Working with GIT

Florido Paganelli
Lund University
florido.paganelli@hep.lu.se
Fysikum, Hus A, Room 403
Visiting time: 11:00-12:00 Every day
Or use Canvas
Or use github!

Required Software

- **Git** - a free and open source distributed version control system
- **Gitg** – a fast git repository viewer (there are many!)

Command line installation (bash):

sudo apt-get install git gitg

Note: this software is NOT installed by default by the Lubuntu system installation.

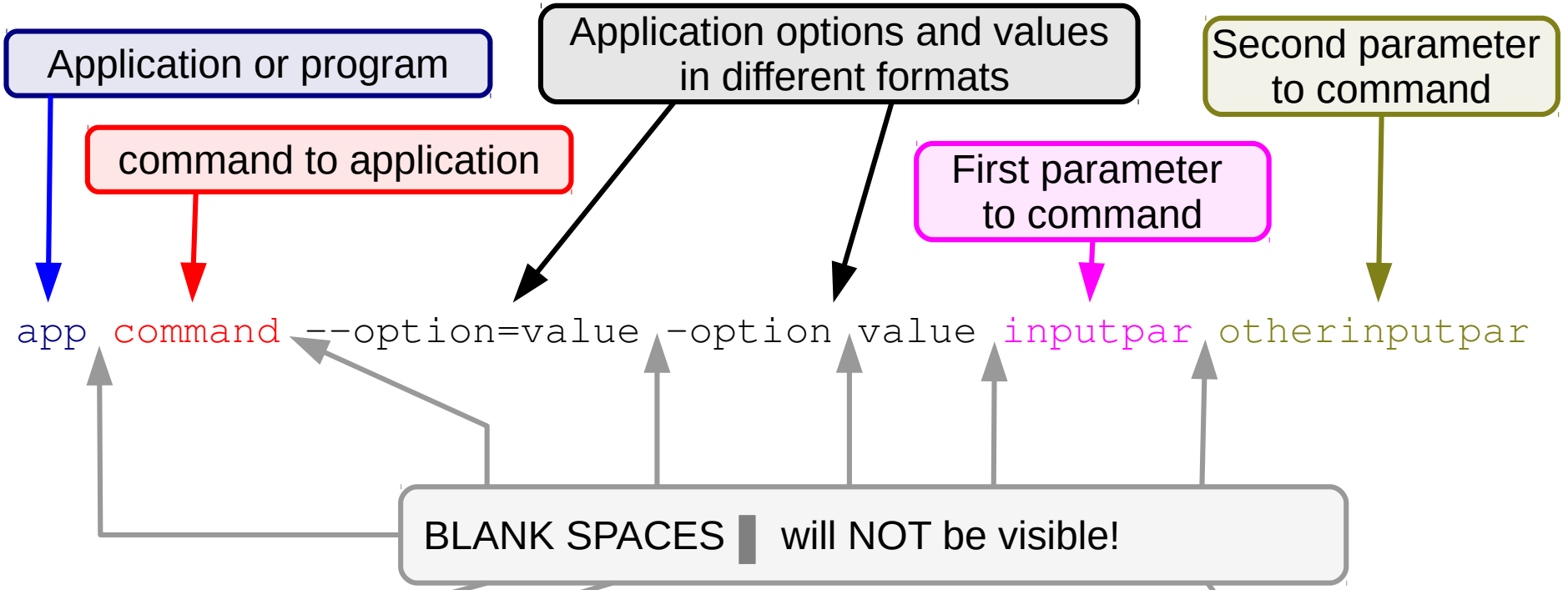
Platform	Package names
Ubuntu, Debian	**git, gitg**
RedHat, CentOS, Fedora, SuSE	**git, gitg**
Windows	http://www.jamessturtevant.com/posts/5-Ways-to-Install-git-on-Windows/
Mac OS	http://www.macworld.co.uk/how-to/mac-software/how-use-git-github-on-your-mac-3639136/

Outline

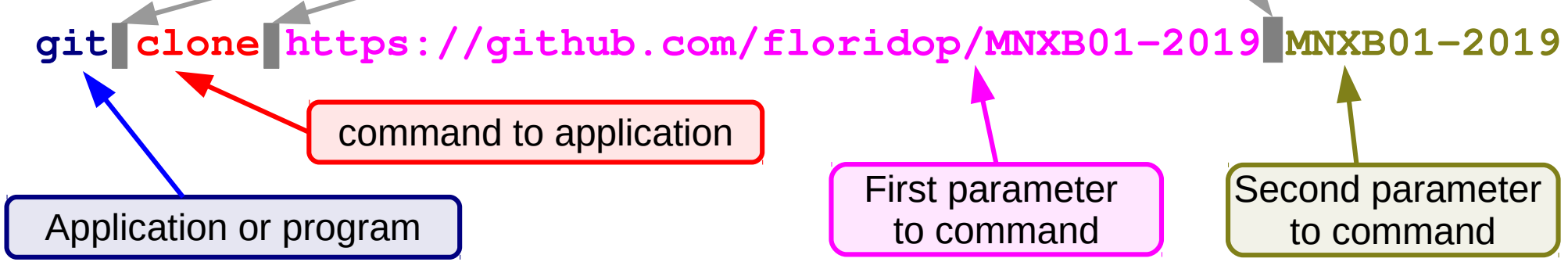
- What are version/revision control systems
  - Generic concepts of version/revision systems
- git
  - Generic concepts of git
  - git tutorial
  - Additional useful commands

Notation

- I will be using the following color code for showing commands:



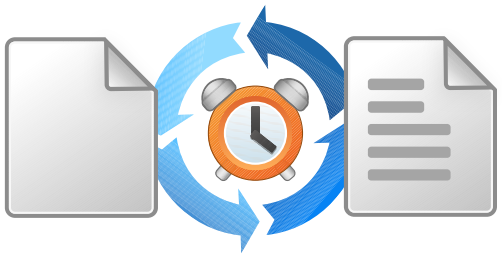
- Example:



Revision systems concepts

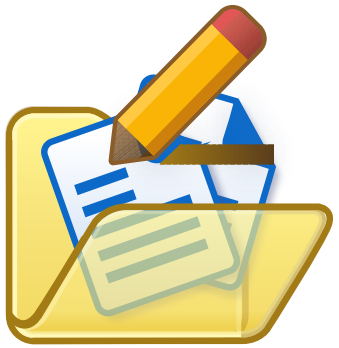
Why version/revision systems?

- Say you wrote some computer program in a text file.
- You discover a bug, something that does not work as it should, and you want to change it.
- You fix the bug, save the file. Try the program again and... **it doesn't work anymore!**
- **What went wrong?** Would be nice if you could **compare** what you **changed**...
- **Solution:** make a backup copy before every (important) change!
- Version systems make it easy to backup and compare **changes**



Why version/revision systems?

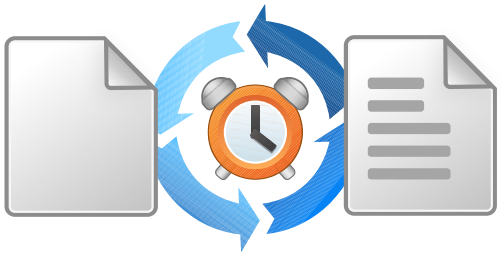
- If you do *many changes*, you might not remember what changes you made. Version systems allow you to attach a **comment** to the change.

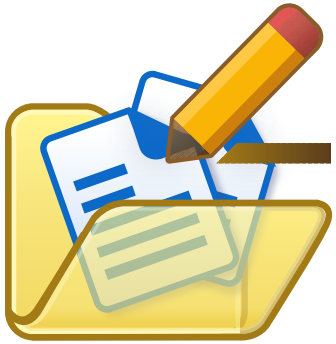


- If you want to *share* your code with *other developers*, it's nice if everybody can see who changed what. Version systems allow you to **author** the changes so the others know what you're done. This helps **sharing** code.

Why version/revision systems?

- Summary:
  - **Backup** each change in your code
  - **Compare** different versions of your code
  - **Comment** and annotate each change
  - **Share** among developers





Version systems: products and features

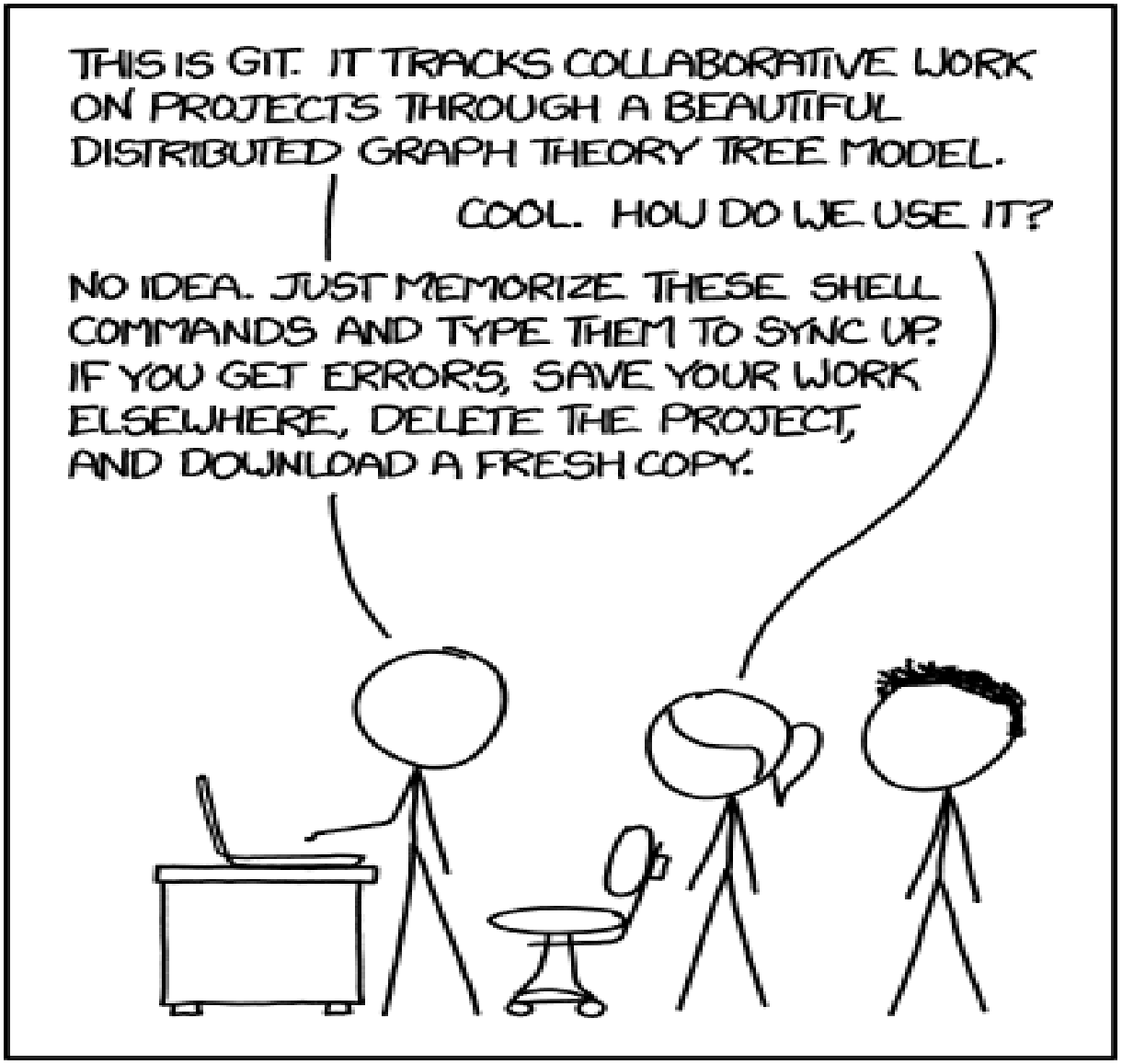
Product	staging	Local commit	diff	Fork/branch management	Distributed/ Collaborative	Compatibility
CVS (Current Version Stable)	N	N	Y	Y	N	?
SVN (SubVersioN)	N	N	Y	N	N	?
Git	Y	Y	Y	Y	Y	SVN CVS

Git: vocabulary and concepts

What and why git

- Was created by Linus Torvalds especially for kernel development
  - Highly distributed community contributions
  - Lots of people *forking* and writing their own version of drivers (later I'll explain this term)
- Nowadays there are many collaborative websites systems that use it to share code (github, gitlab) and make it easier to integrate everyone's work with discussion and code revision/testing tools
- Is being used by many because is a free solution that helps distributed cooperation
- Becoming the most used among research projects
  - In other words, mostly **fashion**

Git ain’t the best.



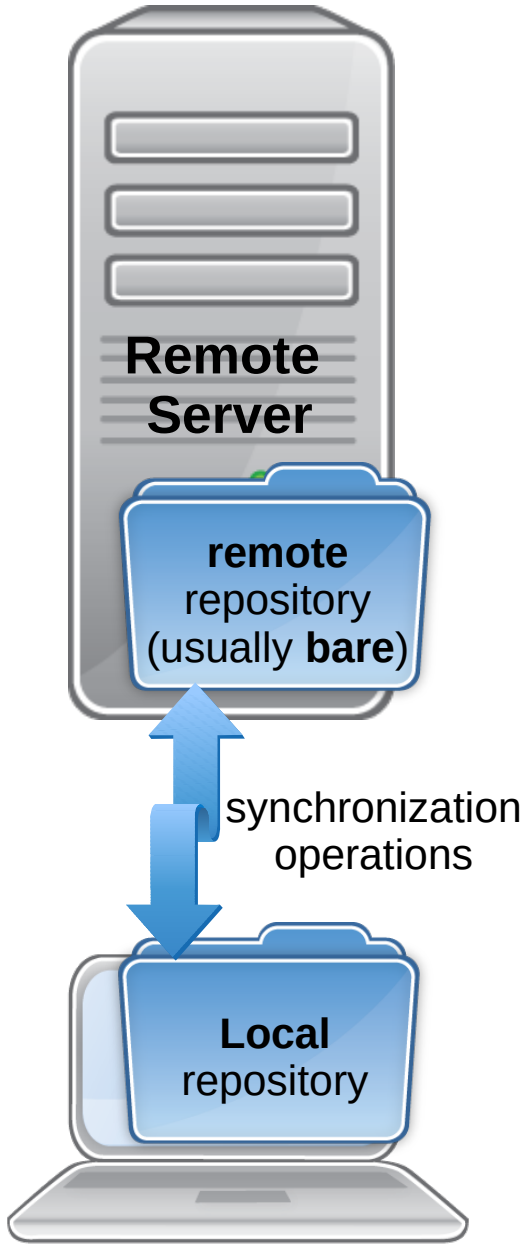
https://xkcd.com/1597/

Why using git in this course

- The VM you are using can be deleted any time. Everything you save in its virtual harddisk can be lost anytime.
- The VM runs on the machine you're sitting and it can be accessed by other users. Other users can change what you did on the machine and you will lose all data.
- You will become a better programmer (but not necessarily a better person)
- Suggestion: at the end of each tutorial, **push your changes to the remote github repository** we will create in the Homework.
- **The final course project** material you will create **can be only handed out using a github repository**, so get familiar with git!

