Workshop 1 : Revision control with Git

It is required that you utilize Git for the project of the course, no other revision control software is allowed. Revision control software not only allows you to track the content of files in the project, but also when that content was added and by whom. It also enables you to backtrack changes and pull out old content from the repository. Before you start this workshop, it is recommended that you gain some understanding of Git and it's benefits compared to other revision control software. A good starting point would be the Wikipedia page[1] and the link collection[2] on the course home page.

Git is fully distributed and does not require a central point to work from. Everybody that makes a copy of a repository has a complete local instance of that repository. Consequently, data and history can be shared directly between users without the need of involving the whole development team.

Despite it's distributed design, it is still preferred to have a bare central repository that everyone can reference when working on the source code. In this workshop, we will create such a repository and do some simple exercises. The central repository does not require any special server. The only requirement is a storage point for storing the repository files. For this purpose, the Git client can communicate directly with a local file system or via one of it's supported protocols (GIT, SSH and HTTP).

You should follow the following work flow when working with Git:

- Before you start work on a new task, you should normally pull (fetch) all the recent changes from the central repository into your local repository. This is done so that you work is based on the latest version of the project.
- You edit the content of any files and documents in the local repository that are part of the task you are working on.
- When you are done, the changes made in you local repository are pushed (sent) to the central repository. This makes your changes available to everyone else in the team.

For the project, it is recommended that you use one of the online git hosting services that exist. If you do, you will also have access to a wiki and an issue tracker (plus many other features). Examples of services that support git and offer good functionality for development include google code and github.

1 Creating a bare central repository

The first step is to create a bare repository that will act as the central repository for the project. A bare repository is a git repository without an actual local representation of the content the repository holds. Thus, it is especially useful when creating a central

[1] http://en.wikipedia.org/wiki/Git_(software)
[2] http://www.cse.chalmers.se/edu/course/TDA367#faq

repository. Everything in this workshop is done via a terminal - keep this terminal open throughout the whole workshop.

1. Change the current directory to the group directory:

   ```
   $ cd /chalmers/groups/tda367/ooproj-NN    (NN is your group number)
   ```

2. Create a directory with the name of the project followed by ".git". This suffix is used to denote that we are dealing with a bare repository.

   ```
   $ mkdir project.git
   ```

   Change directory to the newly created project directory:

   ```
   $ cd project.git
   ```

   Initialize the repository as a bare git repository:

   ```
   $ git --bare init
   ```

   A number of directories and files are created.

2 Creating a local repository

Even if we now have a central repository, we can't really use to do any actual work (this is because we created it as a bare repository). Therefore, we need to create the local directory from which we will conduct all work.

1. Change the current directory to the home directory:

   ```
   $ cd ~
   ```

2. Create a local copy of the central repository. This can easily be done with the clone command:

   ```
   $ git clone /chalmers/groups/tda367/ooproj-NN/project project
   ```

   If you look closely, you can see that the first argument to the clone command is missing the ".git" suffix. This suffix can be omitted (just as in the command above) or included. Git will still find the correct directory.

   When you run the command, Git will give you a warning and say that the repository you have cloned seems to be an empty archive. This is normal and obviously quite correct. Git will continue to consider the repository empty until we make an initial commit.

3. Change the current directory to the newly created local repository and check the status of the Git repository:

```
$ cd project
$ git status
```

Git will say something similar to:

```
# On branch master
# ...
```

Git enables developers to work on different versions of the repository in parallel (these are called branches). The default base branch that normally contains a working version of the project is usually called "master". As you can see, this branch is created by default. We will cover branches later in this workshop.

3 Configuring your Git settings

Before we begin adding files to the repository, we should configure the default Git settings. This can easily be done with the graphical Git-gui tool.

1. With the current directory set somewhere in the local repository, execute the following command:

   ```
   $ git gui
   ```

   This will open the main window of the Git-gui tool. This tool will be covered more thoroughly later in this workshop.

2. Go up to the menu and choose Edit > Options.

   a) Set the user name to your full name. Do not use an alias.

   b) Set the email address.

   c) As all commit messages should always be in English, the spelling dictionary should be set to "en_US".

   d) Set the default file contents encoding to Unicode (UTF-8). Never use any other file encoding.

