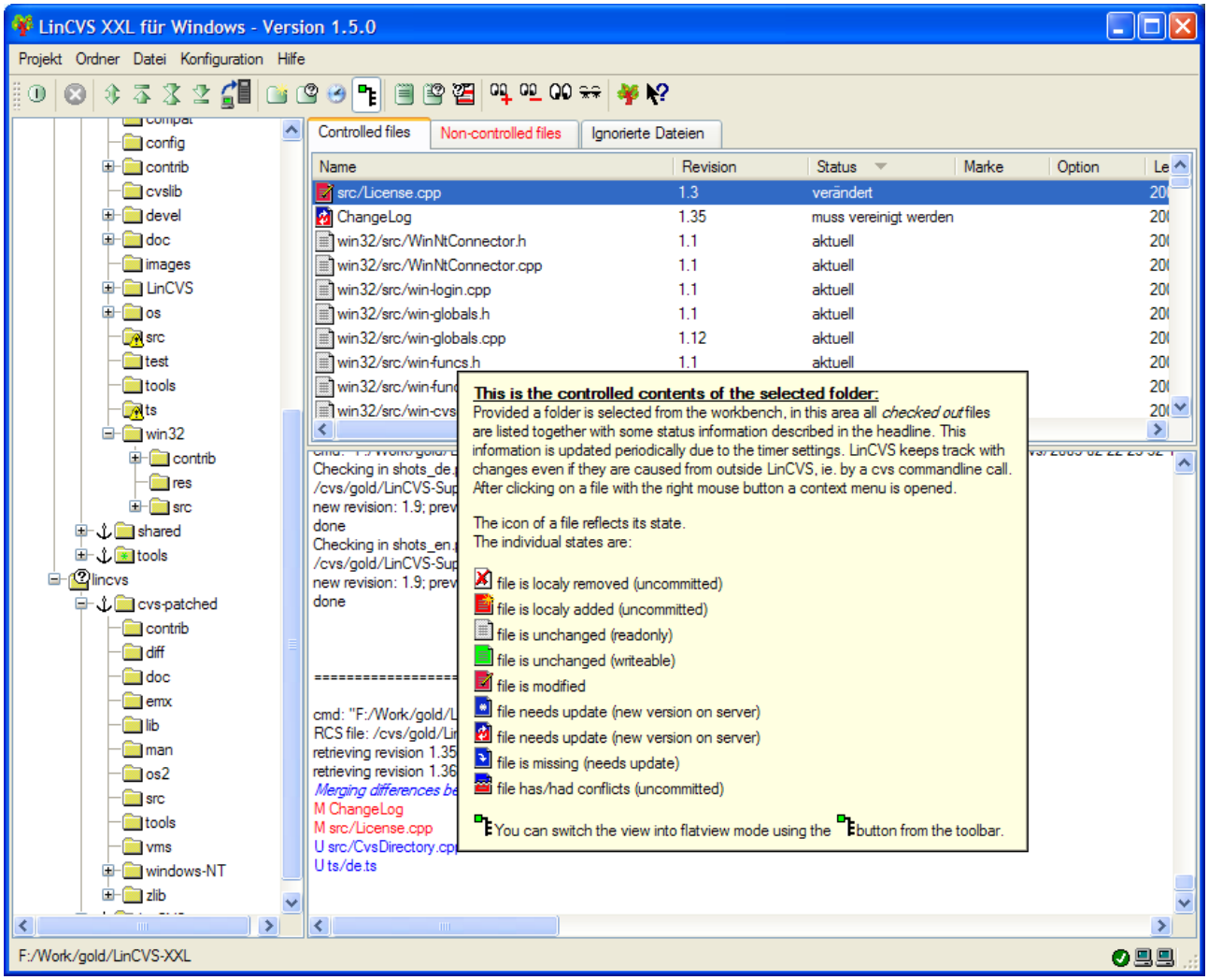


Illustration 4: Main window of LinCVS XXL on *Windows* - source: http://lincvs.com/shots/shot4.png

96 http://lincvs.com/

The free version of *CVSNT* is used and shipped with most *Windows CVS* clients, because it is possible to contact older *CVS* and newer *CVSNT* servers with the same binary. Another very popular *CVS* / *CVSNT* client is *TortoiseCVS*[97]. *TortoiseCVS* acts as Windows Explorer shell extension, which provides the most needed *CVS* commands on right-click. This program is Open Source as well, but unfortunately only available for *Microsoft Windows*.