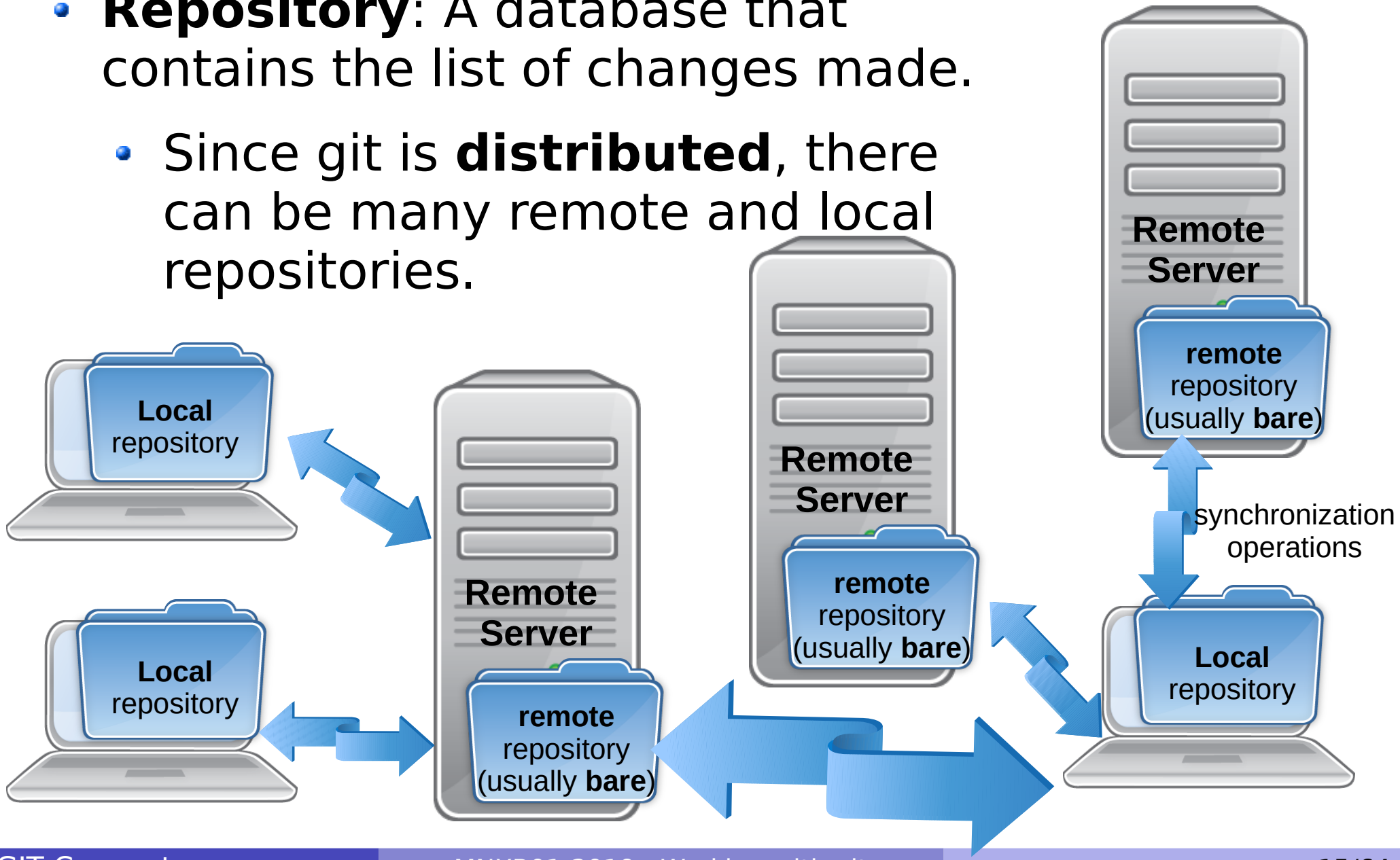
Concepts of version systems in git

- **Repository**: A database that contains the list of changes made.
  - A **local** git repository is shared *locally on your machine* in the **`.git`** invisible folder
  - A **remote** git repository is shared on a *remote server* and can be reached using a URL, like `https://github.com/floridop/MNXB01-2019.git`
  - A **bare** git repository can be stored in any folder and contains data in a form that only the git code understands. Can be used to have multiple copies of the same repository. It can be used to share a repository without GitHub.



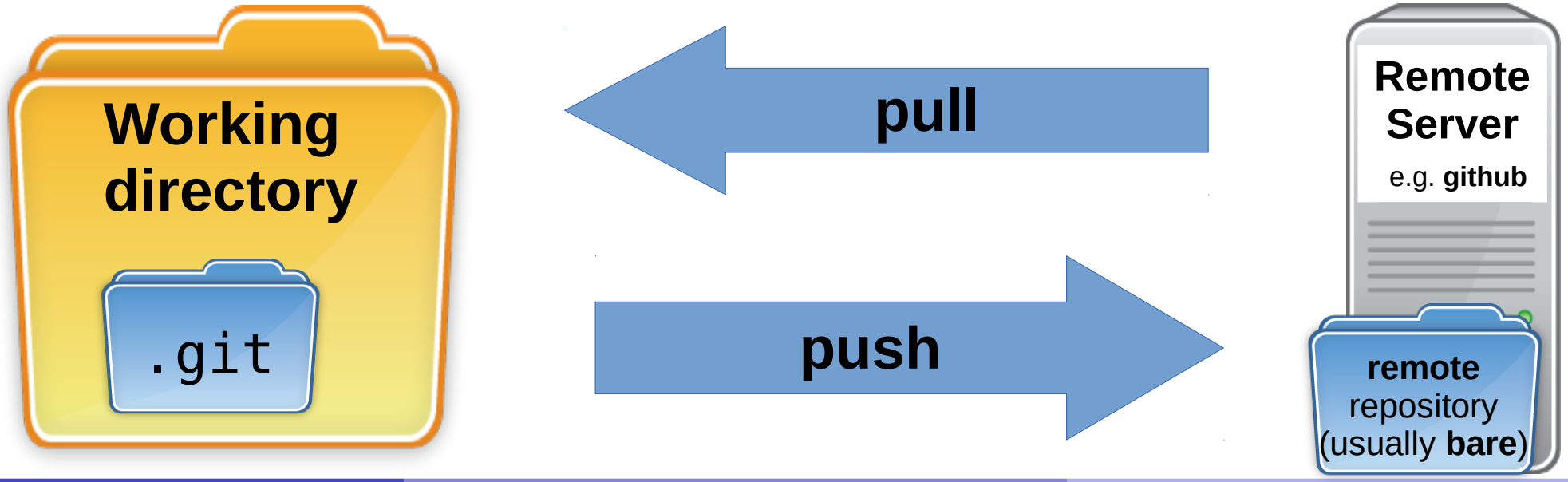
Concepts of version systems in git

- **Repository**: A database that contains the list of changes made.
  - Since git is **distributed**, there can be many remote and local repositories.



Concepts of version systems in git

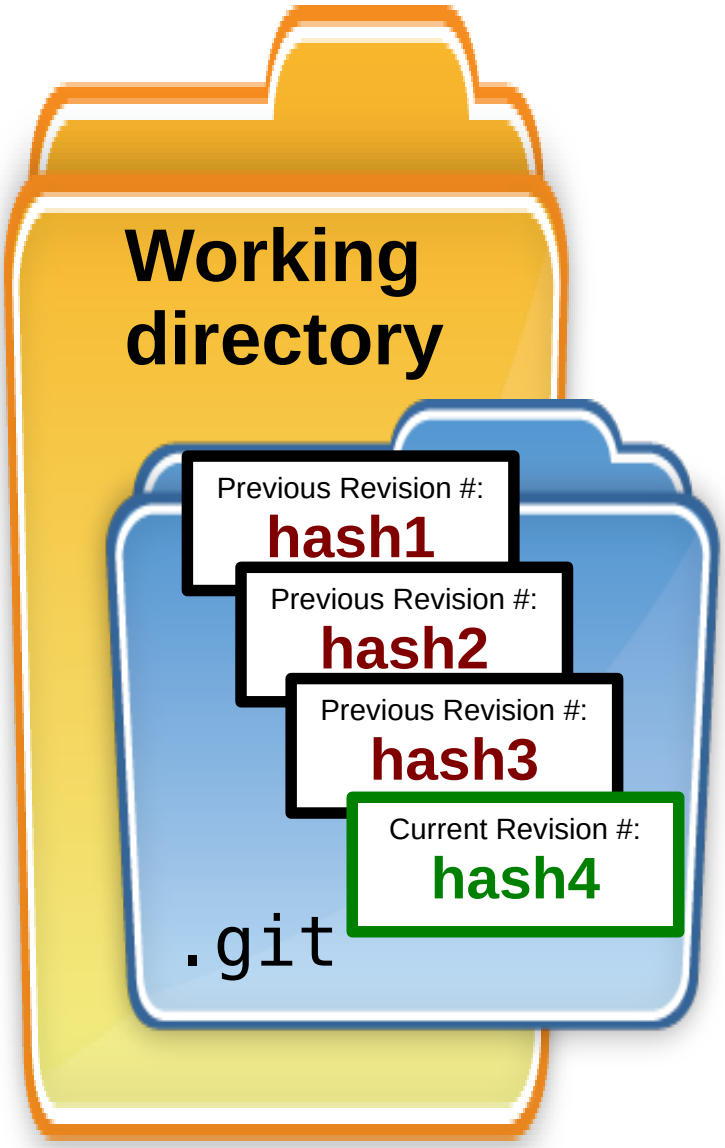
- **Working directory**: the latest version of a set of files that you want to work on. This is usually **local** to your machine.
  - It is usually the result of a **clone**, an exact copy, of some remote repository
  - You can synchronize the local git repository with remote ones using the **push** (send changes) and **pull** (retrieve changes) commands.
  - A bit like DropBox but NOT automatic.



Concepts of version systems

- When one is happy with the changes they made, it records them in the database by doing a **commit**
- A *committed* set of files is called a **revisions** and gets a **commit ID**: every “version” of one or more files gets a **revision tag**. This can be a number, a label, a string.
- In git usually is an **hash***, a strange sequence of symbols. It:
  - **Identifies the repository and other details** of when the changes where made
  - It’s **universally unique**, everywhere in the world that commit will represent a defined sequence of changes.
  - For this reason these systems are also known as **Revision Systems**, as every revision gets a **label** that depends on **time** and **person** who made the change.

*Hash: a special injective function that returns a value from a finite set of strings. The return values are unique under certain conditions.

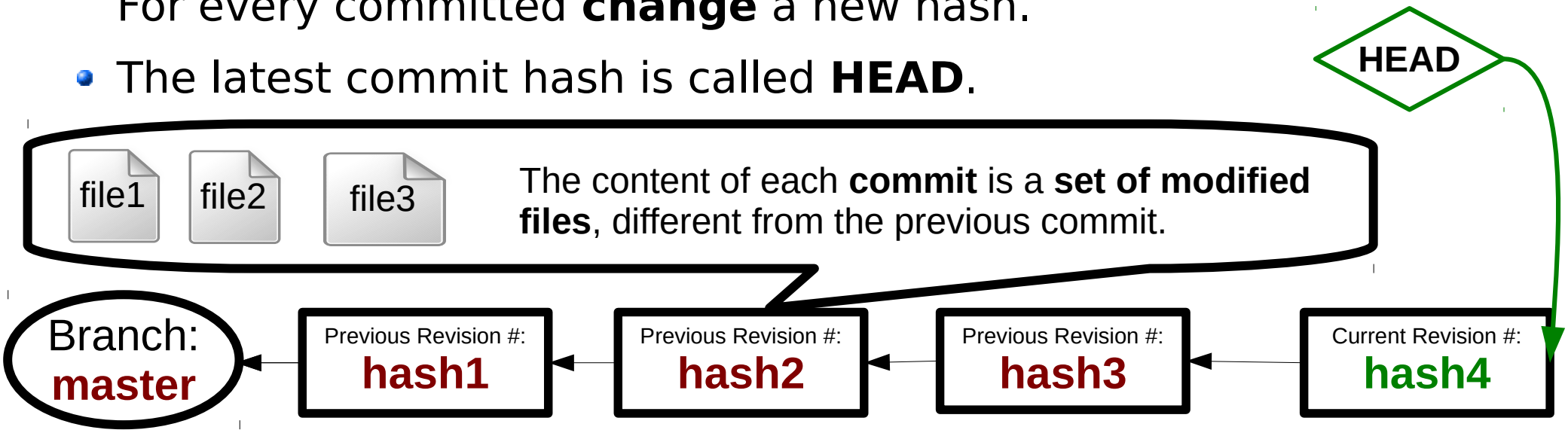


Commit example

```
commit 245dceab78b387020bf5c9de18e5ec599237e4dd (HEAD -
> master, origin/master, origin/HEAD)
Author: Florido Paganelli <florido.paganelli@gmail.com>
Date:   Thu Sep 12 14:24:32 2019 +0200

    fixed wrong formatting

    changed italic to bold

README.md
```

