# Collaboration and Version Control using CVS

Lars Bendix, sneSCM.org

## 1. Introduction

This is a lab aimed at giving an introduction to concepts and principles in Software Configuration Management exemplified through the use of the version control tool CVS. The most common CVS commands are explained and tried out in order to get familiar with the tool. The first part of the lab is well "routed", giving detailed descriptions of how to perform all steps and what to explore. No particular model of how to use CVS is taught. In the second part of the lab, there will be more scope for exploratory investigations on your own.

There are many reasons to pick up a version control tool and start using it [Ehnbom et al., 2004]. Some of the goals could be:

- to co-ordinate the work of people working in parallel
- to know exactly what is/was released to the customer
- the ability to "roll back" a project if a serious bug in discovered/introduced
- to handle the maintenance of older releases

However, no tool (even CVS;-) will be a substitute for good and clever behaviour. So part of what you get through in this lab is also to figure out *how* to best use a version control tool and *what* the tool will not take care of automagically.

CVS (Concurrent Versions System) started out as a bunch of shell scripts written by Dick Grune [Grune, 1986] for managing his collaboration with a couple of students on a project. These scripts, that allow for concurrent editing of files, were released in 1986. CVS was later redesigned and coded in C by Brian Berliner with a first release in 1989 [Berliner, 1990]. Further design, implementation and maintenance was taken over by the Open Source world [nongnu] and development ceased in 2005/6.

Additional information about CVS can be found in: "An Overview of CVS, Basic Concepts" in "Open Source Development with CVS" by Karl Fogel and Moshe Bar [Fogel et al., 2000]. A manual on CVS can be found at 'http://ximbiot.com/cvs/manual/' (in both pdf and html formats). Also try the 'man-pages'.

This lab has been designed so it can be used in two different contexts or ways – as a 2-hour introductory lab or as a 4-hour more complete lab. If you take it as a 2-hour lab, you should do only the **compulsory tasks** (numbered **C.x**), students doing a 4-hour lab should do also the **optional tasks** (numbered **O.y**). The compulsory tasks will lead you through the most basic version control concepts and principles, whereas the optional tasks will explore more advanced ones and in some cases dig a little deeper into the basic ones.

It would be an advantage if you bring **pen and paper** – you might want to draw pictures of the states of your workspaces and repository (I will for sure when I have to help you out with problems). Bring also **two laptops**: one for doing the lab and another for writing your lab report as you go along (if you have drawings you want to put in the report, use the 4P method: pen, paper, photo (use your smartphone) and paste).

## 2. Scenario

The scenario in this lab is that two developers, together, should develop a program. It is important, that two accounts are used simulating two developers, Calvin and Hobbes. The program developed includes a binary search tree, which will be the code worked on during this lab. The two developers, Calvin and Hobbes, will create a shared repository and two private workspaces – and co-ordinate their modifications to the code using CVS. Now, write down who of you is Calvin and who is Hobbes:

| | |
|---|---|
| **Calvin:** | |
| **Hobbes:** | |

A suggestion would be to organize your lab-windows as indicated in Figure 1. In the upper windows you can browse the repo and the two workspaces and use your favourite editor to make changes to the files in the workspaces – in the lower windows you can use CVS commands to administrate your files. Then it will also be very clear to you when you are in "working mode" (upper windows) and when you are in "administration mode" (lower windows) – and who is Calvin and who is Hobbes. In a normal project you will be in working mode most of the time and only administrate files from time to time. In this lab you will only spend a minimum of time "working" (editing text) – and an "absurd" amount of time administrating (to learn exactly that).