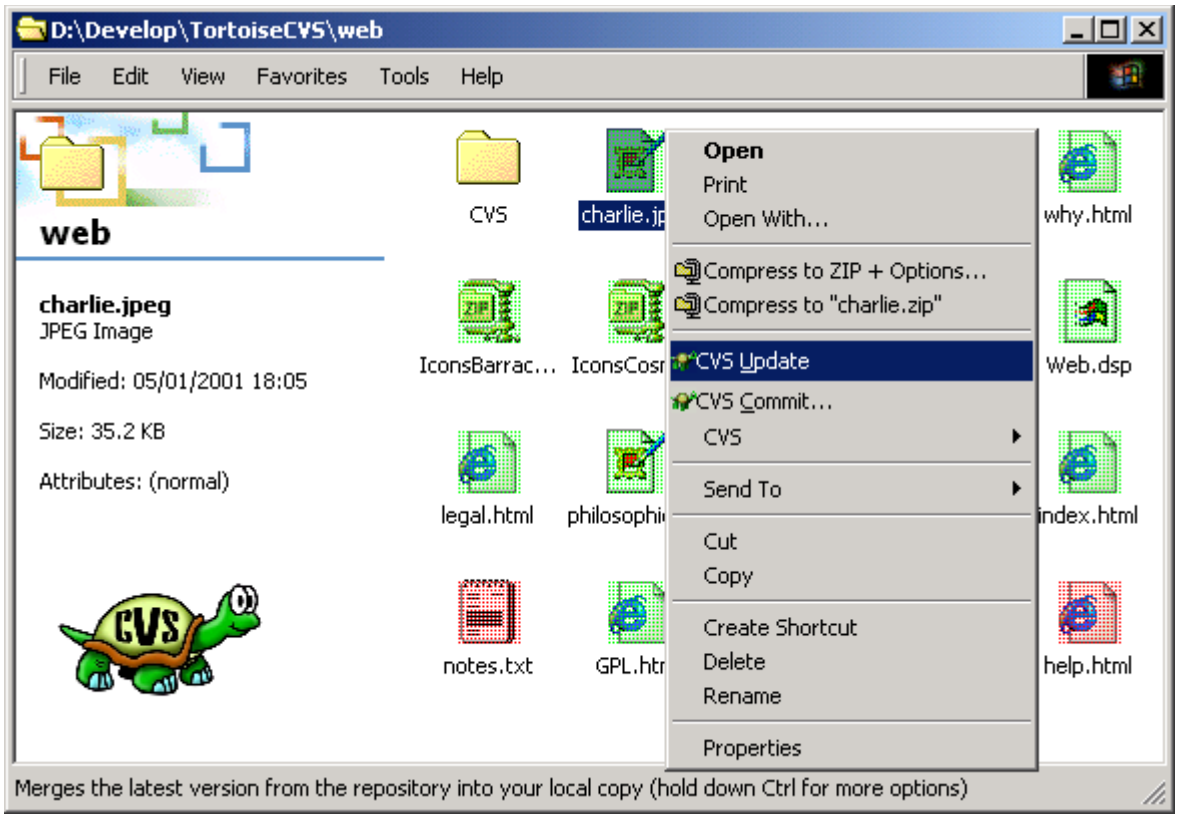


Illustration 5: TortoiseCVS Explorer extension - source: http://www.tortoisecvs.org/screenshot1.png

In the end it is no longer recommended to use the original *CVS* for new development, since there are really better alternatives available. If one likes to stick with the way *CVS* "works", one could upgrade to *CVSNT*[98] and receive a variety of functionality improvements as well as the option of professional support. If meta data and rename support is immanent, one of the other two tools reviewed here may be more suitable.

97 http://www.tortoisecvs.org

98 Since *CVSNT* still uses the *CVS* / *RCS* file format, upgrading the repository while containing the history is an easy task. However, there is no downgrade path from *CVSNT* back to *CVS*. See chapter 2.3.2 for more information about the *RCS* file format.

2.4.2 The New Kid On The Block: Subversion

With the obvious flaws that *CVS* had, many people sought alternatives over the years. In 2000 *CollabNet, Inc*.[99] started looking around for developers which would be able to write a replacement for CVS, which they could use in their collaboration software *CollabNet Enterprise Edition*[100]. They came in contact with *Karl Fogel*, the author of *Open Source Development with CVS* (Coriolis, 1999), who already discussed with his friend *Jim Blandy* the characteristics for a new *Version Control* system they called *Subversion (SVN)*. Fogel agreed to work for CollabNet and hired a few other people for the task. The design goal was to match the features and "feel" of *CVS* while not making the same failures. SVN therefore also uses the centralized model explained above.

After *Subversion* became self-hosting in August 2001 (that means the code of *Subversion* could now be managed by *Subversion* itself), the final version 1.0 was released in February 2004 licensed under an Open Source license[101]. The following review tests *Subversion* 1.2.3 released in May 2005.

Subversion has a very similar command set like *CVS*. Basic actions include *checkout*, *add*, *log*, *remove*, *update* and *commit* which are known from the latter tool. This should make it fairly easy for somebody who is familiar with *CVS* to understand *Subversion* as well.