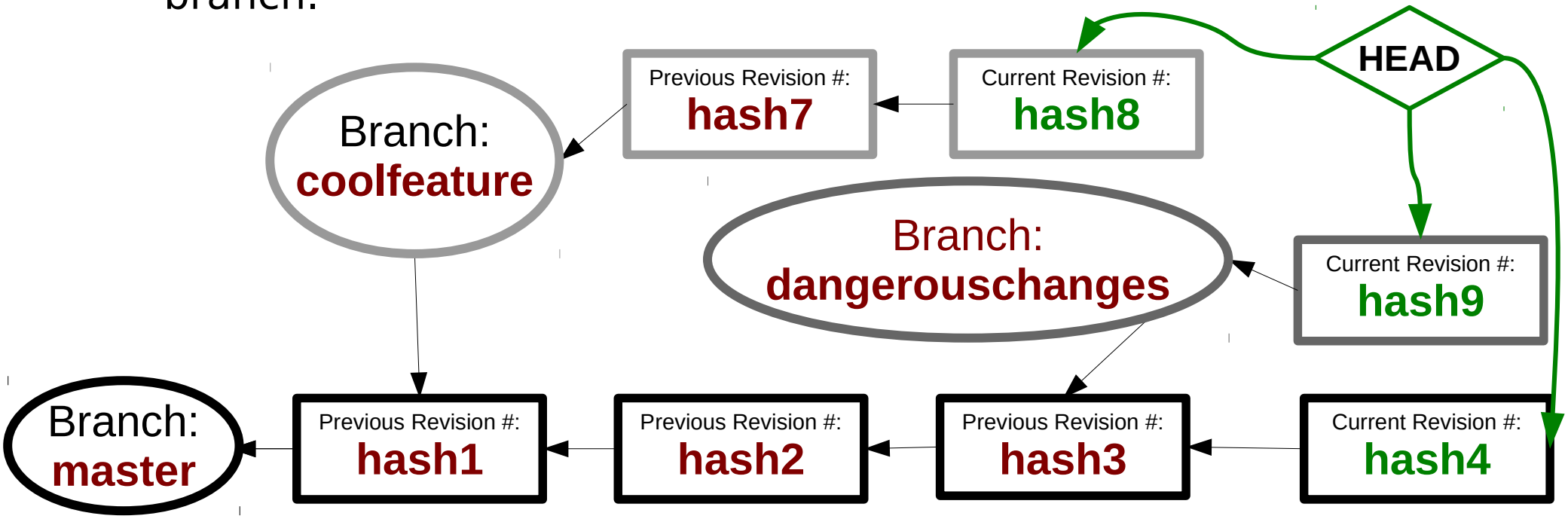
Concepts of version systems
git basic terminology

- The git command accepts **subcommands** to do operations on the database.
- A **brand new git repository** is created with the command `init`
- A brand new git repository always starts with a **branch** called **master.**
- **For every set of changes there is a `commit`.**
  Every commit generates a *new revision* with a different **hash**.
  This can be represented as an ordered graph like the one below.
  For every committed **change** a new hash.
- The latest commit hash is called **HEAD**.



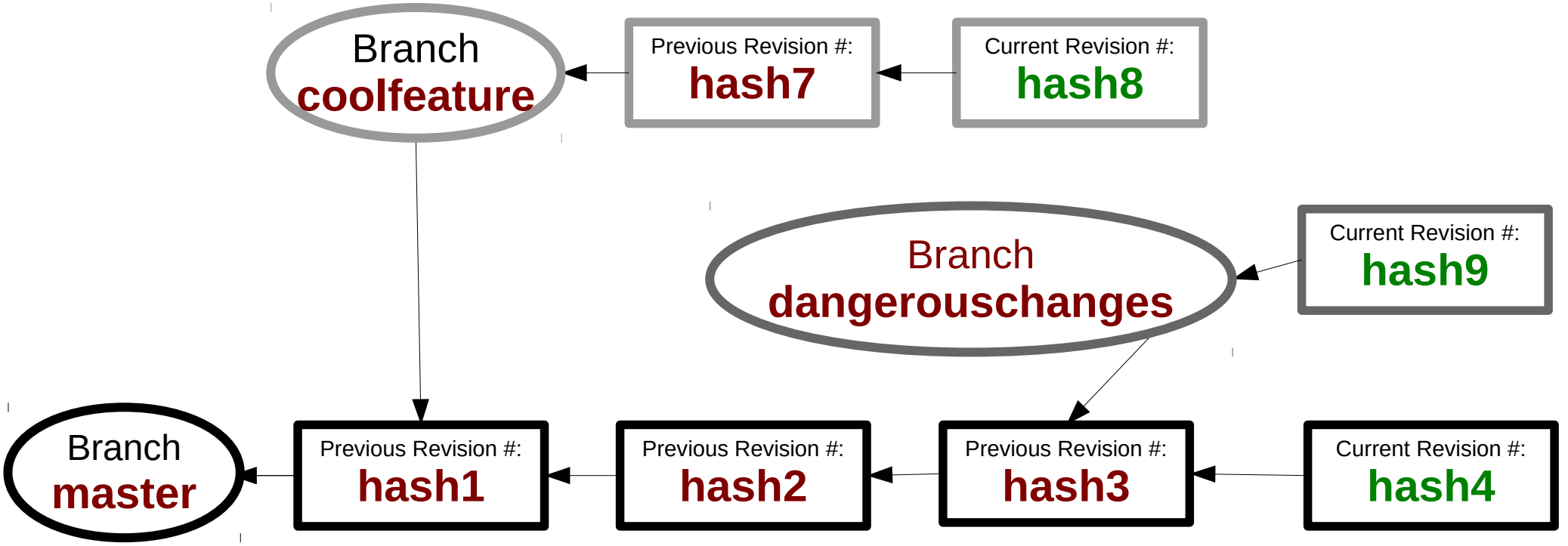
Concepts of version systems git branches

- A repository might have one or more **branch**es, that is, different version of the same repository which modify or propose different features.
  - They're called branches because they can be visualized like a *tree* as they diverge from some initial branch, usually called **master**. Every branch has a **name**.
  - The *latest commit* of each branch is called the **HEAD** of that branch.



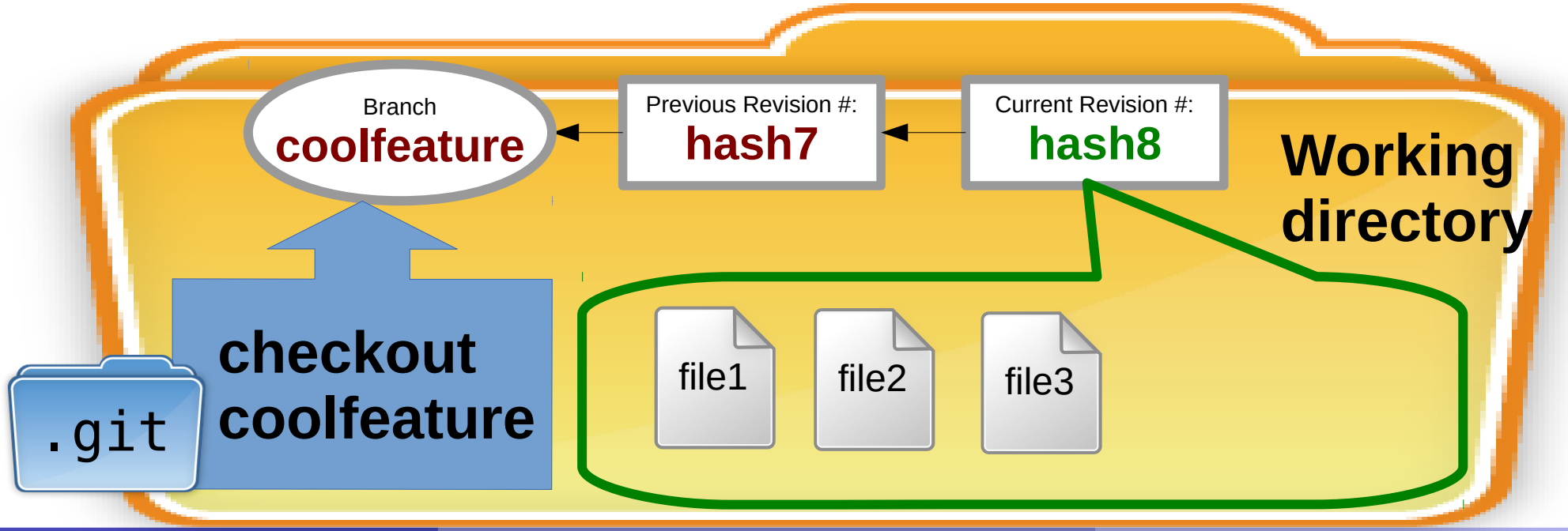
Concepts of version systems
git branch

- Every branch **history** is a continuation of the history where the master was branched.
- It is possible to branch from a branch, not just from the master. Use with care, can be confusing!



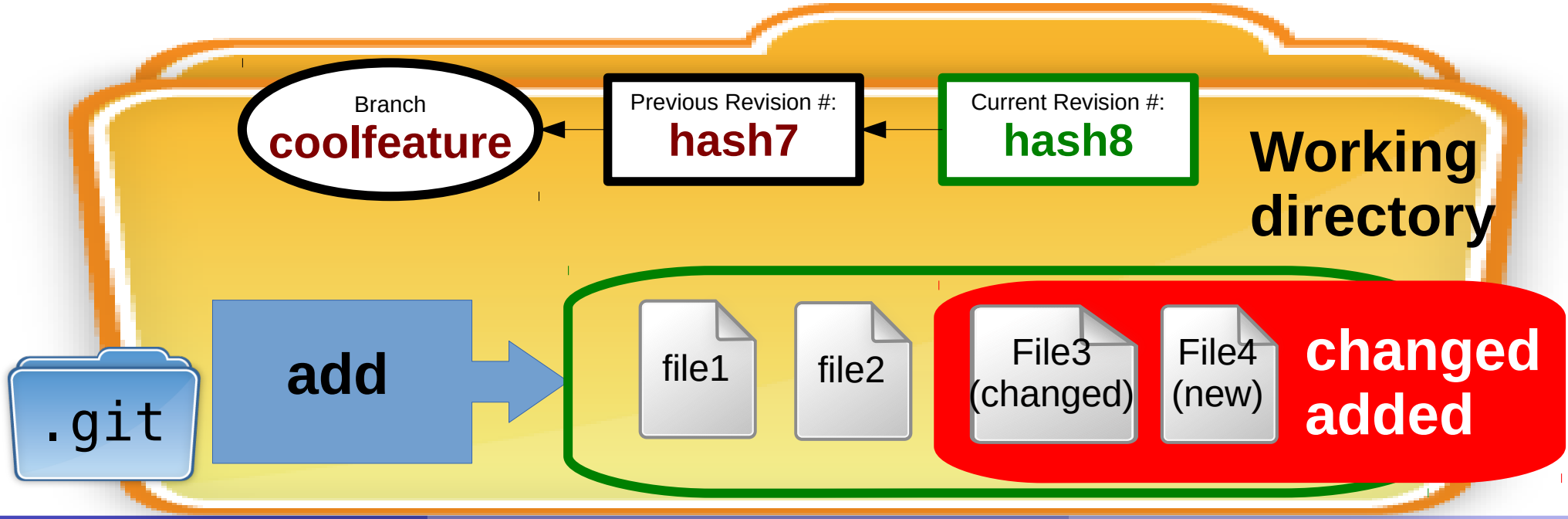
Concepts of version systems
git **checkout**

- A branch can be made *active* with the **`checkout`** operation. When a branch is checked out you will be able to **see its files in your working directory**.
  - To **`checkout`** a branch means to select a certain history of changes.



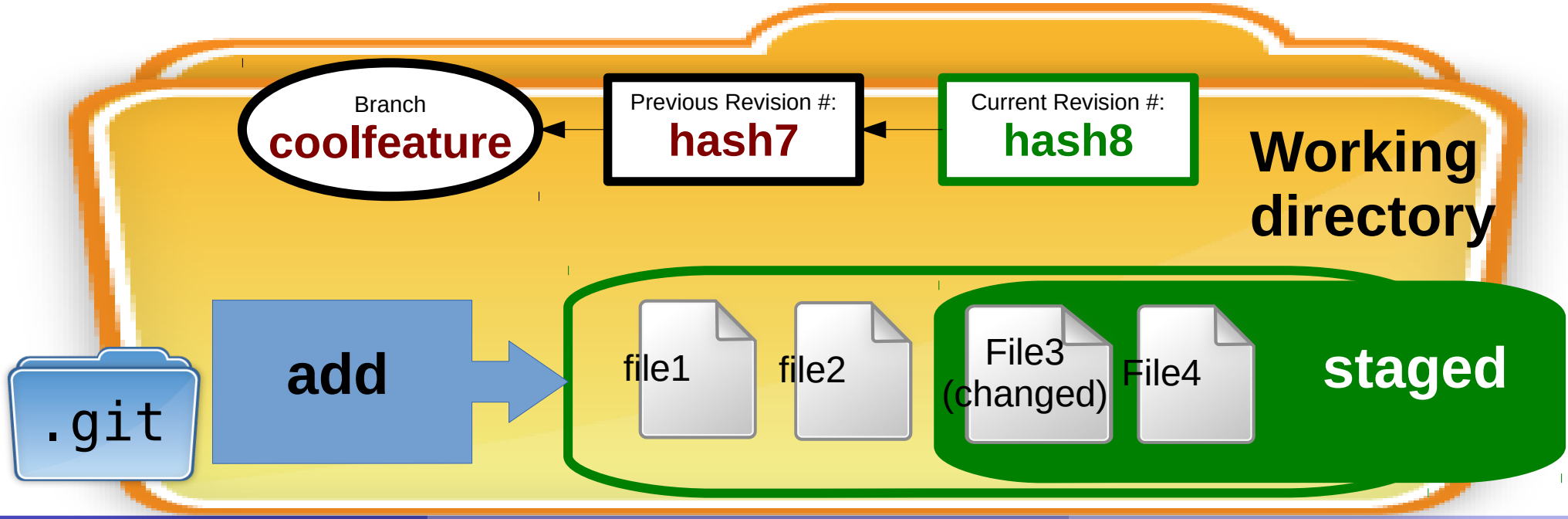
Concepts of version systems
git **add**

- If one modifies or changes files contained in a certain revision, git can see it, and reports to the user.
- Git gives the choice to **add** (include) these changes to the database.



