GIT Work Flow Summary

(Version 1.1 [12/05/13])

Be careful with copy&paste!
Some PDF viewers may not copy characters, such as the dash.

Table of Contents

GIT work flow and life cycle

The work flow of GIT resembles a lot that of subversion (SVN), although some major advantages with the introduction of *local* and *remote repositories,* as well as the *staging area*, have been made. The following describes the basic principle of each station. How to transfer data from one to another is then described further on.

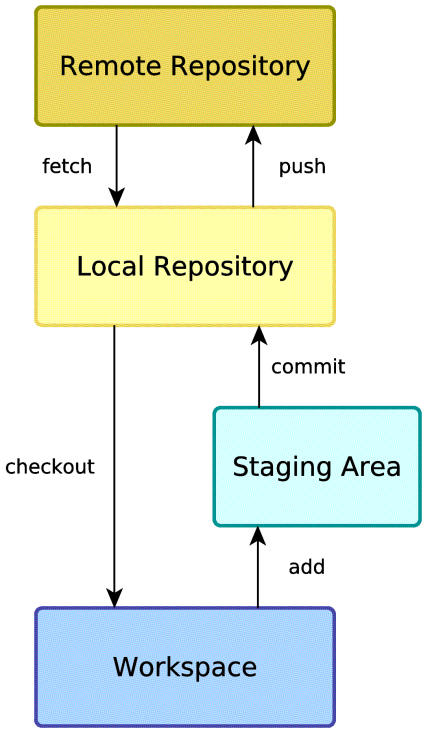

Illustration 1: The life cycle of version controlled data within GIT

Workspace

The workspace denotes the actual physical spot (directory) on a hard drive where all the version controlled files go. There are no editing limitations here, except for the ***.git*** configura-tion files, which should be left untouched most of the time.

Staging Area

The staging area is a transient layer for the evaluation of changed, deleted or added files. With the help of various functions, a review and comparison of the original and the new code can be made (see *Staging files* on page 6). With the *commit* command, tagged changes from the staging area are propagated to the *local repository* (see *Commit to local repository* on page 7).

Local repository

The local repository is, as the name suggests, a local version control. Initially it is a mirror of the current remote repository (see *Getting the code (ROS / NI vision specific)* on page 5). Any subsequent changes or additions to the initial code are then first evaluated through the *staging area* and are then committed to the local repository.

Remote repository

When changes have been committed to the local repository, propagating the changes to the remote repository is the final step in the life cycle of GIT, and makes the changes available to other users. This is performed by the *push* command (see *Push to remote repository* on page 9).

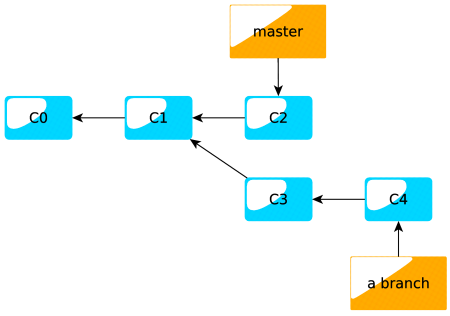

Illustration 2: Controlled versions and multiple branches pointing at different versions

Within the remote repository there may exist different branches of version controlled files. Each of these branches are denoted with a pointer (see illustration 2). The main branch is called the *master* branch. If after some development a user decides that the master branch (“trunk”) and his personal branch should become one, a *merge* is performed. Depending on the amount of changes that occurred, this can be either an easy or a difficult process. If, for instance, changes within a function occurred both in the master and in the user branch, merging can become tricky. Yet with the help of various tools, the process of merging files has become considerable.