However, the tool is distinct from *CVS* in many other aspects. One point is that Subversion uses global revision numbers instead of per-file revision numbers. A revision therefore records the state of the whole repository tree, not just of a single file. One does not say "give me revision 115 of file foo.c", but rather "give me file foo.c as it appears in revision 115". Branching as we know it from *CVS* does not work this way, since there are no revisions like "115.1". Instead, *Subversion* branches are *copies* of a subtree which are saved in another *Subversion* directory, so branching is often called "copying" in this regard. These copies are cheap, because *Subversion* does not really copy all contents somewhere else, but rather links to the original contents. If changes are applied on a copy, only the differences (deltas) to this linked original contents are saved.

99 http://www.collab.net
100 http://svnbook.red-bean.com/nightly/en/svn-book.html#svn.intro.history
101 Apache/BSD-style, while retaining the rights on the use of the name of the product as well as the name "Tigris", which is used as development platform: http://subversion.tigris.org/project_license.html

To manage branches easier, a common repository layout has been evolved over the time, obviously one is not bound to use that if one works with *Subversion*:

- *project/trunk* - Contains the mainline development
- *project/branches* - Contains copied subtrees of *trunk*
- *project/tags* - Another branch directory, which is however never changed (useful for recording "stable" versions)

Another point in which *Subversion* differs from *CVS* is its network capabilities, which is far the best feature of SVN. Users can connect to SVN's *svnserve* server process (optionally secured via an *SSH tunnel*) which uses a proprietary protocol, use the file:// protocol to connect to a local repository, or use *WebDAV*[G] via *HTTP*[G] or *HTTPS* (*SSL*[G]-secured *HTTP*) to connect to *Subversion* over the Internet. In the latter case *Subversion* is installed as a module in the popular *Apache Web Server* and can use its full facilities for access restrictions. Another advantage of this approach is that *Subversion* repositories can be reached easily even by restrictive configured networks and can even be mounted as network shares in many operating systems. If one configures so-called *auto-versioning*[102] it is even possible to save new revisions without the use of a native *SVN* client.

Technically *Subversion* is very up to date. It supports atomic commits, has full rename support (implemented as "copy and drop"), allows the setting of fine-grained rights on a repository (configurable through *svnserve.conf* file or through the *authz_svn_module* available in Apache), and also offers meta data support. These meta data are called *properties* in SVN and are versioned as like the versioned object itself. The value of a property can be arbitrary data - not only text information. The SVN manual gives the example of an image repository where each image can get a thumbnail image assigned as a property[103]. In fact, properties are used by *Subversion* itself for special purposes like identifying the mime type of the file (*svn:mime-type*), or if it is executable (*svn:executable*). All these *system properties* are prefixed with a "svn:".

The repository back end in *Subversion* was database-driven (using *Berkeley DB, BDB*) until version 1.1, but since 1.1 a new, default file-based format called *FSFS* has been used. The new format has some advantages over the previous DB-format, the most important here is probably that the format is platform independent. It also allows smaller repository sizes, works faster with directories containing many files and supports backups via standard backup software. However, there are certain downsides

102 A full explanation how to enable Autoversioning in SVN 1.2.3 can be found under http://thomaskeller.biz/work/thesis/SVN_Autoversioning-HOWTO.txt
103 http://svnbook.red-bean.com/nightly/en/svn-book.html#svn.advanced.props

with this format, as the code base for the new format is less mature and a little slower in certain areas[104]. Therefore, one can still use *BDB* as data back end when this is explicitly given as a parameter while creating the repository.

Speaking of setting up *Subversion*, one has to mention that this software is very easy to deploy. An option to import *CVS* repository files is available, though it is not possible to import the newer *CVSNT* file format with this option. The documentation is very good; the official book *"Version Control with Subversion"*, published by O'Reilly, is licensed under *Creative Commons* (a free license) and is constantly updated with every new version[105]. It has been written by *Ben Collins-Sussman*, *Brian W. Fitzpatrick* and *C. Michael Pilato*, three active developers of *Subversion*. The community is also very active, mailing lists and more are available through the project's website[106]. A variety of graphical user interfaces support *SVN* - apart from every *WebDAV*-capable program there exists support for *Subversion*'s native protocol in *TortoiseSVN*[107], a Windows Explorer shell extension similar to the previously mentioned *TortoiseCVS*, and *RapidSVN*[108], a client available for Windows and Unix platforms, to name only a few.

Commercial support is offered by *CollabNet*, the primary sponsor of *Subversion*. There are three support plans available, "Silver", "Gold" and "Platinum", each with a certain level of responsiveness[109]. Unfortunately there are no prices mentioned on *CollabNet's* website for the different services, so one has to file an official request first to get more detailed information.

