Collaboration and Version Control using CVS

Lars Bendix, sneSCM.org

1. Introduction

This lab will give you an introduction to important concepts and principles in Software Configuration Management exemplified through the use of the version control tool CVS. Version control tools started out in the 1970s as rudimentary *version control*. To be able to follow the evolution of a file over time; to snapshot its state at a specific point in time; to recover and restore a previous version of a file if necessary. However, over time a lot of functionality was added to allow the *collaboration* between developers and address the three co-ordination problems identified by Wayne Babich [Babich, 1986].

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]. In the 1990s and 2000s CVS was the most widely used version control tool in small-medium enterprises and the Open Source world, who took over the further design, implementation and maintenance of CVS [nongnu]. The development on CVS ceased in 2005/6, but it is still an excellent teaching tool with a friendly learning curve (simple things are simple, complex things are more difficult). 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] – or try out the 'man-pages'.

This lab has been designed so it can be used in two different contexts or ways – as a 2-3 hour *introductory lab* or as a 4-5 hour more *complete lab*. If you take it as a 2-3 hour lab, you should do only the **compulsory tasks** (numbered **C.x**), students doing a 4-5 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. Setting up things

The underlying scenario in this lab is that two developers collaborate to 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:	

You should organize the overall structure of the lab as indicated in Figure 1. First create a folder for this lab – call it CVSlab. If you are on a Windows machine, you place the CVS executable in the CVSlab folder (and set the right path so Windows will find it). You then create three new folders in CVSlab: one that is the global, shared Repository, one that is the local workspace where Calvin will have the files he works on, and one that is the local workspace where Hobbes will have his files. In a real setup, the Repository would be on a server somewhere, Calvin would have his workspace on his own computer and Hobbes would have his workspace on a different computer. However, using this setup with everything on one computer we avoid having to deal with servers and internet connections.

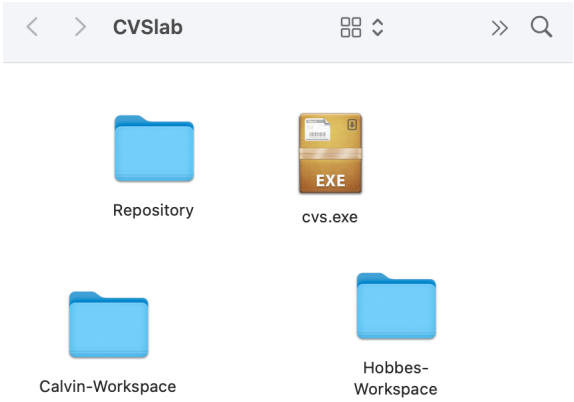

Figure 1. Organizing the folders.

Now Calvin gets prepared to work, he opens the Repository folder (so he can browse the Repository and see what files are there and what they contain – normally he wouldn't care, but for this lab he is curious). He then opens the Calvin-Workspace folder, so he can see what files he has locally and can easily open them when he wants to edit them. He now downloads the zip-file with the code to work on to the folder Calvin-Workspace and un-zips it (and removes the zip-file).

