SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 4

Fabian Wermelinger
Harvard University
CS107 / AC207

Tuesday, September 13th 2022

LAST TIME

- Configuration files read by the shell at startup.
- Login shells and non-login shells.
- Redirection of file content and `stdout`/`stderr`.
- How to use environment variables and how to set them.
- Basics of writing shell scripts.

TODAY

Main topics: ***Process management, Version control systems (VCS), Centralized and distributed models, Intro to Git***

Details:

- Managing Jobs and processes in Linux, suspending and continuing execution.
- Introduction to version control systems
- Centralized and distributed approaches
- Essentials of Git
- Interactive `git rebase` demo

AGENDA CHECK:

- Start building your project groups (4-5 students). Milestone M1A is due next Thursday 09/22. Once you have formed your group and obtained your team ID, completing this milestone takes 3 minutes or less.
- We will also have the first quiz on 09/22 (material of lectures 1–5).

INTRODUCTION TO VERSION CONTROL SYSTEMS

What is version control?

Version control is a system (abbreviated VCS) that records changes to a file or set of files over time so that you can recall specific versions later.

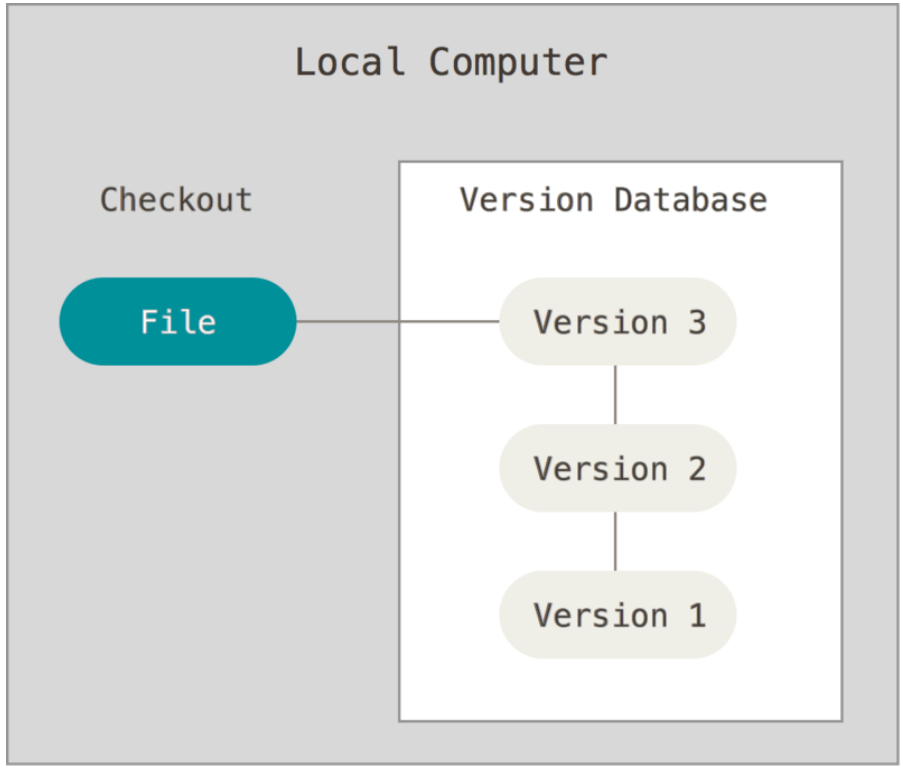
- VCS is absolutely *essential* if you plan on developing serious software *(academic and industry standard)*. If you were not serious about it you would not be in this class.
- Working on a software project without VCS ***simply does not scale***. *The project is doomed to fail.*

Disclaimer: Some content and figures in these slides are based on the free Pro Git book written by Scott Chacon and Ben Straub.

LOCAL VERSION CONTROL

If you would only need version control on your local machine (e.g. laptop), you would probably make backup copies of your files into some directory you dedicate for VCS.

Discuss with your neighbors:
If you were to implement such a tool with a shell script, which of the available basic Linux commands would you think are useful for this task?



LOCAL VERSION CONTROL

Discuss with your neighbors:
If you were to implement such a tool with a shell script, which of the available basic Linux commands would you think are useful for this task?

- A useful command for this task is `diff`. The strategy is to design a data structure that stores a sequence of *file differences* for a particular file and apply them using the `patch` command. Previous states of the file are then *reconstructed* applying patches in reverse.
- You could also use the `rsync` tool together with `ln` to form hard-links to files if they *did not change*. This strategy is similar to *differential backups* and creates *snapshots in time* but would require more disk space than the `diff` approach above.