2.4.3 For Highly Distributed Development: monotone

The last system I like to review here is *monotone*[110], which implements the distributed versioning model, in other words it implements a true $P2P^G$ approach. *monotone* is not quite ready for deployment and use in a production environment; still the tested version 0.26pre2 looks very promising. There are other Open Source VC tools like *git* or *GNU arch*, which also work decentralized, but both are hard to learn and lack on a good implementation for Microsoft Windows platforms. *monotone* is developed since 2003 by *Graydon Hoare* and others under the terms of the *GNU General Public License*.

104 The full list of pro's and con's on both formats can be found under http://svn.collab.net/repos/svn/trunk/notes/fsfs
105 The official startpage of the online version is http://svnbook.red-bean.com/
106 http://subversion.tigris.org
107 http://tortoisesvn.tigris.org/
108 http://rapidsvn.tigris.org
109 http://www.collab.net/subversion/index.xhtml and http://www.collab.net/support/comparison.html (Comparison of different support plans)
110 http://venge.net/monotone

In *monotone* everything is about *hashes*[G]. The system uses the cryptographic *SHA-1* function (*Secure Hash Algorithm* utilizing 160 bits) to create checksums of the contents of any versioned object. The resulting 40 bytes long string is then used as identification for one revision of a certain file. Each commit creates a new repository version[111]; this is because the repository version is determined by the *SHA-1* hash of the *manifest*, an internal data structure which lists all versioned objects and their *SHA-1* hashes. If one of these hashes changes, the hash of the manifest changes as well, and a new version is created. Since it is unhandy to deal with 40 bytes long strings *monotone* offers a smart UI which "guesses" the complete revision identifier even if only a few characters are given.

The system implements a three-way architecture, consisting of a local repository, a local working copy (*workspace*) and remote repositories (see Drawing 7 on page 35). The repositories are single files which contain the back end for the underlying SQLite database system *monotone* uses. They are therefore easy to backup. Synchronization between different repositories happens via the *netsync* protocol. The project got the port 4691 granted by *IANA (Internet Assigned Numbers Authority)* for this purpose[112]. *monotone* uses the same program in client and server mode. To start a server process it is enough to give the database and the branch(es) as parameters to the program:

```
monotone --db=foo.db serve "com.example.project.branch"
```

For globally unique branch names it is proposed to use the syntax

```
TLD.DOMAIN.PROJECT.BRANCH[.SUBBRANCH[...]]
```

which ensures that a branch is never used twice anywhere in the world. Though there is a discussion going on whether this is really the best way[113], *monotone* uses this java-package-style syntax for their self-hosting project.

Meta data information is managed with *certificates*. A certificate consists of a key (the name of the certificate), a value and the revision / file id to which this certificate is associated. *monotone* handles all kinds of data (even system data like the revision's commit date, the log message, the author of the commit, tags, branches, aso.) through this mechanism. To put in an amount of extra security each certificate is signed by the user who issued it. This signature clearly identifies the user through the common public / private key system known from *PGP (Pretty Good Privacy)* and others by adding the cryptographic "checksum" of the certificate's data to the resulting certificate. By exchanging the public part of the user's key, other people can check the validity of any data and therefore accept or decline access from foreign sources if they have a local server running.

111 It is not possible to make commits to single files without changing the overall version.
112 "Monotone Network Protocol": http://www.iana.org/assignments/port-numbers
113 http://www.venge.net/monotone/wiki/BranchNamingConventions

monotone currently lacks of good graphical user interfaces. There are a couple of approaches, but none has gone behind alpha state as of now. The community around the project is very active though, and before *monotone* hits a stable version a *GUI* should be ready. The documentation is very nicely organized, one can receive help through mailing lists, *IRC* (*Internet Relay Chat*), the wiki or the online manual which covers all aspects thoroughly. Unfortunately no professional support is yet available for the system.

In conclusion *monotone* is definitely worth a try, though it should not be used in production environments. It runs therefore out of competition in the comparison.

2.4.4 Feature Matrix

The following feature matrix summarizes the gathered results of the above reviewed *Version Control* systems. Columns which do not contain textual descriptions have been applied a grade system, where one star (*) means insufficient or bad, two stars (**) mean moderate or fair, and three stars (***) mean very good or excellent. If no star is applied the feature or aspect is not available or does not apply.

Subjects / Systems	CVS / CVSNT	Subversion	monotone
Technical Aspects			
Support for Atomic Operations	No	Yes	Yes
File and Directory Renaming and Copying	No / Partly	Yes	Yes
Replication Support	No[114]	No[115]	Yes [116]
Permissions on the Repository	Partly[117] / Yes	Yes	Yes
History Tracking	Yes	Yes	Yes
Meta Data Support	No	Yes	Yes
Status and Deployment			
Activity	-[118] / ***	***	***
Deployment	** / ***	***	***
Networking	* / **[119]	***	**[120]
Ease of Use	***	***	*[121]

114 Replication support is offered by 3rd party tools like *WANdisco*. *CVSNT* 2.6/3.0 might offer built-in replication.
115 Replication support is offered by 3rd party tools like *WANdisco* for SVN.
116 This is by design, since in a decentralized environment there is no single server node, thus no single point of failure.
117 It is possible to expand CVS with ACLs, while the ACL functionality is already built into CVSNT.
118 CVS is no longer actively developed further, though ximbiot cares about bugs of the original software.
119 Though *CVSNT* offers a variety of access protocols (including Kerberos authentication and SSPI support under Windows), its rather hard to get connected in a restrictive environment, where only certain services are allowed. There is no WebDAV support like in Subversion.
120 Since communication happens over a special port, this could be a problem in complex setups with Proxies and Firewalls.
121 No graphical clients are available to date.

Subjects / Systems		CVS / CVSNT	Subversion	monotone
Documentation and Support				
	Documentation and Help	***	***	***
	Professional Support	No / Yes	Yes	No

In the end its not easy to pick a clear winner, since both, *CVSNT* and *Subversion*, try to keep the "feeling" of *CVS*, while removing the lacks of the original software. Subversion does that and starting with a complete new code base, while *CVSNT* originally was thought as *CVS* fork for Windows, thus still needs to care a lot about backwards compatibility to *CVS*.

monotone has a very nice approach and implementation, but is not quite ready for deployment in production scenarios and offers no commercial support yet either, so its not our current scope. Therefore *Subversion* is probably the best choice for a *Version Control* tool to date in a professional environment.

2.5 Best Practices for Version Control

Over the decades many people have used *Version Control* in their projects, and through this daily usage many common practises have evolved. Of course this is slightly biased depending on the tool one uses; since the centralized versioning model is the most wide-spread one, I will discuss only those practices which can be applied with tools using this model here.

2.5.1 Check-in Only What Is Really Needed

This has been mentioned already in the chapter about Product Space and Version Space. Derived objects, such as *C* object files or other automatically created resources should not be included into the repository. Only those files which are needed to build these derived objects are subject to *Version Control*. The reason is simple: these binary components are often platform-dependent, so if another user checks them out and works on a different platform, he cannot use them anyway, because he needs to rebuild them. This overwrites the original files and marks them as changed, thus they could be accidentally committed again, and now the original user has unusable files on his next update.

There are certain exceptions though, when it could be useful to include derived objects as versioned objects. This could be for example test results from test runs, which should be kept together with the project's files. Another example would be derived objects which take very long to be created dynamically or do not change often, so it is useful to have them always available.

Almost all *Version Control* systems come with a technique to "hide" derived objects from a user's eye by ignoring them. Normally there already exists a default configuration right from the start, so the system "knows" of common derived objects and ignores these automatically. However, one can add additional file patterns to the VC configuration to hide project-specific files which should not popup as "unknown" and therefore would be possible to add to the repository.

2.5.2 Commit regularly, in Small, Grouped Portions and only Working Code

People who have never or only a little dealt with *Version Control*, tend to commit many files at once when they start using VC tools. Most of the time, these commits happen before they stop working at the end of the day. A comment, which should describe the actual changes is then applied to the whole state of the module and is often unsuitable for most of the files, since it is written too "generic" or does not even apply to every committed file. Often, many smaller changes are completely forgotten to be mentioned if the change was done many hours ago.

The situation gets worse, if the user's comment is not very descriptive and basically tells nothing about the changes made to single files. One could argue that reading the diff output every VC tool offers should be enough to understand what has happened between two distinct versions, but the practice proves that wrong: even if the code itself is well documented through comments; complex functionality changes that span over several files cannot be examined properly by looking at a single diff.

Moreover good comments make it easier for other developers to follow the development, even if they do not currently work on a particular module. In concurrent development teams there are often automatic email systems setup, which notify other developers on changes. These "commit emails" then contain the comment of the commit as well as all changed files. A descriptive comment therefore helps a lot to give others an overview what has changed.

Leaving the comments aside, another thing is very important if one works together with many other developers. As long as you do not work on your own branch alone, you probably commit your code to some mainline which other users use as well. What happens if your just committed code does not work and produces errors? Suddenly the other developers can no longer build a valid version as soon as they update their sandbox. Doing regular updates with the others developers work is immanent in concurrent development, so broken code actually stops all concurrency immediately. A developer thus should always make sure that the code he commits passed some basic testing on his local machine.

In conclusion, what should all be considered when committing changes to a central repository?

- Do not describe the obvious things ("changed letter a to letter b"), since this could be really examined by looking at the diff output. Instead, comment the reason behind and the functioning of a change.
- Group files which belong together for a change.
- Commit regularly, because naturally you remember more of the actual changes if these changes did not happened too long ago.
- "Do not break the tree" - Do not commit untested code which leaves other developers clueless why their updated version does not work any longer. This stops concurrent development!