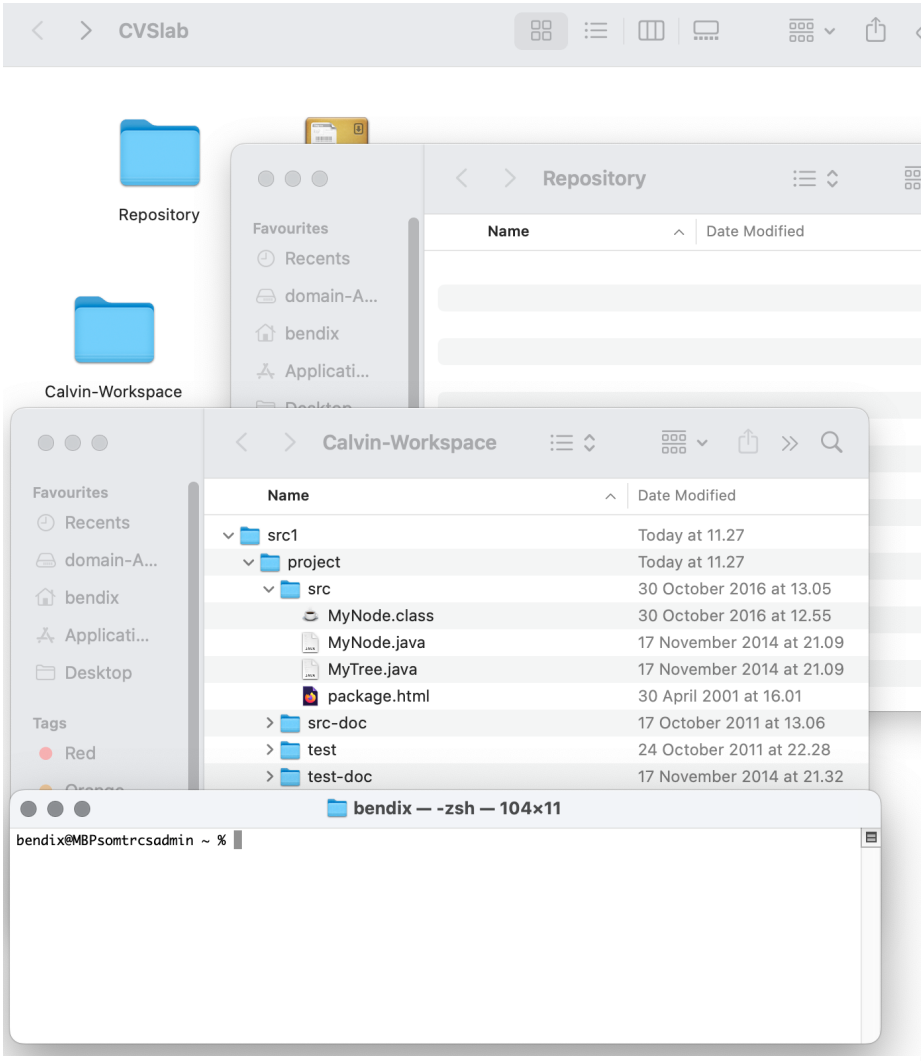

Figure 2. Organizing Calvin's working environment.

Finally he opens a terminal where he can execute the CVS commands when he wants to administrate the files. This should now look like in Figure 2. This should make it very clear to you when you are in "working mode" (upper window) and when you are in "administration mode" (lower window) – 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). Further on when Hobbes joins the project, he will open the Hobbes-Workspace folder and his own terminal to create his own personal working environment. Doing this on the same computer will make it easy for Calvin and Hobbes to follow also what happens in the other person's working environment.

This lab should be performed in small groups of two or three people. 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 commenting and cheering you on). Both developers will use the same computer, looking at the same screen and taking turns at the keyboard.

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.

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

3. CVS in a few lines…

Most SCM tools, including CVS, use a global, shared 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, private **workspace** (sometimes called a sandbox). All work is then made within the workspaces. Changes made in a 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. There used to be many different GUI-clients for CVS, but they are not around anymore. Furthermore, in this lab you will use the command line when you interact with CVS, which in my opinion is far better than using a GUI client (spend a minute or two to figure out why). 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. You will definitely have to do that later for specific commands to learn what options are available for that command and what they do.

4. 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=/…some path…/CVSlab/Repository

and if you are running Windows it looks slightly different:

{Calvin}: set CVSROOT=C:\…some path…\CVSlab\Repository

When Hobbes joins the project later on, he 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/terminals and commands executed in one shell/terminal will not have effect on the other.