LOCAL VERSION CONTROL

- Doing all of this through a shell script is, of course, *error prone* and associated with quite some overhead to get the script to a state with production quality.
- Additional complexity will be added when you must account for *multiple users* which are possibly not working on the same machine. (Each user has a local VCS database, how do you synchronize it?)
- There are many tools that can do the job much more efficient, with a minimal extra overhead for you to maintain your code/files.

WHY VCS IS A REQUIREMENT

- Every change you commit is tracked (author, time, changelog)
- You can easily revert changes
- The VCS allows you to *bisect* your commits in case you need to isolate a bug (e.g. `git-bisect`)
- You know exactly *when*, *where and who* introduced *what* changes. This is crucial in large projects with many developers.
- ***General rule:*** Think carefully about changes you commit in large projects. You should always try to create commits that follow a logical pattern. Favor many *small* commits rather than one huge commit. Bisection for debugging will be useless if the commit granularity is not fine enough.

EXAMPLES OF VERSION CONTROL SYSTEMS

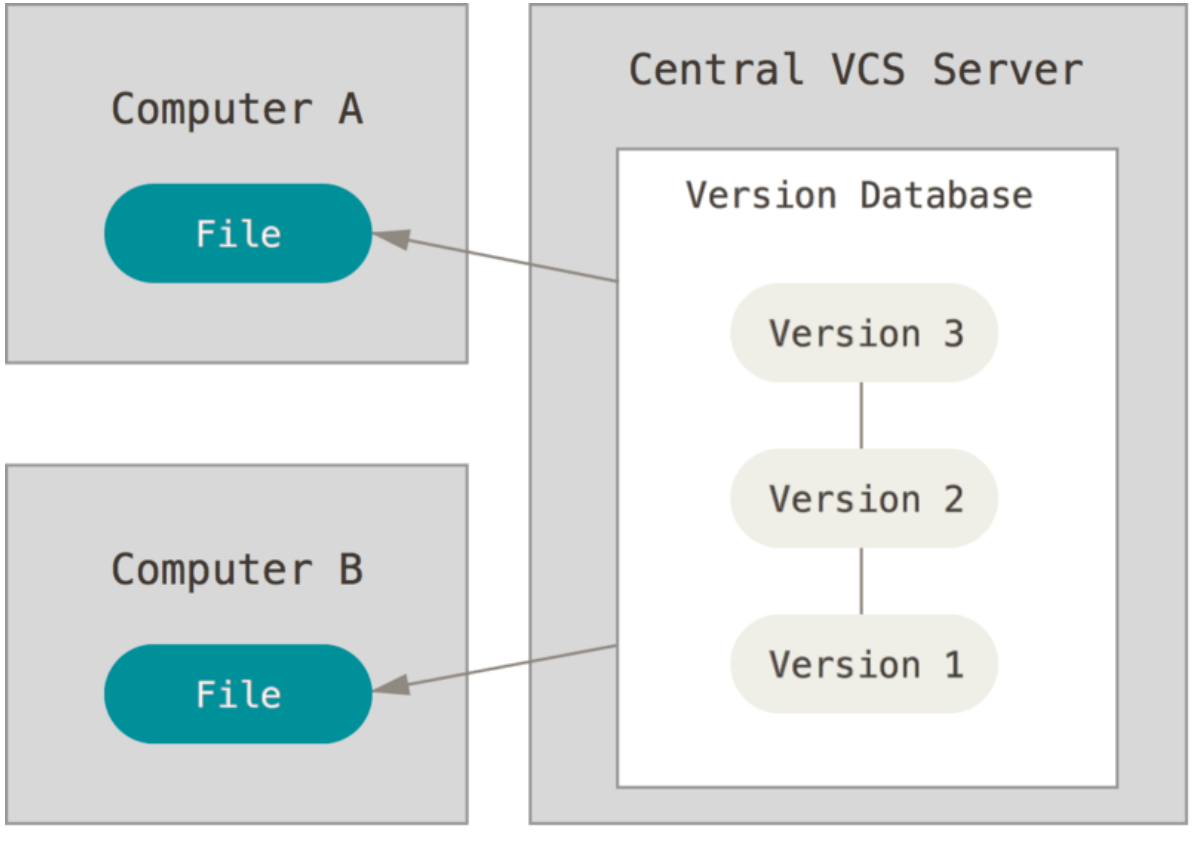
Concurrent Version Systems (CVS)	http://cvs.nongnu.org/
Subversion (SVN)	https://subversion.apache.org/
Helix Core (proprietary)	https://www.perforce.com/products/helix-alm
Bazaar	https://bazaar.canonical.com/en/
Darcs	http://darcs.net/
Git *(industry standard)*	https://git-scm.com/
Mercurial	https://www.mercurial-scm.org/