Getting the code (ROS / NI vision specific)

Assuming that you have no project files or no personal repository on a hard drive so far, you may get the most recent master branch from the remote repository with this command:

```
git clone username@antares.ni.tu-berlin.de:/mnt/raid/repos/computervision.git
```

where username has to replaced with your user name. The files that are downloaded will be placed in a subfolder within the current folder, named *computervision*. This is now your *workspace* and *local repository*. It has to be linked with the *ROS overlays* directory. For this, be sure that there currently exists no ***~/ros_overlays*** directory. If so, rename and back it up. Finally, execute the following command:

```
ln -s $(readlink f ./)/computervision/ros_overlays/ ~/ros_overlays
```

This will create a symbolic link between your local repository and the *ros_overlays* directory. You may then compile the project via:

```
rosmake pcl_visualization --rosdep-install
```

GIT workspace setup

Assuming that multiple users are using one machine, switching between those users (and thus local repositories) has to be considered. The following two steps are mandatory to perform a switch:

- reset the *ros_overlays* link
- the current GIT user has to be set up accordingly

The latter can be done with:

```
git config --global user.name "name"
git config --global user.email "email@address.com"
```

where name and email@address.com should be replaced with your user name and mail address, respectively. Once this is set you may work on the code within your workspace.

Staging files

Assuming that you have changed files within your workspace: the next step would be to stage those files, i.e., prepare them for a *commit*. One way would be to directly stage the entire file:

```
git add filename
```

where filename denotes the name of the file that you have changed (or added). Alternatively the following command allows you to observe changes for all files before actually staging them:

```
git add --patch
```

After executing this line, changes are printed out to the console to review. You will have the following options for each observed change:

Key to press	Action
y	The observed change will be staged.
n	The observed change will not be staged.
d	The observed change will not be staged, as well as all following changes.
s	Splits the observed change into smaller observable chunks. This can be applied if there are unchanged lines between the observed change.
e	Edit the outlined change manually.

Undo staging

If you have accidently staged a file that should not be committed, you may revert the staging status by using the following command:

```
git reset HEAD filename
```

The file (=filename) will afterwards be treated as before it was staged (i.e., added).

Commit to local repository

Files that have been staged (i.e., selected and prepared) have to be committed. This can be done with:

```
git commit -m "comment"
```

where comment should be replaced with a brief, yet precise description of the change. This is quite useful, as it sums up the essence of the change, without the need to, later on, observe the code in order to comprehend.

Pull from remote repository

If you are sure that your changed or added code is working, you may want to push your content to the remote repository. Yet you should be sure that you files are synchronized with the ones of the remote repository. This should be checked and conducted each time *before* a push with the following command[1]:

```
git pull origin master
```

All files are then updated locally. Your changes can be finally pushed, if no conflicts occur. Dealing with the latter is described in the following.

1 Depending on the activity within the GIT, this should be done each day you before you start working on the code. It is always better to deal with only a few conflicts. They may otherwise accumulate and eventually seem to be overwhelming if they appear at once.

Dealing with conflicts

Conflicts occur when files in the local and in the remote repository differ. If, after a commit, a push is performed, affected files in the remote repository may be newer than your local files *before* change. To check if this is the case and get a list of affected files, do:

```
git status
```

In most cases, another user has made a push on the remote repository that hasn't been pulled yet. For this case there exists multiple graphical tools for direct comparison. On of them is the **mergetool**, which can be opened simply by typing:

```
git mergetool
```

This command will let you select a graphical interface and finally allow you editing conflicting files. If you consider a file merged you may re-stage this file (see Staging files on page 6).

Push to remote repository

Finally, after a commit and a pull has been performed and conflicts have been resolved successfully, the push command comes into play:

```
git push origin master
```

You are subsequently asked to enter your password. If for the password of the *wrong user* is asked, check if you are really in the right local repository. Further check the ***config*** file in the ***computervision/.git/*** directory. It should contain your name right under ***[remote "origin"]***.

Developing on another branch

You are, by default, working on the master branch. With the following command you can switch to another branch in order to develop independently from the master branch:

```
git checkout -b branch_name
```

where branch_name denotes the name of the branch. Note if this branch already exists in the remote repository it will be downloaded and set up to be your current working branch within the local repository. After a series of push and pulls, different branches can be merged with the following command:

```
git push origin master
```

Merge and rebase

A merge is connecting the master branch and a user branch. Assuming that you previously worked on a branch, committed changes and then switched to the master branch again. The following command will perform a merge:

```
git merge branch_name
```

where branch_name denotes the branch you have been working on. Again, conflicts may appear. After resolving them (by adding and committing), the command should be executed again.

