GIT - advanced tutorial

David Parsons, Soraya Arias, Thomas Calmant

June, 2018

Contents

1 Preamble **2**
1.1 Checking your **git** configuration 2
1.2 Setup 2

2 Toying with git and some of its concepts and tools **2**
2.1 Introduction 2
2.2 Detached HEAD, Remote-Tracking Branches and Upstreams 2
2.3 Patching and Cherry-Picking 5
2.4 Bisect, Stash and "Partial Commits" 6

3 Rewriting and correcting your history **10**
3.1 Introduction 10
3.2 Amend and rebase -i – an appetizer 11
3.2.1 Setup 11
3.2.2 Amending a commit 11
3.2.3 A simple interactive rebase 11
3.3 Rewriting history – difficult stuff 13
3.4 History and undoing 14
3.4.1 First use case 15
3.4.2 Second use case 15
3.5 Cleaning up your mess before pushing 16

4 Pull-Request / Merge-Request contributions **16**
4.1 Introduction 16
4.2 Pull request mode 17
4.3 Patch mode 17

5 Conclusion **18**

A One Solution for rewriting history **19**

1 Preamble

1.1 Checking your git configuration

Before using **git**, check that your **git** configuration is set correctly (at least you should have told **git** your committer name and email address).

```
$ git config -l
```

1.2 Setup

For this practical work, you will need :

- to retrieve a **git** repository from `github.com`
- to retrieve archive files from `https://parsons.eu/git/advanced`
- and eventually a GitHub account: see `https://github.com`.

2 Toying with git and some of its concepts and tools

2.1 Introduction

The aim of this section is double:

- To better grasp the concepts of `DETACHED` head, remote-tracking branches and upstream branches
- To experiment various tools from **git** such as patches, cherry picking, stash and bisect

For this section you need to retrieve (clone) a public project on `GitHub`, following these instructions:

```
$ # Create system aliases for git and gitk
$ alias g=git
$ alias gk="gitk --all"
$ # Clone project
$ git clone https://github.com/david-parsons/tp-git-intro-completed.git git-adv-s1
$ cd git-adv-s1
```

NOTE: Even though we define aliases in this handout we won't actually use them for the sake of clarity. Feel free to use those proposed here or others of your choice, they should make your life easier.

2.2 Detached HEAD, Remote-Tracking Branches and Upstreams

```
$ # Have a look around the repo you've just cloned
$ gitk --all&
```

You can notice that there is one local branch (`master`) and 2 remote tracking branches (`remotes/origin/master` and `remotes/origin/serialization`).

Another way of seeing that is by running:

```
$ # Create shorthand for branch
$ git config --global alias.br branch
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/serialization
```

This also tells you that the "default branch" on the remote is 'master' (`remotes/origin/HEAD` points to `origin/master`).

Since you've just cloned a repository, you "are on" the "default branch" (`master`).
You can convince yourself that's true *e.g.* by launching **gitk** (master is in bold) or with one of the following commands:

```
$ cat .git/HEAD
ref: refs/heads/master
$ # Create shorthand for status
$ git config --global alias.s status
$ git status
On branch master
[...]
```

Let's checkout some other commit:

```
$ # Create shorthand for checkout
$ git config --global alias.co checkout
$ git checkout 9550
Note: switching to '9550'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 9550cf9... Add function loadSphere
```

OK, **git** is pretty clear that we are in 'detached HEAD' state...
Let's check how this shows with the same 2 commands we've already tried:

```
$ cat .git/HEAD
9550cf915497227e25a51aff23afb1bf7d18dbc5
$ git status
HEAD detached at 9550cf9
[...]
```

We see that HEAD is pointing directly to a commit while it usually points to a branch

As **git** prompts us, we can "make experimental changes and commit them", let's do that:

```
$ echo "Hi there" > foo
$ git add foo
$ # Create shorthand for commit
$ git config --global alias.ci commit
$ git commit -m "experimental change"
[detached HEAD 398c0ec] experimental change
[...]
$ git branch
* (HEAD detached from 9550cf9)
  master
```

You can see (*e.g.* using **gitk**) that you have created a commit, that you are "on" it but that no branch references it, even indirectly.
This is what happens when committing in a detached HEAD state. Since HEAD was pointing *directly* to a commit (not via a branch), it is HEAD (.git/HEAD) itself that's been updated, not the branch (*e.g.* .git/refs/heads/master).
If this commit remains unreferenced, it will eventually be garbage-collected.

git has also suggested us to create a branch, let's do so:

```
$ git switch -c serialization
Switched to a new branch 'serialization'
$ # We now have 2 local branches (and still 2 remote-tracking branches:
$ git branch
  master
* serialization
$ git branch -r
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/serialization
```

We are no longer in 'detached HEAD' state, our commit won't get lost.

Now, if you run **git** branch again, but this time with option `-vv`, you can see that `master` has an upstream, namely `origin/master`, but serialization has none:

```
$ git branch -vv
  master        e444c1e [origin/master] Merge branch 'use_math_constants'
* serialization 398c0ec experimental change
```

One of the situations where having an upstream branch set is nice, is when you want to push your branch (**git** knows **where** to push it to).
Here a **push** will fail unless you tell **git** to which branch of which remote you want to push your changes:

```
$ git push
fatal: The current branch serialization has no upstream branch.
To push the current branch and set the remote as upstream, use
```

```
    git push --set-upstream origin serialization
```

Here, **git** prompts us to set our branch's upstream **while pushing**.
We can also just set the upstream (without pushing) with the following command:

```
$ git branch -u origin/serialization
Branch serialization set up to track remote branch serialization from origin.
$ # Now we have an upstream on both master and serialization:
$ git branch -vv
  master        e444c1e [origin/master] Merge branch 'use_math_constants'
* serialization 398c0ec [origin/serialization: ahead 1] experimental change
```

2.3 Patching and Cherry-Picking

There are basically 2 ways of creating a patch: using `git diff` or `git format-patch`.
Let's do both and compare the results.

```
$ # Create shorthand for diff
$ git config --global alias.d diff
$ git diff HEAD~ > exp-change.patch
$ git format-patch HEAD~
0001-experimental-change.patch
$ meld exp-change.patch 0001-experimental-change.patch
```

You can see that the (automatically named) patch generated by format-patch contains the exact same information as the other one, plus some additional data (committer, commit date, commit message, ...)

