

Collaboration and Version Control using Git

Lars Bendix & Christian Pendleton

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 Git. 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 smooth collaboration between developers and address the three co-ordination problems identified by Wayne Babich [Babich, 1986].

CVS (Concurrent Versions System) was one of the first version control tools to also address collaboration aspects. It 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, which 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 community, who took over the further design, implementation and maintenance of CVS [nongnu]. The development on CVS stopped in 2005/6 when the Open Source community started developing Subversion as a new tool that would improve the deficiencies in CVS. However, it turned out that they were "overtaken" by Linus Torvalds when Git [Git] came out in the late 2000s and Git has over the years grown to become the standard for both the Open Source community and many commercial companies.

There are many reasons to pick up a version control tool and start using it. 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 is discovered/introduced
- to handle the maintenance of older releases

However, no tool (even Git;-) 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 automatically.

This lab has been designed as a 2-3 hour introductory lab. Its scenarios will lead you through the most basic version control concepts and principles and how they can be used to help coordinate a group of developers that have to collaborate on a shared codebase. In chapter 10 ("Afterwords") and on the web-pages for the lab, you will get suggestions for how you can move on to explore more advanced scenarios and dig deeper into some of the more exotic commands in Git.

It would be an advantage if you bring pen and paper – for some situations you might want to draw pictures of the states of your workspaces and repositories (it will surely make it easier for the lab assistant to help you out of problems or answer questions). Bring also two laptops: one for doing the lab and another for taking notes 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

In this lab we are **not** trying to teach you Git - that would require a lot more than two hours. What we aim at is to show you how certain functionality in version control tools can be used to implement the abstract concepts and principles you saw in the lecture note. We also want you to get practical experience with how that can help you *manage the collaboration* in a group of people. As a side-effect of using Git as the version control tool in this lab, you will also learn a little about Git.

For this lab, we have tried to make it as easy as possible for you to get started using a version control tool to explore the concepts and principles from the lecture note. So we are “cheating” a little to hide some of the complexities of Git, so you don’t have to deal with too many new things at the same time.

In the following we will first explain and motivate the basic setup we have created. Then we address some specific Git terminology and differences from traditional version control tools and the lecture note. After that we lay out the general context for the lab and explain how you set up your workspace. Finally we give some directions for what to do in case of an emergency.

2.1 The basic setup

We assume that you show up prepared for this lab - if not go back and follow the instructions for the preparation. So you have checked that Git is installed and you have unzipped the file you were asked to download. Things should now look like in Figure 1. We have provided you with a setup where there is a folder for the Shared Repository and two folders for Calvin and Hobbes’ Workspaces. If you open the folder for Calvin Workspace, you will see the project that you have to work on and in the project folder the different types of things the project works on like the source code, tests and so on.

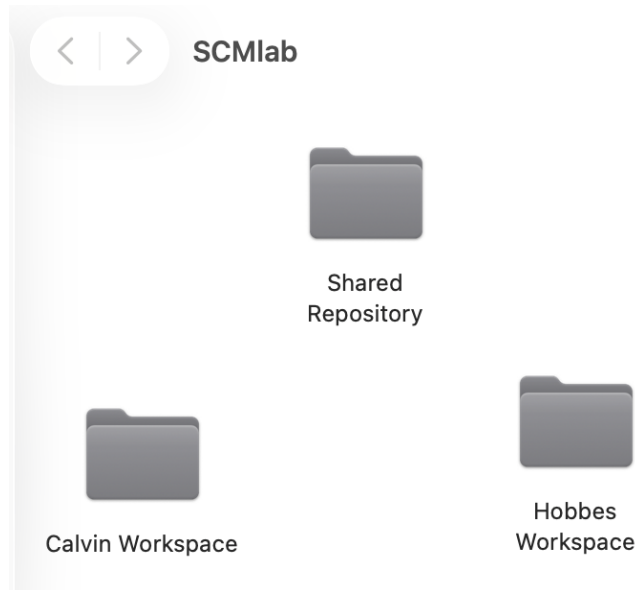


Figure 1. Basic setup.