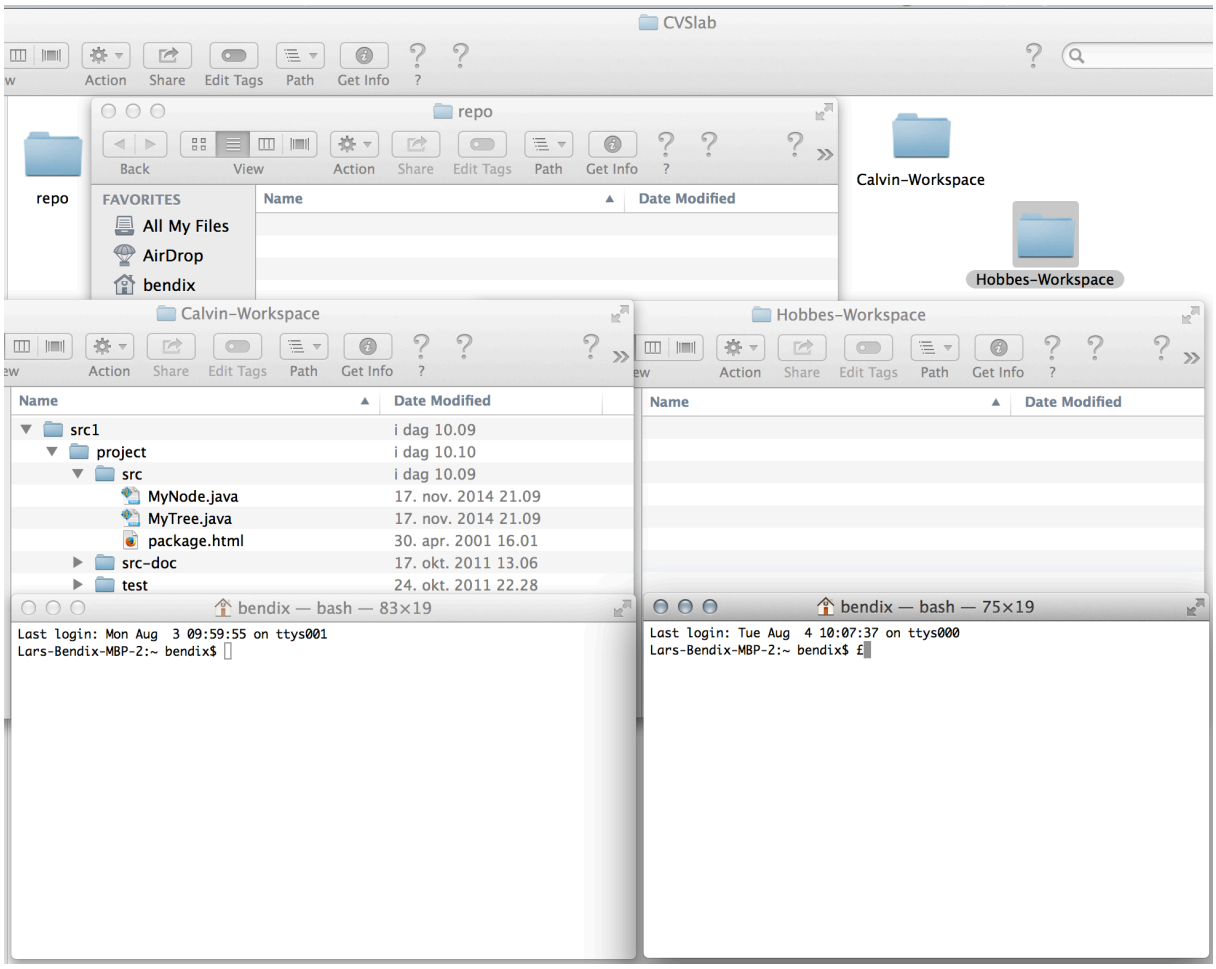


Figure 1. Organization of repo, workspaces and command windows.

## 3. Setting up your project

This lab should be performed in groups of two or three people – so if you are a group of four people, you should split up into two sub-groups. Two "accounts" are used to simulate two developers, Calvin and Hobbes (so if you are three people in a group, the third will be passively observing and actively cheering you on). Both developers will use the same computer, looking at the same screen and taking turns at the keyboard.

Create the folders for the repository (*repo* in Figure 1) and the two workspaces (*Calvin-Workspace* and *Hobbes-Workspace* in Figure1). Move the downloaded (see the course homepage) and unzipped *src1-folder* to Calvin's workspace. Open two command windows – one for each developer. You should now have a setup that looks like the one in Figure 1 – and you are ready to start. There is a place for the repository, there are places for Calvin and Hobbes to work – and Calvin has already done "hours of work" (the files in the src1-folder).

### General observations:

For all the CVS commands you should work in the command-line windows – for all the "programming" (text editing) you can use your favourite editor in Mac, Linux or Windows.

Whenever there are some Unix commands you should use the corresponding Windows command or functionality if you are running Windows.

The javac and ident programs (page 6) will probably not be on your computer, but you should get the idea ;-)

Note that you will ***never*** change any of the files in the repository directly! All changes made to the source code are made in the private workspaces, and files in the repository are updated by CVS and CVS only.

# 4. CVS in a few lines...

Most SCM tools, including CVS, use a database to store all revisions of the files managed. This database is often called a **repository** (but other terms like vault and depot are also used). From this repository users can check out a project (files and directories) into their own local workspaces. All work is then made within the **workspace** (sometimes called a sandbox). Changes made in a local workspace can later be committed to the repository to make them visible to other users. When committing changes new revisions of the involved files will be added to the repository (rather than overwriting the existing).

CVS is a command based Unix tool. Graphical layers on top of CVS also exist, for example tkCVS that you will get familiar with in the second part of this CVS lab. You can find the CVS commands either in the manual, or using the cvs help from a Unix shell:

cvs --help-commands or cvs -H <command>.

The syntax of a CVS command in a Unix shell is:

cvs [cvs-options] cvs-command [command-options] [command-argument]

Examples of common CVS commands are:

- **cvs update** – updates the workspace
- **cvs commit** – checks in a file, a file tree, or a project

- **cvs status** – gives an overview of the differences between the workspace and the repository
- **cvs diff** – views the difference between two versions of a file
- **cvs log** – shows the history of a file or a group of files

- **cvs add** – adds a file to the repository
- **cvs remove** – removes a file from the repository
- **cvs tag** – sets a tag on a file, a file tree, or a project
- **cvs release** – secure removal of a directory in the workspace
- **cvs checkout** – checks out a file, a file tree, or a project to the workspace

- **cvs init** – initiates the repository
- **cvs import** – imports files to the repository

Try to ask for "help" on some of these commands.