We can now apply each of these patches on 2 dedicated branches based on `master`:

```
$ git switch master -c patching-1
Switched to a new branch 'patching-1'
$ git apply exp-change.patch
$ # Applied the patch but did not commit (nor even stage the changes)
$ git add foo
$ git commit -m "exp-change"
```

```
$ git switch master -c patching-2
Switched to a new branch 'patching-2'
$ git am 0001-experimental-change.patch
Applying: experimental change
$ # Applied the patch and committed it with original commit's author and date
```

You can check that the results are identical:

```
$ git diff patching-1
```

Note that, because format-patch patches include information about the commit, only git diff/apply can be used for uncommitted changes.

On the other hand, `git format-patch`/`git am` allow you to keep track of commit metadata. `git format-patch`/`git am` also allow you to create/apply series of patches:

```
$ git format-patch -o patches f5d6..serialization
patches/0001-Add-method-dump.patch
patches/0002-Add-function-loadSphere.patch
patches/0003-experimental-change.patch
$ git switch master -c patching-3
$ git am patches/*
Applying: Add method dump
Applying: Add function loadSphere
Applying: experimental change
```

Actually, generating a patch from a [series of] commit[s] somewhere in the revision graph to apply it [them] somewhere else is so common that there is a dedicated command for that: `git cherry-pick`.

```
$ git switch master -c cherry-picking-1
Switched to a new branch 'cherry-picking-1'
$ # Create shorthand for cherry-pick
$ git config --global alias.cp cherry-pick
$ # Cherry-pick a single commit:
$ git cherry-pick serialization
[cherry-picking-1 6958add] experimental change
[...]
$ git switch master -c cherry-picking-2
Switched to a new branch 'cherry-picking-2'
$ # Cherry-pick a series of commits:
$ git cherry-pick f5d6..serialization
[cherry-picking-2 777bda5] Add method dump
[...]
[cherry-picking-2 59b9706] Add function loadSphere
[...]
[cherry-picking-2 977301c] experimental change
[...]
```

You may have noticed that the commits generated by either `git am` or `git cherry-pick` can have their committer distinct from their author

2.4 Bisect, Stash and "Partial Commits"

For this section, we will create a repository from a series of patches.
Run the following commands to get started:

```
$ # Create dir and cd into it
$ mkdir section_2.4 && cd section_2.4
$ # Init empty repository
$ git init
$ # Download and extract the archive
$ wget https://parsons.eu/git/advanced/hands-on/section_2.4.tgz
$ tar zxf section_2.4.tgz
```

```
$ # Have a look around
$ ls -1
fix#13.patch    # A patch containing a bugfix
patches         # A series of patches
section_2.4.tgz
wip.patch       # A patch containing work in progress
$ # Apply a series of patches (will create commits)
$ git am patches/*
$ # Apply another patch without committing, this is work in progress.
$ git apply wip.patch
```

You are in the middle of something (your working directory is 'dirty') when your boss tells you there is a bug in the app which needs to be fixed **right now**.

You have no choice but to stop what you're doing on the spot and address the bug, but you don't want to lose the work you were doing...

```
$ # Keep your work in progress safe by stashing it:
$ git stash
Saved working directory and index state WIP on master:
  a00081e Add missing new lines at eof
HEAD is now at a00081e Add missing new lines at eof
$ # View stashed changesets:
$ git stash list
stash@0: WIP on master: 02f1c61 Add missing new lines at eof
$ # Check that your working directory is clean
$ # (at least of modified and staged files)
$ git status
On branch master
nothing to commit, working tree clean
```

OK, you've saved you work for later. Now... the bug...

Before even trying to correct it, you should file it and try to reproduce it.

Boss tells you there's an error when running the program with no argument. Indeed, when you run it:

```
$ mkdir build && cd build && cmake ..&& make
$ # NOTE: if you don't have cmake you can use your compiler directly,
$ # e.g. g++ -o 2_4 ../*.cpp
$ ./2_4
terminate called after throwing an instance of 'std::logic_error'
  what():  basic_string::_M_construct null not valid
Aborted
```

OK, you can reproduce the bug. But to file it properly you should tell when it was introduced (and you have no idea !).

To find it with ease, we will use git bisect.

Git bisect needs to be run from the top level of the working tree. To make things easier, we will open a new terminal and cd into that directory.

In a new terminal:

```
$ cd /path/to/section_2.4
$ # Now let's start bisecting
```

```
$ git bisect start
$ # We've just tested HEAD and know it has the bug, i.e. is 'bad'
$ git bisect bad
```

We need to mark an earlier revision as 'good' for bisect to start bisecting. Yet, we have no idea where the error may have been introduced.
Let's checkout the very first commit and check if the bug existed then:
In the first terminal:

```
$ git checkout HEAD~7
[...]
$ # Test the revision
$ make && ./2_4
[...]
Hello, World!
```

This revision does not present the bug, we can mark it as 'good'
In the second terminal:

```
$ git bisect good
Bisecting: 3 revisions left to test after this (roughly 2 steps)
[c3bc7b5d7ace4b45a943affbac2d11b04f1367ad] Extract function hello_text()
```

Now that we've marked both a 'good' and a 'bad' commit, the bisection actually begins.
Have a look with gitk, you'll see that a commit has been marked as `bisect/bad` and another one as `bisect/good`. You'll also notice that you're "on" a third commit in between the `good` and `bad` commits.
The idea is that git will automatically checkout a commit to be tested and wait for you to mark it as either `good` or `bad`. It will then checkout another commit and so on until you find the "first bad commit", *i.e.* the commit that introduced the bug.

