Version Control and Git

Sergio Gutierrez-Santos and Keith L Mannock

2017

1 Version control

What is version control? It is something as simple (and as difficult to make right) as keeping track of changes in some piece of work, over time. You are probably familar with some very rudimentary version of version control. Have you ever listed the documents in a folder and seen something similar to this?

- myDocument
- myDocument-2
- myDocument-3
- myDocument-final
- myDocument-final2
- myDocument-final-final
- myDocument-definitive
- myDocument-defitive-USE-THIS-ONE

If so, you already understand the most important idea behind version control: our work is never created in one go, it changes over time and sometimes we want to make sure we can go back in time to a former version of it...just in case.

Many modern programs have version control embedded into them, e.g. word processors like Microsoft Word, OpenOffice Write, or Google Docs (also known as Google Drive). Very often they just track the changes made to the document, sometimes they allow the user to go back and forth in time to review, accept, or discard changes. This is also very common in wiki sites like the Wikipedia.

Version control systems were initially created to track changes in *source code*. They were created by programmers for programmers. We have come a long way since those days, and now version control is spread over many different applications, but they still pursue the same two goals:

Reversibility: the capacity of going back in time if you mess up and introduce bugs in your code (sorry, I meant *when* you introduce bugs in your code).

Concurrency: the capacity of working together with other people on the same project, on the same file, at the same time.

These two capacities are basic for any modern programmer, and that is why version control is (or should be) part of every programmer's daily life. Modern programs are big and complex, and several programmers work on them. Without appropriate version control, they cannot work at the same time: they need to take turns, pass the baton... this is really unproductive, good programmers do not work like that. Additionally, programmers —even good programmers, as long as they are human— make mistakes all the time, sometimes serious mistakes that break their programs completely, and they need to go back in time to the point where everything was working fine and start again (in large and complicated programs, this may mean a long time).

In this chapter we will learn to make version control on your source code files *right*, not as shown above (e.g. myProgramOLD.jdc, etc). We will use a program for doing so called *Git*.

2 Git

There are many programs for performing source code version control nowadays. For this course, we are using Git, a version control system created by Linus Torvalds, the same guy that created Linux. It is not difficult to use but it is very powerful.

Other very common version control systems are Subversion (also known as svn) and Mercurial. There are many more. There are many sources online, starting with Wikipedia, that will tell you the history of version control, the differences between different systems, and much more. I encourage you to go and read about it if you think the topic is fascinating. If you just want to learn to use Git normally in your life as a programmer, this section will help you learn the basics.

As you can imagine, a full book could be written about all the possibilities and options that come with Git —actually, several ones have been published already. In this section I will just present the most important 10% of Git, which is all you will need 90% of the time.

2.1 Starting a new project

Let's start by creating a new programming project. There are two ways of doing this, locally and using an online repository manager. As the latter is simpler from the point of view of sharing your code (more on that later), we will explain that option here.

We will use GitHub (www.github.com), one of the most well-known online repository managers[1]. It is also costless to use, which is a big plus.

Creating an account on GitHub is easy and free. Click on the "Create a repo" button, choose a name and a description, and you are ready to go. It could not be easier.

Now that you now how to create repositories, we will learn how use them to keep track of the evolution of your source code. For the purposes of the following discussion, I will assume that you have an account at GitHub and that your account name is "ilovegit". In some cases I will need a sample repository to explain some of the features; unless the text says otherwise, I will assume that you have created a repository

[1] There are many others: BitBucket, Google Code... the list is endless.

