POLITECNICO DI TORINO

Introduction to Version Control using

Git

A simple guide to manage Microelectronic Systems laboratories

Professor Maria Grazia Graziano mail@polito.it
Mauro Guerrera mauro.guerrera@studenti.polito.it

2015/2016

Special thanks to Valentina Nerino
Written in LaTeX
Last version: March 7, 2017

Contents

Introduction

Git and Version Control

Git is a Version Control system [1] that allows to record the history of changes made to a project, improving collaboration and productivity. It can be easily integrated in different situations, from programming to web design and even LaTeX composition. The aim of this tutorial is to introduce users to the Git system and increase the workflow of Microelectronic Systems' laboratories.

How it works

The basic idea of Version Control is to have a remote repository where the project can be stored and easily downloaded to any computer equipped with Git.
Users can modify their local copy, create a **commit** (that is, a *snapshot* of the project) and upload it back to the remote repository. This allows to collaborate on the same project and have a remote backup of data.
Git keeps track of the changes made by each user and gives the possibility at any time to switch back to a previous *commit* of the project. It also can show the differences between the current version and the previous ones.
In case of errors introduced in one of the last changes, this function is extremely helpful in trying to figure out what could have caused the problem.

This tutorial

Chapter 1 is about Git setup and basic usage.
Sections 1.1 and *1.2* guide the user through the setup of the Git system on Linux based machines. *Sections 1.3* to *1.8* introduce the basic commands of a Git environment. *Section 1.9* explains how multiple users can collaborate to the same project.
Chapter 2 shows a few advanced commands.
For futher reading see the official Git website [1] and the Pro Git book [3].

Using Git

What you need

First of all, install Git on your Linux machine. This tutorial is based on version **2.8.0**.
For Ubuntu-based systems type in the command line

```
sudo apt-get install git
```

Once installed, configure your username and email (they are mandatory to create new commits). In a command line type:

```
git config --global user.name "Your Name"
git config --global user.email "youremail@domain.xx"
```

Your system is now ready to manage a Git repository.
The next step is to create an account on a hosting website. Although there are several available choices, it is advisable to use GitLab [2].
GitLab is an online repository manager that allows to store private projects for free. It also gives the opportunity to add unlimited collaborators to your projects. The only drawbacks are the lack of support and the limited bandwidth of the server, but it is entirely sufficient for small repositories.
Once logged in, you will have access to your *dashboard*, from where it is possible to create groups/repositories, add collaborators and manage your account.

Creating a Git repository

A Git repository is basically a folder with a `.git` hidden folder inside. The `.git` folder stores all data related to the repository history and configuration, hence it is important not to delete or manipulate it.
There are two ways of creating a Git repository:

- create a local folder and initialize a Git repository, using the `git init` command;

- create an online repository using GitLab and download a local copy of it on your machine.

The second option is faster and allows to automatically set some parameters (as the remote URL of the repository).
First of all, log in on GitLab and create a new group (it allows to share projects with other people, as explained in *section 1.9*). Click *Groups* in the left pane of the dashboard and then *New Group* in the top right corner. Choose a name, select **private** as visibility level and add an optional description. Once done, click on the *New Project* button. It will require to name the project, to provide an optional description and to set the *visibility level*. Choose again **private** and click on *Create project*.
Done! The new repository is now ready to be downloaded.

Cloning a repository

Once the repository is created, GitLab suggests a set of commands to **clone** (that is, download a local copy) your repository.
There are two options: the first one is to use an *SSH* connection, which will be discussed later (*section 2.1*), while the second one is to download the repository using the *HTTP* protocol. *HTTP* doesn't allow to have a secure connection and asks for your username and password every time you upload/download data to/from the remote host.
Use

```
git config --global credential.helper cache
```

to keep your credentials for 5 minutes or

```
git config credential.helper 'cache --timeout=600'
```

to provide a different timeout.
Clone the repository using the *HTTP* URL(switch between *SSH* and *HTTP* URLs using the drop-down menu):

```
git clone https://gitlab.com/your_project.git
```

Commits

The `git clone` command creates a folder with the same name of the project.
Move to the project folder and create a new file `README.md` using a text editor of your choice. Add a description of the repository to the file and save it.
Type

```
git status
```

to show the repository status. This will be an extremely useful command. The output should look like this:

```
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

README.md

nothing added to commit but untracked files present
(use "git add" to track)
```

Each file in a Git repository can be in one of the following states (figure 1.1):

- *untracked*, when it is not tracked by Git; any change to the file will be not recorded. It is the case of the `README.md` file;
- *staged*, when a file under version control has been modified and it is ready to be commited;
- *unmodified*, when a previously *staged* file has been committed;
- *modified*, when an *unmodified* file has been changed but not staged yet.