Keep checking revisions and mark them as either `good` or `bad` until you find the "first bad commit". The ouput should look something like that:

```
$ g bisect good
9de35b4dce2d8fb307d3677268fffbda5d48c099 is the first bad commit
commit 9de35b4dce2d8fb307d3677268fffbda5d48c099
Author: David Parsons <david.parsons@inria.fr>
Date:   Mon Mar 5 14:57:05 2018 +0100

    Pass argv[1] to hello_text

 main.cpp | 5 +++--
 1 file changed, 3 insertions(+), 2 deletions(-)
```

You can now file the bug (*e.g.* in your gitlab's issue tracker).
We won't actually file the bug today but it could contain the following information:

```
Title: std::logic_error when program lauched with no argument

Full message:
```

```
terminate called after throwing an instance of 'std::logic_error'
  what():  basic_string::_M_construct null not valid
Aborted

Introduce by commit 626f0ec0f7046c564bbbabf60f29157b6364195f
```

Let's consider our issue tracker has given our new issue the number #13. We can now create a branch dedicated to fixing this bug:

```
$ # Clean up after our bisect:
$ git bisect reset
Previous HEAD position was 86189c2... Add french "hello" text
Switched to branch 'master'
$ # Create a branch to fix the bug:
$ git switch -c hotfix/13
Switched to a new branch 'hotfix/13'
```

This is where you would work on fixing the bug. But before we do that let's see how we can retrieve our stashed changes:

```
$ # Checkout the branch where your stashed feature belongs to:
$ git switch master #Note: working on master is [very] bad habit!
$ git stash pop
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

	modified:   hello_text.cpp

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@0 (e9713d83ef2abeb94092f4edaf4e91d2a90a4360)
$ git stash list # Stash list is empty
```

OK, let's stash our work again and, this time, actually work on the bug:

```
$ git stash
Saved working directory and index state WIP on master:
  a00081e Add missing new lines at eof
HEAD is now at a00081e Add missing new lines at eof
$ git switch hotfix/13
Switched to branch 'hotfix/13'
```

While correcting the bug, you can't resist the temptation, you correct the typo "Bojour le monde" => "Bonjour le monde" and some other stuff. Run the following command (this simulates you fixing the bug):

```
$ git apply fix#13.patch
```

Now that you've completed the fix, you end up trying to write a commit message like: "Fix bug, add default value to param lang of fn hello_text and fix a typo"

Well, I have bad news, being tempted to write "X **and** Y" in a commit message could be referred to as a "git smell". Here you've made 3 changes and should hence split your changeset into 3 different commits. Plus, your fixing a typo or adding a default value to a function parameter has nothing to do in branch 'hotfix/13'

So, now we will try to build a "partial commit" containing the bugfix and only the bugfix: To do that, open `git gui` and add to your staging area only those modifications that should be part of the bugfix (remember you will have 2 more commits to do: typo and default param value).

Once you are satisfied with your modifications, commit your changes, stash the rest of your changeset and make sure your commit is OK (run tests etc). If you've missed something you can try again by unstashing your changes and ammending your commit. Iterate as many times as you need to get the whole bugfix and nothing else into you commit.

Right. Your bugfix is ready. You can use whatever procedure you have to push it to prod. This procedure may involve CI (Continuous Integration), MR (Merge Requests), code review and maybe even CD (Continuous Deployment).

For the purpose of this exercise, let's say the procedure is as simple as fast-forward merging the hotfix into `master`:

```
$ git checkout master
$ git merge hotfix/13
```

You can now commit the typo and default param value (in 2 different commits of course)

```
$ git stash pop
$ # Break the remainder of your changeset into 2 commits
$ [...]
```

And at long last you can now go back to the work you had in progress when your boss ordered you to fix the bug.

```
$ git stash pop
$ [...]
```

3 Rewriting and correcting your history

3.1 Introduction

The aim of this section is to take you through **git**'s history rewriting process on a local repository.

The first exercise presents a simple use case where you use `--amend` and start experimenting with `rebase -i`.

The second exercise is a little trickier because you will be less guided on how to do things and you will face a more extensive use of `rebase -i`.

3.2 Amend and rebase -i – an appetizer

3.2.1 Setup

Keep in mind that all the actions to be performed here concern a local repository.

- First, create and initialize a new **git** repository locally.

```
$ mkdir appetizer
$ cd appetizer
$ git init
```

3.2.2 Amending a commit

- Create a `README.md` file and add it to you **git** repository

```
$ touch README.md
$ git add README.md
$ echo "Hello" >> README.md
$ git commit
   [master (root-commit) c66c567] add readme
   1 file changed, 0 insertions(+), 0 deletions(-)
   create mode 100644 README.md
$ # Visualize your commit graph (keep gitk window opened)
$ gitk --all&
```

- You realize you've actually added an *empty* readme. Yet you wanted to commit the content as well. You can amend your commit using `git --amend`

```
$ git stage README.md
$ git commit --amend
   [master 2c3df5a] add readme
   Date: Mon Feb 5 22:50:02 2018 +0100
   1 file changed, 0 insertions(+), 0 deletions(-)
   create mode 100644 README.md
```

- Refresh your **gitk** window, you should see both your initial commit with the empty readme (which is now dangling) and the new one (pointed to by `master`) that adds the readme file and its content in a single commit.

  Warning: do not do that if you've pushed the faulty commit. As easy as it might seem, a `commit --amend` is still a history rewrite!

3.2.3 A simple interactive rebase

- Add a file, `config.ini`, containing a login and password in plain text. This could be a file that would be necessary during your development and you think you'd provide it to your fellow developers.

```
$ echo "login=admin" > config.ini
$ echo "passwd=12345" >> config.ini
$ git add config.ini
$ git commit
   [master 0ae90a7] add config file
```

```
 1 file changed, 2 insertions(+)
 create mode 100644 config.ini
```

- Then, create a `howto.txt` and update your repository.

```
$ touch howto.txt
$ git add howto.txt
$ git commit
  [master 051a221] add howto
  1 file changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 howto.txt
```

- Oh no! You eventually realize you have added sensitive information (login and password in plain text) in a previous commit. Now because the faulty commit is not at the tip of a branch, you can't use `git commit --amend` to amend it.