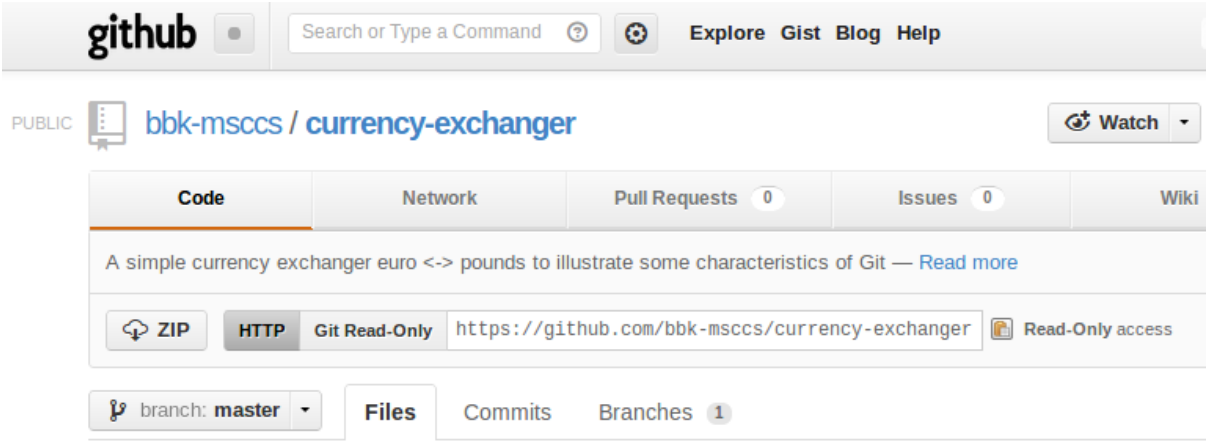

Figure 1: Screenshot of GitHub. You can see the name of the user ("bbk-msccs"), the name of the repository ("currency-exchanger"), and the URL.

in GitHub called "my-currency-exchange" and the examples will refer to this repository created by "ilovegit".

2.2 Think global, act local

A repository is just a place (somewhere on the internet or in a private intranet) where the source code for a software project is stored. There are two types of repositories for any project: *local copies*, where you do the work; and *one public copy*, that other people can look at. Any machine on the internet can be used to host a public copy of a Git project and GitHub (`www.github.com`) is a convenient place that many people use.

You cannot write your programs on GitHub, you can only do it in your local copy. Therefore, the first thing you have to do after creating your repository on GitHub is making a local copy, a process known as *cloning*.

```
> git clone https://github.com/bbk-myName/my-repository.git
```

This will create a local copy of the remote repository you have just created. You can now make changes to it: create new files, modify the existing ones, etc.

(You can also clone repositories that were created long time ago and/or by other people. More on that later.)

2.3 Keeping track of changes

Let's start by writing some code. For example, let's create a simple *Hello World* application in Python. In other words, we will edit a file called `helloworld.py` and write something like:

```
print("Hello World!")
```

If we execute this little program, it will print the words "Hello World!" on the screen. So far, so good. Time to start filling up our *version journal*!

The first step is to tell Git that we want to keep track of this file. In other words, we *add* it to the list of files under Git's responsibility.

```
> git add helloworld.py
```

And now we must perform the most important operation in any version control system: *committing* our changes.

```
> git commit
```

You will be asked to introduce some description of what this *commit* is about. Git automatically adds information about which files are committed and what changes have been performed on them. The programmer must provide some additional information: a short message to explain to other programmers what the changes are about. Note that "other programmers" can mean yourself in two weeks time —when you have forgotten what you committed at this point. Typical messages are "First commit", "Fixed bug #1304", "Added a new feature for…"; examples of bad non-informative commit messages are "new commit", "More code", or "Fixed it AT LAST!" (what is *it*?). Write whatever you want, but make a (small) effort to think what message will be useful for people reading it in the future.

When you finish writing your commit description, save it and close the editor. The commit will be performed, and you will be given some output from Git. That output will include information about the branch (by default, it is called *master*), the identifier for this commit (a unique identifier similar to "12a0006"), your commit text, and some statistics about what the commit did: lines added, edited, or removed, files added or removed, etc (*see Figure 2*).

```
> git commit
[master 12a0006] Added first line: just says "Hello World!"
1 file changed, 7 insertions(+), 1 deletion(-)
>
```

Figure 2: Example of Git's output after a commit.

Now your project has a history! It looks more or less like Figure 3. Not very impressive, but we are just starting.

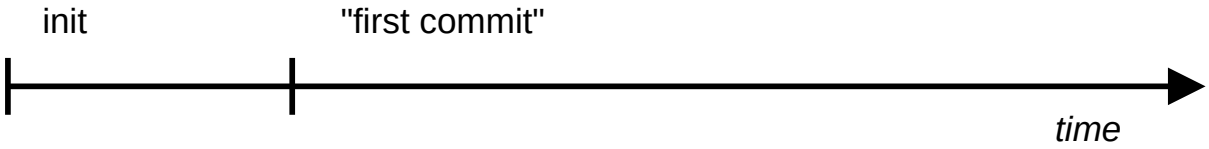

Figure 3: Initial history of this project

What to do when you mess up

Let's say we are not happy with our program. It does not do much. We can modify the program to look like this:

```
println "Hello World!"
println "What's your name?"
String s = System.console().readline()
println "Hello " + s + "!"
```

Once we have saved the changes, we can commit again (`git add helloworld.groovy; git commit`, plus a commit description).