# 5. Getting ready to work

Start by telling CVS where to find the repository. If you are running Unix/Linux/Mac you do it this way:

{Calvin}: export CVSROOT=/Users/…some path…/CVSlab/repo

and if you are running Windows it looks slightly different:

{Calvin}: set CVSROOT=C:\CVSlab\repo

Hobbes must also tell his CVS-program where to find the repository (when working in his command-line window). Even if you work on the same computer, Calvin and Hobbes are working in two different shells/windows and commands executed in one shell/window will not have effect on the other.

In this first part, Calvin will be the active part and Hobbes will curiously investigate what happens.

### Task C.1 Create the repository

Calvin starts by initializing the repository that will be used. The CVS command **init** turns the directory pointed to by the CVSROOT into a CVS repository (and if the directory does not exist it will create the directory too):

```
{Calvin}: cvs init
```

Let Hobbes see what happened in the directory that is pointed to by the CVSROOT (i.e. the repository). Don't touch anything – just look and admire ;-)

Calvin should be responsible for "loading" the first version of the project into CVS since he has made the initial prototype. To minimize the programming (=writing) effort during this exercise, the initial version of the source code was copied to Calvin's workspace from a lab-directory (src1.zip), simulating many hours of hard work.

**Bring all the files into the repository**

The CVS command **import** is used to import new files or entire directory structures into the repository. Now Calvin will import the contents of the current directory (so the CVS command should be executed from the src1 directory) into the repository that is defined by his CVSROOT.

```
{Calvin}: cvs import BST Klein start
```

The tag 'Klein' is a so-called vendor tag and 'start' is a release tag. In this example they have no function, but must be included anyway (not optional) – and we will return to them later on (Task C.21).

Now, Calvin has created a repository and imported the initial version of the source code. In the repository the root directory of this project is named BST (Binary Search Tree, in this case). The CVS terminology for a project is "module". Let Hobbes look at what has happened in the repository – again, don't touch anything! Did you find the "xxx,v" files? What are they? Would CVS be happy if you changed any of them?

Let Calvin delete all files in his workspace. They are now safely stored in the CVS repository. Let Hobbes take a look if Calvin doesn't trust it ;-)

## 6. Doing your daily work

Calvin and Hobbes will now switch roles, so Hobbes becomes the active person and Calvin the one to curiously investigate how – and what – things change. We will now simulate that Hobbes makes some improvements to the code.

### Task C.3 Create workspaces

First, both Calvin and Hobbes will check-out the whole BST project to their respective workspaces. This is done using the CVS command **checkout**. Execute the command from Calvin's WS directory and from Hobbes' WS directory:

```
{Calvin}: cvs checkout BST
{Hobbes}: cvs checkout BST
```

The command works recursively checking out the entire directory tree structure rooted at BST. You just use this command when you start working on a project – if you already have the files in your workspace and just need to update them, you use – the update command (more about that later). The actual source code is found in the 'src' directory. Test programs (unit tests) are found in 'test', and JavaDoc documentation is found in 'src-doc' and 'test-doc' respectively. Browse quickly through it if you are curious, but do not modify anything.

Calvin has noticed that in every folder of the project (in the workspace) there is now a folder named "CVS" – what is in that folder and what is the use? Would CVS be happy if you changed any of those files?

**Task O.2 (Unique identification):** Hobbes has heard that there is a way for CVS to create unique identification of files – even after they have been compiled into binaries. He adds an attribute to both classes in the src directory (i.e. in the files 'MyNode.java' and 'MyTree.java'):

```
private static String cvsID="$Id$";
```

CVS will later replace the string $Id$ with more detailed information about the revision, which will also be compiled into the class files. Thus, this makes it possible to identify the revisions used to produce a class file (using the Unix command ident).

Hobbes has now changed the two java files and saved the changes in his workspace. He will move on to other tasks before picking up this task again later.

## Task C.5 Querying the status of files in the workspace

The CVS **status** command returns the status for a file – or all the files in the current directory and below:

```
{Hobbes}: cvs status
```

What is the result when Hobbes queries the status? What is the result when Calvin queries the status? Why is there this difference?

A file within the workspace is in one of the following states – during the rest of the lab you should be able to see cases of all states:

- *Up-to-date* – The file is identical to the latest version in the repository.
- *Locally modified* – The file is modified within your workspace. Your changes have not yet been committed to the repository.
- *Needs patch* – The file has been changed by someone else and committed to the repository. I.e. the version in your workspace is older and needs to be patched (brought up to date).
- *Needs merge* – The file has been changed by you in your workspace, and a new version has been created in the repository by someone else. I.e. the file has been changed in parallel by you and another developer and the changes need to be merged.
- *File had conflicts on merge* – CVS has made a merge of the file as a consequence of an update command, but it contains conflicts that must be resolved manually.