  Thankfully, you have not pushed the faulty commit yet. You can hence use the interactive rebase feature to get rid of the commit adding sensitive data.

- Your goal here is to entirely remove the commit that added this file.

  To do that, you will run an interactive rebase.

  - First, identify the commit reference corresponding to the commit you want to rebase from, *i.e.* the one before adding `config.ini`.

    ```
    $ git log --pretty=oneline
    aa602108334ee488f407c51b75bfcc87190b9e06 (HEAD -> master) add howto
    4314b0df88b810b27e90c289bdcfa9d08d5f0964 add config file
    3fa3ded324772c0fe4e76b4b7762e69b6bb00e2e add readme
    ```

    According to the logs above, the commit we need to reference (the last commit we want to preserve as-is) is `3fa3ded3`.

  - Run the `rebase -i` command with the commit reference you have identified:

    ```
    $ git rebase -i COMMIT_REFERENCE_FOR_REBASE
    ```

    Your favorite text editor should pop up with the following instructions (or similar):

    ```
    pick 4314b0d add config file
    pick aa60210 add howto

    # Rebase 3fa3ded..aa60210 onto 3fa3ded (2 commands)
    #
    # Commands:
    # p, pick <commit> = use commit
    # r, reword <commit> = use commit, but edit the commit message
    # e, edit <commit> = use commit, but stop for amending
    # s, squash <commit> = use commit, but meld into previous commit
    # f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
    #                    commit's log message, unless -C is used, in which case
    #                    keep only this commit's message; -c is same as -C but
    #                    opens the editor
    # x, exec <command> = run command (the rest of the line) using shell
    # b, break = stop here (continue rebase later with 'git rebase --continue')
    ```

```
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified); use -c <commit> to reword the commit message
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
```

This file is basically a script that we can edit to do as we like.

Here, what we want is to get rid of the commit that added the configuration file altogether. We do however want to keep the commit that adds file `howto.txt`. Hence, we can simply `drop` the commit that introduced the configuration file.

– Edit the file to drop the faulty commit:

```
drop 4314b0d add config file
pick aa60210 add howto

# [...]
```

Note that you could also have removed the line about the commit to drop.

– Save the file and exit the editor for the rebase process to run. Once it is done, you should get the following message:

```
Successfully rebased and updated refs/heads/master.
```

– Refresh your **gitk** window, you should see both your initial 2 commits (which are now dangling) and the new one (pointed to by `master`) that adds the howto file.

3.3 Rewriting history – difficult stuff

For this use case, you will be less guided.

1. First you need to work on a **new directory**.
2. Then you need to retrieve and uncompress the following archive file:
   - `https://parsons.eu/git/advanced/hands-on/section_3.3.tgz`

   This archive contains a local **git** repository, not yet associated to a remote.
3. Take a look at the commit history of this repository:

   ```
   $ git log
   ```

   You might also use **gitk** to better understand the different succession of commits.

   You can state from this history that some commit messages are misspelled, some commits should have been performed differently to ensure atomic commit, etc.

   Therefore, this history needs a good cleanup: `git rebase -i` is now your best friend.

4. So we ask you to rewrite this history keeping in mind that you need to:
    - ensure commits are atomic
    - smarten commits that are related
    - fix spelling for commit messages

   To start rewriting this history, we recommend you use `git rebase -i --root`. This way, you begin at the origin commit (`root`) of the repository.

   Instead, you can use the reference to a commit, i.e. `git rebase -i COMMIT_REFERENCE_FOR_REBASE`.

Some hints

- Sometimes trying to do everything at once is not the best way to go...
- Look at the commit messages, they describe what you might want to do during the rebase operation
- If you want to split a commit, you have to mark it as "edit" when using `git rebase -i`. Once you are editing this commit, run `git reset HEAD~` to go back to the state right before this commit. After having committed your changes run `git rebase --continue`.
- Use `git status` often: it will tell you the current state of the repository and give you advice on what to run next.
- You may need to perform partial commits, see `git add -p` (you can also use `git gui`) and `git diff --cached`.
- After resolving merge issues, run `git rebase --continue` instead of `git commit` (**git** will tell you if you run `git status`).
- Use **gitk** to ensure your progress is correct.

For a solution to this problem, see section A.

3.4 History and undoing

In this subsection you will be less guided regarding the commands needed to complete the required task.

Here we focus on how to undo committed modifications both locally and on a remote repository.

We propose two very similar use cases but that require two different sets of **git** commands to be solved, so be smart/cautious. We do give you some hints :-) !

Hints

- Check the help for `git revert` and `git checkout`
- You will be asked to undo things when commits have been pushed (to a remote repository). To that end, you can emulate a remote repository by creating a local "bare" repository. Here are the commands you will need to create and link to a "bare" repository (don't do it just now):

```
$ mkdir ../pseudo-remote.git
$ cd ../pseudo-remote.git
$ git init --bare
$ cd -
$ git remote add origin ../pseudo-remote.git
$ git push origin master
```

3.4.1 First use case

1. First you need to work on a **new directory**.

2. Retrieve the archive you can find at:
   `https://parsons.eu/git/advanced/hands-on/section_3.4.1.tgz`.

3. Untar this archive and check the status of this local repository:

   ```
   $ tar xvfz section_3.4.1.tgz
   $ cd section_3.4.1
   $ git log
   $ # OR
   $ gitk
   ```

   You should identify the following situation:

   - Four successive commits have been made.
   - Some files added in the first commit have been removed in the second one.

4. We ask you to:

   - **"Undelete" the deleted files** (they should be present at the tip of master after your intervention). How can you do this?
   - Consider both cases:
     - none of the commits have been pushed to a remote repository yet,
     - all commits have already been pushed (see Hint paragraph above for the use of "bare" repo).