However, if we try to run the program, Groovy will complain. We have messed up! At this point, we have two options:

- If we know where the problem is (and in this simple example, we do) we can just fix it and commit again. A good commit message would be "Fixed typo in line 3: readline() should be readLine()". The history of the project is represented in Figure 4.
- If we did not know where the problem is, as it is usually the case in big programs, we can go back in time until we find the commit in which the problem started. Looking at the changes on that commit we can see how the *bug* was introduced. Thanks to version control, finding bugs is much easier (and finding bugs is 80% of the job).

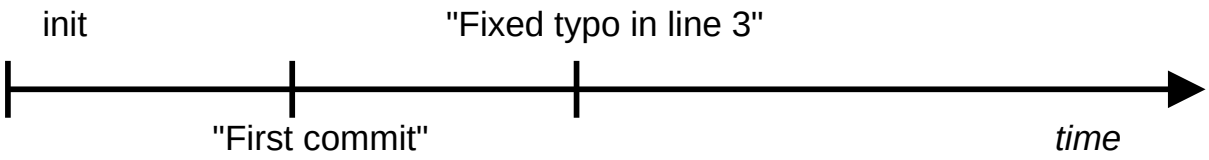

Figure 4: History of this project after second commit

The simplest way to travel in time is by looking at the change log of a file, and we are going to see how to do that in the next section.

2.4 Looking into the past

Contrary to most novelists, professional programmers do not usually start working on a project from a blank editor page. The most common situation is to work on a project that has already been going on for some time. Maybe the programmer is a new employee in the company, or has been transferred to a different project, or has "inherited" a project that somebody else started. Or it may be that the programmer wants to contribute to another project because the project is appealing or famous[2]. Or maybe the programmer starts a project from scratch and then realises that somebody else is doing the same thing, only they started long ago and have made a lot of progress, so it makes sense to join their team instead of reinventing the wheel. For any of these reasons —and many others— you will find yourself in a situation where you want to get a copy of the source code that other people have written. This is very easy to do in

[2]Examples of interesting free/open-source projects with many contributors include web browsers like Firefox or Chrome, the Linux kernel, the Android operating system (based on Linux), the LibreOffice suite of applications, mail programs like Thunderbird, and many others.

Git. You just need to be given a URL[3] to the source code, and clone it. Cloning other people's project works in the same way as cloning your own project:

```
> git clone https://github.com/bbk-msccs/currency-exchanger.git
```

Now that we have a copy of the Currency Exchanger project, let's look at it. You will see that it has only two files: a "read me" file called `README.md` (that explains what the project is about) and a Groovy file called `currencyConverter.groovy`. This is the interesting one.

Look at the code of `currencyConverter.groovy` and understand what it does and how it works. Then come back and continue reading. Go on, I will wait for you here.

As you can imagine, this little program was not written in one go. You can see a summary of the history of the file by looking at its *commit log*:

```
> git log currencyConverter.groovy
```

This will show the list of commits on your screen, ordered chronologically. For every commit, you can see the commit ID, the author, and the date and time on which the commit was made. You can also read the commit message.

You can pass arguments to `git log` so that Git only shows you commits for one author, or between two specific dates, and many other options. Type `git help log` for more information. You can use `git help <command>` to get help on any other command.

The problem with `git log` is that it does not show how the code changed from one commit to the next. But there is a way to do this: `git diff`. Let's have a look at an example:

```
> git diff ab9d6 9ecf9
```

This command shows the changes between those two commits. Note that I did not have to write the whole commit ID (40 characters!) but only the first five are fine *as long as they are unique*. If the shortened IDs you use are not unique, Git will complain. In this case, the short IDs are unique and Git tells that only one line was changed:

```
index 2d97cf4..5650842 100644
--- a/currencyConverter.groovy
+++ b/currencyConverter.groovy
@@ -23,7 +23,7 @@ while (!finished) {
    case 2:
        print "How many euro would you like to convert? ";
        double euro   = Double.parseDouble(System.console().readLine());
-       double pounds = euro * poundOverEuroRatio;
+       double pounds = pounds * euroOverPoundRatio;
        println euro + " will give you č" + pounds;
        break;
     case 0:
```

[3] A Uniform Resource Locator (URL) is a specific character string that constitutes a reference to an Internet resource. Examples of URLs are `http://www.bbk.ac.uk` and `ftp://gb.archive.ubuntu.com/ubuntu/`.