Besides the information about the status of each file, the command gives a lot of additional information. What does the line “Working revision” mean? What does the line “Repository revision” mean?

## Task C.7 Seeing what has changed

The CVS **diff** command allows you to view the actual changes made to the files since they were checked out (i.e. comparing the current workspace version with the version in the repository):

```
{Hobbes}: cvs diff MyNode.java
```

What does Hobbes see when he asks for a diff on the files MyNode.java and MyTree.java? What does Calvin see when he asks for a diff on the files MyNode.java and MyTree.java? Why is there this difference?

We will pick up on exactly how diff behaves in different situations later – and move on to something else.

## Task C.9 Commit changes to the repository

Hobbes will now **commit** his changes to the repository. It is most convenient to let CVS keep track of which files have been changed and need to be committed. This is done by committing the entire project, rather than individual files. CVS then commits changed files only. Move to the BST directory and commit the whole project:

```
{Hobbes}: cvs commit
```

Note that CVS does *not* remove the files from the workspace on commit. The normal work process is to continue to work and repeatedly commit whenever a task is finished.

What valuable information does CVS provide for the commit message? Why is this information so valuable? Why do most people make it go away?

**Task O.6:** There is a “–m” option to the commit command – why should you never use that?

Now, let both Calvin and Hobbes check status for the files again. Are they as expected? What is the status for MyNode.java – for Calvin, for Hobbes?

If for some reason you want to remove all the files in your workspace, you can do it by either using the normal Unix rm command or by using the 'cvs release -d' command. The latter is to prefer, since CVS also checks that files have been committed before removing them. However, do *not* remove the files.

## Task C.7 (continued) Seeing what has changed

We will now continue to explore the potential of the **diff** command. Now Hobbes has committed his changes to the repository – and Calvin has seen (using the status command) that some of his files "needs patch". So Calvin goes on to use the diff command to see what is changed:

```
{Calvin}: cvs diff MyNode.java
```

What happens? Why does this happen (and "CVS stinks" is not the right answer)? What happens if Hobbes runs the diff command on MyNode.java? Is that surprising? Let Calvin make a small change to a comment in MyNode.java – and then run the diff command again. What does it show? Now you should be ready to answer this question: exactly what is it that the diff command compares?

**Task O.2 (continued) (Unique identification):** When Hobbes looks at the source code of his two files (MyNode.java and MyTree.java), he will see that CVS has "expanded" the attribute he added to the files ($Id$). This information about the revision can be extracted from both source code files and compiled files using the program '**ident**'. Hobbes could now compile the two files (if he has a Java compiler on his computer):

```
{Hobbes}: javac *.java
```

Or he could more simply use the already compiled file MyNode.class. He then checks that the ident program actually works as promised (if present) – both for text files and for binary files:

```
{Hobbes}: ident *.java
{Hobbes}: ident *.class
```

If Hobbes does not have an ident program, then he can look at MyNode.class (and MyNode.java) in a text editor and try to find the unique identification – after which he would probably appreciate if ident had been part of his environment.

During the rest of this lab you will not compile the code. It is *not* important for this course that you write code – and we trust that you could write good code and tests if you really had to.

## Task C.11 Adding and removing files

No project is static and from time to new files will be needed or some files will become superfluous. The **add** command can be used to add new files to the repository and the **remove** command to delete files from the repository.

Hobbes wants a new file "README.txt" that explains how to use the two classes MyNode and MyTree – and he doesn't like the file "package.html". With his favourite editor he creates the file "README.txt" and writes some text. He now turns to CVS to add the new file:

```
{Hobbes}: cvs add README.txt
```

What does CVS say about that? What is the status of README.txt in Hobbes' workspace? What will Calvin see if he does a status?

Now it is time to get rid of the unwanted file:

```
{Hobbes}: cvs remove package.html
```

What does CVS say about that? Now Hobbes should do as CVS instructs him to – and then do the cvs remove again. What does CVS say about that? What is the status of package.html in Hobbes' workspace? What will Calvin see if does a status?

Now Hobbes is finally happy and will commit his changes. What will Hobbes expect to see if he runs status? What will Calvin see now if he runs a status for his workspace? Was that surprising? We will see later that CVS actually behaves correctly when Calvin tries to synchronize his files with the repository.

**Task O.8:** It happens that you regret having removed a file. How can you get the file back – if you regret before you commit, if you regret after you did the commit?

### Task C.13 Following the logical history of a file

Come so far, Calvin has become curious about what has happened to the file MyNode.java and why changes were made. The CVS command **log** shows both information about who did changes and when – and the text Hobbes wrote (hopefully) when he did the commit:

```
{Calvin}: cvs log MyNode.java
```

Is the information that Hobbes wrote on the commits useful for Calvin?

**Task O.10:** There is no rename operation in CVS – how would you recommend that people carry out renaming of files?

### Task C.15 Gain confidence

Continue to let Hobbes make small changes until both Calvin and Hobbes feel comfortable with the way the commands status, diff, log and commit work.