Centralized VCS

Distributed VCS

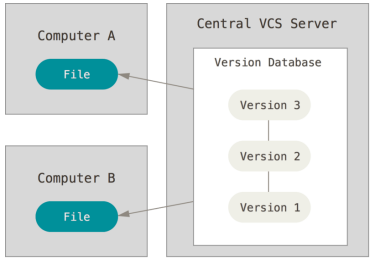
Google Drive and Dropbox are **no** examples of VCS

CENTRALIZED VCS MODEL (CLIENT-SERVER)



- Instead of hosting the VCS database *locally*, it is hosted on a *central server*.
- Administrators have fine grained control over access rights and policies.
- Everyone knows to a certain degree what everyone else does.

CENTRALIZED VCS MODEL (CLIENT-SERVER)



Workflow in a centralized VCS (CVCS):

1. A developer is checking out a file to do work on.

   What should be the policy for other developers who want to checkout the same file?

   i. ***File locking:*** local read-only, write access through locking.

   ii. ***Version merging:*** local read- and write-access, conflicts resolved through merging algorithms.

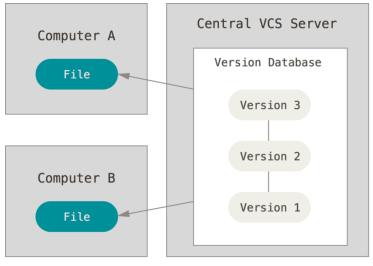
2. After the work is done, the changes are committed and checked-in on the central repository:

   i. ***File locking:*** the check-in process is trivial. The updated file is simply added.

   What is the main drawback?

   ii. ***Version merging:*** if multiple developers modified the same file, individual changes must be *merged*. Expensive operation for chunks of changes and difficult to resolve automatically.

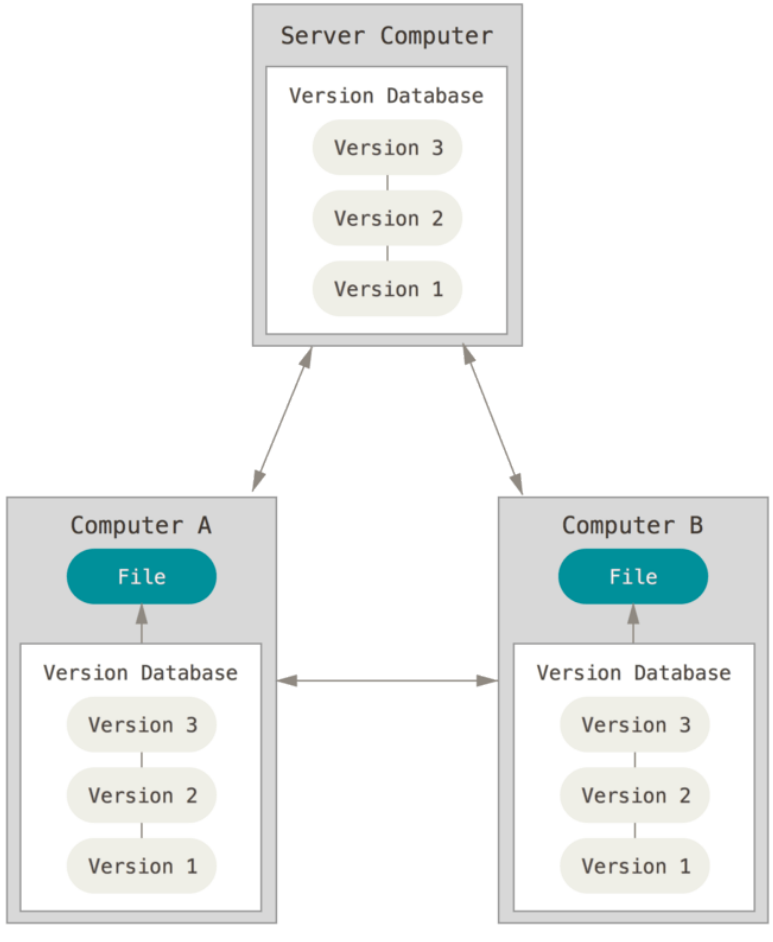
CENTRALIZED VCS MODEL (CLIENT-SERVER)



The main problems with centralized VCS are:

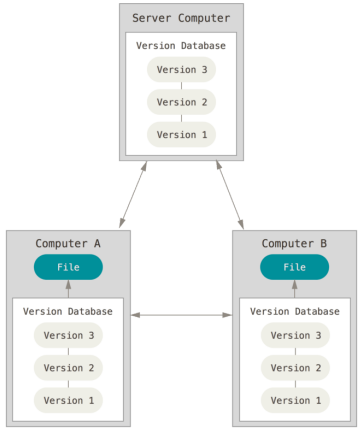
- ***Single server instance***. If the server goes down (e.g. a fire), you better have a backup somewhere else!
- During server downtime (or you are offline), you can not work on the project!
- ***Branching*** for feature testing and implementation is not trivial in CVCS.
- Resolving merge conflicts can be an expensive problem due to the *sequential* nature of the CVCS model.