3.4.2 Second use case

1. First you need to work on a **new directory**.

2. Retrieve the archive you can find at:
   `https://parsons.eu/git/advanced/hands-on/section_3.4.2.tgz`.

3. Untar this archive and check the status of this local repository.

   ```
   $ tar xvfz section_3.4.2.tgz
   $ cd section_3.4.2
   $ git log
   $ # OR
   $ gitk
   ```

   You should identity the following situation:

   - Four successive commits have been made.
   - Only a single line within file `readme.txt` has been modified by the second commit.

4. We ask you to:

   - **Remove the update** made in `readme.txt` within the second commit. How can you do this?
   - Consider both cases:
     - none of the commits have been pushed to a remote repository yet,
     - all commits have already been pushed (see Hint paragraph above for the use of "bare" repo).

3.5 Cleaning up your mess before pushing

This exercise aims at illustrating how to use the `git reset` functionality within 3 independent use cases but based on the same local **git** repository.

1. First you need to work on a **new directory**.
2. Retrieve the archive you can find in:
   `https://parsons.eu/git/advanced/hands-on/section_3.5.tgz`.
3. Untar this archive and check the status of the following working tree.

```
$ tar xvfz section_3.5.tgz
$ cd section_3.5
$ git log
$ # OR
$ gitk
```

4. Each of the following use cases is independent from the other 2. You need to restart from the original contents of `cleanBeforePushing.tgz` archive for each case.

Case 1 You realize your last changes on `license.txt` are unnecessary and you want to go back to a previous version:

```
$ git checkout COMMIT_REFERENCE_WITH_PREVIOUS_LICENSE -- license.txt
```

Case 2 You have added `license.txt` to the index but realize the changes made to it should not be included to the next commit. How can you go back without deleting `license.txt` file?

```
$ git reset --mixed        # Resets the whole index, or
$ git reset -- license.txt # Removes license.txt from the index
```

Case 3 You have modified files and added some of them in the index, but you realize everything you did there was wrong. Therefore, you want to get back to the state of the latest commit and discard the changes you had made.

```
$ git reset --hard # Resets the index AND the working dir (be careful)
```

4 Pull-Request / Merge-Request contributions

4.1 Introduction

The aim of this section is to take you through the merge-request (a.k.a. "pull-request") mode when contributing to a **git** project.

For this exercise, we provide a project on `https://github.com` that you need to fork.

Why fork this project and not clone it directly? Because in "pull-request" mode, you are not allowed to push directly to the upstream repository.

Forking is actually copying the content of the repository into your own remote repository (a new project entry on your account page). Then you can clone it on your computer.

Here are summarized the prerequisites for this exercise:

1. Check you can sign in on `https://github.com` (otherwise you need to sign up).
2. The base project you will build upon is located at `https://github.com/david-parsons/tp-git-advanced-prmr`. Fork the project by clicking on the *fork* button in the upper right corner of the project page.

3. Go back to your profile page, and check that a `tp-git-advanced-prmr` project is listed.

4. Finally, *clone* the forked repository. You will find the url of your git repository by clicking the *Code* button (choose the SSH URL).

4.2 Pull request mode

We ask you to:

- Take a look a the repository files. The files of interest are `runsphere` and `sphere/sphere.py`. The last Python file computes sphere related attributes such as its surface, volume, diameter given a radius.

- Run `runsphere` with option -svd and check if the results are correct. For instance, if radius is set to 2.0, you should get:

  - sphere surface: 50.265
  - sphere volume: 33.51
  - sphere diameter: 4.0

- Correct the `sphere/sphere.py` (if needed :-)) and commit locally.

- Push your changes to your github fork.

```
$ git push
```

- To contribute to the original repository `david-parsons/tp-git-advanced-prmr` with your changes, you need to perform a *pull request*.
  Go to the GitHub Web page corresponding to your forked project, find the `Code` tab and check that your push has been done. Then click on the `New pull request` button and follow the instructions to create the request.

Note If you want to retrieve in your own repository changes made on original/upstream repository `david-parsons/tp-git-a`
you can use:

```
$ git remote add source https://github.com/david-parsons/tp-git-advanced-prmr.git
$ git fetch
$ git branch -a # to check the branches
$ git rebase # to merge modification from source original repo to your local repo
```

4.3 Patch mode

Another way to contribute to a project you don't have permission to push to is to send a [list of] patch[es] containing your modifications.[1]

To do so:

- From the current state of your local repository, create a patch between the original/upstream repository version and your current repository state:

```
$ git format-patch  REFERENCE_TO_THE_ROOT_REPO_BASE_COMMIT
```

- You should get a file `.patch` per commit. These files include the commit author name and email.

[1] If you are using Inria Gitlab, `http://gitlab.inria.fr`, it is one of the methods recommended to contribute. See Inria Gitlab FAQ for more information.

- You can e-mail the files to the owners of the root repository.
- See `git am --help` to understand how to apply this kind of patch sent by mail.

5 Conclusion

You should now have performed a lot of reasonably advanced **git** operations, locally and using a remote **git** repository.