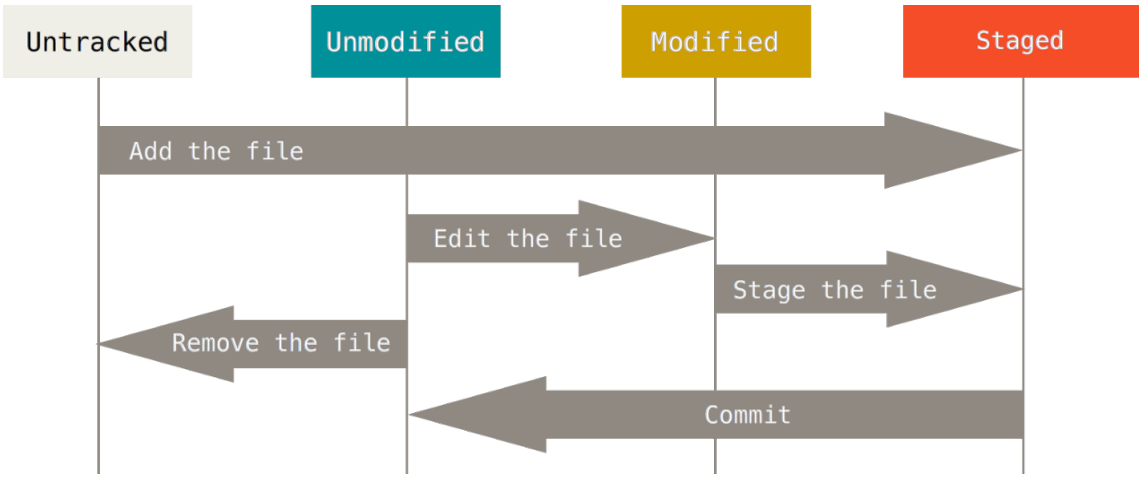

Figure 1.1. Lifecycle of the status of files in a Git repository (image from [1])

A **commit** is a snapshot of the current *staged* files. To stage the `README.md` file, type

```
git add README.md
```

`git status` now shows the file in section *Changes to be committed*, which identifies the staged files.
To create a commit, use

```
git commit -m "Commit message"
```

where the `-m` option allows to add a message to the commit (it should be something meaningful, that summarizes the changes introduced with the commit).
The new commit is local and it has not been uploaded yet to the remote repository. Type

```
git push -u origin master
```

to upload the changes.
The option `-u` identifies the *upstream* (remote) location. The word `origin` refers to the URL of the main upstream repository (which is stored in the `.git/config` file), while `master` is the name of the default **branch**.
A branch is just a pointer to a commit and allows to move through the history of a project and to work on different parts of the project at the same time. Branches will be described in details later (*section 1.6*).
Now edit the `README.md` file. After saving changes, `git status` will show that there are new changes to be staged. The command

```
git diff
```

shows the differences between the last committed version of each *tracked* file (a file is tracked if it has been staged at least once, like the `README.md` file) and files in the current working directory.
Use again

```
git add
git commit -m "New commit message"
```

to stage and commit the `README.md` file. It is possible to use

```
git commit -a -m "New commit message"
```

where the `-a` option stages all tracked files before creating the new commit. Remember that a commit will include only staged files, either with the `git add` command or with the `-a` option.
To see the history of commits type

```
git log
```

which will show for each commit the *commit ID* (a unique identifier associated to the commit), the author and the date of the commit and the commit message.
At this stage, last commit is stored only in the local `master` branch, as shown by `git status`

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit
...
```

It is worth mentioning that local and remote branches are not strictly related. Any local branch can be linked to any remote branch. However, it is a good practice to have local and remote branches with the same name connected togheter.
Using the option

```
git config --global push.default simple
```

Git will automatically link local and remote branches with the same name.

Now use

```
git push
```

to upload the new commit to the remote server. The *simple* push mode set before will link the local `master` branch and the remote `master` branch.
`git status` should confirm the upload