In this first part, Calvin will be the active part and Hobbes will curiously observe 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
```

Task O.2 (Contents of a repository): Calvin is curious to 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 is responsible for "loading" the first version of the files in 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 the downloaded zip-file (src1.zip), simulating many hours of hard work.

Task C.3 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 – Create a release).

Task O.4 (More contents of a repository): 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". Calvin is curious to see what has happened in the repository as a result of the import – 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?

Calvin deletes all files in his workspace (simply remove all the files in the folder Calvin-Workspace), so he has something nice and clean to start from. All the files in the project are now safely stored in the CVS repository.

5. Working alone

We will start out by Calvin "working alone" mostly interested in getting information about how the files in his workspace relate to the versions of the files in the repository.

Task C.5 Create a workspace

Calvin will now check-out the whole BST project to his (empty) workspaces. This is done using the CVS command **checkout**. Execute the command from the Calvin-Workspace directory:

```
{Calvin}: 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.

Task O.6 (Administrative files): Calvin now notices 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?

CVS will execute commands in the place where you are when you give the command. So, if you are in the root the status command will give the status for the whole project (a lot of files and a lot of information) – and the diff command will look for the file only in the root folder (and not longer down the paths). It will be easier for you if you always work (and execute CVS commands) in the 'BST/project/src' directory for most of the following tasks.

Task C.7 Querying the status of files in the workspace

Calvin now decides that he wants to add a small comment in both 'MyNode.java' and 'MyTree.java'.

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

```
{Calvin}: cvs status
```

What is the result when Calvin queries the status?

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.9 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):

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

What does Calvin see when he asks for a diff on the files MyNode.java and MyTree.java?

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

Task C.11 Commit changes to the repository

Calvin 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:

```
{Calvin}: 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? What kind of information would you like get from another person's commit?

Task O.8 (Commit option): There is also a "-m" option to the commit command – when should that option be used?

Calvin now checks the status for the files again. Are they as expected?

Task C.13 Status and seeing what has changed (continued)

We will now build on the insights from C.7 and C.9 and continue to explore the potential of the **diff** and the **status** commands. After Calvin has committed his changes to the repository (C.11), it is time for Hobbes to also join the work. Hobbes creates his own workspace the same way as Calvin did in C.3.

Calvin makes some small changes in the beginning of the files MyNode.java and MyTree.java, but does not commit the changes yet. Hobbes decides to make a small change at the end of the file MyTree.java and does not commit the change yet.

Now both Calvin and Hobbes use the status command to get information about the current status of the files in their workspace. Is the status as you would expect?

Since the file MyTree.java is locally modified in both workspaces, both Calvin and Hobbes are curious to see what has been modified and use the diff command to get that information. What does diff tell Calvin? What does diff tell Hobbes? Is that as you would expect?

Calvin now decides that it is time to commit his changes to the repository and does that as in C.11. Since both Calvin and Hobbes know that something has happened, they are curious to know what is now the current status of their files and use the status command to get that information. What does status tell Calvin? What does status tell Hobbes? Is that as you would expect? What does it mean that a file "Needs patch"? What does it mean that a file "Needs merge"?

Hobbes has the suspicion that it was Calvin's changes that turned the status of MyTree.java into "Needs merge" and use the diff command to see what the differences are for MyTree.java. Does the result surprises you? A little puzzled Hobbes also try to see what the differences are for MyNode.java that has the status "Needs patch". Does the result surprises you? Now you should be ready to answer this question: exactly what is it that the diff command compares?

Hobbes decides that he'd better integrate into his workspace the changes that Calvin previously committed to the repository. He does that by using the **update** command that will synchronize a workspace with the latest changes from the repository. So from the BST directory Hobbes does:

```
{Hobbes}: cvs update
```

What does CVS tell Hobbes about the update? What does that mean? Is that what you would expect? What is the status of Hobbes' files now? Inspect both MyNode.java and MyTree.java and see what has changed in the files (Hobbes should remember what he has changed). You should also be ready to answer the question: what is the difference between "Needs patch" and "Needs merge".

Task O.10 (Unique identification): Calvin 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="$Id$";
```

Calvin commits his changes to the repository after which CVS will expand the Id string with some information about the files. This information can be extracted from both source code files and compiled files using the program '**ident**':

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

If the ident command is not available on your system, look at MyNode.java in a text editor and try to find the unique identification from the "$Id … $" string. Then try to look at the file MyNode.class with a text editor (or the command less) if it allows you to read binary files – you should find the "$Id … $" string about 3-5 lines down in the file. After which you would probably appreciate if ident had been part of your environment. What information is there? How could that be useful?