DISTRIBUTED VCS MODEL



- Clients do not only checkout the latest snapshot, but *mirror the full repository*.
- If a server instance dies, *everybody* has a backup locally.
- It allows you to work completely offline.
- Allowing for multiple server instances (remotes) is trivial and allows for *hierarchical* collaboration.

DISTRIBUTED VCS MODEL



Workflow in a distributed VCS (DVCS):

1. A developer *clones* a repository if it does not exist locally yet.
2. Modifications are committed on the *local* repository. There are two ways these commits can be distributed to collaborators:
    i. Sending *patches* to collaborators via email. Receivers then patch their local repositories *(you would not need a remote server computer in this case)*. The way Git was designed. The Linux kernel and Git itself use this distribution model. See https://git-send-email.io/ for more about this.
    ii. Hosting the repository on a *public* remote, where commits can be *pushed* to or *pulled* from *(as in the figure on the left)*. This workflow does not originate from Git itself but was introduced by platforms such as GitHub and GitLab. To prevent unauthorized pushes to a remote, the *fork* and *pull-request* model is used on these platforms.

What is the main potential problem you deal with when designing a distributed version control system?

ENTER GIT

WHY GIT?

We have seen that there are many alternative VCS. The reason Git exists is because of the Linux kernel development:

- The Linux kernel is a very ***large*** project
- There are many contributors developing in ***parallel***
- On many different (feature) ***branches***

Linus Torvalds created Git. The home of Git is https://git.kernel.org/pub/scm/git/git.git/

WHY GIT?

The characteristics of the Linux kernel project therefore have the following VCS requirements:

- The VCS must be *fast*
- It must have a *simple* design
- Strong support for *non-linear development* (thousands of parallel branches)
- Fully *distributed*
- Ability to handle large projects efficiently in terms of speed and data size

These are all features implemented in the Git version control system. And it is open-source of course.

Side note: before Git, the Linux kernel used BitKeeper which changed licensing policies to commercial focus in 2005 (the year Git was born). Now the Linux kernel uses Git exclusively and BitKeeper became an unused open-source project.

GIT REFERENCES

Recommended Git references:

- Scott Chacon and Ben Straub, *"Pro Git book"*, open access https://git-scm.com/book/en/v2
- Excellent Atlassian tutorial: https://www.atlassian.com/git/tutorials
- Git get started on GitHub: https://docs.github.com/en/get-started/quickstart/set-up-git
- StackOverflow beginners guide: https://stackoverflow.com/questions/315911/git-for-beginners-the-definitive-practical-guide

Selection of projects and companies that use Git:

THE ESSENTIALS OF GIT

Before we start:

The first steps you should do when working with Git is to ensure that your contact information is setup correctly. Every commit contains the author information with an email address. All your global Git customization is done in the file `~/.gitconfig`. Verify that the file exists and you have at least the following lines in the file:

```
[user]
    name = FirstName LastName
    email = you@domain.com
```

THE ESSENTIALS OF GIT

Before we start:

The first steps you should do when working with Git is to ensure that your contact information is setup correctly. Every commit contains the author information with an email address. All your global Git customization is done in the file `~/.gitconfig`. Verify that the file exists and you have at least the following lines in the file:

```
[user]
    name = FirstName LastName
    email = you@domain.com
[core]
    editor = vim  ; and possibly this for a preferred editor (otherwise defaults to system)
```

See `git help config` for all configuration details.