   These steps should always be taken when installing Git on a new machine or when cloning or creating a new repository. Notice that there are two configuration columns. One is for this repository, while the other holds the default values used for all other newly cloned or created repositories.

4 Creating an initial project

Next, we should commit an initial version of the project, making the repository non-empty.

1. To accomplish this, we can create a new project in NetBeans and use the cloned repository as the location for the project files.

a) Create a new project (Category:Maven and Project:Java Application)

b) Set the project name to: Project

c) Set the project location to the local project repository we created earlier.

d) Set the GroupId to: edu.chalmers

2. Build the project.

3. Go back to the terminal and run the following command somewhere inside the local repository:

```
$ git status
```

Git will report two untracked entries; .gitignore and the Project/ directory. The .gitignore file is a blacklist used to inform Git about files in the directory structure that it should ignore and not place under revision control[3]. This file was automatically generated by NetBeans when the project was built. If you inspect the file you will notice that NetBeans automatically added the /Project/target directory to it. This is where compiled Java classes are placed during build. The .gitignore file is usually committed to the central repository, so that everybody can keep track of blacklisted files together.

4. We will tell Git about the files that we want to include in the next commit. This can be done with the command:

```
$ git add --all
```

This tells git to stage all untracked files and changes for the next commit.

5. Run the status command again:

```
$ git status
```

Git will now report a list of several files all staged for the next commit.

6. To finally commit the files and permanently add them to the local repository, we use the commit command and specify a commit message.

```
$ git commit -m ''Initial commit.''
```

Git will show as a list of all the created files together with the number of total rows added.

5 Your first local changes

We will now do some modifications in order to see how Git behaves when files in the repository are modified.

[3] http://git-scm.com/docs/gitignore

1. Go back to the project in NetBeans and open App.java in the editor.
2. Modify the file by adding an additional printout to the main method and save your changes.
3. In the terminal, run the status command again:

   ```
   $ git status
   ```

   Git should report that it has detected the modified file.
4. You can also view your changes with the Gitk command:

   ```
   $ gitk
   ```

   The entry at the very top in the Gitk window (red dot) shows the current modifications that are not yet committed. Gitk is a great tool for viewing your local changes, enabling you to get a simple overview of each modified file. We will cover more of the Gitk tool in the next section.
5. To commit all modified files, we can run the commit command in the following manner:

   ```
   $ git commit -a -m "Added an additional printout to main()."
   ```

   Git should report some information about the commit, including the files modified and the total number of rows that were added and removed.

6 Viewing the history

Git offers two standard tools for viewing the modifications and history of the repository; Gitk (which you previously tried) and git log.

1. First, let's try the log command:

   ```
   $ git log
   ```

   This should output the two commits that we have done so far, including the date of the commit, the author, and the SHA-1[4] hash representing the version of that particular commit.
2. We can also tell the log command to be more verbose:

   ```
   $ git log --stat
   ```

   This gives us some additional information while also showing us the number of modified rows.
3. Next, let's try the Gitk tool:

[4] http://en.wikipedia.org/wiki/Secure_Hash_Algorithm

```
$ gitk
```

As you can see, the output is similar, but gives a better overview of the actual changes. Gitk also offers us a simple way to search the log. It can be extremely useful when the commit log grows.

7 Sharing your changes with the central repository

So far, we have made changes in our local copy of the repository. We also need to share those changes with the rest of the development team. We do this via the central repository.

1. To push the changes to the central repository, we just use the push command:

   ```
   $ git push
   ```

   However; the first time we push to an empty central repository, the reference to master does not exist. Because of this, the above push command will fail and report an error; saying that there are no refs. This can be easily fixed by specifying that we want to create the reference to master when we push:

   ```
   $ git push origin master
   ```

   This should push the commits and report that the master branch was created in origin (the central repository).