Since this is a lab, we want to eliminate all possible irrelevant “details” and have you focus on experimenting with the functionality that version control tools provide for managing collaboration. So in this lab you will have everything locally on one computer. This way you avoid network problems and having to address authentication keys for a centralized repository hosting service like GitHub.

This will simulate that you join an already running project with a Shared Repository and where someone (the configuration manager?) has provided you with a Private Workspace that contains the latest version of all the project’s files. In real life things might not be that simple - there might not be a configuration manager who prepares everything for you. We will return to that in Chapter 10, where you will get more insight into how to create a project using Git on your own.

2.2 Terminology and technicalities

In the lecture note we have talked about the Shared Repository as the place where we store and *exchange* versions - and the Workspace as the place where we *work*. All nice and simple. In Git things are a little more complex and terminology differs from the standard terminology used in the lecture note.

In Git they most often refer to the Shared Repository as the “*Remote repository*”. They do so because the Workspace in Git is not just a simple folder in a file system, but a complete “repository-workspace setup” on its own. This is one of the coolest things about Git and other distributed version control systems like Mercurial [Mercurial]. You have full version control functionality in your Workspace - a thing developers have dreamt about for years and usually obtained by abusing branches in the shared repository. This means that conceptually the way you have to think about your private workspace is like you see in Figure 2. So Calvin Workspace consists of Calvin’s local private repository (usually called the “*Local repository*” in Git terminology) with a connected local private workspace where he actually works when

he modified files (usually called the “Working directory” in Git terminology). This looks quite similar to Figure 1 from the lecture note. There are a couple of differences though. Usually you don’t see or care about the local private repository, you use it for checkpointing versions (using the command *commit*) so you have the possibility to undo undesired changes. The exchange of changes with other people is not between the local private workspace, but between the local private repository and the global shared repository (using the commands *push* and *pull* as you will see later). Actually there is a little more to the story, but this is sufficient for now and we will tell you the rest later on in section 9.

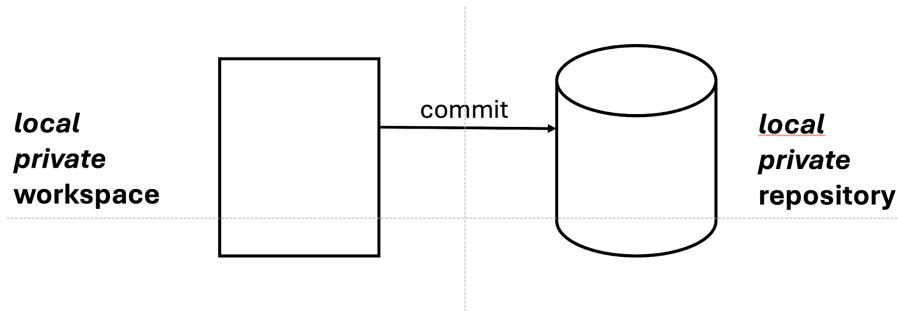


Figure 2. A private Workspace in Git.

2.3 General context

The underlying theme in this lab is that two developers collaborate to develop and release a program. It is important that two distinct environments are used to simulate the 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 have been given a shared repository and a private workspace for each of them – and they can now explore how to coordinate their modifications to the code using different functionality in Git. You work in pairs in this lab, so you have a partner that you can get help from and with whom you can discuss the experiments and their outcomes. This is a whole lot easier when you are both sitting in front of the same computer and can easily see what the other person does and what happens. Now, write down who of you is Calvin and who is Hobbes:

Calvin:

Hobbes:

2.4 Getting ready to work

From the basic setup, that you saw in figure 1, we have provided you with a Shared Repository and two Private Workspaces. When Calvin opens his workspace folder, he will see the project folder - and opening the src folder you will see the source code files that you will work on in this lab. In order to create the full lab setup, that you can see in figure 3, Calvin will have to open a terminal window. If you are using Linux, Unix or MacOS, just bring up a terminal window. If you are using Windows, we recommend you to use Git Bash (that comes with the installation of Git). Once Git is installed in your Windows environment you will find Git Bash among the applications in the Windows/Start menu. Later on, Hobbes will do the same.