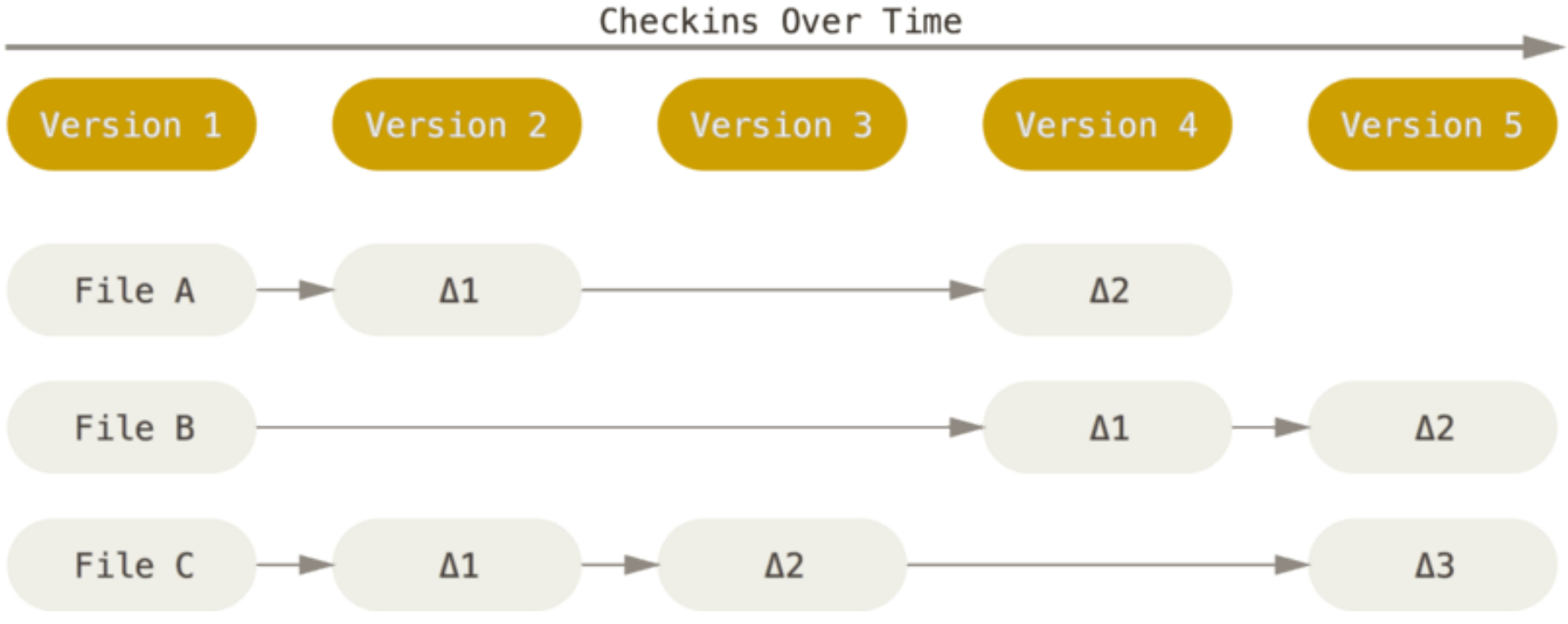
THE ESSENTIALS OF GIT

The major difference between Git and any other VCS is the way Git treats its data

- Most other VCS store the data as a ***list of file-based changes***
- We refer to this as ***delta-based*** version control
- Think of "delta" as a *difference*, often denoted by the Greek Delta Δ
- ***Recall the earlier slide where we tried to implement our own VCS*** → using the `diff` command to implement a VCS system is a Δ-based approach.

THE ESSENTIALS OF GIT

Δ-based:



Storing data as a series of differences relative to a base version (delta-based VCS).
Think of the Δ hunks as patches applied to the individual files.