The rebase works almost like the merge, yet the difference lies in the integration of all versions of a branch into the master branch. This makes sense if, for instance, work on a branch turns out to be fully relevant for the main development line.

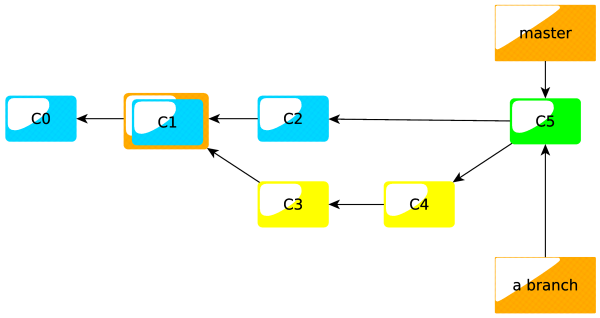

Illustration 3: c1 is the common version of the master and the user branch. Finally, after a successful merge, a new version C5 deriving from C2 and V4 is created, which both the master and the user branch are then pointing at.

Examine the GIT repository

With the help of the tool **gitk**[2], all versions of the GIT repository can be displayed. This is a quite useful tool to comprehend the development steps through comments of controlled versions that lead to the current version of a repository.

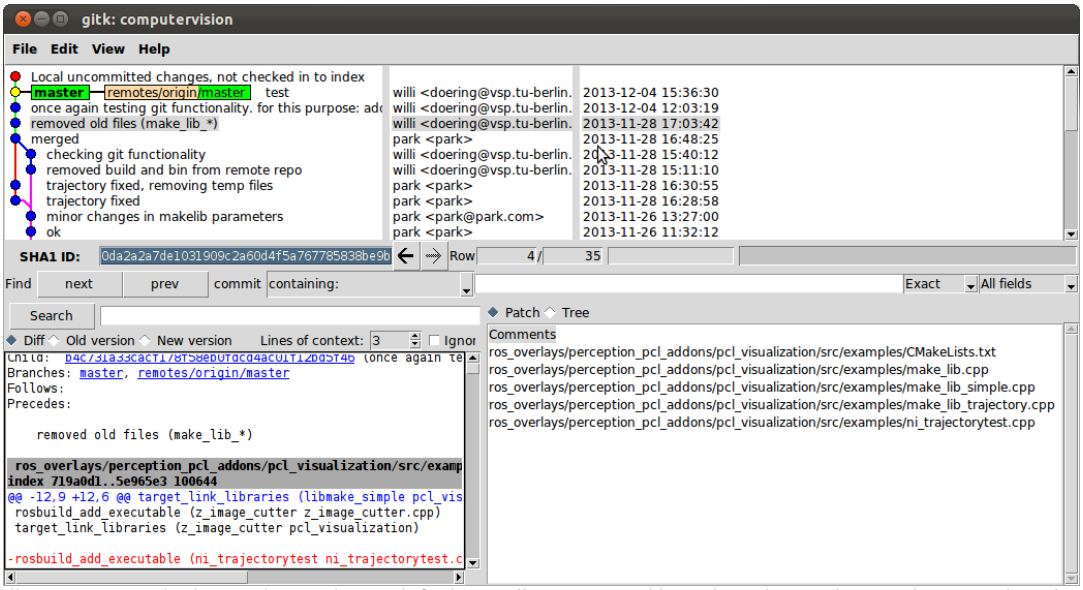

Illustration 4: gitk: the window in the top left shows all versions and branches; the window on the top right side shows the users and the times of push; the bottom windows displays changes that occur within a selected controlled version.

Ignoring files

Some files should not be put either in the local or in the remote repository. For this purpose lists of files to ignore can be set up with a file named ***.gitignore***, which is to be placed in the same directory as the ***.git*** directory. Not that files you are adding to this are already within version control are not considered. They have to be remove with:

```
git rm --cached filename
```

where filename is the file you want to remove from version control.

2 **gitk** is also available through the Synaptic Package manager or the Ubuntu Software center.