```
On branch master
Your branch is up-to-date with 'origin/master'
...
```

Fetch, Merge and Pull

A new cloned repository is up-to-date and synchronized with the online repository. What happens if a group member adds a new commit and pushes it to the online repository? Or if the same user adds a commit from a different machine and then gets back to work on the previous one?
The local repository is now outdated. It is **ESSENTIAL** to keep the local repository up-to-date with the online version. Otherwise, if there are conflicting commits (e.g. a file modified and committed at the same time by different users), Git will not be able to push new commits on the remote repository, therefore the `push` option will fail.
To download new data from the online repository use

```
git fetch
```

This will only download new commits on the local machine, without applying changes to the working directory. To synchronize the changes on the **master** branch, type

```
git merge origin/master
```

These two operations are condensed into

```
git pull
```

which automatically fetches new data and merges it into the local `master` branch.
NOTE that to merge the local branch with the remote one, it is mandatory to have a clean working directory (all tracked files must be previously committed).

What to do if a merge fails

If you modify a file locally (which has also changed on the remote repository) without having pulled, the merging fails.
In this case there are three options to recover:

- if you have not committed yet in your local repository, **stash** your working directory (that is, save the changes made since last commit in a temporary area and clean the working directory) using `git stash` (this command will be explained later in details). Then pull from the remote repository;
- if you already have made one or more commits, reset the local repository to a previous commit using `git reset --hard <commit ID>`, to reset the repository to the commit preceding your new local commits (the `<commit ID>` can be retrieved using `git log`). Then pull.

- manually merge conflicting files, using a merge tool (`meld` is a powerful GUI tool that allows to merge two different versions of the same file).
  Once all conflicting files are merged, create a new commit and push to the remote repository.

Branches

This section explains how to manage branches in a multi user context.
As already mentioned, a **branch** is a pointer to a commit. The history of a repository is a linked list of commits: each new commit is connected to the previous one and stores a copy of the modified files in that commit.
The default branch for both local and remote repositories is the `master` branch. Every time a new commit is added locally, the `master` branch points to that commit (figure 1.2). A branch can only move forward. This avoids history manipulation that could lead to an inconsistent state of the repository. In fact, moving back to a previous commit, say `C1`, and creating a new commit would cause the loss of all commits following `C1`. In other words, a branch can store only sequential commits and points always to the last one.
Git keeps track of the current version of a repository using the `HEAD` pointer. It points to the current branch but it is not a branch itself, hence it can be used to move backwards through the history (see 2.3 for further details).

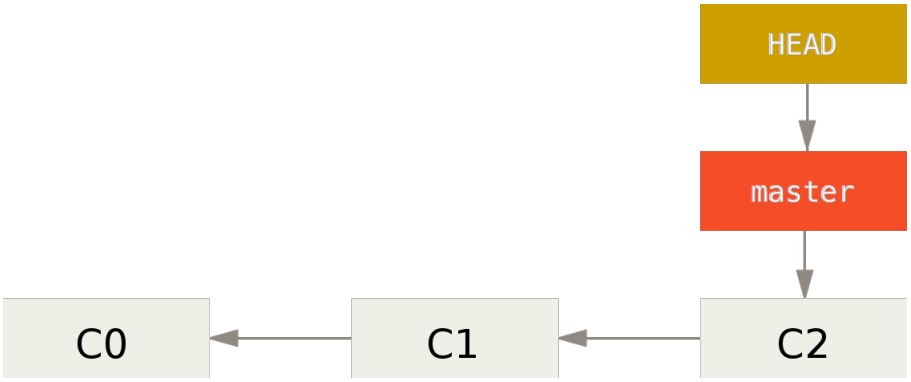

Figure 1.2. `master` branch and `HEAD` point to last commit (image from [1])

When creating a new commit, only the local `master` branch is affected. The remote `master` branch points to last pulled commit. It is automatically updated by spushing on the remote repository or by pulling new commits (if available).
The most important aspect of using branches is the possibility of splitting the repository history, add changes in parallel (to different files or different parts of the same file) and then merging all the new features in the `master` branch.
This can be achieved by creating new branches, add changes to them and then merge

the changes into the `master` branch.
Let's make an example: suppose the only branch in your repository is the `master` branch and that it points to the last commit, `C2`. Imagine that there is an open bug on one of the tracked files and that you want to fix it. Create a new branch called `fix` using