The "-" sign on the margin shows a deleted line while the "+" sign shows an added line; the other lines were not changed. If you take everything into account, the only thing that changed in this commit was the name of two variables on that line.

You can also use some special tag names for referring to a commit, as shown on Figure

ID	Identifies...
9ecf9	...any commit whose 40-digit ID contains *9ecf9* as long as there is only one
HEAD	...the latest commit done
^	...this suffix identifies the commit before another one
HEAD^	...the commit before the latest
9ecf9^^	...the commit before the commit before *9ecf9*

Table 1: Commit names in Git

2.5 Sharing is caring

2.5.1 Push

If you wanted to improve the former program (the currency converter), you could edit it yourself, add new features, change some code, etc. You would commit regularly to make sure you always have the history of your project up to date (so you can see what changes you introduced over time).

At some point you will want to make your changes public (in Git jargon, this is called *pushing your changes*), e.g. to allow other members of your team to see what you have done. Usually, you push to the same public copy that you cloned the code from; by default, this public copy is called "origin" by Git, although you can change this name and/or have more than one remote copies where you push your changes ("git help remote" is the place to start if you want to learn more about this). Pushing changes is very easy:

```
> git push origin
Username for 'https://github.com': ilovegit
Password for 'https://ilovegit@github.com':
To https://github.com/ilovegit/my-currency-exchange.git
6457568..55ca3da  master -> master
```

The system will ask for your username ("ilovegit" in this example) and your password. Then, if everything goes according to plan, it will inform you that it has pushed your commits. Note that this step could fail for several reasons, like your network connection going down.

2.5.2 Pull

Once you have pushed your changes, how do other people see them? How do they take the code you have *pushed* into your remote repository and put it in their local copies of the source code? As you can imagine, they *pull* it.

```
> git pull origin master
```

When you pull, you have to specify the remote repository you are pulling from ("origin" in this case) and the branch you are pulling from. We will talk later about branches, but for now it suffices to say that the default branch in most projects is called "master", and sometimes it is the only one. If you are not sure which branch you are pulling from, "master" is usually a good guess.

Once you pull, Git will download the code (or, in other words, the changes made public at that remote repository and merge them with your local copy. Once it finishes, you have the latest copy of the project's source code, including those changes that somebody else (maybe you) made in a different computer.

At this point, you have surely noticed that Git can be used to synchronise with other team members, each of them pulling whatever changes others have pushed so that everybody is up to date; but it can also be used to keep up to date with yourself: you can work on different computers (at home, at the office, on your laptop on a plane) and you can push your changes and then pull them from the other computers. There is no need to carry around USB sticks with folders called "currencyExchangev2", "currencyExchangev3", or "currencyExchangev4usethisnottheother". Git will always have the latest version as long as you do not forget to commit and push timely (Figure 5).