Task C.15 Adding and removing files

No project is static and from time-to-time 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 he does a status?

Now Hobbes is finally happy and will commit his changes. After the commit, 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.12 (Removed files): 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.17 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.14 (Renaming files): There is no rename operation in CVS – how would you recommend that people carry out renaming of files?

6. Co-ordination mechanisms

Calvin and Hobbes now want to explore together the different mechanisms for collaboration in a version control tool. They want to know what mechanisms are present in CVS and understand in detail how they work. To make sure that we get a "clean start" both Calvin and Hobbes will start with a commit, so we are sure that they have no uncommitted local changes floating around. They should also both do an update, so we are sure that they are both up to date with the repository. Check with status that all files are "Up-to-date" for both Calvin and Hobbes.

Task C.19 Concurrency check

First Calvin and Hobbes wants to make sure that CVS is actually a version control tool. A tool that does not have "concurrency check" when you want to commit something to the repository, runs the risk that you will have Wayne Babich' Simultaneous Update problem. We don't want to consider tools that do not protect us from the Simultaneous Update problem as version control tools.

To check that Calvin first makes a change in the beginning of the file MyNode.java and commits it. Then Hobbes also makes a change to the file MyNode.java, but towards the end of the file. If CVS would allow Hobbes to commit his changes directly, it would allow the Simultaneous Update problem since the commit would add a version to the repository (the latest that everyone usually uses) that does not include the changes that Calvin just committed. This is also known as the infamous "return of the zombie bug".

Let us see what happens when Hobbes try to commit his change. What happens? What does CVS tell Hobbes to do? 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? What is the status of MyNode.java for Calvin? Let Calvin do an update to take in the latest version from the repository – and check that it still has his change (and that of Hobbes).

To make sure that we get a "clean start" for the next task both Calvin and Hobbes will do a commit, so we are sure that they have no uncommitted local changes floating around. They will also both do an update, so we are sure that they are both up to date with the repository. Check with status that all files are "Up-to-date" for both Calvin and Hobbes.

Task C.21 Long transactions

Calvin and Hobbes now want to check if CVS supports Long Transactions. The purpose of Long Transactions is to keep a logical change (possibly covering several files) together as one transaction. That means that when we try to commit the logical change, the tool should either commit all the changed files or nothing (abort the commit). The worst possible situation would be if it committed to the repository only the files that we were up-to-date with and left the others in our workspace. That would mean that part of a logical change would end up in the repository and that would not work.

First Calvin makes a change to MyNode.java in the beginning of the file and commits that. Then Hobbes makes a change to both MyNode.java and MyTree.java towards the end of the files. Hobbes is now ready to commit his logical change that spans two files. Let Hobbes check the status of his files.

If the commit in CVS works such that it adds a new version in the repository of MyTree.java (where Hobbes is up-to-date) but leaves the locally modified version of MyNode.java in the workspace (because it misses the concurrency check and cannot be committed), then it would leave the repository in a sort of inconsistent state. Which would not be good.

Let Hobbes commit and see what happens. What does CVS tell Hobbes? Let Hobbes do as CVS instructs him and then commit his changes. Did they go through this time? What is the state of the files for Hobbes? What is the state of the files for Calvin?

Task O.16 (Breaking Long Transactions): Can you break the Long Transaction mechanism in CVS?

To make sure that we get a "clean start" for the next task both Calvin and Hobbes will do a commit, so we are sure that they have no uncommitted local changes floating around. They will also both do an update, so we are sure that they are both up to date with the repository. Check with status that all files are "Up-to-date" for both Calvin and Hobbes.

Task C.23 Strict long transactions

Calvin and Hobbes now want to check if CVS supports Strict Long Transactions. The purpose of Strict Long Transactions is to make sure that we do not "merge into the repository". No sane version control tool would try to merge files that "Needs merge" since it might give a physical merge conflict and we do not want that to end up in the repository. Some tools, however, allow you to commit files that you are up-to-date with even if there are other files (that you have not changed) that you are not up-to date with. "Physically" that is not a problem and does not create any physical merge conflicts in the repository. However, now your changes exist in the repository together with the changes of the files where you are not up-to-date – and no one has tested if they work together. If we want to work in a sane and secure way we do not want to end up in that situation and therefore use a tool that has Strict Long Transactions.