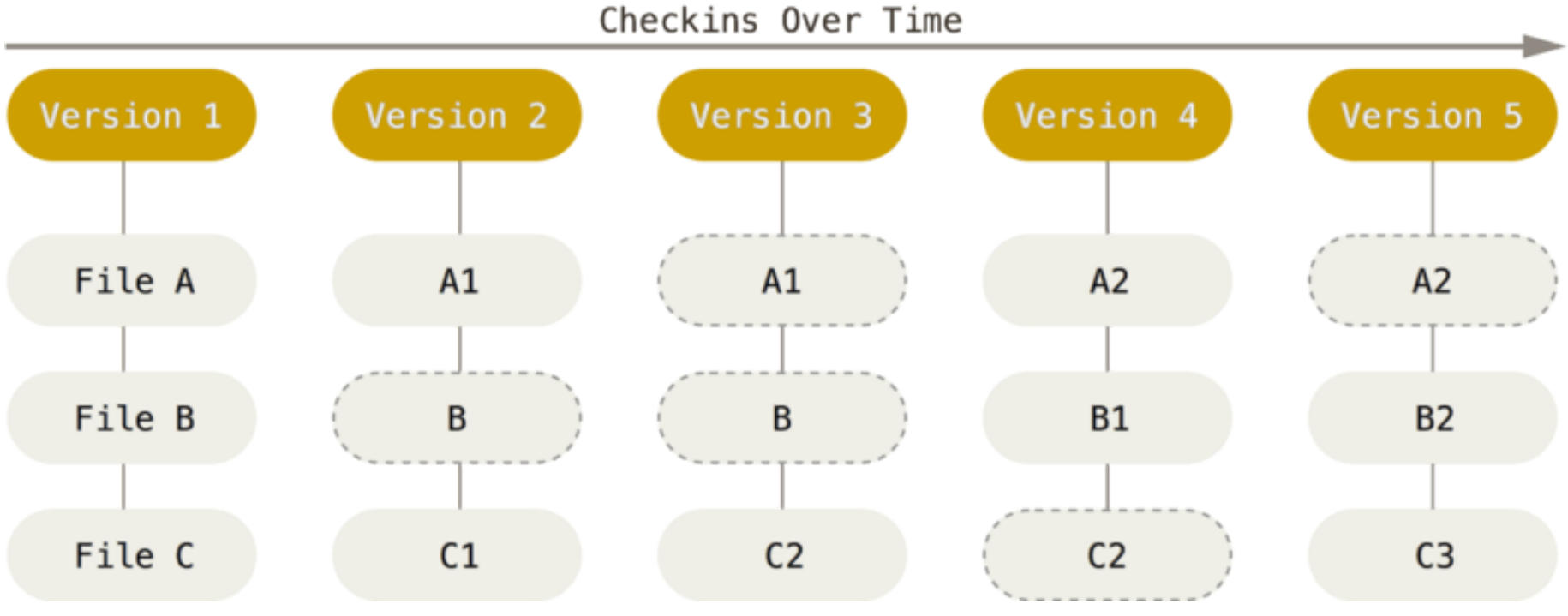
THE ESSENTIALS OF GIT

Git sees its data not as individual files but a whole file system. It stores the *state* of the full file system for a particular instant in time.

- Such a frozen state is called a ***snapshot***.
- Think of it as if Git is taking a picture of all your files every time you create a commit.
- If a file has not changed Git will *not store it again* but simply add a ***reference*** to the already stored file.
- ***Recall the earlier slide where we tried to implement our own VCS →*** using the `rsync` command (with hard-links) to implement a VCS in the style of a differential backup is more like Git handles its data.

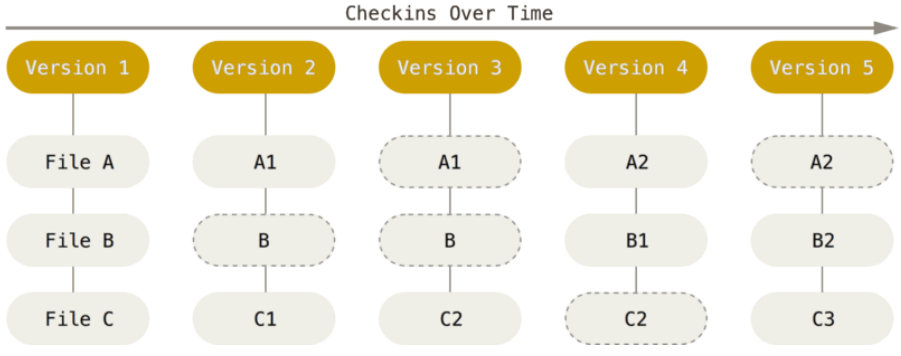
THE ESSENTIALS OF GIT

The way Git treats data for versioning:

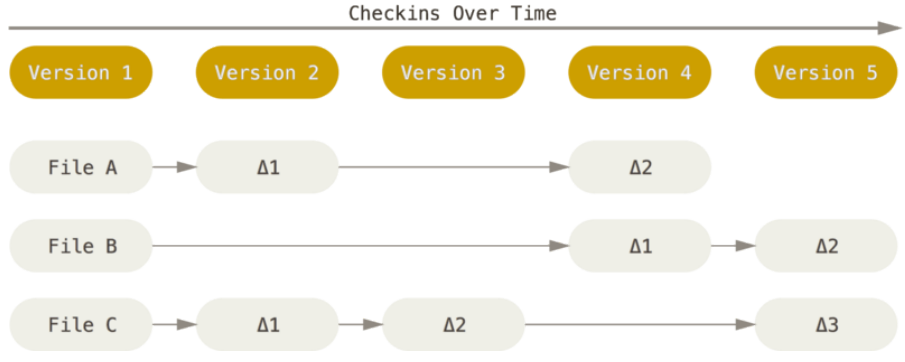


Storing data as a *stream* of *snapshots*: each vertical version is a snapshot of the complete file system within the repository. Dashed lines indicate references to the file of the previous version because the file has not changed.

THE ESSENTIALS OF GIT



Git: each version is a snapshot of the *full* file system



Most other VCS: each version records a set of differences between *individual* files

Time flows from left to right.

THE ESSENTIALS OF GIT

Git *does not* reference its files by name!