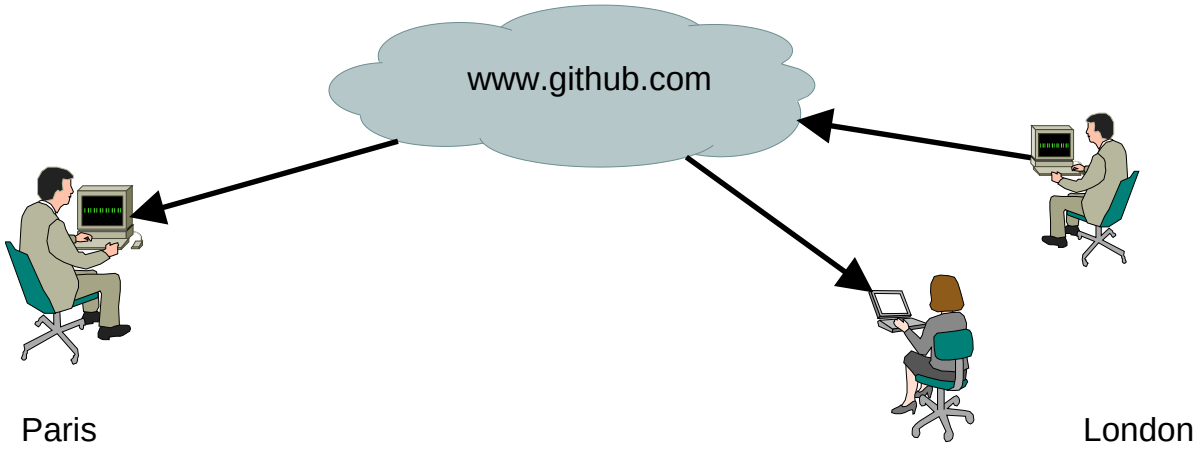

Figure 5: Pushing and pulling: when Alfred pushes his changes to make them public, Bertha will be able to pull them and work on the last version of the software. If Alfred takes a plane to Paris, he will be able to pull the changes from Paris to get the last version…including any changes that Bertha and other co-workers may have pushed while he was on the plane!

2.6 When should I commit?

As a general rule: *commit early, commit often*. It is generally a good idea to make small commits so that it is easier to go through the history of your code and understand every step if you have to. The cost of committing is almost zero so you should not be afraid of having too many commits (there is no such thing as "too many commits").

Of course, this does not mean that you commit every single character that you write. Committing every single line is also probably too much, unless those lines are

really special (e.g. they fix a bug). *Do not spend more time writing commit messages than writing code*. That said, if you have not committed for the last half hour, either you have written a lot of code and you should commit early (as in *now*); or you are tired or distracted, and not really programming, so maybe you should have a break.

If you are not sure what you have changed since the last commit, you can use `git diff`. Another —less verbose— possibility is to use `git status`. The latter will tell you which files have changed since the last commit, without details of the changes.

Additionally, you should *push* your changes to your public repository fairly often. Some people say *push early, push often*, but it does not mean that you need to push after every commit. Sometimes you cannot even push because you do not have a network connection, although this is becoming more and more uncommon these days, where you can connect to the internet from planes, submarines, and almost a robot wandering around Mars[4]. As a rule of thumb, push every time you have finished a coding session (e.g. before turning off the computer or before standing up to grab some lunch), or as soon as possible after that.

2.7 Back to the past...

If you have been reading carefully until now, you will have noticed that we do not know yet how to *move back in time*. So far we have only learnt how to look into the past using `git log`. In order to be able to *roll back* when we mess up things, we need to do one of two things: either we can *revert* changes or we can *branch* the history of our source code (Section 2.8).

2.7.1 Time reversal

The simplest form of time reversal is by using the command `git revert`. This command makes a commit that cancels all the commits that you specify. Confusing?

Maybe, but think that the typical use case is cancelling just the last commit: you make some changes, you commit them, and then you realise that you forgot to do something before committing...maybe you forgot to add a file, or something of the sort. At this point, you can follow two courses of action. Either you do the things you had forgotten and then commit them (using a message like "this is what I forgot to do in commit 25cec5a... 4"), which is not optimal; or you revert and then commit again, like in this example.

```
(modify file1 and file2)
> git add file1
> git commit
(realise you forgot to add file2)
> git revert HEAD
> git add file2
> git commit
```

[4]Actually, a robot in Mars cannot connect to the internet, at least not what we call "the Internet" in 2012 (computers interconnected using the TCP/IP and related protocols). Radio waves travel *only* as fast as light-speed, and it takes them so long to reach the Earth that internet hosts would timeout before a connection could be stablished. You *could* connect to the Internet from the Moon, though, but your bandwitdh would be very poor compared to your home connection.

HEAD is a special tag that means[5] "the last commit" (see Table 1). Git will ask you to introduce a commit message to explain why you are reverting that change. You are not limited to revert the last commit. You can revert many commits, as in this example (remember the special tags from Table 1) that reverts the last four commits (you will need to add a message for each reverting commit):

```
> git revert HEAD HEAD^ HEAD^^ HEAD^^^
```

Git will take care of all the changes to the source code for you.

2.7.2 Conflicts

You can also revert any combination of commits that you want.

```
> git revert a827e5fe 067cac919 938f6821a
```

Usually, Git will do all the necessary changes to revert all those commits and give the resulting files. However, reverting arbitrary commits may result in a *conflict*. A conflict happens when Git cannot reliably make a change in the code to accommodate your wishes. This is uncommon, but may happen if you are not careful when reverting changes or merge two very different pieces of code.

When a conflict happens, Git will tell you which files have conflicts so that you can fix the source files yourself. A file with a conflict looks like this:

```
(some source code here)
<<<<<<< comit-ID-1
(source code as in ID-1
=======
(source code as in ID-2
>>>>>>> commit-ID-2
(more source code here)
```

Your role as programmer gifted with a human intelligence —Git is able to solve most conflicts itself, but not all— is to decide which code should stay: the code between <<<< and ===== or the code between ==== and >>>>. Maybe none is correct and you need to write new code to fix the conflict (and remove the <, =, and > symbols, of course).

In any case, once the conflict is resolved (by you), you have to commit the changes.

2.8 Branches, branches everywhere...

Last, but definitely not least, we need to learn about branches. You have already met the most important one, called *master*. This is the default branch in any Git repository, and sometimes it is the only one. But a Git repository can have an unlimited number of branches, and this is very common for large projects.

Branches are important in Git, and in any version control system. Branches allow programmers to advance development without compromising the stability of the code

[5]Strictly speaking, it means the last commit *for the current branch*, but for now we are assuming we only have one branch, called *master*.

released to clients, to try experimental features that may or may not be worth been added to the project, and to collaborate with external programmers that want to help in the development.

There are many interesting things that you can do with branches, and this is one of the most important features of Git. However, for the moment we are going to see just the basic functionality.

You create a new branch with the command `git branch <branch_name>`, and change branches with the command `git checkout <branch_name>`. Whatever you commit to a branch is only visible to that branch, at least until you merge it with another branch.

Let's see what branches are about with a small step-by-step example:

1. Create a new empty Git repository. Clone it.
2. Create a new file called `mainFile` inside the local copy of the repository, and then make some simple changes to it (e.g. add a few lines of text). Commit your changes (to *master*).
3. Create a new branch: `git branch testing`. Change to the new branch: `git checkout testing`
4. You are now in `testing`. Create a new file `experimentalFile` and add some lines to it. Commit. Add some lines to `mainFile` too. Commit.
5. See the history of your files using `git log`.
6. Go back to the *master* branch: `git checkout master`.
7. See the history of your files using `git log`. Do you see any difference?
8. Add some lines (different from before) to `mainFile`. Commit.
9. Change branches again: `git checkout testing`. Read the content of mainFile. Check the history: `git log mainFile`. Is it the same history as in the other branch?

As you can see, opening branches allows you to do experimental or not-well-tested code without compromising your main line of development (see Figure 6). After branches diverge, any new commits you make in `testing` are not visible in `master` and viceversa.

If at a later point you become convinced that your work in testing is worth being merged into the main branch, you can do so with:

```
> git checkout master
> git merge testing
```

This will merge both branches and create a new commit in *master* that includes all the changes from *testing*. Note that, depending on the changes you have made in both branches since they separated, a conflict may arise and you will need to fix it (and then commit) manually (see Section 2.7.2).

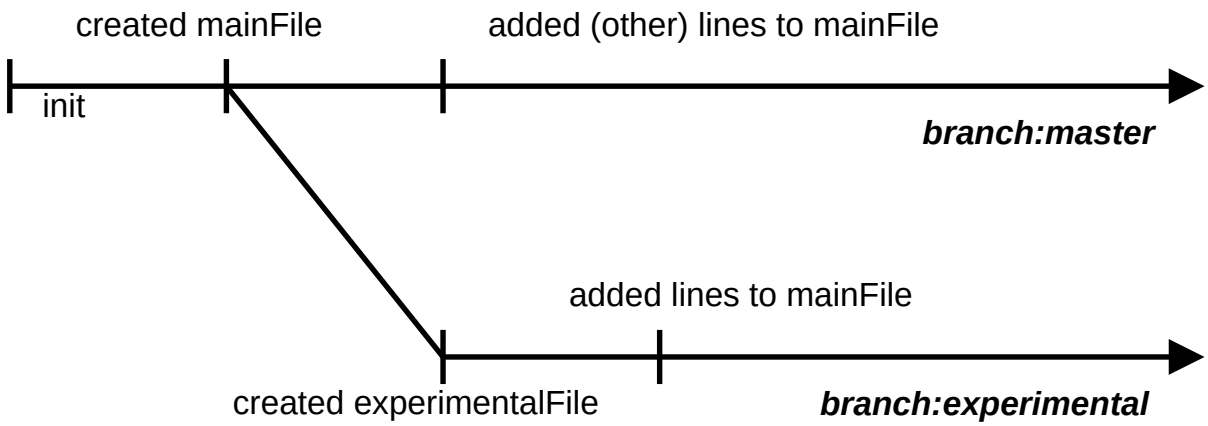

Figure 6: Project with two branches: master and experimental

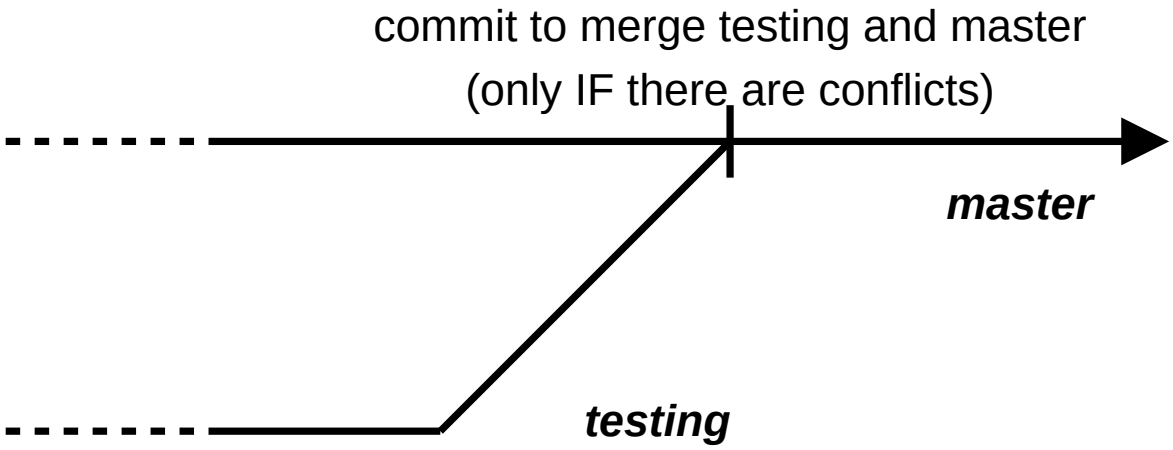

Figure 7: Merging branch “experimental” into “master”

Replacing a branch

You can also use branches to go back in time and change the way you are doing things. For instance, think of a situation where your work in *master* was based on one technology and, further down the line, you discover another technology that is much better. Another possibility is that you messed up but do not realise until several commits laters.

You could of course just delete the code of the old technology and start writing your new code for the new technology. But there is a better way. You can move back in time, branch, and then continue your development in the new branch... and call it *master*. The process would be as follows:

```
> git checkout 33gfg32    [1]
> git branch newMaster    [2]
> git branch -m oldTech   [3]
> git branch newMaster
> git branch -b master    [4]
```

The first command [1] puts you at the right point where you want to start a new branch. Note that you can use `git checkout` to change branches or to change to a specific commit[6]. Then you create a new branch [2] as we have seen before. Then you change the name of the current branch [3], using `git branch -b` (look up the help for more details). Finally, you change branches and rename the new branch to *master* [4]. Then you continue your development in the new branch.

There is much more that can be said about branches and how they make it easy to have several levels of development in your projects, and how they make it easy for many programmers to work on the same project, but we will see all of that at a later point.

2.9 Ignoring files

Sometimes you want Git to ignore some files. You do not want to keep them under version control, cannot delete them, and do not want to have them appearing on the screen every time that you check the status of the repository. This is a common occurrence for temporary files like `.o` files in C/C++, `.class` files in Java, `.dvi` files in LaTeX, automatic backup files, etc.

The way to tell Git to ignore these files is by listing them in a file called `.gitignore` at the root of the repository. This file can contain name files, but can also include wildcards. The typical content of a `.gitignore` file can look like this:

```
*.o
*.class
*~
```

For Python these contents will be slightly different and a sample `.gitignore` can be seen at `https://github.com/github/gitignore/blob/master/Python.gitignore`.

[6] Actually, branches in Git are implemented as aliases of commits, so it is basically the same thing. In other words, a branch is just a pointer that points to a commit and that gets updated every time there is a commit and that branch is active.

As you can see it is possible to add comments to a `.gitignore` file (with "#"), add exceptions to wildcard rules (with "!"), and many more useful tricks. Remember, `git help gitignore` is a good place to start looking for help.

The file `.gitignore` can (and should) be added and committed to your repository like any other file.