Calvin as usual make a change to the file MyNode.java and commits it. Hobbes makes a change to the file MyTree.java. What is the status for Hobbes' files? Now Hobbes tries to commit his changes – what happens? What does CVS tell Hobbes to do? Hobbes does that – and checks the status of his files. Is Hobbes able to commit now? Why/Why not?

Task O.18 (Breaking Strict Long Transactions): Can you break the Strict Long Transaction mechanism in CVS?

To make sure that we get a "clean start" for the next task both Calvin and Hobbes will do a commit, so we are sure that they have no uncommitted local changes floating around. They will also both do an update, so we are

sure that they are both up to date with the repository. Check with status that all files are “Up-to-date” for both Calvin and Hobbes.

Task C.25 Merge conflicts

Now Calvin and Hobbes want to explore more in detail the merge functionality. They have previously seen that if they make changes in different parts of the same file, the merge functionality has no problem in integrating both changes. Now it is time to see what happens if they change the same line in different ways.

So, Calvin makes a change to line 5 in the file MyNode.java – he adds his name to the beginning of the line. He is happy with the change and commits it. Hobbes gets the same idea, so he also adds his name to the beginning of line 5 in MyNode.java. Hobbes now checks the status of his 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 Hobbes about the file MyNode.java?

Hobbes take a look (don’t change anything) at the file MyNode.java to see what the merge functionality 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 is curious to see what Hobbes did to “solve the merge conflict” and does an update.

Task O.20 (More merge conflicts): 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.22 (“Un-merged” files): 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.24 (Gaining experience): Repeat making changes, status, diff, updates and commits until you feel comfortable with the way CVS works and detects conflicts.

Task C.27 Un-committed changes

Hobbes continues to work and make a couple of changes to both MyNode.java and MyTree.java. He then realises that it is time stop the work for today and go home. Before logging out from and closing down his computer he wants to check whether he has any un-committed changes his workspace. How does he do that?

Hobbes see that there are some un-committed changes (“Locally modified”) but decides to *not* commit them before logging out and closing down his computer. Will they still be there tomorrow morning when Hobbes start up his computer and log in? Why/why not?

7. The necessities and the goodies

Now Calvin and Hobbes want to move on to explore some functionality that is necessary for a team to handle releases and maintenance – and one more thing that is convenient for managing team collaboration.

This final part of the lab consists of three main areas:

- *Creating a release* – remembering the state of the repository at some specific point in time.
- *Maintain an old release* – create a branch from an old release in which we maintain the old release in parallel with the development of a new release. We use merge to implement the same change to both branches.
- *Awareness* – we explore how the CVS functionalities watch/edit and notification can be used to increase awareness between developers.

Both Calvin and Hobbes should start with “clean” workspaces. They can to that either by deleting their workspaces and check out new workspaces, or by committing and then updating.

Task C.29 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 label or 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 options for status. Check which versions of each file got tagged.

Task O.26 (Exploring tag): Are you sure that you know exactly how CVS decides which version to tag? Maybe you should make a couple of small experiments to discover if it is what you have in your workspace that gets tagged, if it is the latest version in the repository that gets tagged, if it is some other version in the repository that gets tagged – or if it is something completely random.

First, Calvin makes a small change to MyNode.java 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 MyNode.java do you expect got tagged? Use log to check if you were right or not.

Second, Calvin makes another small change but does *not* commit – 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 "forgot" to commit his change – so he does that.

Which version of MyNode.java do you expect got tagged? Use log to check if you are right or not.

Now you should be prepared to explain exactly how CVS decides which version to tag? Is that a good decision? In which situations? Is it a bad decision? In which situations?