### Task C.17 Getting the latest changes from the repository

Now that Calvin has been looking at how the repository has changed due to the work that Hobbes has done, he is eager to get those changes into is own workspace. That would take him to the bleeding edge of the development. The **update** command in CVS will synchronize a workspace with the latest changes from the repository. It is possible to update single files, but it is both safer and easier to just update everything. So from the BST directory Calvin does:

```
{Calvin}: cvs update
```

What does CVS tell Calvin? What does that mean? What will Calvin see if he does a status now? Let Hobbes do an update too – what does CVS tell Hobbes? What does that mean?

## 7. Co-ordinating with other people

Calvin and Hobbes now become "equal partners" – both will make changes and both will curiously investigate what happens in workspaces and repository. They will concurrently modify the common code, and different situations will occur in order to see how CVS (and its users) manage them.

There exist different policies on how files may be checked out from and committed to the repository. A common policy is to lock a file when it is checked out by someone, prohibiting others from checking out and working on the same file. This is not suitable for development situations (maintenance?) where people would want to work on the same files from time to time. There is a more flexible policy called copy-modify-merge. This policy lets several users retrieve the same files from the repository and perform changes in their local workspaces. When the tool discovers concurrent changes it will try to merge the involved files. Copy-modify-merge is illustrated in figure 1.

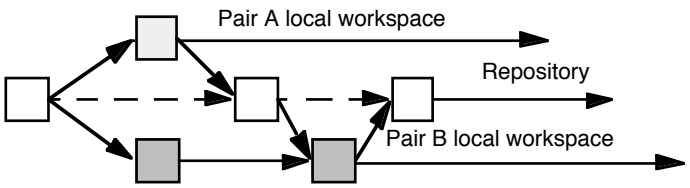


*Figure 1. Copy-Modify-Merge*

**IMPORTANT!** The CVS command update (and checkout too) modifies the files within the workspace. Therefore, you should never have any files "loaded" in any text editors when executing this command. A file

within an editor will not be updated and if saved, the old (loaded) version will overwrite the newer version just fetched from the repository (an example of the "simultaneous update" problem [Babich86]).

To make sure that we get a "clean start" Hobbes will start with a commit, so we are sure that he has no uncommitted changes floating around.

## Task C.19 Parallel development

**Scenario 1:**
Calvin's first task is to remove some unnecessary code from the class MyNode in MyNode.java. The attribute 'father' and corresponding methods 'getFather' and 'setFather' should be removed. Do that.

Calvin will now commit his changes to the repository. Let both Calvin and Hobbes check status for the files again. Are they as expected? What is the status for MyNode.java?

Now Hobbes modifies the text of one of the comments in MyNode.java. Check status of both Calvin's and Hobbes' workspaces? Let Hobbes try to commit his changes – what happens? Hobbes should do as CVS instructs you to and then try to commit again – what happens now? Look at the file MyNode.java and see what has happened to it? Why is this "good behaviour" and which of Babich's problems is CVS trying to help us with?

Well, Hobbes is happy with the result and commits it to the repository. Calvin does a status and sees that there is something new – he cannot resist and does an update to get this exiting new stuff.

**Scenario 2:**
Calvin now goes on to make changes to both MyNode.java and MyTree.java in the beginning of the files. In parallel Hobbes make a new change to MyNode.java towards the end of the file. Hobbes is happy with his work and commits it.

Let both Calvin and Hobbes check status for the files. Are they as expected? Now let Calvin try to commit his changes – what happens? Why is this "good behaviour"? What could happen if CVS did not try to protect us? Calvin does as instructed and then commits. Hobbes wants to stay tuned, so he does an update.

**Scenario 3:**
Hobbes finds that there is a small imperfection in MyNode.java – he changes it and commits. At the same time Calvin has been working on adding a few lines of text to MyTree.java.

Let both Calvin and Hobbes check status for the files. Are they as expected? Now let Calvin try to commit his changes – what happens? Why is this "good behaviour"? What could happen if CVS did not try to protect us? Calvin does as instructed and then commits. Again Hobbes wants to stay tuned, so he does an update.

**Scenario 4:**
Now, both Calvin and Hobbes are very unhappy about the name of the class MyTree – it is too anonymous and they both want to make it their own. So Calvin changes the name of the class (not the file) to CalvinsTree – and commits his change. Hobbes is also working on changing the name of the class – in his case to HobbesTree.

Let both Calvin and Hobbes check status for the files. Are they as expected? Now let Hobbes try to commit his changes. What do you expect to happen – does that happen? By now Hobbes has learned to always do as CVS instructs him to. What do you expect will happen now – does that happen? What did CVS tell about the file MyTree.java?

Take a look (don't change anything) at the file MyTree.java to see what CVS has done to it. What does that mean? If you had written proper code, would the file then compile? Hobbes decides that he doesn't care and tries to commit now that he has updated – what happens? Hobbes fixes the problem and commits – Calvin also wants to be in on the latest and does an update.