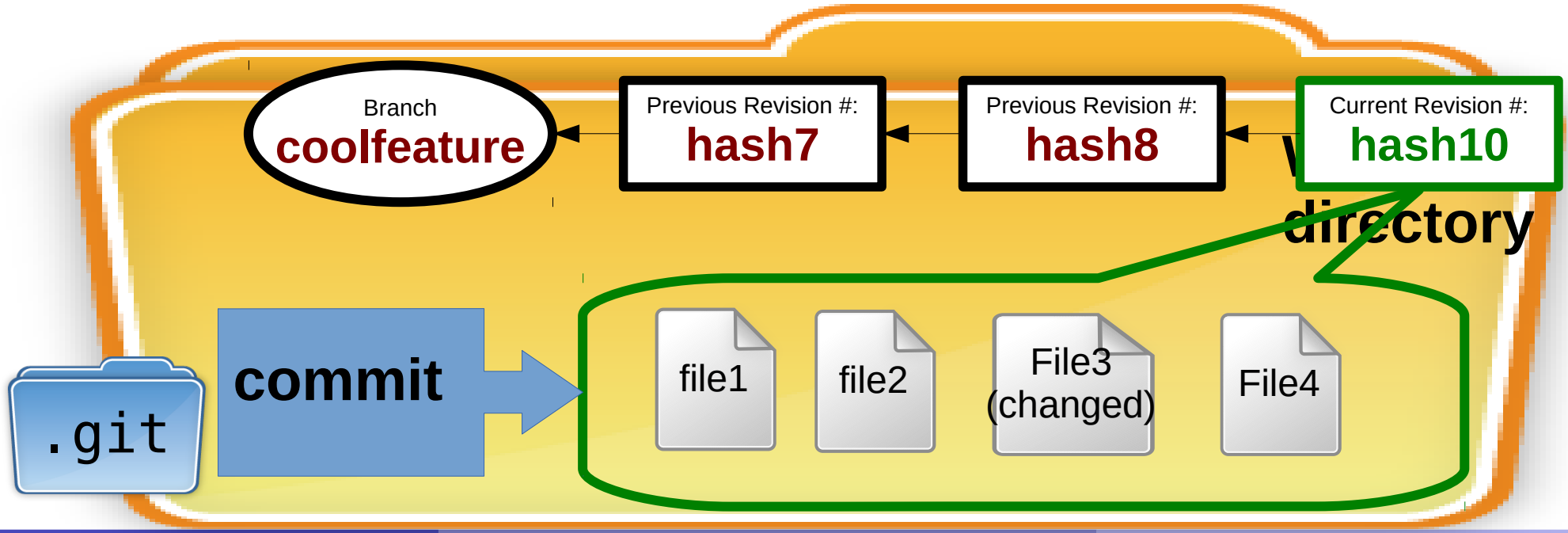
Concepts of version systems
git **add**

- Once files are **add**ed, they are *marked* to be part of a next revision, but they’re not yet saved in the database.
- In git slang, they’re **staged –** shortlisted to be part of the next commit.



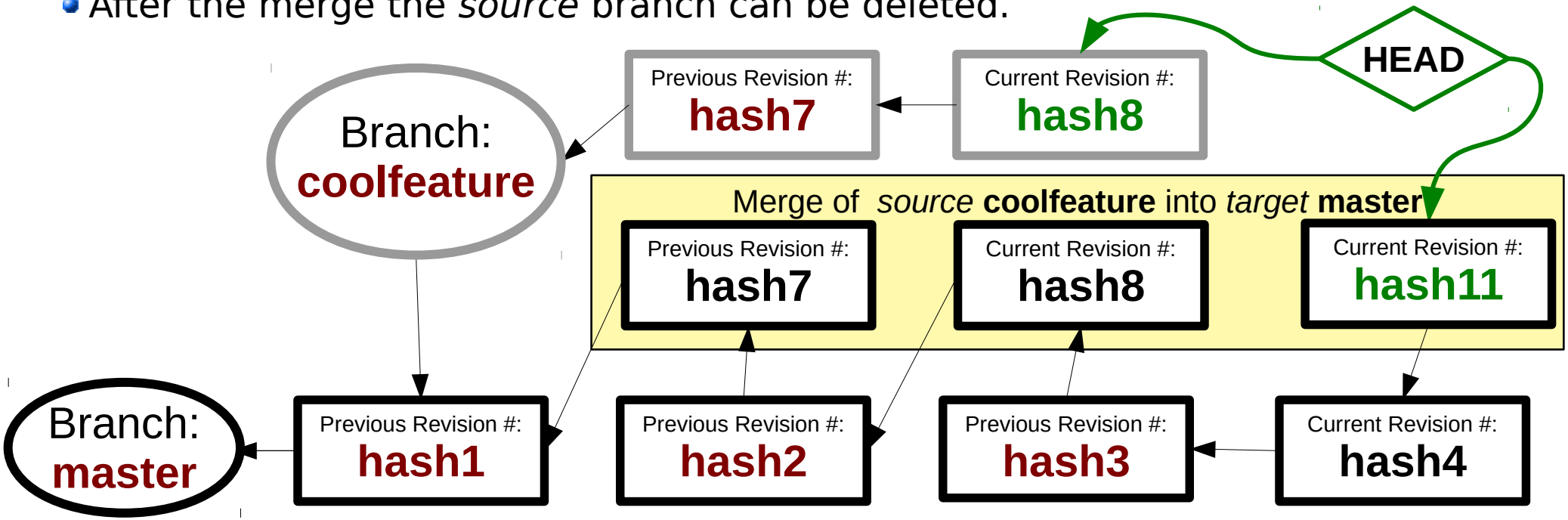
Concepts of version systems
git commit

- *Staged* files will then be actually become part of a new revision in the database once the user **commits** them.



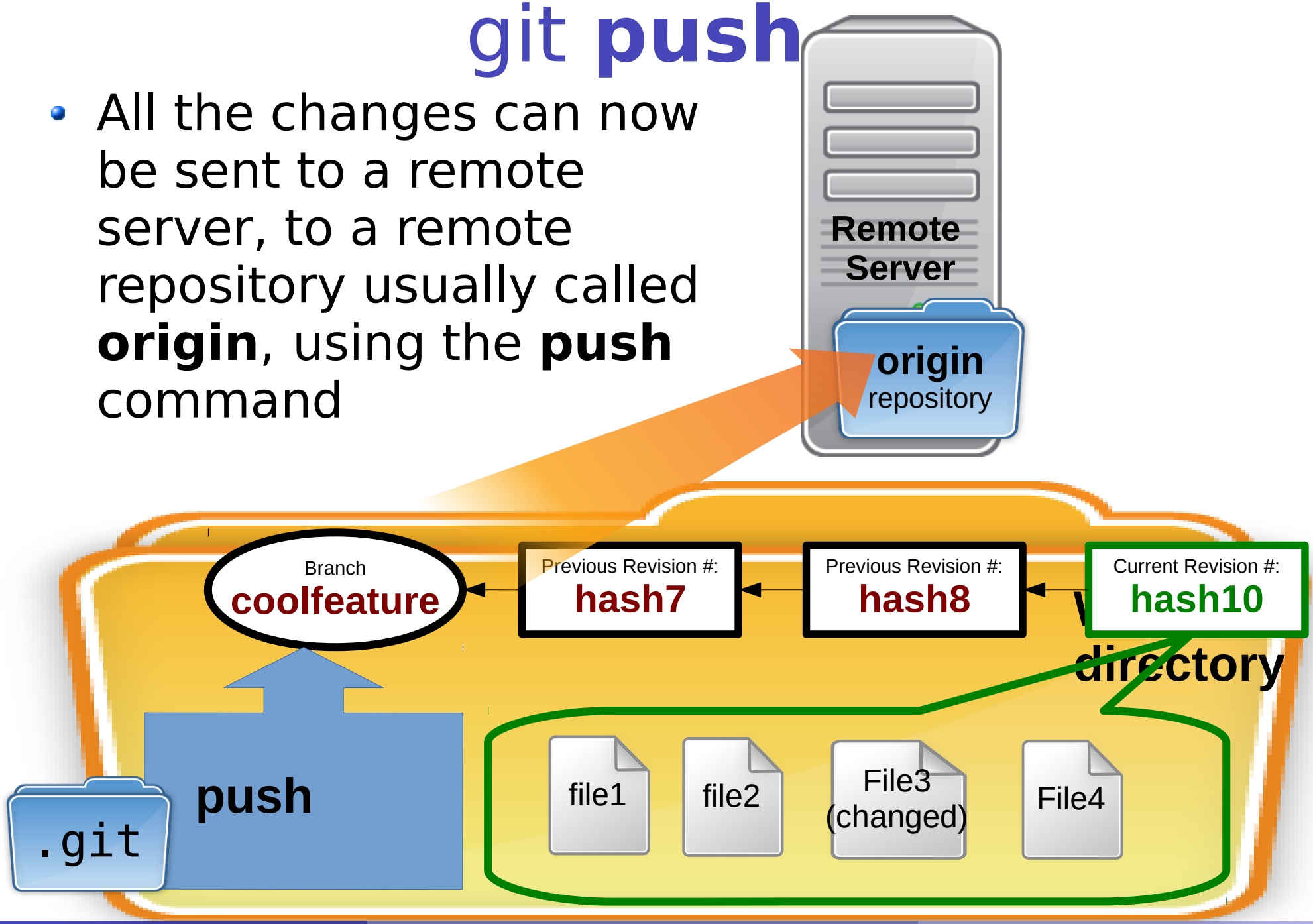
Merging

- Once the changes in a branch are accepted, these are usually integrated back in the master with the `merge` operation
- The result of a merge operation between a *source* and a *target* branch is a merged history of commits between the two branches. **The commits in the source branch are copied to the target branch.**
- A new HEAD is created with a commit that says that there was a merge in a given moment in time.
- The result of this two operations may look like the one in the example below
  1. Checkout *target*: `git checkout master`
  2. Merge with *source*: `git merge coolfeature`
- After the merge the *source* branch can be deleted.



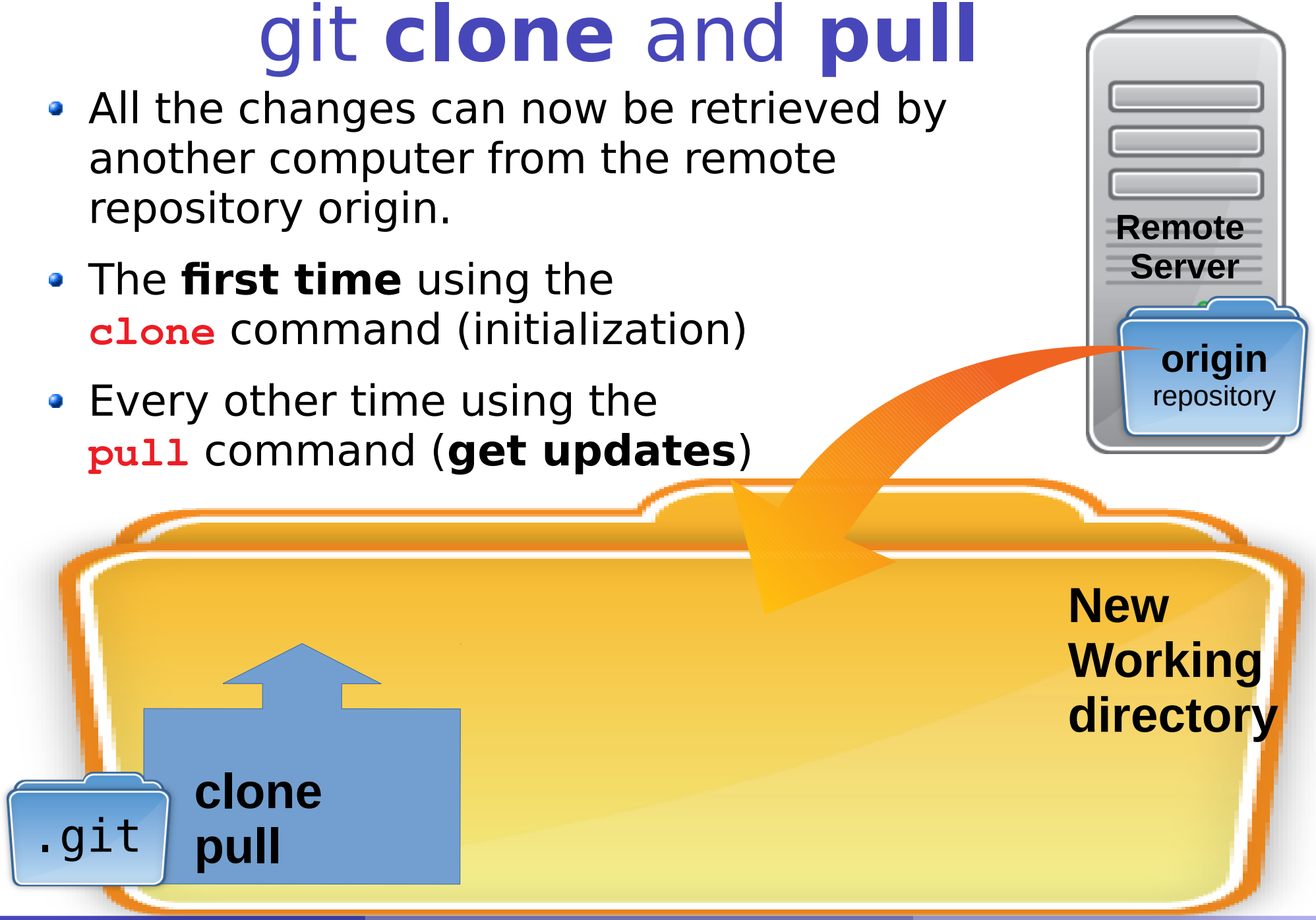
Concepts of version systems
git **push**

- All the changes can now be sent to a remote server, to a remote repository usually called **origin**, using the **push** command



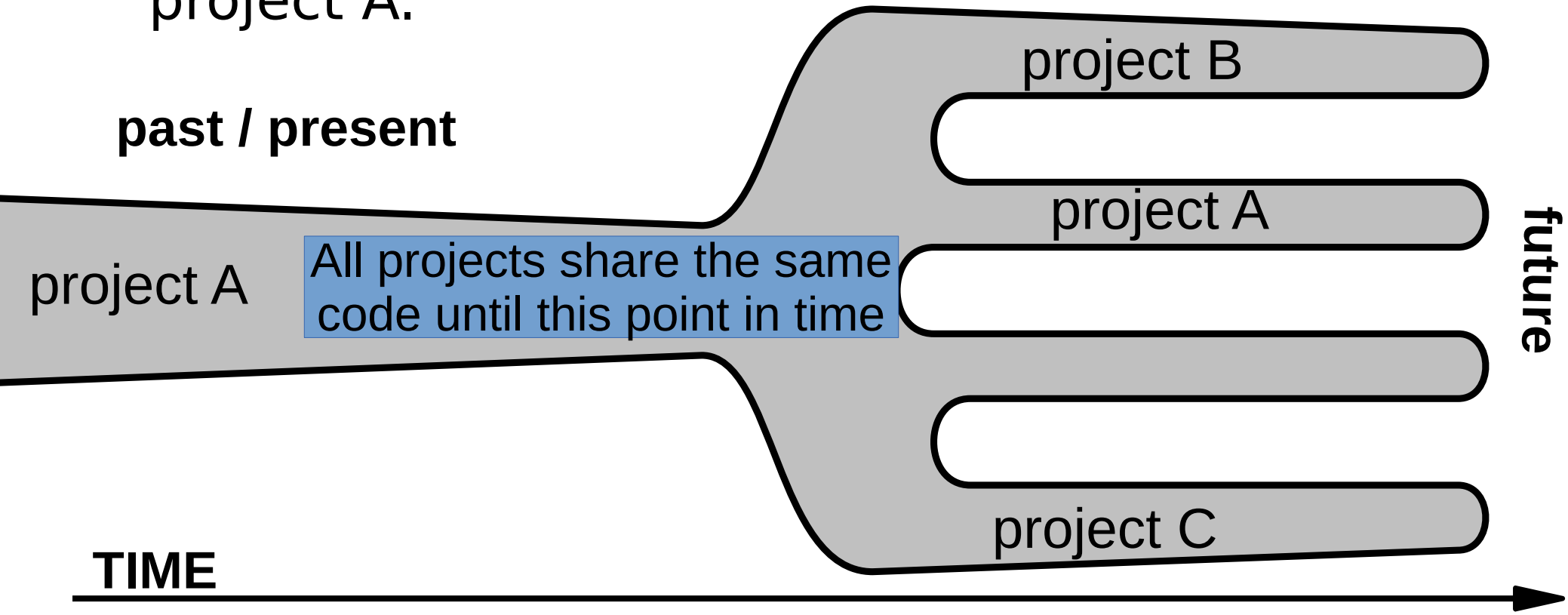
Concepts of version systems
git **clone** and **pull**

- All the changes can now be retrieved by another computer from the remote repository origin.
- The **first time** using the `clone` command (initialization)
- Every other time using the `pull` command (**get updates**)

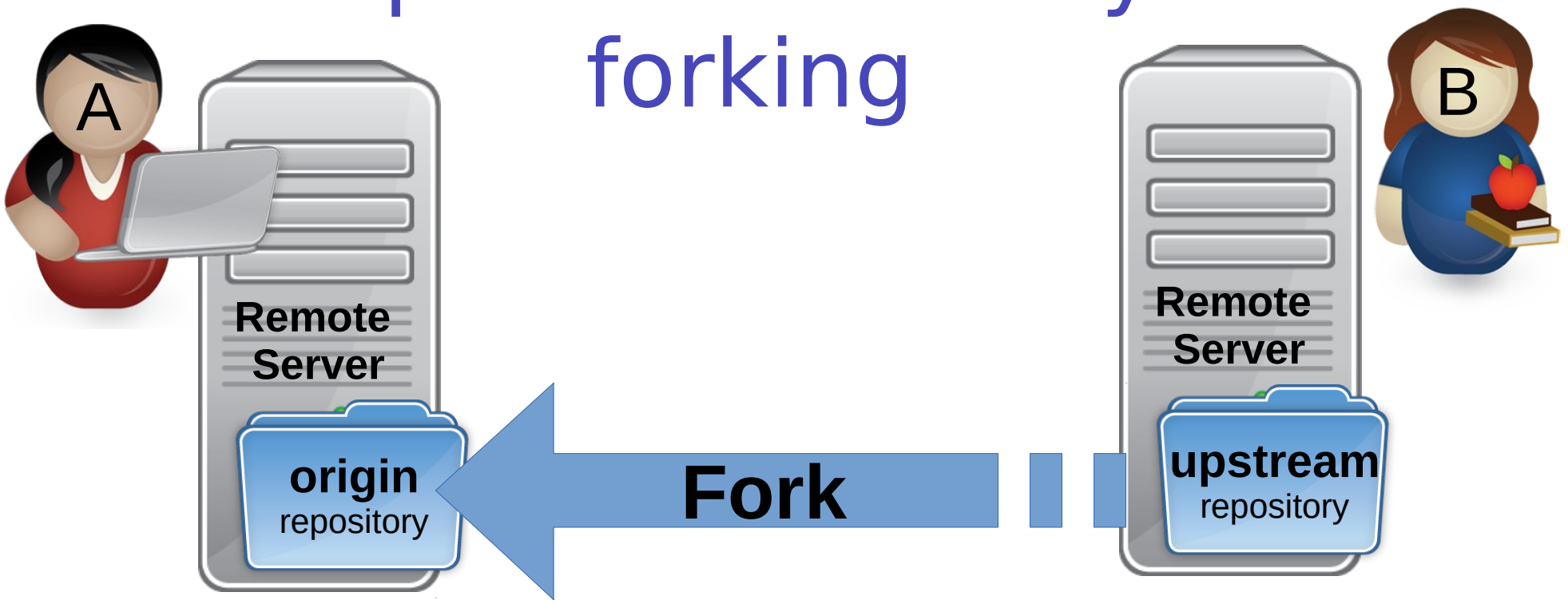


What is a software fork

- In software engineering, a **fork** of a software project A it's a copy of the software source code of A to develop features for a project B,C,... that follow completely independent choices from project A.



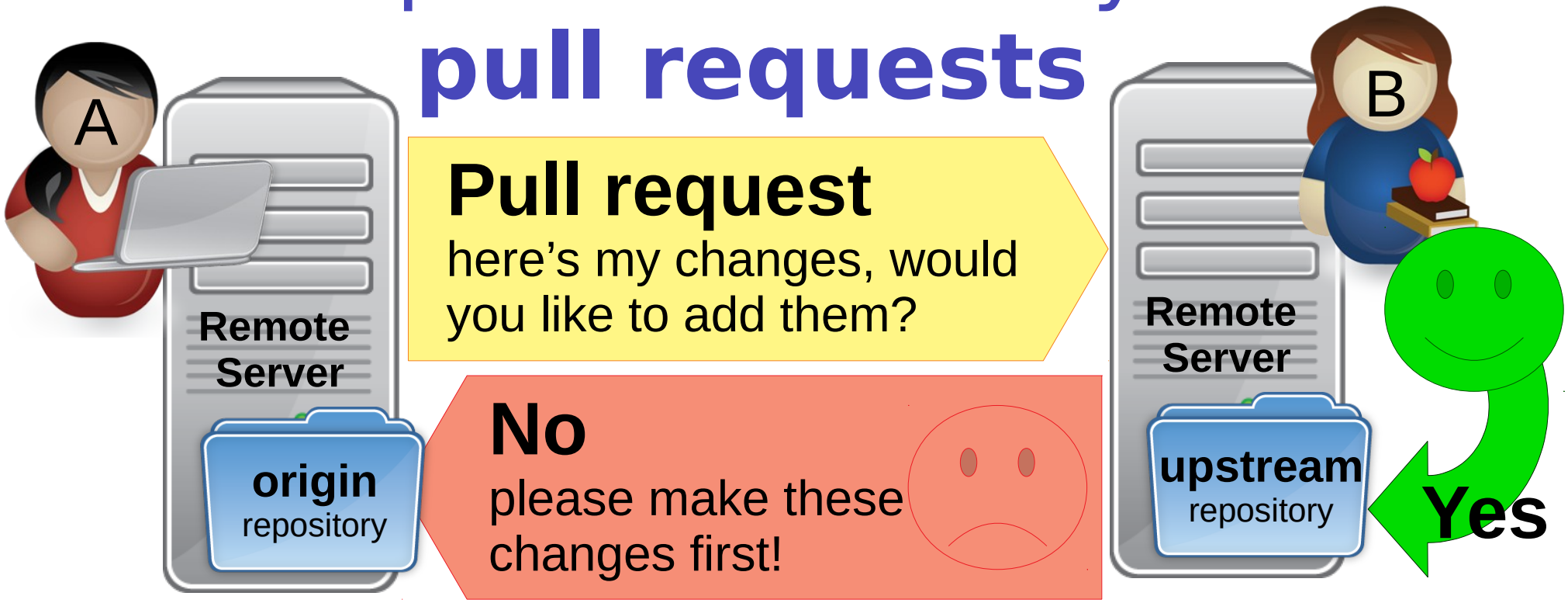
Concepts of version systems forking



- A fork happens usually between two users or organizations writing software, A and B
- A **fork**s B's repository in git is done by duplicating (cloning) the repository of a project you want to work on, called **upstream**
- User A works on their fork **independently**. At some point they might want to send the changes they made back to the B's upstream repository.

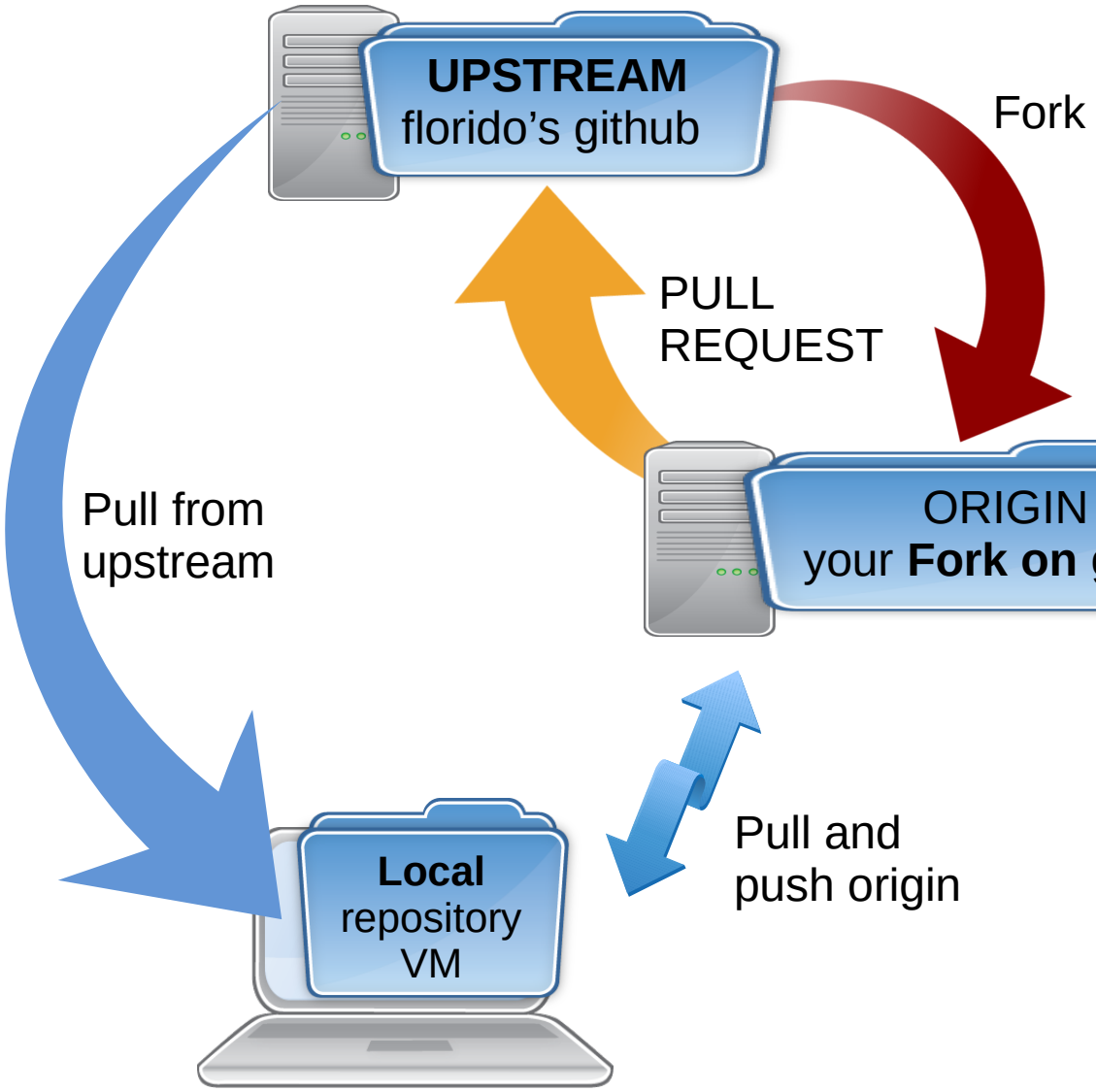
Concepts of version systems

pull requests



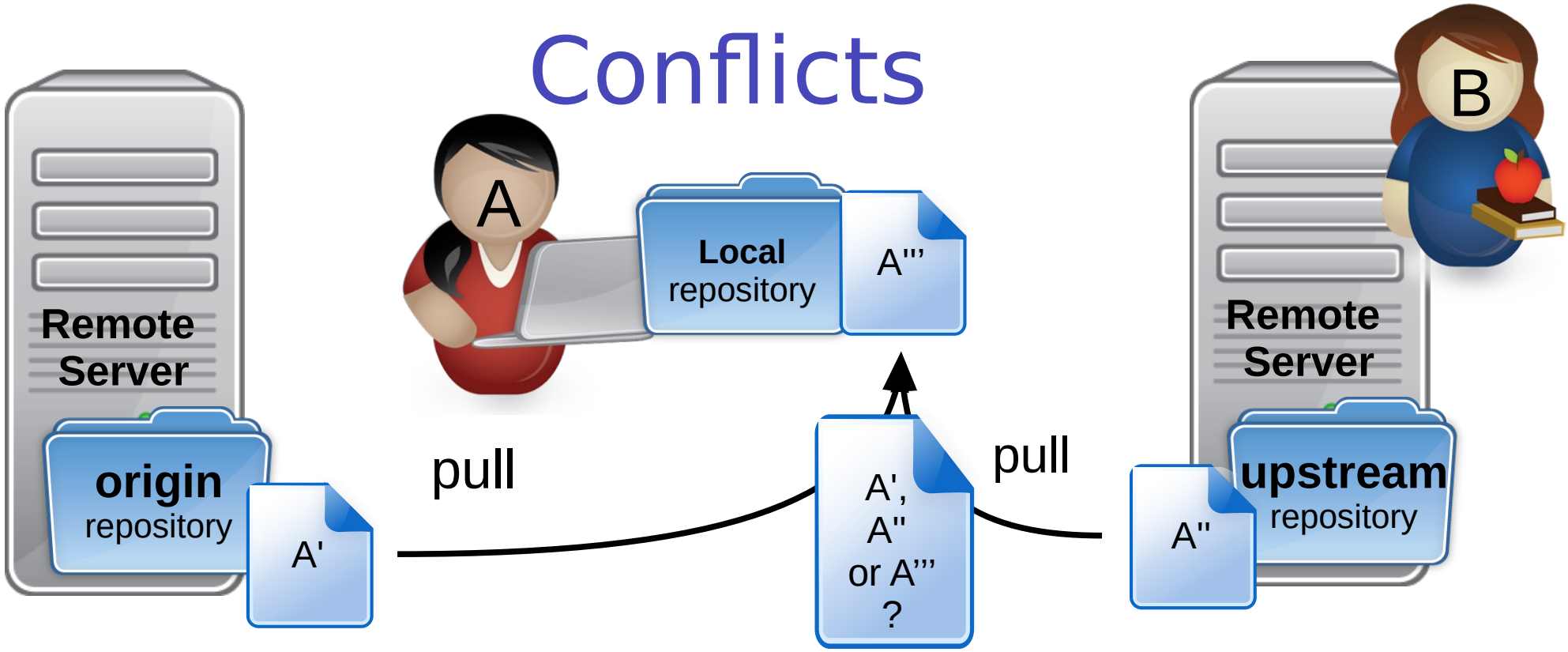
- Pull requests are a way to **propose changes** to the forked repository so that the owner of the upstream repository can **review** them and discuss them before approval
- If they are accepted, they will be integrated
- If they are rejected, a discussion can be made about why and how to make them acceptable.
- After this process the user A will need to **pull** the changes from upstream for origin to be in **sync** with upstream.

Upstream, origin, local
A Tale of a River



Kachemak Bay, AK. Photo credit: Alaska Shorezone.
https://medium.com/@AKSalmonProject/where-the-river-meets-the-tides-salmon-and-estuaries-a9e7aaf78519

Conflicts



- In a distributed development environment, different user may modify the same file
- When the same file is modified more or less around the same line, and you try to pull from repositories where the modifications have been made, you may incur in a **conflict**
- A **conflict** is a set of changes that must be reviewed in order to sort out which of A', A'' or A''' should go into the final result
- This usually can only be solved by a developer knowledgeable of the code, and it resolves in a **n-way-merge**. An example of 3-way merge is at slide 50.
- The result is often an A'''' file that integrates all the changes all the developers made.

GitHub

Octocat

- A cloud service that offers for *free*:
  - hosting for git projects (they run the git server)
  - A web interface to collaborate on projects
- Acquired in 2018 by Microsoft, now offers also
  - Private projects (can’t be seen by other users)
  - Enterprise services
- They claim they will not use your code except for the purposes of their service and that you retain all the copyrights on the code.
- It is **not** an open source project.
  https://www.github.com/
- Open source alternatives: Gitlab
  https://about.gitlab.com/

Homework Tutorial 5

1)Create a github account (you should already have it after the tutorial)

2) Fork my repository on github:
`https://github.com/floridop/MNXB01-2019`

3) Clone the repository you forked on your local machine or virtual machine, and enter the cloned folder.

4)Using the **git remote** command, add:

- your fork repository `https://github.com/yourgithubusername/MNXB01-2019.git` as the remote *origin* (should be already there!). Change *yourgithubusername* to the username you created during the tutorial.
- My remote repository at `https://github.com/floridop/MNXB01-2019.git` as the *upstream* remote repository

5)Create a **new branch** named **hw3hw5** and *checkout* the branch

6)At the root of the repository, create a folder with your name and the first three letters of your last name. For example my name is Florido Paganelli, I created:
floridopag

7) In the above folder, create a folder called tutorial3 and copy all the the files contained in my tutorial3 folder.

```
cp -r ../../floridopag/tutorial3 .
```
Don't miss the dot at the end of the line! "It means copy in the directory where I am"

8) **git-add** the newly copied files and **commit. Remember to write an explanatory comment in the commit. Stupid comments will be rejected.**

9) **push** to your remote fork *origin* the hw3hw5 branch

10) Submit me a **pull request** for that branch on github.

11) Copy the link of your github fork **and** a link to your pull request on Canvas.

1)An example of github fork link is as follows:
https://github.com/floridop/git-it-electron

2)An example of github merge request link is as follows:
https://github.com/jlord/git-it-electron/pull/204

Git interactive tutorial

Preparing for the tutorial

- Create a folder in your home folder
  - **mkdir** ~/gittutorial/
  - **cd** ~/gittutorial
- Download the tutorial app:
  - For the LubuntuVM (32 bit):
    wget http://www.hep.lu.se/staff/paganelli/fileshare/gitmnxb01-ia32.tgz
  - If you're using your own laptop (64 bit):
    wget http://www.hep.lu.se/staff/paganelli/fileshare/gitmnxb01-x64.tgz
- Unpack the tutorial app:
  - 32bit: **tar** zxvf gitmnxb01-ia32.tgz
  - 64bit: **tar** zxvf gitmnxb01-x64.tgz
- Enter the created directory:
  - 32bit: **cd** Git-it-linux-ia32
  - 64bit: **cd** Git-it-linux-x64
- Start the tutorial app:
  - ./Git-it &

Reminder: the ~ symbol means "my home folder", that is /home/courseuser/
the listed commands will create (make directory) and go inside (change directory) /home/courseuser/gittutorial/

Note: only the **English** version of the tutorial is customized for this course. I did not modify the versions for other languages, so if you change language things might be a bit different – but you will still learn!

Have fun with the Git-it tutorial!

- Created by jlord, see https://github.com/jlord/git-it-electron
- Contributed by various authors
- Written in JavaScript and HTML using a framework called node.js
- Once done with the interactive tutorial, read the slides for some other useful commands and tools.

Best practices in the lab

- Since the VM is shared, I suggest that at the beginning of each lecture, after turning on the VM, you open a terminal and:
  - Always check the global variables and make sure they refer to your user:
    ```
    git config --global --list
    ```
  - Always redefine the git --global user.name and user.username as in the tutorial with your own name:
    ```
    git config --global user.name yourusername
    git config --global user.usernamename yourusername
    ```
  - Make sure there is a folder `~/git/yourGITusername` with your username. For example, I'd do:
    ```
    mkdir -p ~/git/floridopag (will give you an error if the folder already exists)
    cd ~/git/floridopag
    ```
  - Get into your local copy of your fork
    ```
    cd MNXB01-2019
    ```
    - If the above folder does not exist, clone your git repository (use your username not mine!):
      ```
      cd ~/git/floridopag
      git clone https://github.com/floridopag/MNXB01-2019.git
      cd MNXB01-2019
      ```
  - **Print this slide as a reminder of what to do!**

A word on privacy and security

- When you fork my MNXB01-2019 repository and submit pull requests, everything will be public.
  Others will see your code.
  - It is perfectly ok for me because I believe one learns coding by looking at other people's code and sharing/discussing coding with others.
  - If you're not happy with it, you can create your own **private repository** to store the material produced during the MNXB01 course, so that **nobody else can see it**.
- However, for the Tutorial 3 (bash) I will require that your ultimate submission happens as a pull request to my repository. **From that moment on your bash code for the homework will be public.**
- The grading will be done on canvas, not on github
- If you prefer not to write your name on the github repository, you can write your nickname, but **make sure I know who you are**. I will not correct submissions if I don't have a mapping nickname→student. You can send me this information privately if you don't want others to know who you are.

Useful git commands

Setting your default editor with git

- If you commit without the -m option, git will automatically **open a text editor** for you to write a commit comment.
- It is good practice to:
  - write a commit title
  - leave a blank line
  - describe your commit in more detail.
- We will use geany as the default editor, but you can use any editor you like.
- If you don't configure anything, the default is a text editor called **nano**, which for some is a bit weird at first. But I suggest to use it so you just use the command line. Press “CTRL + O” to save the file, ”CTRL + X” to exit.

Setting geany as the default git editor

- Run:

```
git config core.editor geany
```

- Note that the commit will only happen ONCE when you save the file in geany.
- Test by running

```
git commit
```

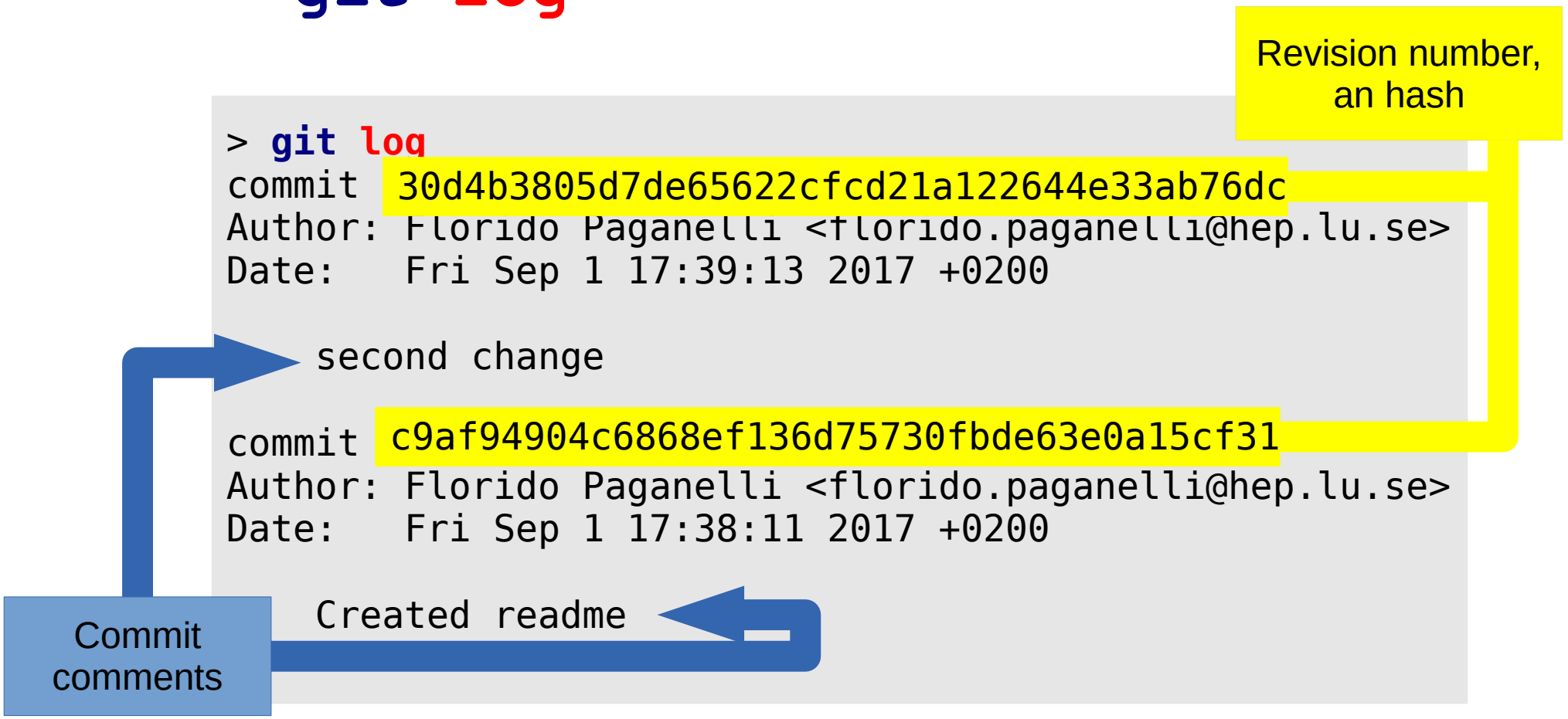
- **If you don't like it, revert to default by writing**

```
git config --unset core.editor
```

Git log, commit history, revision numbers

- All the commit history with you messages can be browsed using the command

  `git log`



Git log, commit history, revision numbers

- To see which files have changed for each commit:

```
git log --name-status
```

Removing or renaming a file from the git database

- **Removing**: Sometimes one can decide that files in the directory should not be part of the repository anymore. Rather than deleting them with the rm command, one can use

```
git rm filename
```

  - Remove a file using the above command.
  - Check the output of `git status` .
  - `git commit -m 'I have deleted file filename'`
    Remember: CLEARLY STATE that you removed some files in the commit message!

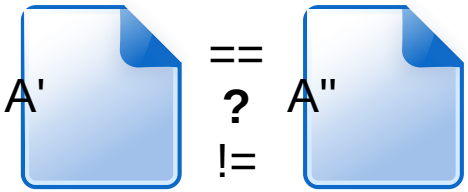
- **Renaming**: `git mv oldfilename newfilename` is equivalent to

```
git rm oldfilename
```

followed by

```
git add newfilename
```

Graphical Diffing



- Run
  git diff

```
> git diff
Index: thisisfloridofile.txt
===================================================================
--- thisisfloridofile.txt (revision 6)
+++ thisisfloridofile.txt (working copy)
@@ -1 +1,2 @@
 Hello! this is florido's file.
+I am adding this change.
```

Line numbers of the two files:
-1 : showing line 1 of of file ---
+1,2 : showing lines 1 to 2 of file +++

- If you want a graphical tool to check the diffs, I suggest **meld** (should be already installed, but if not: `sudo apt-get install meld`
- Use meld as the default diff tool:
  `git config diff.tool meld`
  `git difftool thisisfloridofile.txt`

Undoing not committed changes

- Say that we are not happy with the changes we just made **to a single file** and we want to go back to the latest commit (also called HEAD)
- Change one of the files in your repository and issue `git status`.
- The best to do is a **simple checkout** of the file from the last commit
```
git checkout thisisfloridofile.txt
git diff
```
- **Careful! You will lose all the changes done and not committed!!!**
- Note that this is equivalent to checkout the file at the **latest revision HEAD**:
```
git checkout HEAD thisisfloridofile.txt
```
- Checking out HEAD of all files in a directory will cancel all the changes done to the uncommitted files in that directory.
```
git checkout HEAD *
```
- **Play a bit with these commands by changing files and see what happens.**

Reverting to a previous revision

- Say that we don't like the current revision state, and we want to roll back the code to a state of a different revision back in time.
- The main suggestion is:
  try to never go back in the revision history.
  This is actually nice because in a collaborative environment, keeps track of who-did-what with no cheating allowed :)
  Unfortunately git allows for “cheating” by changing the revision history. It can be useful sometimes, but must be used with extreme care. **Changing the revision history gives no UNDO.**
- To experience with this, change some files and commit.

Reverting to a previous revision the safe way: revert

- The **revert** command restores the state of all files at a certain revision to the current working dir.
- Usually the output of a revert gives hints about the steps to take before committing.
- Make sure you have at least three commits (check git log)
- Create a fourth commit

Reverting to a previous revision the safe way: revert

- Try to git revert everything to your second commit in the log:
  `git revert commithash`
- Example:
  `git revert c9af94904c6868ef136d75730fbde63e0a15cf31`
- Read the git status output to see what changed
- Take action to make the files ready for commit, and commit
  - Git will automatically start a commit and open the text editor for you. It will add the “Revert commithash” comment to your commit and wait for your input.

Reverting to a previous revision the unsafe way: reset

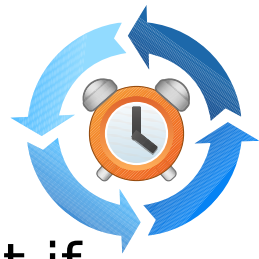
- The reset command does something different. It does not preserve history and allows you to modify an existing commit. For a detailed explanation see https://www.atlassian.com/git/tutorials/undoing-changes
- Use it **only** on a **private branch** and **never** on a branch you share with others (typically a master branch)
- Additionally, I suggest to use it **only** when one of these two happen:
  - You already staged some changes to a file and you want to unstage them

    ```
    git reset filetounstage
    ```

  - You are totally unhappy with whatever you did so far and want to **unstage all staged files**:

    ```
    git reset
    ```

Fixing commit mistakes



- Commit allows you to amend or change the latest commit if, for example, you forgot a file or you wrote the wrong comment:

```
git commit --amend
```

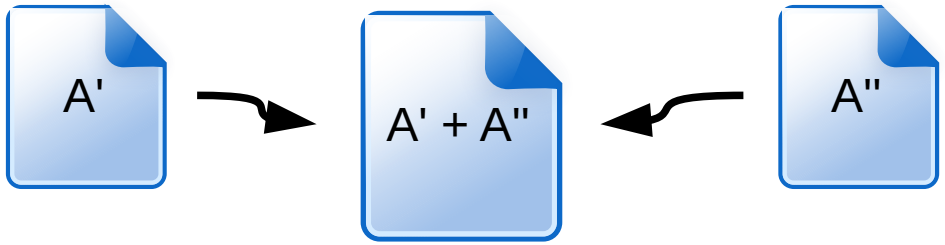
- Note that this will create a new revision hash, and will **DELETE** the previous commit hash. So be sure you are done with `amend` before you push to your remote repository.
- **NEVER DO THIS AFTER YOU PULLED YOUR LOCAL BRANCH TO A REMOTE REPOSITORY UNLESS YOU'RE THE ONLY USER OF THE REMOTE REPOSITORY.**
- Seehttps://git-scm.com/book/id/v2/Git-Basics-Undoing-Things

Graphical Clients

- Want to try a graphical client?
  - Minimalistic one: in the folder where a git repository exists, run
    `gitg &`
    - Check out how it shows branches!
  - Feature-rich one (not available in repositories): `https://www.gitkraken.com/`
    - This one is NOT available on Lubuntu repositories. You need to download it from the internet if you want the latest version.

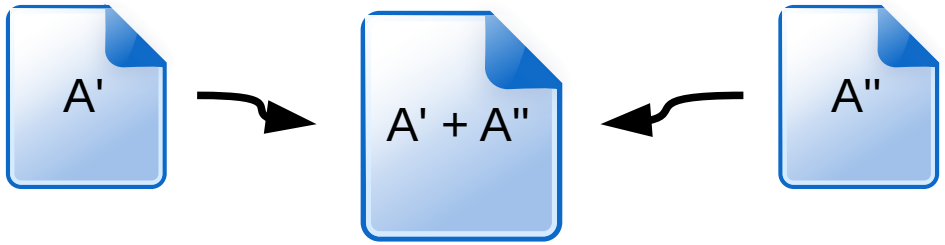
Additional material

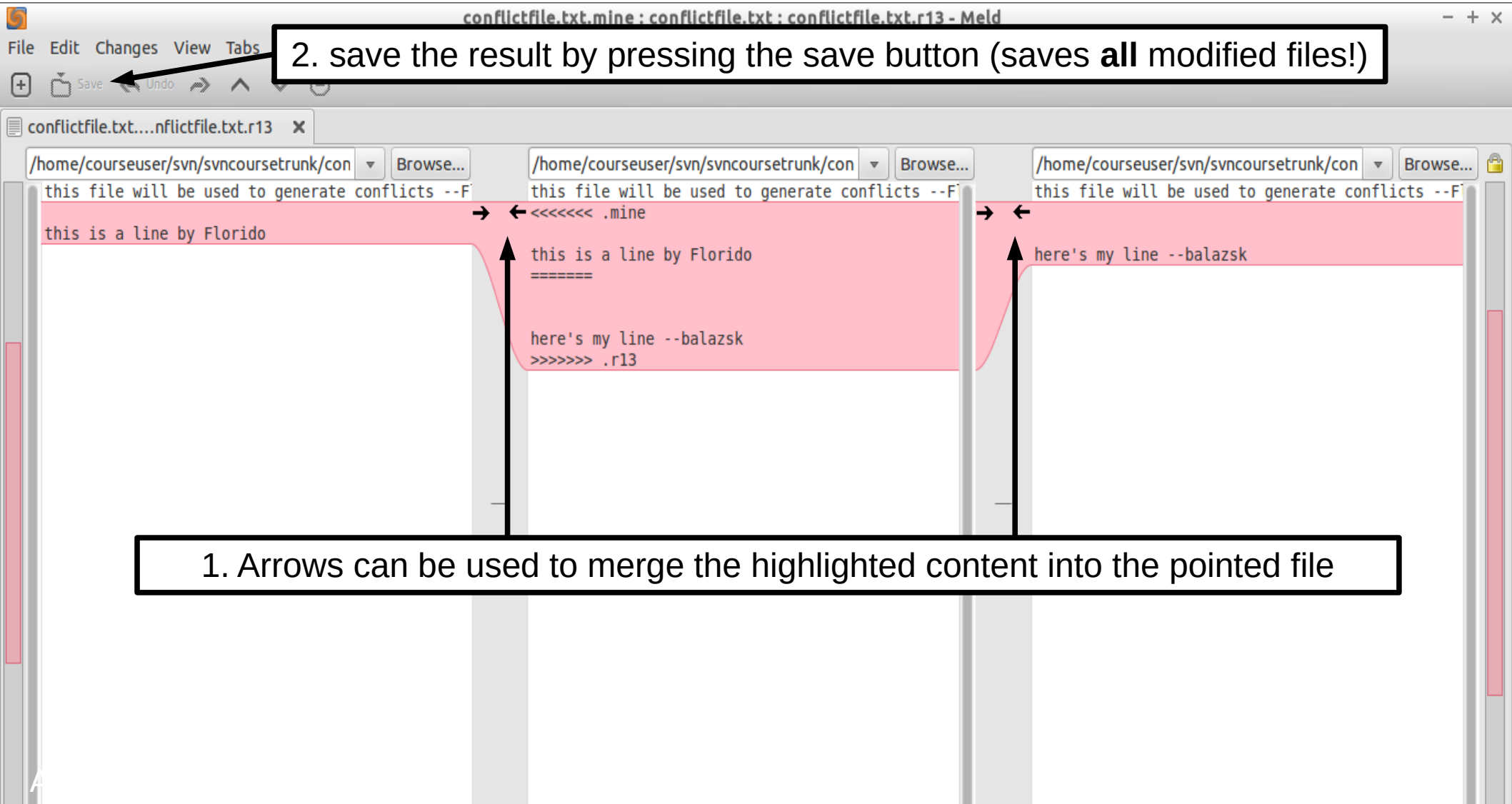
Merging



- Suppose we have two versions of a document with different contents
- We want to make one out of two
- This is often referred as **three-way-merge**
- We need to choose which part of each document we want to keep
- There exist tools to do it, for example the excellent **`meld`**
- git can attempt to do merges for us:
  - If the merges are simple, i.e. the changed content of A' can be easily mixed with that of the content of A''. For example, the documents differ a little but the changes in each document are not overlapping.
  - If we provide it with some hint on how to do the merges
  - If the above fail, it will ask us to do the merge manually, for example using `meld`
- The most frequent case of merge is in case of **conflicts**, we will not see them in this course.

Merging with meld





References

- git cheat sheets:
  https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf
  https://jan-krueger.net/wordpress/wp-content/uploads/2007/09/git-cheat-sheet.pdf
  https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet
- Quick guide to githttp://rogerdudler.github.io/git-guide/
- Jlord's git-it:
  https://github.com/jlord/git-it-electron
- Merging with meld
  https://www.youtube.com/watch?v=3Qynj8WUwgs
- Reverting
  https://www.atlassian.com/git/tutorials/undoing-changes
  https://git-scm.com/book/id/v2/Git-Basics-Undoing-Things

Pictures references

- https://openclipart.org/
- http://www.libreoffice.org/

Creating and editing a file

- During the tutorial you'll be asked many times to do things with files. For those of you not familiar with **file editing**, here's a small how-to.
- There are many ways of creating a file.
  One way is by using a **text editor**
- The favorite text editor for this course is called **geany**. Can you find the icon in the menu? Open it by clicking on the icon.

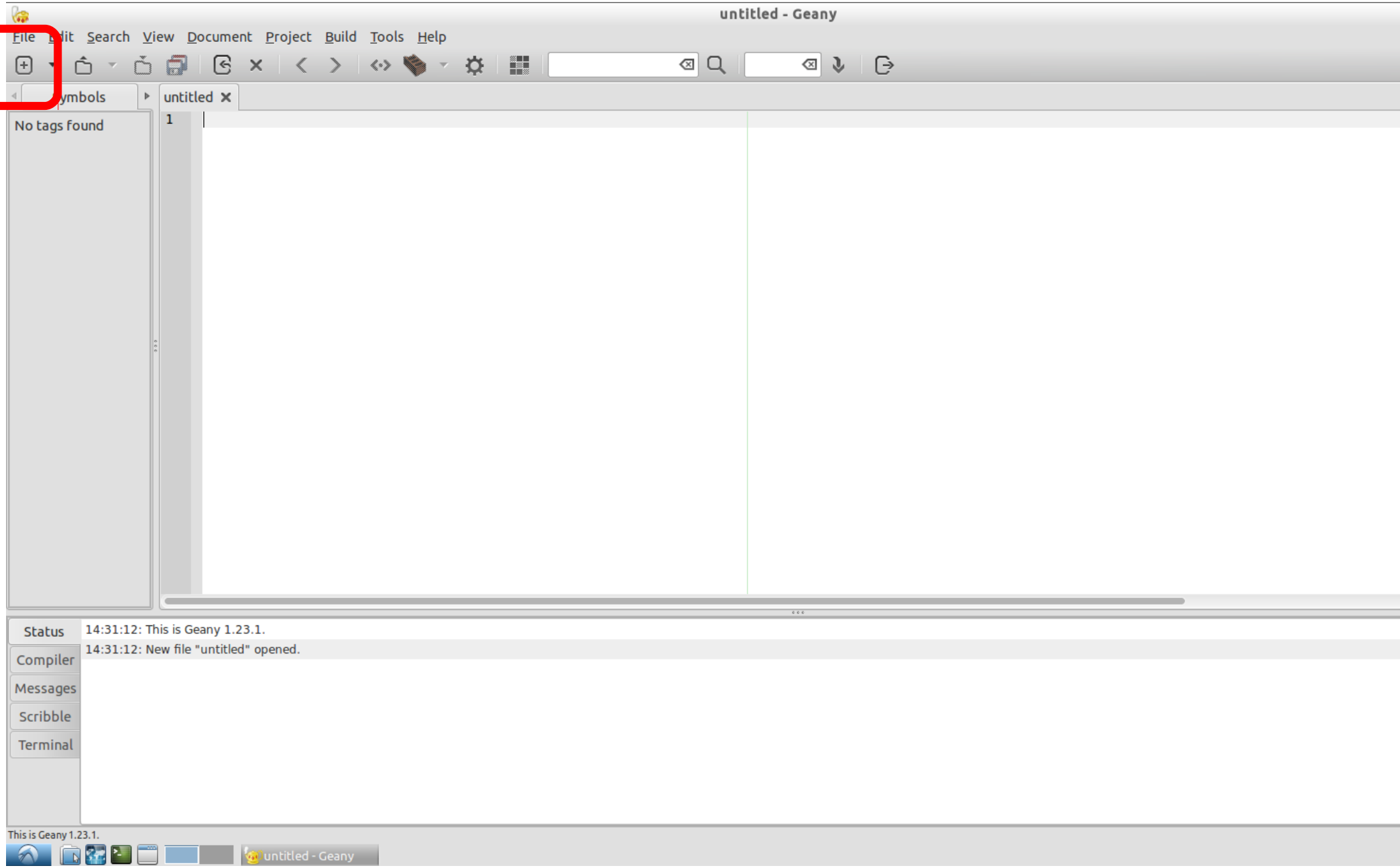
- Alternatively, open a terminal 

and write the command:

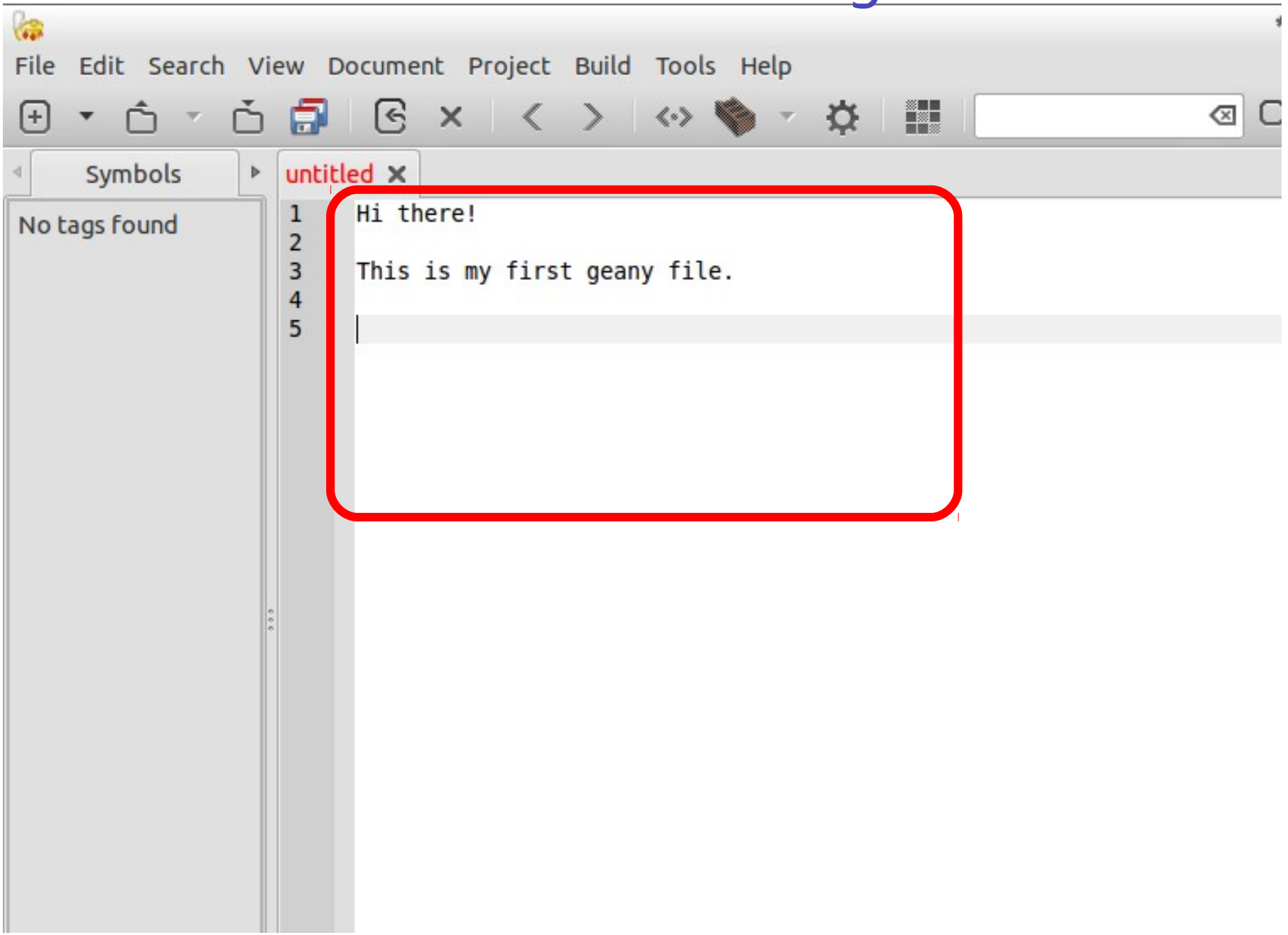
```
geany &
```

(the & symbol sends the command execution in background, see tutorial 1b!)

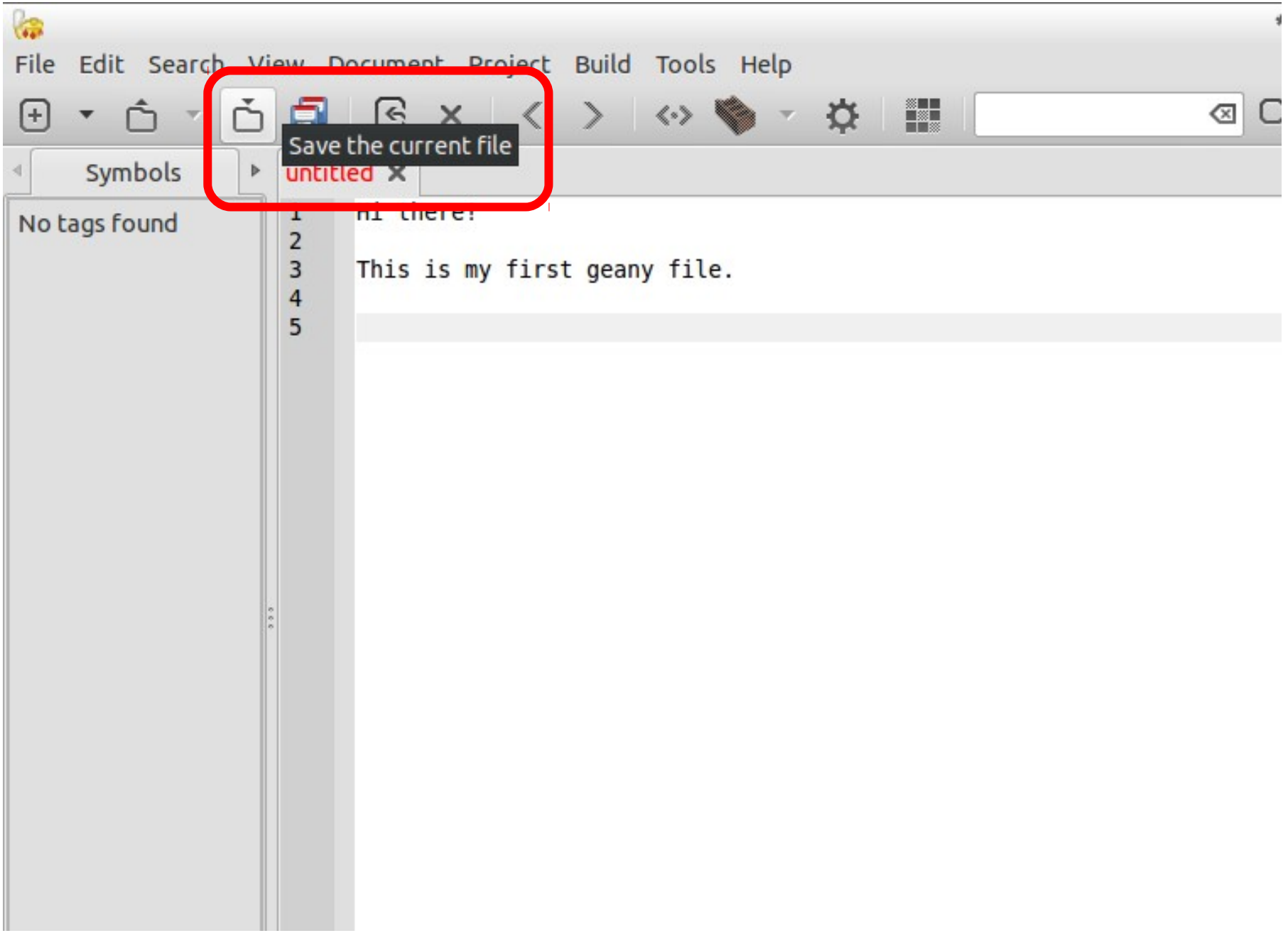
Editing and saving a file: create new



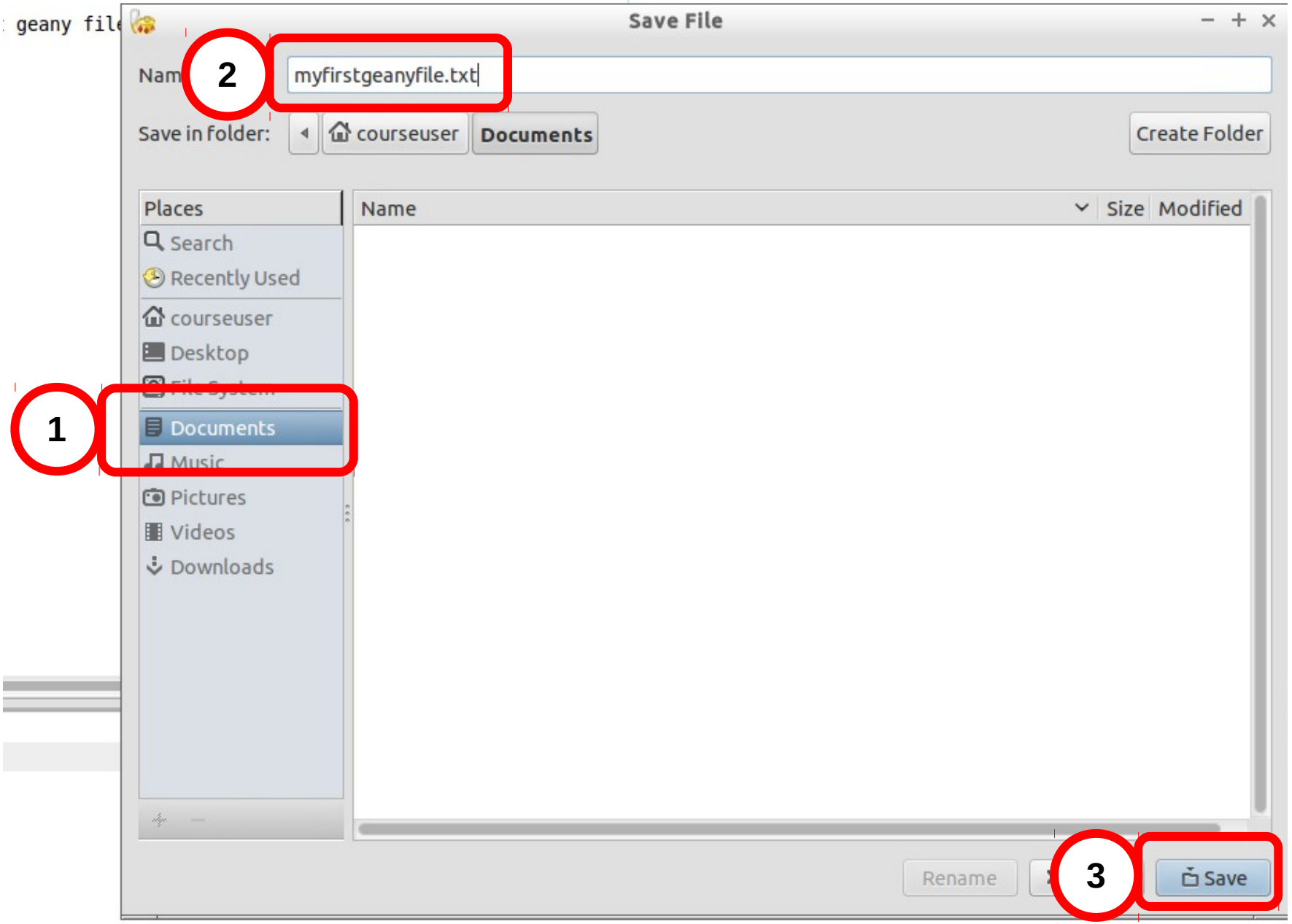
Editing and saving a file: write something



Editing and saving a file: save or save as



Editing and saving a file: choose location and filename



Editing and saving a file