A One Solution for rewriting history

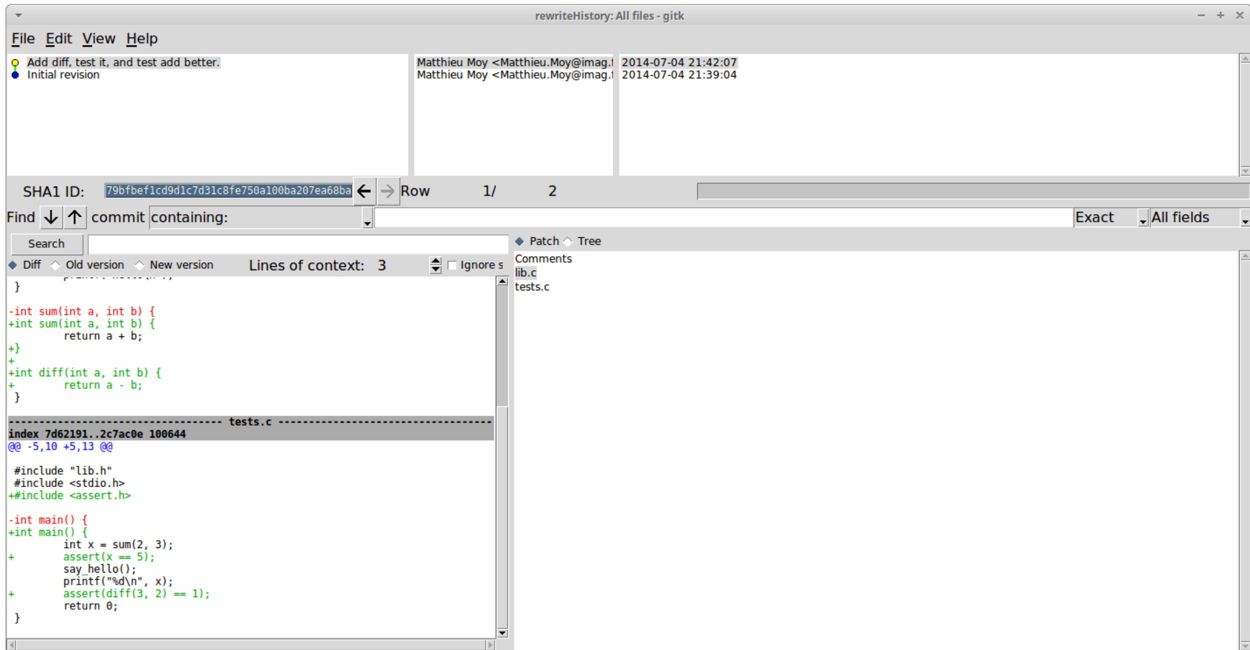
We start rebasing interactively the local repository from `root`.

```
$ git rebase -i --root
```

We want to edit the second commit to add first the sum method, then in a another commit the adding and testing of diff method.

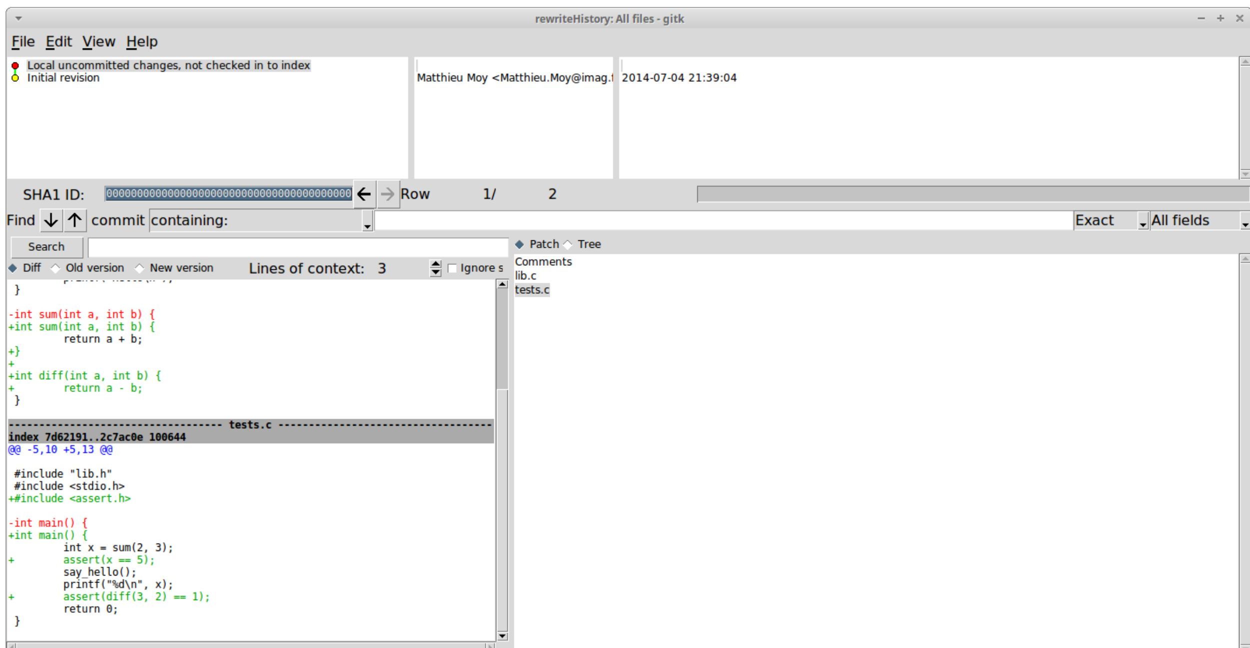
```
pick c5603bc Initial revision
e bd28b12 Add diff, test it, and test add better.
pick 1c5ca4c Rename diff to sub
pick 2976e39 Rename diff to sub in tests too
pick 495c6a0 add fnuctino f

# Rebase 495c6a0 onto 260caa8 (5 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
-- INSERT --                                            4,1           Top
```

```
$ git status
interactive rebase in progress; onto 46ee46e
Last commands done (2 commands done):
   pick c5603bc Initial revision
   e bd28b12 Add diff, test it, and test add better.
Next commands to do (3 remaining commands):
   pick 1c5ca4c Rename diff to sub
   pick 2976e39 Rename diff to sub in tests too
  (use "git rebase --edit-todo" to view and edit)
You are currently editing a commit while rebasing branch 'master' on '46ee46e'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

nothing to commit, working directory clean
$ gitk
```



As we want to reconsider the current commit (the one concerning adding diff and test add better) in order to split it into one commit for testing add and another to add and test diff, we need to change the HEAD of the current repository to be able modify this commit

```
$ git reset HEAD~
$ gitk
```