Task C.31 Getting back to a release

Calvin is lucky. The testers accidentally erased the release that Calvin sent them for testing and want 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.28 (Getting a "clean" release): As you already noticed earlier (Task O.6) CVS places a CVS directory in all the directories in the workspace. Most probably our customer (or the testers) 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?

Task C.33 Working with branches

To maintain an old release (or many old releases) in parallel with the continued development of new releases is a very common situation. Often customers do not want to get (or pay for) the latest version, but want a bug to be fixed in the version they have. Almost equally as common is the situation that we have to create a release, but we need to put some final things right before the release is perfect. So we want a couple of developers to do the perfection, while allowing the rest of the team to continue with the development of the next release instead of sitting idle.

The concept of **branches** is ideal to handle such situations. They allow work to go on in parallel and they allow both the continued development and the maintenance/release effort to add new versions on their own branches. However, the problem we face is that we cannot possibly avoid the "double maintenance problem" – so we need to find out how to manage that problem with the help of CVS.

In this scenario Calvin will create and work on a branch for fixing a few bugs in release1, while Hobbes will continue to develop the product on the main line/branch. Both Calvin and Hobbes should start with "clean" workspaces. They can to that either by deleting their workspaces and check out new workspaces, or by committing and then updating.

Both Calvin and Hobbes now have the latest version of the system in their workspaces. That works perfect for Hobbes, who implements new functionality for the next release. However, it does not work for Calvin, who fixes bugs in release1 since the versions in release1 might no longer be the latest versions. So Calvin has to return to release1. How does he do that? *Hint*: go back to task C.31. Now, Calvin should create a new branch originating from the release1 tag. How does he do that? *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 wants to fix a problem to release1. He does not feel that his work is recognized and changes the first line in MyNode.java to be his name. Calvin then commits the change to the maintenance branch. Did that go smoothly? Follow the "advice" from CVS and try again. Hobbes also has a problem with having his work recognized and decided to change the first line in MyNode.java to be his name – and then commits the change to the main line. Explore the current status of the files in both Calvin's and Hobbes' workspaces.

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

When Hobbes does the merge, CVS tells him that there is a merge conflict in the file MyNode.java. Calvin changed the first line in one way and Hobbes changed it in a different way. CVS does not want to solve the conflict and leaves it to Calvin and Hobbes to work it out. They talk together and decide that since this is a perfect example of the "double maintenance" problem, they want to honour Wayne Babich and puts his name as the first line to resolve the merge conflict. Now that Hobbes has done that he commits the result to the main line.

Task O.30 (Merge tracking): Hobbes makes a few more changes to the beginning of some files and commit to the main line. Calvin makes a few more bug fixes towards the end of some files and commits to the maintenance branch. Calvin tells Hobbes that there are more bug fixes ready – and Hobbes happily tries to merge them into the main line now that he knows how to do that. What happens? Why does that happen? What can be done to fix it now? What could Hobbes have done when he merged the first time?

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

Task O.32 (Switching tasks): Right in the middle of implementing the new feature the boss comes storming in and yells that there is critical security bug that needs to be fixed NOW – and he tells Calvin to do that as Calvin is used to fixing bugs. Calvin asks if he can finish the implementation of the new feature before he starts fixing the bug – NO! Is there a way that Calvin can save his work on the feature – and pick it up later when he is done with the bug?

Task C.35 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 investigate 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 Watch 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 certainly haven't) – you could set notification on e.g. commit and have CVS send you an email every time someone commits. 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 an old, 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.

8. 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 Perforce and git, but nothing that cannot be fixed by using a little 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.

In this lab we have not looked at "access rights", that is who can and cannot edit files. As long as you are a student you probably don't care and are happy to allow everyone to edit anything. However, when you start working in a "real project"™ you might not want the testers to edit the source code, or the programmers to change the test cases – or that the incompetent people programming the graphical user interface to put their dirty hands on your beautiful database code. In CVS (and some other tools) it is difficult (but not impossible) to define different access rights to different people. In other version control tools it seems to be almost their primary purpose and it is quite easy to do.

9. 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", https://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: https://dickgrune.com/Programs/CVS.orig/)

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