So, Calvin now has two windows which mirror the two modes of work that you should use when developing software. Most of the time you are in "*working mode*" ("writing code" in the upper window) and when you are in "*administration mode*" ("talking" to Git in the lower window). Setting up things this way will make it very clear when you are in "working mode" and when you are in "administration mode" - 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" your 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) and reflecting on what happens.

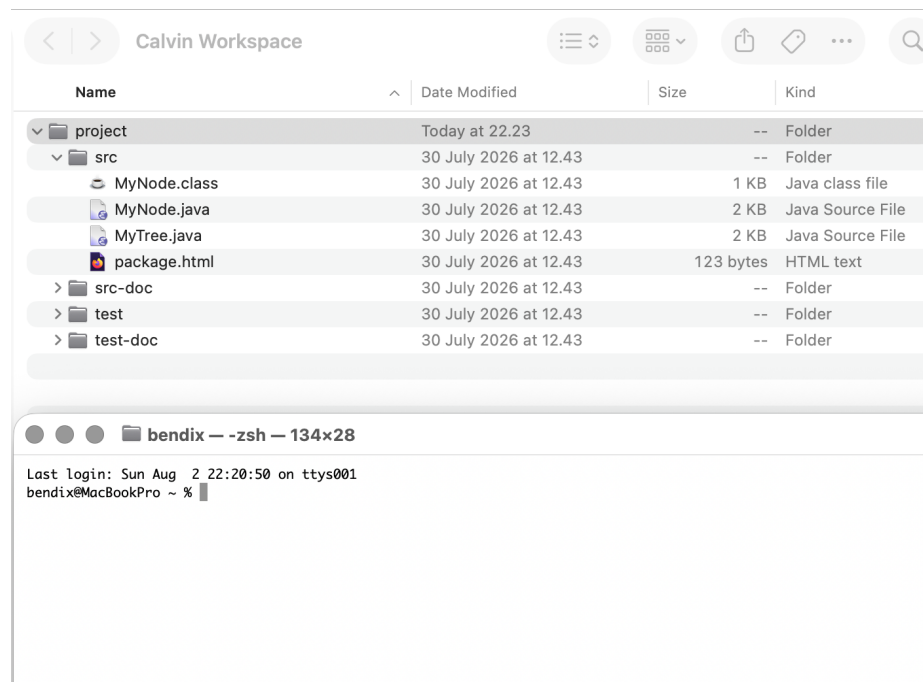


Figure 3. Complete lab setup.

Git is a tool with a large set of features. It is also quite common that there are several ways of solving the same problem. You will get hints about what Git command to use for the different scenarios, but only the first time. If you don't see a hint it means that this is something you have done in a previous scenario and that we expect you to know already.

There are two good ways of getting help on Git commands:

`git <command> -h` and `git help <command>`

explore what is the difference between the two. Which one gives the best explanation of what is the purpose of a command? Which one is a good reminder of the options for a command?

In appendix A, you can find an overview of the most common Git commands (some of which will not be necessary for this lab) with a short description of what they do.

So now you are ready to plunge into work and exploration - or almost. One last piece of advice coming up.

2.5 Emergency procedure

Should things get out of hand with the repositories or if you simply end up in an unknown state that you are unable to get out of, Git is your friend. You can always delete your local repository with its workspace and clone a clean version from the server repository.

1. Go to the directory you created for this lab (SCMLab)
2. Remove the repositories and workspaces with the command

```
$ rm -rf *
```

3. Download and unpack the zip file again

You now have a fresh environment and can start over with the scenario you were working on when things got messy. You are also happy that in this lab you are only working on “fake things” and not real production code.

3. The Scenarios

We will now go through a number of scenarios that are rather common when collaborating using git as version control system. The scenarios connect and relate to the concepts and principles you were taught in the lecture note. Some of the scenarios go a little further and indicate “good ways of working” when collaborating with other people. Take them one by one and do consider the questions in each scenario thoroughly. They are there to make you reflect on what just happened and understand how you ended up in that situation.

Keep calm¹ and get giting.

4. Working alone

Useful commands in this section: `git status`, `git diff`, `git log`, `git commit`

Now we start the work in Calvin’s workspace. Calvin wants to see how he can get information that can give him awareness about what he is doing and what others are doing - and have done.

Scenario 1: A simple change

1. Check the status of your workspace and repository (hint: `git status` is your friend)
2. Check commit history (hint: `git log` is useful and has lots of options) – What branches do you see?
3. Change a file
4. Check the status – Is it different from what you got in step 1?

¹ <https://github.com/louim/in-case-of-fire>

5. See the details of what has changed (hint: `git diff` tells you how your workspace differs from your repository) – Can you see what you did in step 3?
6. Add and commit your change (hint: `git commit -a` should help you)
7. Check the status – Is it different from step 4
8. Check commit history – What has changed since step 2?

5. Simple collaboration

Useful commands in this section: `git pull`, `git status`, `git diff`, `git commit`, `git push`

Now move to the workspace of Hobbes. We will explore how Hobbes can integrate in his workspace the work that Calvin just did.

Scenario 2: Retrieving updates

1. Hobbes gets the latest changes from the server (hint: `git pull` is useful here)
2. Examine the status
3. Examine commit history – Do you see the whole history? Explain what you see.
4. Move to Calvin's workspace
5. Examine commit history – Explain what you see.
6. Push the changes to the server repository (hint: `git push` is your friend)
7. Move back to Hobbes' workspace
8. Pull the changes from the server
9. Examine commit history – Explain what happened.

Scenario 3: Concurrency check

Before we move on we want to check that Git is really a version control tool. This means that we have to make sure that Git performs a concurrency check before sending modified things to the repository so we will not experience the Simultaneous Update problem described in the lecture note.

1. Hobbes makes a change to the beginning of the file `MyTree.java`
2. Hobbes commits and pushes the change
3. Calvin decides to make a change to the end of the file `MyTree.java`
4. Calvin also wants to commit and push his change

What happens? Will Calvin be allowed to overwrite or “shadow” Hobbes' change? Does Git protect us from the Simultaneous Update problem? Is Git a version control tool?

What should Calvin do to be able to commit and push in this situation? Do that.

So now Calvin's workspace contains both his own change and that of Hobbes - but Hobbes' workspace currently only has his change to `MyTree.java`. What should Hobbes do to get the change that Calvin has just committed and pushed? Do that.

Scenario 4: Working in parallel

Now Calvin and Hobbes want to continue to explore how they can work in parallel and try to coordinate and stay in sync.

1. Move to Hobbes workspace (unless you're already there)
2. Make a change to the file MyNode.java
3. Commit the change
4. Push the change to the server repository
5. Go to Calvin's workspace
6. Make a change to the file MyNode.java (on a different line than in step 2)
7. Commit the change
8. Push the change to the server repository – What happened? Why? How can we solve it?
9. Pull changes from the server repository – What happened?
10. Examine the status and commit history – Explain what you see.
11. Now push again – What happened now?
12. Go to Hobbes' workspace
13. Examine commit history
14. Pull changes from the server repository
15. Examine commit history again – What changed?

Scenario 5: A clean start for the next scenario

Now we need to get back to a good start state for the next exercise. So Calvin and Hobbes need to do a little work to get in sync so their workspaces have the same state and contents.

1. Make sure that you're in Hobbes' workspace
2. Pull changes from the server repository to integrate the changes in the workspace
3. Push the new merged changes to the server repository
4. Go to Calvin's workspace
5. Pull the changes from the server repository to align Calvin's workspace with Hobbes' workspace
6. Examine the commit history in both the workspaces. There should now be no differences.

6. Transaction types

Useful commands in this section: git pull, git status, git diff, git add, git commit, git push

We now move on to the two coordination mechanisms that you have seen in the lecture note: Long Transactions and Strict Long Transactions. We want to explore how they work, to check if Git has support for them and to investigate to what degree.

Scenario 6: Long transactions

Calvin and Hobbes want to check if Git 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 only part of a logical change would end up in the repository and that would not work.

1. Go to Calvin's workspace
2. Make a change to one of the first lines in the file `MyNode.java`
3. Commit the change
4. Examine the commit history
5. Push the change to the server repository
6. Go to Hobbes' workspace
7. Change one of the last lines in the file `MyNode.java` and in the file `MyTree.java`
8. Commit the changes
9. Examine the commit history
10. Push the change to the server repository – What happened? Have any of the files been updated in the server repository?

Scenario 7: A clean start for the next scenario (same as Scenario 5)

Now we need to get back to a good start state for the next exercise. So Calvin and Hobbes need to do a little work to get in sync so their workspaces have the same state and contents.

1. Make sure that you're in Hobbes' workspace
2. Pull changes from the server repository to integrate the changes in the workspace
3. Push the new merged changes to the server repository
4. Go to Calvin's workspace
5. Pull the changes from the server repository to align Calvin's workspace with Hobbes' workspace
6. Examine the commit history in both the workspaces. There should now be no differences.

Scenario 8: Strict long transactions

Calvin and Hobbes now want to check if Git supports Strict Long Transactions. The purpose of Strict Long Transactions is to make sure that we do not "merge into the repository". That we only "promote" states that have existed in a workspace (and could therefore be tested) to the remote repository. No sane version control tool would try to merge files that "Needs merge" into the repository 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 supports Strict Long Transactions.

1. Go to Calvin's workspace
2. Make a change to the file MyNode.java
3. Commit the change
4. Examine the commit history
5. Push the change to the server repository
6. Go to Hobbes' workspace
7. Change a line in the file MyTree.java
8. Commit the changes
9. Examine the commit history
10. Push the change to the server repository – What happened? Why? What will Hobbes have to do before he can push his changes? What should Hobbes do after he has done that - and before he pushes?

Scenario 9: A clean start for the next scenario (same as Scenario 5 and 7)

Now we need to get back to a good start state for the next exercise. So Calvin and Hobbes need to do a little work to get in sync so their workspaces have the same state and contents.

1. Make sure that you're in Hobbes' workspace
2. Pull changes from the server repository to integrate the changes in the workspace
3. Push the new merged changes to the server repository
4. Go to Calvin's workspace
5. Pull the changes from the server repository to align Calvin's workspace with Hobbes' workspace
6. Examine the commit history in both the workspaces. There should now be no differences.

7. Merge

Useful commands in this section: git pull, git status, git diff, git mergetool, git commit, git push

Scenario 10: Merge conflicts in collaboration

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 (Scenario 3 and 4), the merge functionality has no problem in integrating both changes. Now it is time to see what happens if they have different ideas and change the same line in two different ways.

1. Go to Calvin's workspace
2. Add Calvin's name in the beginning of line 5 in the file MyNode.java
3. Commit the change
4. Examine the history of the repository. Does it look ok?
5. Is it now possible to push
6. Push the change to the server repository
7. Go to Hobbes' workspace
8. Add Hobbes' name to line 5 in the file MyNode.java
9. Commit the change
10. Examine the version history of the repository
11. Is it now possible to push Hobbes' change to the server repository? Why?
12. Pull the changes from the server repository – What happened? Why?
13. Open the file with the conflicts in your favourite editor. Solve the conflict manually.
14. Commit the merge result
15. Push the change to the server repository

Scenario 11: A clean start for the next scenario (same as 5, 7 and 9)

Again we need to get back to a good start state for the next scenario. So Calvin and Hobbes need to do a little work to get in sync so their workspaces have the same state and contents.

1. Make sure that you're in Hobbes' workspace
2. Pull changes from the server repository to integrate the changes in the workspace
3. Push the new merged changes to the server repository
4. Go to Calvin's workspace
5. Pull the changes from the server repository to align Calvin's workspace with Hobbes' workspace
6. Examine the commit history in both the workspaces. There should now be no differences.

Scenario 12: 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 Git 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 Git to do?

1. Go to Calvin's workspace
2. Make a change to one of the first lines and to one of the last lines in the file MyNode.java
3. Commit the change
4. Push the change to the server repository

5. Go to Hobbes' workspace
6. Make changes (but different ones) to the same lines as you just did in Calvin's workspace
7. Commit the change
8. Pull changes from the server repository – What happened?
9. Open MyNode.java in your editor
10. Solve the merge conflict in the start of the file, leave the other one untouched
11. Commit the change – Was it possible? How does the repository look now? Will it build? How would you have liked git to behave?
12. Repeat making changes, pull, push etc in the two workspaces until you feel comfortable with how git handles conflicts and updates.

Scenario 13: A clean start for the next scenario

By now you should have learnt how to do that. During “real life” development it is often helpful to do the same to bring the different developers' workspaces in sync with each other - and the shared repository. This makes the collaboration work more smoothly and avoids the syndrome of “it works on my computer”. This technique has a name: “Continuous Integration” [Continuous Integration]. Now, do some continuous integration including pull and push until the two workspaces have the same status and history.

8. Baselines

Useful commands in this section: `git tag`, `git checkout`

From the lecture note, you know that a *baseline* is a frozen configuration - a snapshot - that will remain the same forever and that is given a name such that we can reference it. It could be a working and tested version of the code that is frozen and given the name “Release 1” after being approved at a Configuration Audit. The programmers can continue to work on the code and create new versions, but the configuration that was frozen and named “Release 1” will never change. In case a customer reports a bug in Release 1, we can go back to that exact release and fix it.

More informally taken there is a range of uses of baselines. Locally for a single developer a baseline of the code in the workspace could be used to create a checkpoint that can be returned to if something goes wrong. Most editors have the possibility to *undo* an edit operation, but if you have worked for a whole day it will be a lot of work to undo hundreds of edit operations. A version control system offers you a better possibility. You create a baseline of the working configuration that you have in the morning and if something goes wrong you can *roll back* to that configuration with just one command. The same can be done for a team. If we discover after a couple of hours that something is wrong with Release 2 and we need some time to figure out what it is and how to fix it, then we can *roll back* to the working Release 1 with one command.

Baselines can also be used for *deliveries*. To allow developers to continue work on the shared code base, while making it possible to know exactly what was delivered and allowing others to access exactly that. It could be a developer “delivering” a feature to the test team, a developer “delivering” a release candidate to the team, or a team delivering a release to a customer. In all cases we will freeze a given configuration of the code and give it a unique name - in short, create a baseline. Git, like other version control tools, has built-in functionality for freezing and naming a given configuration, namely the command `tag`. In Git all commits are snapshots of the local workspace - and each commit has a name. So in theory it should be easy to return to any given commit by giving its name. However, the name that is used for a commit is the checksum for the commit - a number like this:

```
“a6b4c97498bd301d84096da251c98a07c7723e65”
```

which does not make much sense to most people. With the `tag` command in Git, you can give a commit a more telling name. Using a good naming standard for your tags will make it a lot easier to find and identify those commits that are of more interest than others.

Now read the help texts for the `git tag` command. You will see that there are a number of options (like with all Git commands) that may or may not be useful for you. Finally it is time to explore baselining and tagging in practice - how does it work? So, let Calvin make some changes to one of the files and commit the changes. Calvin is satisfied with the changes and wants to baseline the commit - how is that done?

Calvin does some more changes and commits them. Now he wants to see if he can get back to the baseline - does he remember the name he gave the baseline? What should Git be told to bring back the baseline to the workspace (hint: `git checkout` is your friend)? Calvin checks that the workspace now contains the first changes he did and none of the second changes. Now Calvin wants to get back to the latest commit (he does not want to make any changes based on the baseline) - how does he do that (hint: `git checkout` is again your friend)? Now that he is back, he wants to share all his changes with the rest of the team and does a push.

Hobbes has very eagerly followed what Calvin did and wants to do some exploration on his own. He starts by doing a pull so he should get all the changes that Calvin pushed. Let Calvin check to see if Hobbes' workspace contains all the changes that Calvin just did. Now Hobbes wants to go back to the baseline that Calvin created - how can he do that? What happens? What do you think is wrong? Let Calvin do a `git tag` command. What does it say? Then let Hobbes do a `git tag` command. What does it say? Why don't they give the same result even if Hobbes has pulled all the changes that Calvin did? What should Calvin do so the tag on the baseline is not a private thing that remains in his workspace, but such that it ends up in the remote repository and becomes something that he shares with the whole team? Try to experiment a little with the push that Calvin does - if you cannot get it to work, ask the lab instructor.

9. Further experimentation with Git

You have now been through exploring in practice the most important concepts and principles introduced in the lecture note. This is sufficient to allow you to use SCM and Git for a simple setup like a student project. However, there is much more to Git (and SCM) than this. If you are interested in moving on - maybe after you have consolidated what you have learned so far - we can recommend that you continue with the git-katas [git-katas].

Git katas

A good set of exercises to practice your Git skills is found in the git-katas [git-katas]. The kata concept is copied from the coding kata concept and the idea is that you can do the same exercises over and over again to experiment with different ways of solving the same problem. The katas are downloaded or cloned from the GitHub repository that was linked above and all the required documentation is present in the repository.

10. References

- [Babich, 1986]: Wayne A. Babich: *Software Configuration Management – Coordination for Team Productivity*, Addison-Wesley Publishing Company, 1986.
- [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/>).
- [Berliner, 1990]: Brian Berliner: CVS II: Parallelizing Software Development, in proceedings of USENIX Winter 1990, Washington D.C.
- [nongnu]: <https://www.nongnu.org/cvs/>
- [Git]: <https://git-scm.com/>
- [Mercurial]: <https://www.mercurial-scm.org/>
- [Continuous Integration]: https://en.wikipedia.org/wiki/Continuous_integration
- [git-katas]: <https://github.com/eficode-academy/git-katas>

Appendices

Appendix A: Git commands

The most *important*:

git help and/or git <command> help

git init

Creates a new Git repository in the current directory.

git clone

Creates a local copy of an existing remote repository.

git add

Selects changes to include in the next commit.

git commit -a

Commits changes to files that Git is already tracking. New files must still be added with git add.

git push

Uploads local commits to a remote repository.

git fetch

Downloads new commits and branch information without changing your working files.

git pull

Downloads changes from a remote repository and integrates them into the current branch.

git branch

Lists, creates, or deletes branches.

git merge

Combines changes from another branch into the current branch.

git rebase

Moves or reapplies commits onto a different base commit.

git switch

Changes to another branch or creates a new branch.

git tag

Lists or sets tags in the repository.

git status

Shows the current branch and the state of changed files.

`git diff`

Shows differences between files, commits, or branches.

`git log`

Shows the repository's commit history.

`git log --oneline --graph --all`

Shows a graphic representation of the whole commit history (in ASCII graphics)

`git config`

Lists, sets or changes configuration values for your git environment.