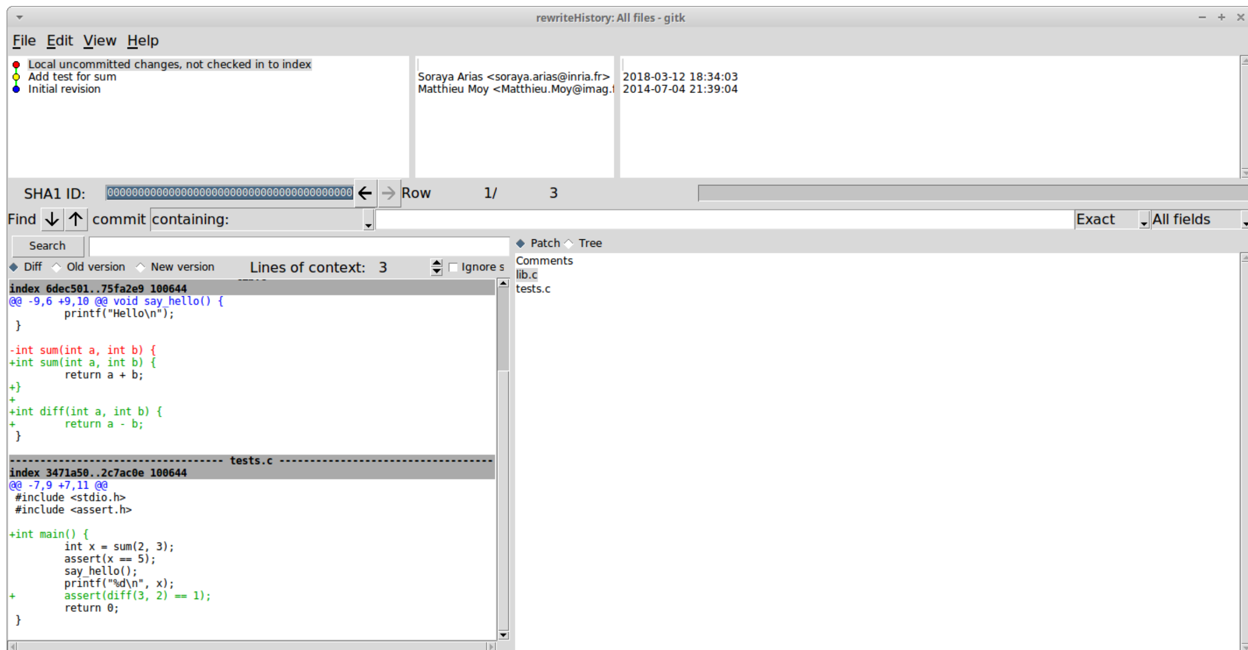
Then, we can modify `tests.c` to take into account only the test on add method. But we do not want to lose the modifications about the diff method, so we perform a *partial* commit modification.

```
$ git add -p tests.c
```

```
# Manual hunk edit mode -- see bottom for a quick guide
@@ -5,10 +5,13 @@

 #include "lib.h"
 #include <stdio.h>
+#include <assert.h>

-int main() {
        int x = sum(2, 3);
+       assert(x == 5);
        say_hello();
        printf("%d\n", x);
        return 0;
 }
# ---
# To remove '-' lines, make them ' ' lines (context).
# To remove '+' lines, delete them.
# Lines starting with # will be removed.
#
# If the patch applies cleanly, the edited hunk will immediately be
# marked for staging. If it does not apply cleanly, you will be given
# an opportunity to edit again. If all lines of the hunk are removed,
# then the edit is aborted and the hunk is left unchanged.
~
```

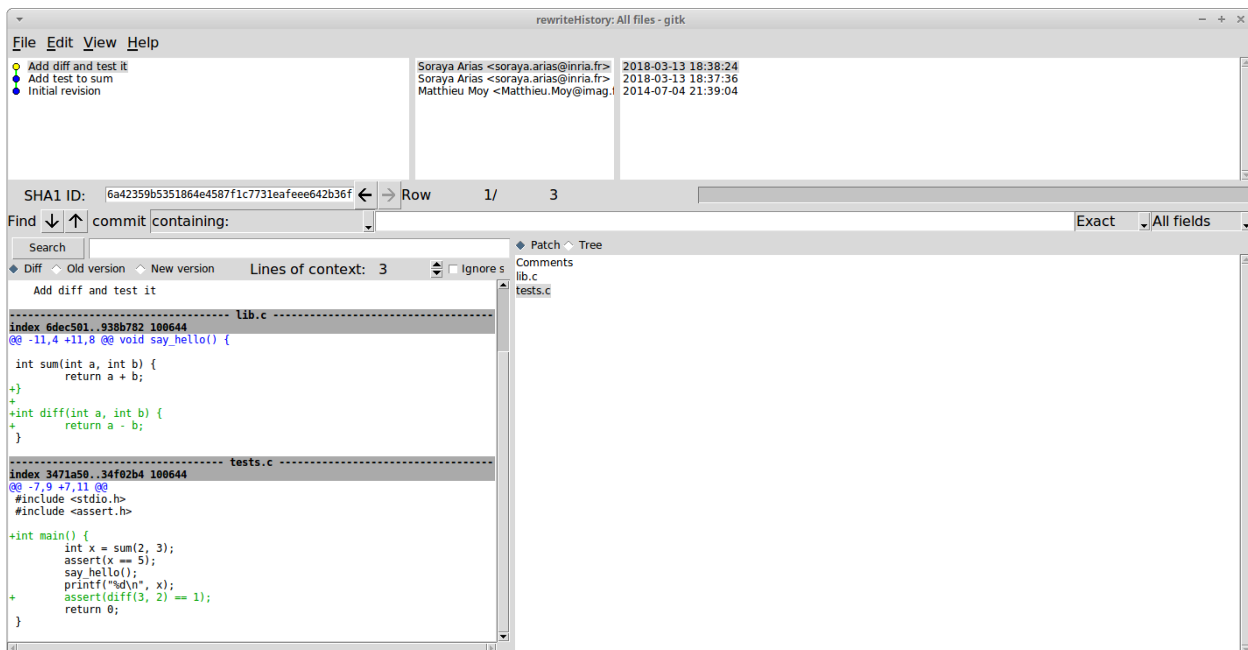
```
$ git status
interactive rebase in progress; onto 46ee46e
Last commands done (2 commands done):
   pick c5603bc Initial revision
   e bd28b12 Add diff, test it, and test add better.
Next commands to do (3 remaining commands):
   pick 1c5ca4c Rename diff to sub
   pick 2976e39 Rename diff to sub in tests too
  (use "git rebase --edit-todo" to view and edit)
You are currently splitting a commit while rebasing branch 'master' on '46ee46e'.
  (Once your working directory is clean, run "git rebase --continue")

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   tests.c

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   lib.c
        modified:   tests.c
```

```
$ git commit -m "Add test for sum"
$ gitk
```



Now you can edit `lib.c` and `tests.c` and delete unnecessary spaces after main() and sum() definitions. Then, you can add and commit theses modifications.