- Everything in Git is referenced by *hashes*. Internally, Git uses the SHA-1 hashing algorithm for this.
- These hashes look similar to something like this: `05682360767630528cc5188a81f88d5e64711608`
- A hash is *uniquely* determined given the content of a file.
- This means that changes in file state you commit ***are final*** once they are shred on a public remote.
- *Before* you share your changes (i.e. everything is still *local* in your repository), you still have options to rewrite commits without affecting any remote tracking histories. *This, and the ease of creating branches, allows for a lot of freedom in your software development process and it is where Git stands out from all other VCS!*

THE ESSENTIALS OF GIT

The anatomy of Git:

- There are ***three*** fundamental objects in Git:
  1. ***Blobs:*** encode the *contents* of files in your repository.
  2. ***Trees:*** model a (directory) hierarchy similar to the hierarchical Unix file system. The leafs of trees hold either *blobs* or other *trees*.
  3. ***Commits:*** contain one tree and they store to references one or more *parent* commits. A *branch*, therefore, is nothing more than *a sequence of commits* and it can be referenced by the most recent commit.
- A *blob* is the fundamental data unit in Git. Everything about Git is really about how blobs are managed.
- Trees and commits are just high-level structures to manage blobs.
- All of these objects are *hashed* and identified with the unique hash value. These hashed objects are stored in `.git/objects`.

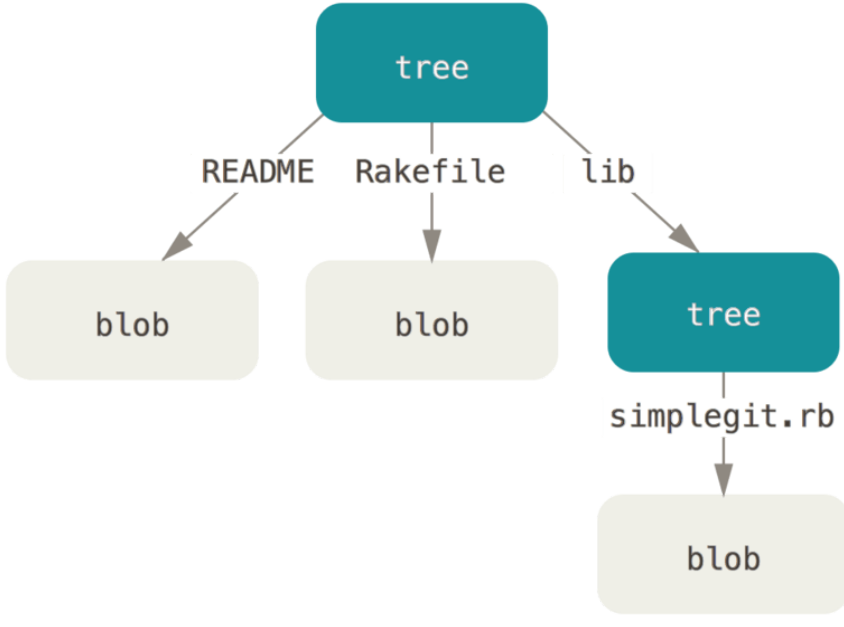
THE ESSENTIALS OF GIT

The anatomy of Git:

Example repository structure:

```
.
├── lib
│   └── simplegit.rb
├── Rakefile
└── README
```

Corresponding Git commit:



- A *blob* is the fundamental data unit in Git. Everything about Git is really about how blobs are managed.
- Trees and commits are just high-level structures to manage blobs.
- ***Highly recommended reading:*** https://jwiegley.github.io/git-from-the-bottom-up

THE ESSENTIALS OF GIT

- The previous slides were addressing *low-level* concepts in Git.
- Understanding them makes working with Git on the high-level easier.
- This is where we turn to now (high-level).

There are ***three*** file states in Git that are *important to understand*:

1. `modified`: you have changed the file but not committed to your local database.
2. `staged`: you have marked a modified file to go into your next commit snapshot.
3. `committed`: the data is safely stored in your local database (SHA-1 hash is computed).

These states apply to files that are *tracked* by Git (under version control). Files not marked for version control (and not ignored by Git either) are indicated as *untracked* files in Git.