8 Creating additional local repositories

We will now test the central repository by cloning it from multiple locations and doing some local modifications in multiple locations. Let every group member log onto their accounts and execute the following steps in a terminal:

1. Configure the global Git settings using the Git-gui tool as described in section [3].
2. Change the current directory to the home directory:

   ```
   $ cd ~
   ```

3. Create a local copy of the central repository. To get access to the repository, we have to clone it using the account of the original author. Write the following on a continuous row in the terminal:

   ```
   $ git clone <user name of central repository creator>@remoteMM.
   chalmers.se:/chalmers/groups/tda367/ooproj-NN/project project
   ```

4. Try opening the project in NetBeans. Everything should be identical to the copy of the original author.

9 Pushing, pulling & rebasing

Now that everybody has created a local copy of the repository, we can start experimenting and testing how Git functions when sharing changes between group members. Let every group member take the following steps while logged in onto their account:

1. Open the project in NetBeans and add a new class. Name the new file according to your name plus surname. For example; if your name was John Smith, the name of the class file would be:

   ```
   JohnSmith.java
   ```

2. Commit your changes and push them to the central repository:

   ```
   $ git add --all
   $ git commit -m "Added class JohnSmith that does nothing."
   $ git push
   ```

   a) When the second person tries to push, Git will complain that your current tip is behind the master branch in the central repository. This is happening because the previous person made changes to the repository **after** you cloned it. This can be fixed by executing the following command:

      ```
      $ git pull
      ```

      When done; Git will inform you that it has merged your changes, while also listing all the files that were added.

      i. When a merge is done with the default strategy (used above), an extra branch is created, showing from which commit the merge originated. This is called a self merge. If we were to push this, it would eventually pollute the Git history with hundreds of minor branches. Unnecessary branching is **never** a good thing. In the next section, we will briefly cover branches and how to work with them. Using the Gitk tool, you can see the extra branch that was created:

         ```
         $ gitk
         ```

      ii. To avoid the branching, we can bring our branch up to date with the remote master, effectively merging our changes onto the top of the current version. This is called rebasing and can be done with the following command:

         ```
         $ git rebase
         ```

      iii. Let's view the history again:

         ```
         $ gitk
         ```

         As you can see, the extra branch is now gone.

      iv. We can finally share our changes:

```
$ git push
```

Now that everybody has committed a new file, we also want to sync up the local repository of each person and make sure that everybody has the latest version.

1. To get the latest changes, each person just executes the following command:

   ```
   $ git pull
   ```

2. Inspect the project in NetBeans. Everybody should now have the same version of the project.

10 Branches

Branching is something Git handles exceptionally well; providing many different ways of merging and sharing branches. When creating a branch, a copy of the head of the currently active branch is created (often from master). Branches are created locally (in the private repository), but can be shared with the remote (central) repository whenever appropriate. Some uses for a branch include:

- Privately testing new features and playing with ideas without polluting the central repository.
- Creating feature branches. If the code of the master branch is in working order and a very substantial feature (which will break the code for a while) needs to be implemented, a common practice is to create a feature branch that is shared via the central repository. This way, the master branch can be kept workable, while everybody works on the new feature. When the feature is completed, it is merged (or rebased) onto the master branch.

Let's experiment with the branch support in Git:

1. Go into your project directory and create a new branch called "feature":

   ```
   $ git branch feature     (creates the branch)
   $ git checkout feature   (checks out the newly created branch)
   ```

   Git has now created the branch "feature" and switched to it.

2. To view all the available branches and show the one that is currently active, you can run the following command:

   ```
   $ git branch
   ```

3. Open the source code in NetBeans and modify in some way. Commit your modification to the repository.

4. Once committed, inspect your changes by starting the Gitk tool in the following manner:

```
$ gitk --all
```

This command tells Gitk to show all the available branches. The default behavior is to show the current branch, plus the master branch. So in this case, providing the flag doesn't change what is shown. Knowing about this flag, however, is important.

5. Let's jump over to the master branch again:

```
$ git checkout master
```

If you inspect the code again, you will see that the modifications you just did are gone. This is because we are on the master branch.

6. Now, let's imagine that we are extremely happy with the changes on the feature branch and want to bring those changes over to the master branch. This can now be done with the merge command:

```
$ git merge feature
```

Git will inform you about the update and report all the files that were imported from the feature branch.

7. Now that we are done with that particular feature, we can safely remove the branch:

```
$ git merge -d feature
```

Note that everything we just did was done locally and that nothing was changed in the central repository (we didn't push anything).