2.5.3 Branch when Needed and Keep your Branch Up To date with the Trunk

Branches are a good way to separate certain development lines from each other. There are several reasons why one wants to branch from the mainline. The most popular reason is probably to separate a "stable" version of the product from a "development" version. This could be important e.g. to restrict access to certain areas of the project which should not be accessible by users which might break something, to give them some kind of "playground".

Another reason is the ability to apply bug fixes on older versions which have been released to the public. If there is only one line of development available, you cannot release an updated version of your product if your sources already contain code parts of a newer version, which may not even run stable.

Branches are in many VC systems nothing more than special marks on certain repository versions, so it is cheap to create those. Like we have heard for example in *Subversion* branches are created by simply copying the contents of one directory to another. While in the real world this would effectively double the needed size on a file system, *Subversion* only applies links to the original versions and stores the differences between those and the applied changes. So again, branches are cheap, and one should just use them, because they make the life with *Version Control* much easier.

It has to be distinguished if a branch is created for forking the current development, or if the branch might be merged back into the mainline (the Trunk) of the repository. In the latter case, one has to make sure that both development lines do not drift too far away from each other, because this will

make a later merge almost impossible, or at least will create many conflicts[122]. A developer should therefore merge new code from the mainline into his branch on a daily basis. This way he can recognize possible conflicts early and act accordingly to resolve them.

2.5.4 Do You Really Need to Lock it?

People migrating from *Visual Source Safe,* or another tool which uses file locks intensively, to a system which supports concurrent development, often fall back into their old behavior pattern and think they need to lock the items which they checkout.

At first one has to say that such a "checkout lock" would lock the whole module in tools which support concurrency, so effectively no other developer would be able to work on the whole project! There are possibilities to lock single files even in CVS (via the "edit" and "unedit" commands), but this functionality is not used very much. Why should it be useful at all?

Secondly exclusive locks are more obstructive than helpful. Imagine you have a team of developers and one of these developers accidentally forgets to "unlock" the code he last worked on? Now of course you can ask him to do so, but what if he is not available at that moment, because he does not sit in the same bureau, is not in the same city, maybe not even in the same country like you? Your development stops immediately.

The fear of having to work without locks is mostly because people think it would create too much conflicts. However, intelligent merge algorithms ensure that most concurrent edits on single files are automatically resolved, and only those edits which concern the same line of code create a conflict. Also, if one checks in the code regularly it is not that often that concurrent edits on the same file happen at all.

Again, it is really not needed to acquire exclusive locks on source code "just to be safe". If you fear that different edits may create too many conflicts a better approach would be to use branches.

122 This is especially true for the centralized model, while the developers of monotone, which uses the distributed model, state, that even merging two complete distinct branches makes no problems. *monotone* allows several heads in one branch, so merging two branches together would probably create multiple heads in the final branch. These heads could then be further merged several times, until one head is the result (see http://venge.net/monotone/faq.html).

Glossary

Atomic Commit

An atomic action, such as an atomic commit, is an action which should either completely succeed or completely fail. It is the task of the software to ensure that such an action is really atomic, which means that no partial changes are stored permanently if an error case pops up, but a complete rollback to the original, pre-action state is performed.

Branch

In SCM a branch is a development line which exists parallel to the main development line in the *Version Control* system without interfering with the latter. It is often used to separate stable from unstable code.

Bug

A Bug is a misbehavior of a software (a defect). Many think the term goes back to the early days of computing where a real bug was once found in a mechanical calculating machine, but in fact already Edison used it to describe little *"faults and difficulties"*[123] in his inventions.

Change Control Board (CCB)

"A formally constituted group of stakeholders responsible for approving or rejecting changes to the project baselines."[124]

Change Management Process

"A procedure to ensure that proposed changes are merited and will not adversely affect other elements of the plan or interdependent plans."[125]

Checkout

A checkout is the action which loads files from a remote or local repository in a local workspace. Some *Version Control* software locks the checked out files for other users (*MS Visual SourceSafe*), while the most VC tools allow concurrent access and apply a merge later on.

Compilation

The word compilation describes the work of *compilers,* special software programs, which translate high-level languages such as *C++* into machine code or byte code. Machine code is dependent on the underlying hardware of a computer, while byte code independent of the hardware, but needs a special runtime environment to be executed (e.g. for the programming language *Java* this is the *Java Virtual Machine*).

123 http://en.wikipedia.org/wiki/Software_bug
124 http://www.welcom.com/content.cfm?page=136#Change%20Control%20Board%20(CCB)
125 http://w2.byuh.edu/projects/tools/definitions.html

Conflict

In *Version Control* systems which support concurrent access to versioned objects a conflict pops up if two users altered an item simultaneously and the automatic merge algorithm of the software is unable to bring together the changes of both authors into one resulting file. This normally happens for most binary file formats, and for text files only if the same lines have been altered.

Delta

The difference between two item sets or two items. Directed deltas are used to re-create revisions in VC systems. They are applied on an earlier revision and create a more recent revision, but they cannot be used the other way around. In a version graph applying a directed delta is represented by the edge which spans between two revisions.

Derived objects

Derived objects are artifacts which are created e.g. during the compilation process. Some of these objects are temporary files which are later used to build the final binary file, which then includes all machine code to execute a certain program. In general a derived object is always dependent on a base object, usually a source file, and can be created from this base object at any time.

Hash

A hash is the output of a hash function which maps an infinite domain to a finite codomain. It is possible that two or more elements of the domain have the same value in the codomain, but this is not very likely with diverse input. Many hash functions use an algorithm which outputs a completely different hash value if only one bit of the input data changes (like for example *SHA-1*).

Head

In *Version Control* the term "Head" often refers to the mainline or trunk of the development. The term probably originates from *CVS* which uses this as label for its main branch.

HTTP

The *Hyper Text Transfer Protocol*, the protocol which is used throughout the Internet to deliver web contents. For *HTTP* there is usually a client, which makes a request, and a server, which responds to the request e.g. by sending data. *HTTP* does not allow to push contents to the client without a request; also the communication between client and server happens always asynchronous.

Intellectual Property (IP)

> *"Property that derives from the work of the mind or intellect, specifically, an idea, invention, trade secret, process, program, data, formula, patent, copyright, or trademark or application, right, or registration."*[126]

ISO 9000 Standard

> *"A family of standards and guidelines for quality in the manufacturing and service industries from the International Organization for Standardization (ISO)."*[127]

Kernel

> The most important part in any operating system; the layer between the software and the actual hardware.

Library

> In Software Engineering a library is a file which contains a piece of compiled machine code which can be used by other programs either on build-time (static libraries) or on run-time (dynamic libraries). A library is a derived object.

Mature Software

> The term "mature" in connection with software implies that the software has been thoroughly tested in the past, is used in production environments and is relatively bug-free. An indication for the maturity of a software project is in Open Source projects usually the version number of the project. If the version number starts with a zero (e.g. "0.12") the software is usually not mature / runs not stable.

Merge

> In SCM the automatic process / algorithm which tries to find and merge together the differences between two revisions of a versioned object. If the process fails, a conflict is created.

Non-Disclosure Agreement (NDA)

> *"An agreement signed between two parties that have to disclose confidential information to each other in order to do business."*[128]

126 http://www.bitpipe.com/tlist/Intellectual-Property.html
127 http://computing-dictionary.thefreedictionary.com/iso+9000
128 http://computing-dictionary.thefreedictionary.com/NDA

Patch

In Software Engineering, a patch is a file with a special format, which records forward deltas based on one or more changed files. The file can then be transferred to another developer which applies the patch on his workspace and recreates the version the first developer created. This process is often used if not every developer gains access to a central repository (e.g. because of missing trust), but can be today circumvented by the usage of *Version Control* tools which use the distributed models.

P2P - Peer-To-Peer

P2P is a technique in which every node (every computer) participating in a network is client and server at the same time. Each node collects data from other nodes and acts as a servant for other nodes needing the acquired data once again.

Proprietary Software

Copyright-protected, closed-source software, owned by a single company or individual.

Proxy (Server)

A proxy server is a computer which arranges the communication between computers inside and outside of a network. Remote computers only see and communicate with the proxy, which directs the data to any requesting local computer. Proxies can act as content caches (e.g. for websites), as filters for malicious requests and more.

Public Domain (PD)

Public Domain means that absolutely no copyright is applied on a certain piece of artwork or software. Everyone is free to do whatever he likes to do with the work, without any restrictions. In some countries, like the UK, PD has no legal status.[129]

Release

A release is the packaging of all files which are needed to build and / or execute a software. This may include the source files, data files, configurations, derived (binary) components etc.

Repository

The data back end of a *Version Control* system which stores all versioned objects of a *Version Control* system. The back end can either use files or a database to store its data.

Revision

A revision is the single state of a single versioned object in a *Version Control* system. Some systems, such as monotone, even refer with this term to the state of the whole repository (all versioned objects).

129 http://computing-dictionary.thefreedictionary.com/public+domain

Sandbox

The local workspace which constitutes one of many states available in a *Version Control* repository. The developer normally makes changes to the sandbox and commits them back to the repository to create a new version.

SSL

Secure Socket Layer, a layer which acts in the OSI stack above the transport and below the application layer, and which enables secure data transfers over public networks.

Tag

A tag is a descriptive mark or label in a repository, which marks a certain version of a software. Versions may have cryptic internal version identifiers applied and these are not very meaningful to a developer, so tags help him / her to recognized important versions much better. Those important versions are versions from the Practical Version Space, i.e. versions which are reasonable and can be built without errors, such as software releases.

Total Cost of Ownership (TCO)

"The cost of using a computer. It includes the cost of the hardware, software and upgrades as well as the cost of the inhouse staff and/or consultants that provide training and technical support."[130]

Trunk

Another term for the main line in a development hierarchy, see *Head.*

Tunnel

A special network technique which allows arbitrary data to be transferred "pickaback" over another, most likely more secure protocol such as SSH. A tunnel has two endings, one on each side, which need to be established first before the actual data transfer can be started.

User Space Programs

Programs which run in User-Space, i.e. without interfering with kernel functionality. Typical kernel functionality is device driver access; if an user space program likes to access a device, it typically makes a system call (syscall) to the Kernel, which returns the requested data.

Variant

A parallel (often conflicting) state of a versioned object, e.g. a revision of a file in a branch.

130 http://computing-dictionary.thefreedictionary.com/TCO

Version

A version marks a common state of all versioned objects in a repository. One can distinguish between Theoretical and Practical Version Space; the first contains all possible versions, while the latter only contains those versions which actually make sense in the development.

Virtual Private Network (VPN)

A VPN offers the connection of two LANs (Local Area Networks) over a WAN (Wide Area Network), such as the connection between two companies. A VPN can be implemented as Tunnel; since VPNs often transfer private data which must not be intercepted by third parties, they are secured.

WebDAV

A standard established by the IETF working group *WWW Distributed Authoring and Versioning* which enables a file system alike access to web resources via the *HTTP* protocol.

List of References

[Perens2006] The Open Source Definition, http://www.opensource.org/docs/definition.php, Bruce Perens, others, 2006, last viewed: 01/25/06

[Rasch2000] A Brief History of Free/Open Source Software Movement, Chris Rasch, 12/26/00, http://www.openknowledge.org/writing/open-source/scb/brief-open-source-history.html, last viewed: 01/26/06

[King1999] Free Software is a political action - In conversation with Richard M. Stallman, J.J. King, 08/18/1999, http://www.heise.de/tp/r4/artikel/6/6469/1.html, last viewed: 01/26/06

[GNU2006] GNU, Multiple Authors, http://en.wikipedia.org/wiki/GNU, 2006, last viewed: 01/27/06

[Linux2006] Linux Kernel, Multiple Authors, http://en.wikipedia.org/wiki/Linux_kernel, 2006, last viewed: 01/27/06

[Licenses2006] Various Licenses and Comments about Them, Jonas Kölker and others, http://www.gnu.org/philosophy/license-list.html, 2006, last viewed: 02/04/06

[Asklund2002] A Study of Configuration Management in Open Source Software Projects, Ulf Asklund, Lars Bendix, 2002, http://www.cs.lth.se/home/Ulf_Asklund/publications/AB02/CM4OSS-s.pdf, last viewed: 01/30/06

[Kotulla2002] Management von Softwareprojekten - Erfolgs- und Misserfolgsfaktoren bei international verteilter Entwicklung, Andreas Kotulla, Deutscher Universitätsverlag, 2002

[Raymond2000] The Magic Cauldron, Eric Steven Raymond, 2000, http://www.catb.org/~esr/writings/cathedral-bazaar/magic-cauldron/, last viewed: 02/08/06

[Negulescu2003] Configuration management, Radu Negulescu, 2003, http://www.macs.ece.mcgill.ca/~radu/304428W02/cm.pdf, last viewed: 02/10/06

[Sink2004] Source Control HOWTO, Eric Sink, 2004, http://www.ericsink.com/scm/source_control.html, last viewed 02/10/06

[Conradi1996] Version models for software configuration management, Reidar Conradi, Bernhard Westfechtel, Technical Report AIB 96-10, RWTH Aachen, 1996

[Perry2005] Managing System Artifacts, Dewayne E. Perry, Introduction to Software Engineering (Lecture 14), 2005, http://www.ece.utexas.edu/~perry/education/360F/L14.pdf, last viewed: 02/18/06

[Fish2006] Version Control System Comparison, Shlomi Fish, 2006, http://better-scm.berlios.de/comparison/comparison.html, last viewed: 02/26/06

Web links to the online documentation of the reviewed Version Control systems:

- CVSNT Online Manual, http://cvsnt.org/manual/html/
- Version Control with Subversion, http://svnbook.red-bean.com/nightly/en/svn-book.html
- monotone documentation, http://venge.net/monotone/docs/index.html

Copyright Notices

License of this Work

This work is licensed under the terms of the *GNU Free Documentation License* (*GNU FDL*) version 1.2, or, at your option, any later version. A copy of the license is available in the chapter GNU Free Documentation License.

License for Wikipedia contents

Resources taken from the *Wikipedia* project (http://wikipedia.org) are licensed under the terms of the *GNU Free Documentation License* (*GNU FDL*) version 1.2. A copy of the license is available in the next chapter.

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies

of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied

verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes

a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new

problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.