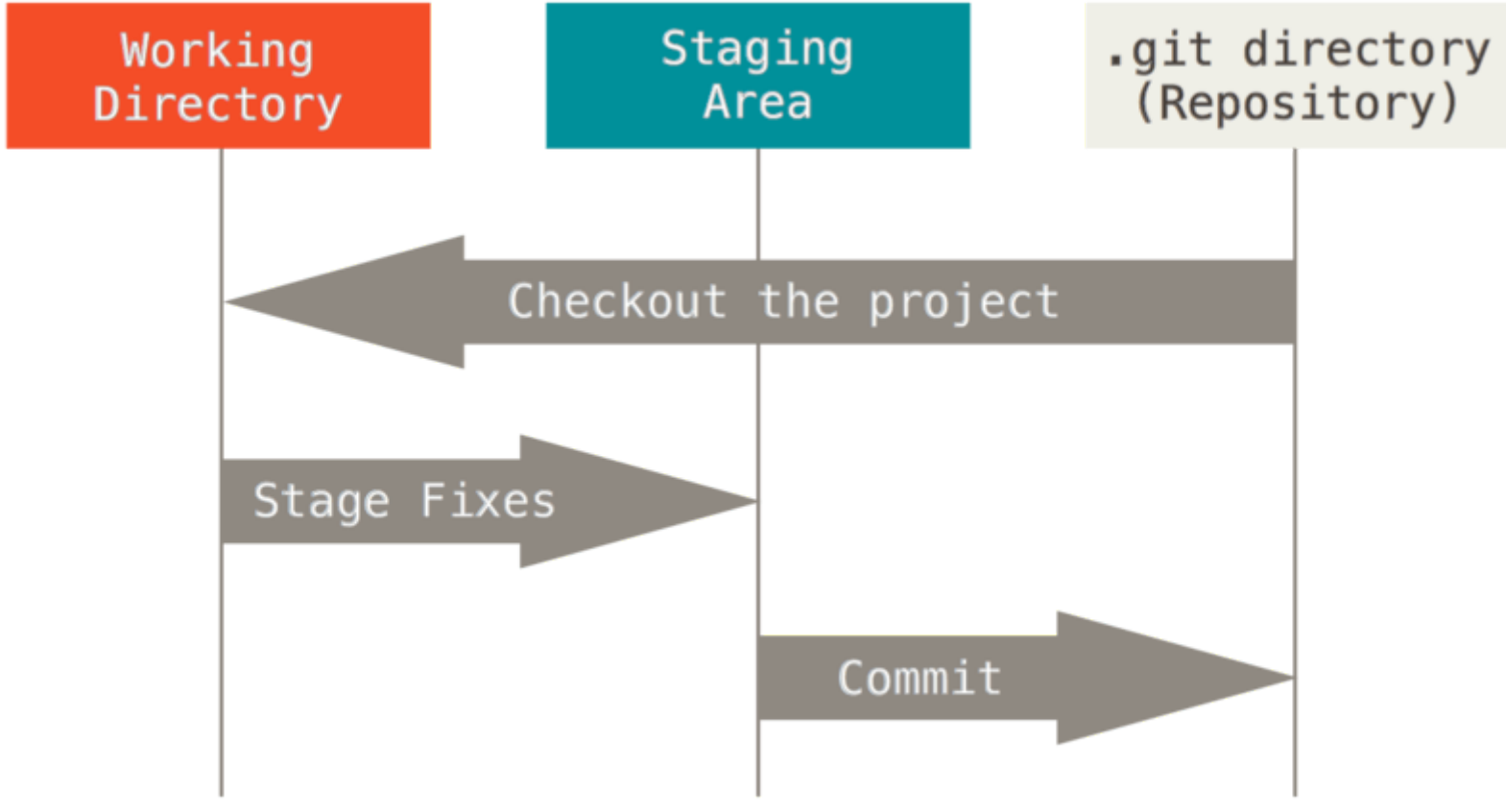
THE ESSENTIALS OF GIT

This leads to three main sections in a Git repository:

1. The ***working tree*** where you *checkout* and modify files
   - Holds the human readable contents of a decompressed snapshot (Git stores data using compression algorithms)
   - If you have a branch checked out, the files you see are in your *working tree*.
2. The ***staging area*** where you put your staged files (also referred to as ***index***)
   - The staging area is simply a file inside the `.git` directory called `index`
   - You *selectively* add changes to the `index` that you plan for the *next commit*
   - *You should add logical, small changes at a time.* If you are faced with a larger problem, divide it into smaller sub-problems and go step-by-step. ***Why is this good practice?***
3. The ***`.git` directory*** is where your commit *history* lives (i.e. the Git repository)
   - This is where Git stores metadata and the object database for your project. *When you remove it, you lose all of your version control.*
   - It is the *heartbeat* of Git and the data that is copied from a remote when you clone a project.

THE ESSENTIALS OF GIT

This leads to three main sections in a Git repository:



The three main sections a certain change traverses in Git and how information flows.

STEP BY STEP GIT EXAMPLE

Step 1: obtain `.git` directory

- To start VCS with Git you need a `.git` directory inside the *root* directory of your project. You have two options:
  1. You can *initialize* a new one
  2. Or *clone* an existing project (you did this in the first homework)
- