TDDC88/TDDC93: Software Engineering

Lab 4

Software Configuration Management (SCM)

Objectives:

- To give you a fundamental understanding and practical experience of how a version control system works in general, some of the things that are possible with Git
- To practice with different workspaces within SCM

Introduction

"Version control is to programmers what the safety net is to a trapeze artist. Knowing the net is there to catch them if they fall, aerialists are free to fly. In the same way, version control enables you to take programming risks that you would never otherwise consider. If something goes wrong, you can always revert back to a known, good-working version of your code. You can experiment in a branch [1] off the main trunk without interfering with other team members. When bugs are discovered in an older version of a shipped product, you can easily check out that specific version to confirm, fix, and generate a patch for the bug. Without version control, you would have to be much more cautious, move more slowly, and generally be less productive."

-Elliotte Harold, Adjunct Professor, Polytechnic University

The main idea of a version control system is to contain the software project in a so called *repository*; from which developers can *check out* copies of a project and in this way create local *working copies*. These working copies can then be edited and eventually uploaded, or *committed*, back to the repository, as a new version, or *revision*, of the file that has been modified. If several developers simultaneously are editing working copies of the same file, the version control system must be able to deal with *conflicts* that might occur when the developers later try to commit their different working copies to the repository. To be sure that a developer always has the latest revision of a file he or she should *update* his copy from the repository frequently, and at least before commit.

The most commonly used version control system for the latest 20 years has been *Concurrent Versions System*, or *CVS*. In contrast to most other version control systems CVS uses a so called *non-locking repository*, which allows several developers to simultaneously edit local working copies of a file. This is in contrast to a *locking repository*, in which only one developer at a time can gain write access to a specific file. Although it is a good system, CVS is starting to get aged and is, because of its limited ability to handle file content, not suitable for all kinds of modern software development. Because of this, in 2004, the first version of its successor *Subversion*, or *SVN*, arrived, developed by the same people responsible for CVS.

[1] The normal convention is to use three root folders for a software project: *branches*, *tags*, and *trunk*. Branches are for experiments. Tags normally identify older, already released versions of the software. However, most of the time, you'll want to work on the main branch, which is called trunk.

Git was developed by Linus Torvalds in 2005 for the development of the Linux kernel. Just as SVN this is also a *non-locking repository* and the main difference from SVN is that Git is a distributed version control while SVN is a client-server system. Git is by default included in the standard Eclipse version. It is possible to work with Git from the terminal (by typing the commands) or from a graphical user interface (e.g. standalone clients or plugins to IDEs such as eclipse).

Very Important:

All instructions in this lab are inter-dependent. If you make a mistake in one instruction, the whole flow will fail, and you will be asked to do the lab again. To avoid the inconvenience, it is important to take a screen shot of each command, you ran in the terminal. These screen shots can be placed in Word along with the number or what you did. These screen shots will help your instructor to debug the error, in case workflow fails. This does not guarantee (i.e., particularly in cases, where you made many mistakes) that you will not be asked to do lab again.

1. Git

These tasks are for demonstrating some features and problems that may occur using version control and the features in Git. To make life a bit easier we will in the later part of the exercise pretend that we are monitoring two different developers, named Peter and Sally, who are working for the same company. This company has recently bought a project called *HelloWorld*. However, the product manager of the company would like to make some changes to *HelloWorld* and has put both Peter and Sally to work on the project. Of course, since Peter and Sally do not exist you will have to carry out the tasks for them. When you have finished the tasks you should report to the lab assistant and give an oral explanation and a demonstration of what you have done and what you learned from it. Be prepared to answer questions about details about your solution.

Note: Using several workspaces can be a bit confusing. Therefore, it is very important that you read what you should do, before you do it! In the worst case you will have to re-do the lab from start if you just miss a single instruction.

Task 1. Setting up a Git repository and importing a project

In this first task you will create a Git repository and import the *HelloWorld* project to the repository. However, since this is something that developers usually don't have to bother about, we will not pay it that much attention, but only go through the task.

Open a terminal and run the following commands:

```
git config --global user.name "<first name> <surname>"
```

Change <first name> to your name and <surname> to your surname

```
git config --global user.email <liu-id>@student.liu.se
```

Change <liu-id> to your liu-id

```
Mkdir -p ~/TDDC88/git_lab
cd ~/TDDC88/git_lab
mkdir remote
cd remote
git init --bare HelloWorld.git
```

*This will create an empty git repository HelloWorld placed in the folder **git_lab/remote** placed in your **home** folder. The ~ is short for your home folder "/home/<liu-id>/".*
**This folder corresponds to a remote repository where everyone has access to. Usually reachable through a network (e.g. internet).*

```
mkdir ~/TDDC88/git_lab/tmp
cd ~/TDDC88/git_lab/tmp
```

This will create and navigate to a temporary folder.

```
git clone ~/TDDC88/git_lab/remote/HelloWorld.git
```

This clones the repository we just created into a folder HelloWorld in the current directory. You should now have a copy of the repository in the current folder (tmp). The project is still empty.

*Download the folder named "Files" on the webpage (*https://www.ida.liu.se/~TDDC88/labs/index.en.shtml*) and place the file HelloWorld.tgz in ~TDDC88/git_lab/tmp. Run these commands:*

```
tar xzf HelloWorld.tgz
cd HelloWorld
git add -A
git commit -m "Added HelloWorld files to repository."
git push origin master
```

This will be explained later, but it will in short we copied new files into the project. Added them to version tracking with Git. Committed the new changes. Then pushed our changes to the remote origin (which is located at git_lab/remote) so that everyone else can see them.

Remove the temporary folder. We don't need it anymore.

```
cd ~/TDDC88/git_lab
rm -Rf tmp
```

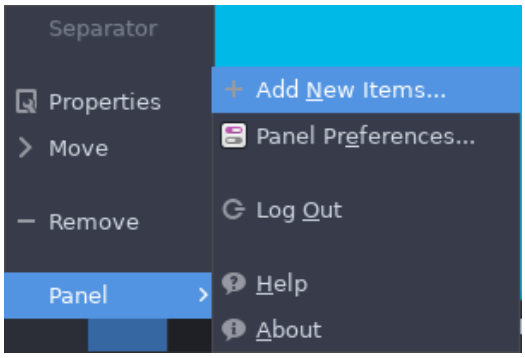
Task 2. Checking out working copies to two different workspaces

In this task we will set up two new workspaces, one for Peter and one for Sally, and check out one working copy from the Git repository to each of the workspaces. You will from now on need two terminal sessions running at the same time. Use eclipse to modify and run the code.

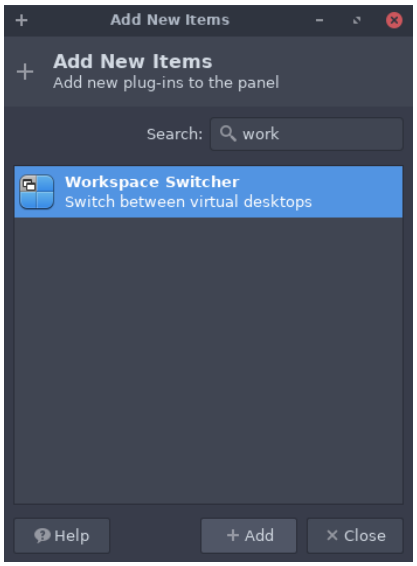
Using two different eclipse and terminal sessions can be confusing since it might not be easy to tell which one belongs to who. So we will use the *Workspace Switcher.* The workspace switcher will keep the two desktops separate. In simple terms it would be as if you had two computers side by side (although it really is not! files, system resources … are still shared). You should see the workspace switcher at the task bar. By default, it has 4 Workspaces.

If you need to install the workspace switcher:

1. Right-click on the taskbar at the bottom of the desktop and select Panel→Add New Items.



2. In the window that pops up, scroll down and select workspace switcher.



3. To make identifying simpler we will let the Switcher show the workspace names. Right click on the switcher panel and choose ***Properties***. Then unmark **Show miniature view**. Click Close.

From now on **Desktop 2** will be assumed **Peter's** and **Desktop 3** will be **Sally's**. Workspace 1 will have every window you have already opened so we will leave that as it is. Use it as you want. Pay attention to which workspace you are working on when you work as one of them.

Now we are ready to start working as Peter. Switch to Desktop 2.
Open a terminal and run the following commands:

```
mkdir ~/TDDC88/git_lab/peter
cd ~/TDDC88/git_lab/peter
```

Create and navigate to peters folder

```
git clone ~/TDDC88/git_lab/remote/HelloWorld.git
```

*Clone a working copy of the repository to peter. You should now have a folder named **HelloWorld** in the current folder (which is **peter**). This is your clone.*

```
git config user.name "Peter"
```

Change the name in this repo to peter. (This is only for the lab and is normally not needed since you have entered a global configuration earlier.)

It is Sally's turn now. Switch to Desktop 3.
open a new terminal and do the same for sally:

```
mkdir
~/TDDC88/git_lab/sally cd
~/TDDC88/git_lab/sally
git clone ~/TDDC88/git_lab/remote/HelloWorld.git
```

Tip: If you forget which terminal is peter and which is sally you can run "git config user.name" to get the name from the terminal. Or check your workspaces.

Back to Peter. Switch to Desktop 2.
Open eclipse.
Choose
`/home/<liu-id>/TDDC88/git_lab/peter`
as your workspace.

Click File → New → Project…
Select Java Project and click Next. As project name type **HelloWorld** after that everything should be grayed out so just press Finish. If you get a popup asking to create a module-info.java, click **don't create**.

You will get a question about the Java perspective which you can select

yes in. Switch to Sally's workspace and do the same for Sally. Use
`/home/<liu-id>/TDDC88/git_lab/sally`
as your workspace this time.

Task 3. Modify-Commit-Conflict that can be automatically resolved

Both Peter and Sally will now begin their modifications of *HelloWorld*. Their first assignment is to add some comments in the file *HelloWorldFrame.java*. Although, it is the only file in the project, the CEO still wants to clarify that it is the main file.

1) Sally starts by adding a comment at the top of the file *HelloWorldFrame.java*, clarifying that this is the main file;
//This is the main file.
Be sure that you are in Sally's workspace, add this comment and save the file!

2) Sally then checks if there has been a new revision to the repository and performs an update. To do this run *git pull* in sally's terminal. Has anything happened since the latest revision?

3) Switch to Peter's workspace.

4) Peter, who is lagging behind somewhat, adds a comment right above the ***main* method**;
//This is the main method.
Add this to his working copy of the file and save it. Peter also checks for new revisions. Do this! Has anything happened? Why/why not?

5) Switch to Sally's workspace.

6) Sally now feels pretty satisfied and decides to commit her working copy to the repository.
Do this for her using the following commands:

git status
To check for unstaged changes, HelloWorld/HelloWorld/src/helloworld/HelloWorldFrame.java Should be marked in red. If you have unstaged .class files don't add those.

git add <file path>
This command puts the file in the staging area. Change <file path> to the unstaged file ex. HelloWorld/HelloWorld/src/helloworld/HelloWorldFrame.java

git commit -m "Enter commit message here"
This commits the changes.

git pull

git push
This updates the server with your changes.

git log -2
This shows the last 2 commits.

7) What is the git log output after the push?

8) What is the staging area for?

9) Why should you do a git pull before pushing?

10) Switch to Peter's workspace.

11) Peter also feels it is time to make a commit. Therefore, commit the file to the repository using the same procedure as above. Remember to enter a commit comment. What happened?

Tip. If you find yourself in JOVE for solving a merge use the commands in the following order to save and exit (don't write the commas): ctrl+x, s, ctrl+x, ctrl+c,

12) Would it have been possible to solve the problem just as easily if Peter had added a comment at the top of the file as well?

13) Which is the three latest commit messages after Peter successfully has committed the file?

Task 4. Modify-Commit-Conflict that must be manually resolved

The product manager has also asked Peter to change the color of the "Hello World"-message from blue to green when pressing the button. However, due to a misunderstanding Sally thought she was assigned to do this, she also thought the color should be red instead of blue.

1) Switch to Sally's workspace.
2) Make a pull for Sally so that she has the latest revision. Locate the section in the file containing the information of the color, and enter the code for red. Save the file and run the application to make sure the text turns red instead of blue. Then make a commit with a suitable commit comment and push it to the server.
3) Switch to Peter's workspace.
4) **Without** first making an update with pull, help Peter change the color to green. Save the file and test the application! When you are done, try to commit and push the file. If it does not work, you must solve the conflict in it and commit the merge!

    Tip switch to Peters workspace in Eclipse and look for rows looking like these and replace it with what it should be:

    <<<<<<< HEAD
    // This is your local version
    =======
    // This is the conflictiong change from the server
    >>>>>>> 9c421ebb4def402d2204d05301aec9b1b07e148a

    Note: Do not push the commit to the remote after the merge has been resolved.

    What happens? Can this problem be solved like before? Why/why not?

Task 5. Roll-back to an earlier revision

The CEO of the company, who has been under a lot of stress lately, suddenly realizes that it is probably best to go with the blue color after all. The product manager is notified about this and asks Sally to fix it.

1) Switch to Sally's workspace and make sure she has the latest version.
2) Sally, being a practical developer, thinks the best way to fix this is to make a so called *roll-back* to an earlier revision instead of changing the code. Help her with this. In this task you can choose to do it in eclipse or in the terminal.

Terminal:

Run git log and find a suitable commit that you want to check. On the commit you want to check copy the string following "commit" looking something like this 775b4b53e2de4d85f794c5932af3e5f133ffde07.

Then run:

git diff HEAD 775b4b53e2de4d85f794c5932af3e5f133ffde07

This command will show what will be changed if we roll back or revert that commit.

When you find the commit you want to revert run:

git revert 775b4b53e2de4d85f794c5932af3e5f133ffde07
git pull
git push

In Eclipse

Right-click on the file and select *Team → Show in history*.

Right-click on a change and choose *Compare with Workspace*

If it is the commit you want to revert right click it again and choose *Revert commit.*

When the revert is done right click on the project and choose *Team → Push to upstream.*

3) What does the log say now, why?

Examination

When you are done and understand all steps in part A contact your assistant during a lab occasion. Be ready to answer questions concerning what you did when demonstrating. You don't need to hand in anything.

References and resources:

Git
git-scm.com