```
git branch fix
```

It just adds a new pointer to the commit indexed by the current branch. Both the `master` and the `fix` branches now point to commit `C2` (figure 1.3).

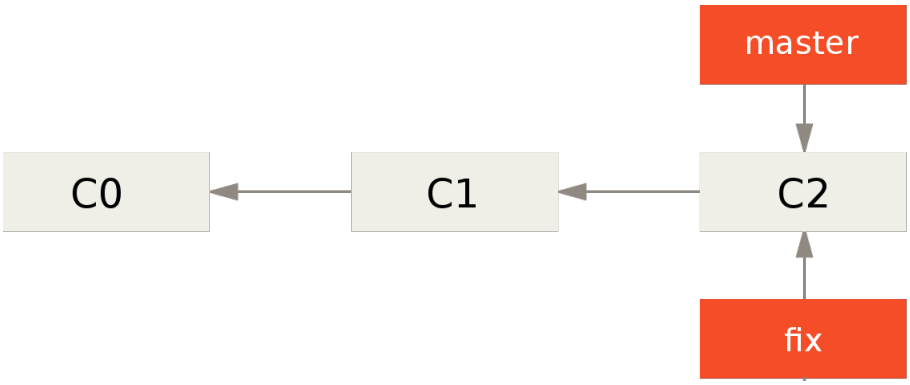

Figure 1.3. Branches `master` and `fix` point to commit `C2` (image from [1])

To know which branch you are currently in, run `git status`. The first line shows the current branch, which should be still `master`.
Now switch to the new branch using the `checkout` option

```
git checkout fix
```

`HEAD` now points to the `fix` branch. Note that the working directory is still the same (both the `master` and the `fix` branch point to the same commit). Modify the file with the bug, stage it and commit changes. The new commit, say `C3`, is now linked to commit `C2` and belongs only to the local `fix` branch (figure 1.4).

The `git diff` tool can also be used with branches:

```
git diff master
```

shows the differences between the current branch (`fix`) and the `master` branch.
Once the bugfix is tested, checkout the `master` branch and merge the `fix` branch:

```
git checkout master
git merge fix
```

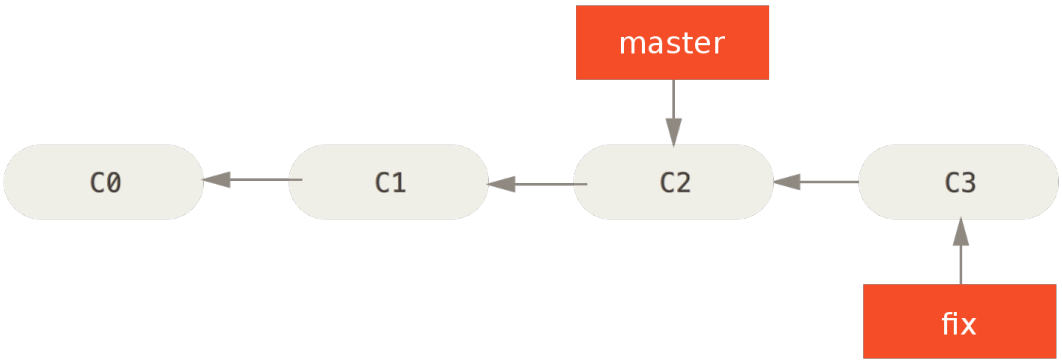

Figure 1.4. Add new commit to branch `fix` (image from [1])

This allows to test changes and add new features without affecting the `master` branch, which is usually considered stable (it compiles and hopefully works).

The `fix` branch can now be removed

```
git branch -d fix
```

This is pretty useful, but think of another situation, in which two bugs are assigned to two different developers. While the first works on the `fix` branch, the second one can create another branch (let's say `fix2`), resolve the bug and add a new commit `C4`. What happens is that commit `C4` is linked to commit `C2`, as well as commit `C3`, but the two commits on different branches are not related (figure 1.5).

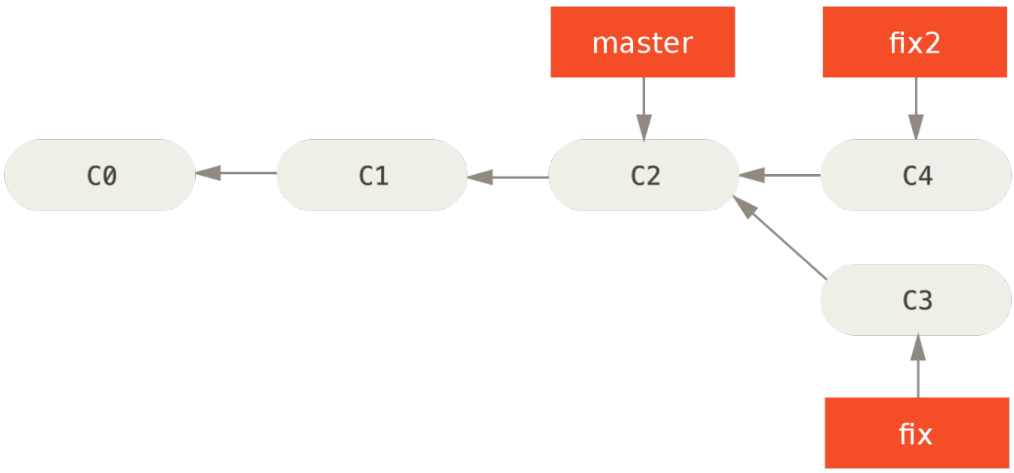

Figure 1.5. Advanced use of branches (image from [1])

At this stage, if the two developers worked on different sections of the project, it

is possible to merge both the `fix` and the `fix2` branches on the `master` branch, including the two bugfixes in a straightforward way.

```
git checkout master
git merge fix
git merge fix2
```

Git adds a new **merge commit** `C5` to the `master` branch, that now includes changes made in commits `C3` and `C4` (figure 1.6).

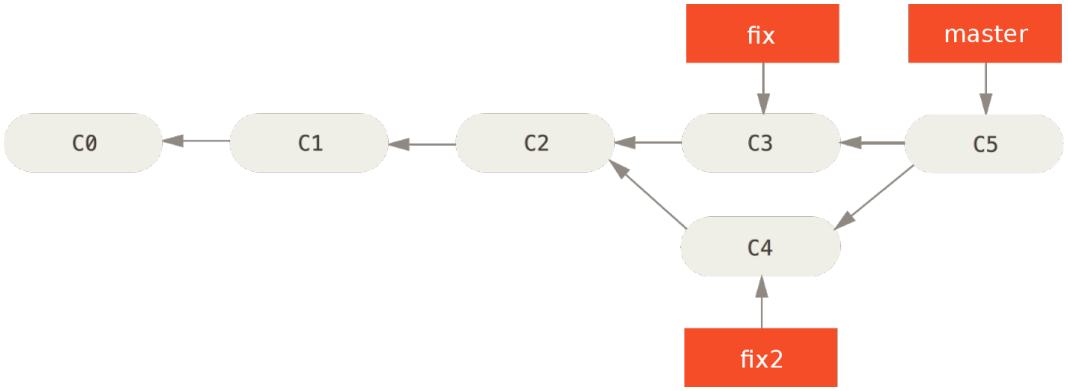

Figure 1.6. Merging of two branches (image from [1])

If the merge fails (for instance, both developers modified the same file), it is possible to recover using one of the methods explained in *section 1.5.1.*

The `fix` and the `fix2` branches are **local** branches, they are not linked to any remote branch. They can be considered as *temporary* branches created only for bugfix purposes and can be removed.
Another possible approach is to create permanent development branches (one for each developer), both on local and remote sides. This way, a user works only on its developer branch and eventually merges changes into the `master` branch. It is advisable to use the first approach, because the maintenance of different remote branches is more complex and could easily end up in merge errors (each user must remember to pull from the `master` branch and to synchronize its development branch before adding any change).

Git submodules

Submodules allow to include a Git repository into another one. This is extremely useful when different projects rely on a common library, which is under Version Control in a separate repository. This way, any improvement to the library can be

automatically included in all projects, without copying it manually every time a change is introduced.
To include a Git repository as a submodule into the current project, type

```
git submodule add <git repository URL> [path]
```

The repository is downloaded into the current Git folder (or into the optional `path` specified at the end of the command). Git adds to the current repository a `.gitmodules` file and the submodule folder, which is treated as a file. In other words, the content of the submodule is not included in the current repository. Both the hidden file and the folder must be staged using `git add` and committed.
When moving to the submodule directory, Git recognizes it as a Git repository and allows to pull new changes or to push new commits.
When switching to another repository that uses the same submodule, run

```
git submodule update
```

and all new changes in the project's submodules will be pulled (previous versions of Git may use

```
git submodule foreach git pull origin master
```

instead of the previous command).
When cloning a repository that already contains a submodule, type

```
git submodule init
```

before the `update` option, to initialize the submodules.

Ignoring files in a Git folder

Every file added to a Git folder is initially marked as *untracked*. When running `git status`, it shows all untracked files in the local repository. Untracked files are not included in commits, but they are listed as part of the repository.
Git allows to ignore files and folders in the repository by listing them into a hidden file named `.gitignore`. Each line can be either the full name of an entry or a pattern that identifies a set of files (e.g., `*.o` idenfies all files with a `.o` extension). As well as the `.gitmodules` file, `.gitignore` must be staged and committed to the repository.
Create a `foo.txt` file in the repository and run `git status`. It should list the `foo.txt` file as untracked. Create now the `.gitignore` file and add to the first line `foo.txt`. Run

```
git add .gitignore
git commit -m "Add gitignore file"
git status
```

The `foo.txt` is not displayed anymore.
This is a simple way of keeping your remote repository clean and to limit the output of `git status`.

Collaboration

Git can be succesfully used for private projects developed by a single user, but it becomes extremely useful when working with other people. To collaborate to the same project, GitLab allows to add group members to a previously created group (see *section 1.2*).
Go back to the GitLab page and select the group created in *section (1.2)*. In the left pane click on *Members* and search other GitLab user by email or by name.
Set the Group Access to **Developer** or **Master** and click on *Add user to group*. The new group member can now clone the repository using the *HTTP* URL and push commits on the remote repository.

Advanced topics

SSH authentication

The *SSH* protocol uses a secure authentication method based on private and public RSA keys. The first thing to do, is to create a SSH key pair. Move to your home directory and type

```
ssh-keygen -t rsa -b 4096
```

This command creates a 4K RSA pair of keys (one private and one public). It then asks for the name of the key (the default one is fine) and a passphrase (can be left blank).
Move to `.ssh` hidden folder and copy the content of the `id_rsa.pub` file. Be sure to copy the **public** key content.
Log in on GitLab and click on *Profile Settings* in the left pane. Then click on `SSH Keys` and paste the content of the public key in the `key` field. Add a title (again the default one is fine) and then click on *Add key*.
Now, when trying to push to/pull from the remote repository, Git will not ask for username and password anymore.
This method is **not** available on the laboratories virtual machine, where only *HTTP* can be used.

Git stash

While developing a project, it is sometimes useful to clean the working directory for a while. Suppose you are doing some changes which partially rely on your colleague's work. You can start working on your part and making changes to the project. At a certain point your colleague pushes his work on the `master` branch, but your work is not finished yet. The working directory is not clean, hence pulling from remote repository would end up with an error. One way is to reset your local repository to the last pulled commit using `git reset --hard <commit ID>`, but all current work would be lost. Another option is to commit changes and then pull, but adding

partial and not tested work to the `master` branch is not considered a good practice. Git helps with the `stash` option.
The stash is a stack where you can temporarily put the dirty state of your working directory, which can be recovered later.
Add some changes to a file.
`git status` will notify that there are some modified files in your woking directory. Run

```
git stash
```

The output of `git status` now should be

```
...
nothing to commit, working directory clean
...
```

All changes made in the working directory have been saved into the stash area. Now it is possible to pull your collegue work and, if changes were made on different parts of the project, the command

```
git stash pop
```

will pop out the previuos changes from the stash area and apply them to the updated repository.
This works unless there are conflicting changes.

Moving the `HEAD` pointer

Unlike a branch, the `HEAD` pointer can be used to move back in the history. Use

```
git checkout <commit ID>
```

to move the `HEAD` pointer back to a previous commit, where `<commit ID>` is the unique identifier of that commit. When the `HEAD` pointer is not aligned to the current branch, you are now in a **detached** state, as shown by `git status`

```
HEAD detached at <commit ID>
...
```

This means that no branches point to the current commit. At this stage, it is possible to see how the repository looked like when the commit was created, but it is **strongly advised** not to add new commits unless a new branch is created. Type

```
git branch <new branch>
git checkout <new branch>
```

to create and switch to a branch that points to the current `HEAD` position. From now on, new commits will be added to the new branch, splitting the history of the repository starting from the point where the new branch was created.
Using

```
git checkout master
```

the `HEAD` pointer is moved back to the last commit on `master` branch.

Checking out a previous version of a file

Sometimes it can be useful to restore a previous version of a specific file.
Type

```
git checkout <commit ID> <file name>
```

to substitute the current version of the specified file with the one saved when the previous commit was created. Running `git status` will confirm that the file is changed and it is ready to be committed (Git automatically stages it).
Note that this operation is completely different from moving the `HEAD` pointer back to a previous commit (section 2.3). In this case, only the checked out file is affected, while `HEAD` is still pointing at the last commit of the current branch.

Bibliography

[1] Git. `https://git-scm.com/`.
[2] GitLab. `"https://gitlab.com/`.
[3] Scott Chacon and Ben Straub. *Pro Git Book*. APRESS, 2014.