There is an important choice that needs to be made whenever you merge a branch; to rebase or not to rebase. The usual thumb of rule is that big feature branches, with many changes, including other long-lived branches are often merged in order to preserve the branching point. Smaller (or short-lived) branches are generally rebased onto master in order to keep the repository history clean.

11 Dealing with merge conflicts

Nasty conflicts can occur upon merging whenever two people modify the same line or the same chunk of code. Sometimes, Git is able to solve conflicts on it's own, other times it will need manual help from you. Let's create such a conflict on purpose and see what happens:

1. Let two people modify the same row in the same class. Each person should make a different change.

2. The first person commits and **pushes** the change to the central repository.

3. The second person commits and then **pulls** from the central repository.

4. Git will promptly complain about merge conflicts when the second person does this, reporting the file with the conflict.

5. You should now open the file in NetBeans and help Git to choose which change should be kept and which one should be thrown away. Depending on the changes you made, the file will contain something similar to the following:

```
<<<<<<< HEAD
System.out.println("This was the old version");
=======
System.out.println("This is the new version");
>>>>>>> 38fc8527df449fa353fa20e9df68f8fd16952108
```

   Git is showing you the two versions that it has trouble choosing between. Everything between "<<<<<<< *" and "=======" is one version, while everything between "=======" and ">>>>>>> *" is another version. You simply select the version that you consider as the correct one, removing all the rows with markings (===, <<<, >>>).

6. Next, you just commit and push your changes. The conflict should be solved.

TIP: If you ever get many conflicts and know that the changes you have locally can be safely thrown away, you can pass the following commands, effectively resetting your local copy to be identical to the master branch in the central repository:

```
$ git fetch origin
$ git reset --hard origin/master
```

12 Tags

Git supports another very useful feature; tagging. Tags can be applied to a given commit or directly to the current head. They are used to tag releases or certain events.

1. Let's try out this functionality by tagging the version 0.0.1 onto the head of our local master branch:

```
$ git tag v0.0.1
```

2. To share the tag with everyone else we can push it to the central repository with the following command:

```
$ git push --tags
```

Tags can also contain annotations. In the case of version tagging, these are often used to add release notes. An annotated tag is created using the following command:

```
$ git tag -a v0.0.1
```

13 A word about the Git integration in NetBeans and Eclipse

It is important that we talk about the Git integration in NetBeans and Eclipse. While it is extremely useful to have direct access to Git commands from within your IDE of choice, it has to be said that the Git support in these IDE's is based on a reimplementation of the Git client, that is completely written in Java (JGit). In a nutshell, this means that there are certain incompatibilities between Git and JGit and that certain features are missing.

For this reason, it is recommended that you avoid branching, committing or pushing from within the IDE. Everything else should be fine, though. Also, the tools provided by Git (Gitk and Git-gui) are superior in almost every way.

14 Some other useful Git commands

Git is ridiculously comprehensive. In fact, you can **easily** write a book, several hundred pages long, and still not manage to cover all of the functionality that Git offers. Obviously, we can never go through it all. That being said, there is some very important functionality in Git that you should check out on your own:

- **git bisect**[5] - If you happen to introduce a bug in the code that you **know** didn't exist in a previous commit, the bisect command can help you track down the exact revision where the bug was introduced. The bisect command can save you many hours of bug hunting.
- **git cherry-pick**[6] - Sometimes, when you have been working on a branch, you only want to merge certain commits onto master without merging the whole branch. This is often the case when you do a lot of experimentation in a branch. This is exactly what the cherry-pick command is for; it let's you apply commits from one branch onto another.
- **git rebase –interactive** - Gives you the ability to merge (squash) commits and edit commit messages. Only works on local changes that are not yet pushed to the central repository.
- **git reset** - Offers functionality to undo commits. Only works on local changes that are not yet pushed to the central repository.
- **git stash [push/pop/drop]**[7] - An extremely useful command to use when you get conflicts and want to temporarily store away local changes. This command also gives you the ability to pull the latest version of the repository and inspect it without the need of immediately merging your changes.

[5] http://git-scm.com/book/en/Git-Tools-Debugging-with-Git
[6] http://git-scm.com/docs/git-cherry-pick
[7] http://git-scm.com/book/en/Git-Tools-Stashing