**Scenario 5:**
Now, both Calvin and Hobbes will make some changes to the same line in the beginning of the file MyNode.java and another line towards the end of the same file. This time Hobbes is the first to commit. When Calvin then tries to commit his changes CVS complains. Calvin does as instructed.

Now Calvin opens MyNode.java with his editor and fixes the merge conflict in the beginning of the file – but "forgets" that there is also another conflict towards the end. He saves the result and tries to commit. What happens? Would you characterize that as "good behaviour"? What would you have preferred CVS to do?

**Task O.12:** If people wait an awful long time before they finally do an update, they might get a gigantic merge conflict that they are not able to fix. Instead they want to have their original file back from before the merge – is that possible?

**Task O.14: Gaining experience.** Repeat making changes, status, diff, updates and commits until you feel comfortable with the way CVS works and detects conflicts.

After this part of the lab, you should be familiar with the most basic ways of using a version control tool – and might want to take a break. But before you do that, you should do Task C.20.

### Task C.20 Un-committed changes

Before logging out from and closing down your computer you want to check whether you have any un-committed changes in either Calvin's or Hobbes' workspaces. How do you do that? If there are un-committed changes and you do not commit them before logging out and closing down the computer, will they still be there the next time you start up your computer and log in? Why/why not?

## 8. The necessities and the goodies

The remaining part of the CVS lab is not 'routed' in such detail as the first part was. Instead we want you to explore some more of the functionality of CVS within three problem domains. We will ask you some questions, which you will answer by first creating a relevant scenario, try it out in practice, and then write down the answer.

The remaining part of the lab consists of five main areas:

- *Creating a release* – remembering the state of the repository at some specific point in time.
- *Gaining more experience* – playing around, also simulating further development after the release.
- *Maintain an old release* – create a branch from an old release in which we maintain the old release in parallel with the development to a new release. We use merge to implement the same change to both branches.
- *Awareness* – we explore the CVS functionality watch/edit and notification used to increase awareness between developers.

You will continue to work with the repository and workspaces created in the first part, using the same accounts. You will also continue to simulate the two developers, Calvin and Hobbes, working in their workspaces. Remember to set CVSROOT again (it is forgotten when you close the terminal).

Both Calvin and Hobbes should check out "clean" workspaces.

### Task C.21 Create a release

Calvin is now satisfied with the project and wants to send it to the customer – but first he wants to make sure that he can always return to exactly the current configuration. The CVS command **tag** can be used to put a 'tag' (a symbolic name) on the current version of all files included in the BST project. Since this is the first release Calvin does he will call is "release1":

```
{Calvin}: cvs tag release1
```

Will the command **status** help you to see which version of each file got tagged? If not, then try the command **log** – or play around with status. Check which versions of each file got tagged.

Are you sure that you know how CVS decides which version to tag? If not, then make a couple of small experiments.

First, Calvin makes a small change to one of the files and commits his changes. Hobbes does not make an update, but decides that he wants to make a release (Calvin forgot to tell him that he already made a release). So Hobbes tags everything with the tag "HobRel1":

```
{Hobbes}: cvs tag HobRel1
```

Which version of the files do you expect got tagged? Use log to check if you were right or not.

Second, Calvin makes another small change – and decides that it is about time for release 2 of the system. So he proceeds to tag everything "release2":

```
{Calvin}: cvs tag release2
```

He now remembers that he did not commit his change – so he does that.

Which version of the files do you expect got tagged? Use log to check if you were right or not.

Now, how does CVS decide which version to tag? Is that a good decision? In which situations? Is it a bad decision? In which situations?

Calvin is lucky. The customer has erased the release that Calvin sent him and wants to have a new copy of release 1. Now, how does Calvin get back to release 1? Hint: he wants to update his workspace – and maybe there is something in "cvs –H update" that can help him? What is the meaning of a "sticky tag"?

**Task O.16:** As you discovered last week – and probably also noticed this week – CVS places a CVS directory in all the directories in the workspace. Most probably our customer does *not* want to have these directories, but just the proper files. Now, is there an easier way to "export" a clean release to the customer than manually deleting all the CVS directories?

In preparation for the next task, Hobbes will make a number of changes and commits to different files.

## Task C.23 Maintaining an old release

To maintain an old release (or many old releases) in parallel with the development for a new release is a very common situation. The problem we face is that we cannot possibly avoid the 'double maintenance problem'. You will now (by yourself) create such a scenario and use CVS to help you as much as possible.

Both Calvin and Hobbes continued to develop the 'product' after the release Calvin did in task C.21, i.e. changing some of the files in the main branch.

Now, Hobbes should create a new branch originating from the release tag and make some changes (simulating fixing a bug). *Hint:* A branch is created by setting a branch tag on the version that you want to branch from. Type `cvs -H tag` for help. Try to draw the situation you want before executing the CVS commands (pen&paper rules!).

Calvin now discovers that Hobbes fixed a bug in the maintenance branch – and he want to have it removed on the main trunk also. How could that be done? Why is "manual fixing" not "good behaviour"? How can – and should – you merge the bug fix(es) from the branch to the main trunk with the help of CVS? *Hint:* You would like to update the main trunk with the changes from the branch. Type `cvs -H update` for help. Try to draw the situation you want before executing the CVS commands (pen&paper rules!).

**Task O.18 (Merge tracking):** Calvin makes a few more changes and commit to the main trunk. Hobbes makes a few more bug fixes that are committed to the branch. Hobbes tells Calvin that there are more bug fixes ready – and Calvin happily tries to merge them into the main trunk now that he knows how to do that. What happens? Why does that happen? What can be done to fix it now? What should Calvin have done when he merged the first time?

Finally Hobbes is done fixing bugs on the maintenance branch and has been assigned some new development on the main line. How does he get back to the main line in an easy way?

**Task O.20:** Right in the middle of implementing a new feature the boss comes storming in and yells that there is critical security bug that needs to be fixed NOW – and he tells Hobbes to do that as Hobbes is used to fixing bugs. Hobbes asks if he can finish the implementation of the new feature before he starts fixing the bug – NO! Is there a way that Hobbes can save his work on the feature – and pick it up later when he is done with the bug?

## Task C.25 Awareness

CVS implements the copy-merge model which makes it possible to change any file you want in order to implement your change. However, even though it is possible, you do not want to end up with a tricky merge situation later when you have to update. To avoid this it is possible to increase the awareness of what other developers are doing. In this lab you will try out two ways of doing that: watch and notification.

Below is an extract from Cederqvist's manual describing both watch and notification:

> *10.6 Mechanisms to track who is editing files*
> *For many groups, use of CVS in its default mode is perfectly satisfactory. Users may sometimes go to check in a modification only to find that another modification has intervened, but they deal with it and proceed with their check in. Other groups prefer to be able to know who is editing what files, so that if two people try to edit the same file they can choose to talk about who is doing what when rather than be surprised at check in time. The features in this section allow such coordination, while retaining the ability of two developers to edit the same file at the same time. For maximum benefit developers should use* cvs edit *(not* chmod*) to make files read-write to edit them, and* cvs release *(not* rm*) to discard a working directory which is no longer in use, but CVS is not able to enforce this behavior.*
>
> *10.6.1 Telling CVS to watch certain files*
> *To enable the watch features, you first specify that certain files are to be watched.*
>
> cvs watch on [-lR] files . . .
>
> *Specify that developers should run* cvs edit *before editing files. CVS will create working copies of files read-only, to remind developers to run the* cvs edit *command before working on them.*
> *If* files *includes the name of a directory, CVS arranges to watch all files added to the corresponding repository directory, and sets a default for files added in the future; this allows the user to set notification policies on a per-directory basis. The contents of the directory are processed recursively, unless the -l option is given. The -R option can be used to force recursion if the -l option is set in '~/.cvsrc' (see Section A.3 [~/.cvsrc], page 88).*
> *If* files *is omitted, it defaults to the current directory.*
>
> cvs watch off [-lR] files . . .
>
> *Do not create* files *read-only on checkout; thus, developers will not be reminded to use* cvs edit *and* cvs unedit*. The files and options are processed as for* cvs watch on*.*
>
> *10.6.2 Telling CVS to notify you*
> *You can tell CVS that you want to receive notifications about various actions taken on a file. You can do this without using* cvs watch on *for the file, but generally you will want to use* cvs watch on*, to remind developers to use the* cvs edit *command.*
>
> cvs watch add [-a action] [-lR] files . . .
>
> *Add the current user to the list of people to receive notification of work done on files. The* -a *option specifies what kinds of events CVS should notify the user about.* action *is one of the following:*
> edit *Another user has applied the* cvs edit *command (described below) to a file.*
> unedit *Another user has applied the* cvs unedit *command (described below) or the* cvs release *command to a file, or has deleted the file and allowed* cvs update *to recreate it.*
> commit *Another user has committed changes to a file.*
> all *All of the above.*
> none *None of the above. (This is useful with* cvs edit*, described below.)*
>
> *The* -a *option may appear more than once, or not at all. If omitted, the action defaults to all.*
> *The files and options are processed as for the* cvs watch *commands.*
>
> cvs watch remove [-a action] [-lR] files . . .