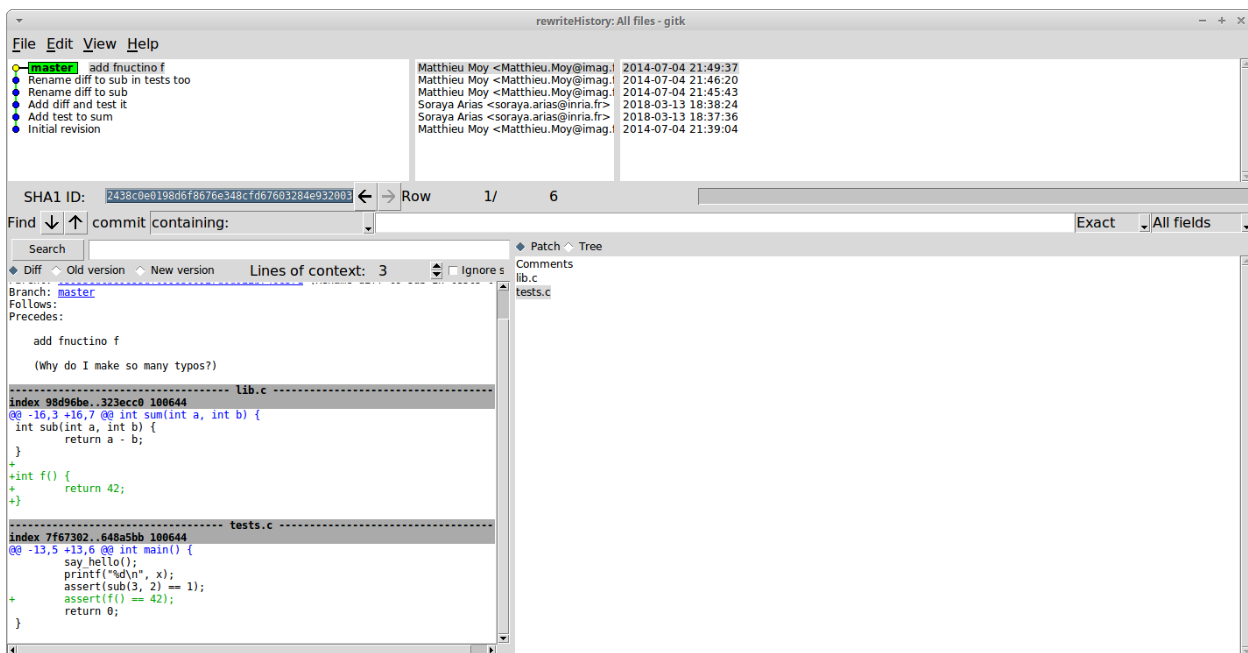
```
$ git stage lib.c tests.c ; git commit -m "Add diff and test it"
$ gitk
```



If we take a look to what `git status` tells us and as we are satisfied with previous changes, we can continue rebasing:

```
$ git status
interactive rebase in progress; onto f76f6a5
```

```
Last commands done (2 commands done):
   pick c5603bc Initial revision
   e bd28b12 Add diff, test it, and test add better.
Next commands to do (3 remaining commands):
   pick 1c5ca4c Rename diff to sub
   pick 2976e39 Rename diff to sub in tests too
  (use "git rebase --edit-todo" to view and edit)
You are currently editing a commit while rebasing branch 'master' on 'f76f6a5'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

nothing to commit, working directory clean
$ git rebase --continue
Successfully rebased and updated refs/heads/master.
$ gitk
```



We can continue rewriting the history to squash or fix diff related commits and correct mispelled comment. So we can use `git rebase -i` to perform these modifications on history.

```
$ git rebase -i --root
```

```
pick 7818e1e Initial revision
pick 97f61a1 Add test for sum
pick 4fc133c Add diff and test it
pick 9708dbc Rename diff to sub
f 0aa493f Rename diff to sub in tests too
r 171a8a6 add fnuctino f

# Rebase 171a8a6 onto e8f0d6b (6 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
~
```

Here is the editor to reword the mispelled comment:

```
add function f

(Why do I make so many typos?)

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Author:    Matthieu Moy <Matthieu.Moy@imag.fr>
# Date:      Fri Jul 4 21:49:37 2014 +0200
#
# interactive rebase in progress; onto e8f0d6b
# Last commands done (6 commands done):
#    f 0aa493f Rename diff to sub in tests too
#    r 171a8a6 add fnuctino f
# No commands remaining.
# You are currently editing a commit while rebasing branch 'master' on 'e8f0d6b'.
#
# Changes to be committed:
#       modified:   lib.c
#       modified:   tests.c
#
~
~
```

You should get an output message telling you the rebase is successfull:

```
$ git rebase -i --root
[detached HEAD cccb2d9] Rename diff to sub
 Author: Matthieu Moy <Matthieu.Moy@imag.fr>
 Date: Fri Jul 4 21:45:43 2014 +0200
```

```
 2 files changed, 2 insertions(+), 2 deletions(-)
[detached HEAD 6c0acf5] add function f
 Author: Matthieu Moy <Matthieu.Moy@imag.fr>
 Date: Fri Jul 4 21:49:37 2014 +0200
 2 files changed, 5 insertions(+)
Successfully rebased and updated refs/heads/master.
```

And finally you should retrieve a more coherent history !

```
$ gitk
```