*Remove a notification request established using* cvs watch add*; the arguments are the same. If the* -a *option is present, only watches for the specified actions are removed.*

*When the conditions exist for notification, CVS calls the notify administrative file. Edit notify as one edits the other administrative files (see Section 2.4 [Intro administrative files], page 16). This file follows the usual conventions for administrative files (see Section C.3.1 [syntax], page 133), where each line is a regular expression followed by a command to execute. The command should contain a single occurrence of '*%s*' which will be replaced by the user to notify; the rest of the information regarding the notification will be supplied to the command on standard input. The standard thing to put in the notify file is the single line:*

```
ALL mail -s "CVS notification" %s
```

*This causes users to be notified by electronic mail.*

*Note that if you set this up in the straightforward way, users receive notifications on the server machine. One could of course write a notify script which directed notifications elsewhere, but to make this easy, CVS allows you to associate a notification address for each user. To do so create a file* users *in* CVSROOT *with a line for each user in the format* user:value*. Then instead of passing the name of the user to be notified to* notify*, CVS will pass the value (normally an email address on some other machine).*

*CVS does not notify you for your own changes. Currently this check is done based on whether the user name of the person taking the action which triggers notification matches the user name of the person getting notification. In fact, in general, the watches features only track one edit by each user. It probably would be more useful if watches tracked each working directory separately, so this behavior might be worth changing.*

*10.6.3 How to edit a file which is being watched*
*Since a file which is being watched is checked out read-only, you cannot simply edit it. To make it read-write, and inform others that you are planning to edit it, use the* cvs edit *command. Some systems call this a checkout, but CVS uses that term for obtaining a copy of the sources (see Section 1.3.1 [Getting the source], page 4), an operation which those systems call a get or a fetch.*

```
cvs edit [options] files . . .
```

*Prepare to edit the working files files. CVS makes the files read-write, and notifies users who have requested edit notification for any of files. The* cvs edit *command accepts the same options as the* cvs watch add *command, and establishes a temporary watch for the user on files; CVS will remove the watch when files are unedited or committed. If the user does not wish to receive notifications, she should specify* -a none*. The files and options are processed as for the* cvs watch *commands.*

*Normally when you are done with a set of changes, you use the* cvs commit *command, which checks in your changes and returns the watched files to their usual read-only state. But if you instead decide to abandon your changes, or not to make any changes, you can use the* cvs unedit *command.*

```
cvs unedit [-lR] files . . .
```

*Abandon work on the working files* files*, and revert them to the repository versions on which they are based. CVS makes those files read-only for which users have requested notification using* cvs watch on*. CVS notifies users who have requested* unedit *notification for any of files. The files and options are processed as for the* cvs watch *commands. If watches are not in use, the* unedit *command probably does not work, and the way to revert to the repository version is to remove the file and then use* cvs update *to get a new copy. The meaning is not precisely the same; removing and updating may also bring in some changes which have been made in the repository since the last time you updated.*

*When using client/server CVS, you can use the* cvs edit *and* cvs unedit *commands even if CVS is unable to successfully communicate with the server; the notifications will be sent upon the next successful CVS command.*

*10.6.4 Information about who is watching and editing*

cvs watchers [-lR] files . . .

*List the users currently watching changes to* files*. The report includes the files being watched, and the mail address of each watcher. The files and options are processed as for the* cvs watch *commands.*

cvs editors [-lR] files . . .

*List the users currently working on* files*. The report includes the mail address of each user, the time when the user began working with the file, and the host and path of the working directory containing the file. The files and options are processed as for the* cvs watch *commands.*

**Lab exercise for Watch**
Set watch on all files. To get the correct Unix access rights you have to release (remove, i.e. option -d) the workspace and check it out again. Explore how it works. What happens if you ignore that another developer edits a file and 'edit&change' it anyway? Also try the command CVS editors.

**Lab exercise for Notify**
If you had a mail-server running (which you probably haven't on your own laptop) and you had all the time in the world (which you don't) – you could set notification on e.g. commit and have CVS send you an email. This is what commercial companies (and Open Source projects) did when they used CVS. **You can limit yourself to read the manual** – and admire what a simple tool like CVS could actually do to help developers coordinate.

**Note:** You should never make changes directly to the files in the repository! To modify files in the CVSROOT directory, check out the entire directory, make the changes and commit – exactly as with any other directory.

# 9. Afterwords

Now you know enough about CVS (and version control) to make you dangerous ;-) CVS has a couple of "inconveniencies" that have been fixed in more modern systems like Subversion, but nothing that cannot be fixed by using some manual bookkeeping together with CVS.

In this lab you may often have worked on a single file to make it more clear to identify exactly what happens and how it works. This is pure convenience and in a "real project"™ you would *always* do commits and updates on the *whole* project.

# 10. References


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

[Berliner, 1990]: Brian Berliner: *CVS II: Parallelizing Software Development*, in proceedings of USENIX Winter 1990, Washington D.C.

[Ehnbom et al., 2004]: Dag Ehnbom, Jon Hasselgren, Anders Nilsson, David Svensson: *The Evaluation of Configuration Management Version Control Tools*, Department of Computer Science, Lund University, June, 2004.

[Fogel et al., 2000]: Karl Fogel, Moshe Bar: *An Overview of CVS, Basic Concepts*, in "Open Source Development with CVS", http://cvsbook.red-bean.com/cvsbook.html#Basic%20Concepts.

[Grune, 1986]: Dick Grune: *Concurrent Versions System, A Method for Independent Cooperation*, IR 113, Vrije Universiteit, Amsterdam, Holland, 1986 (available from: http://dickgrune.com/Programs/CVS.orig/)

[nongnu]: http://www.nongnu.org/cvs/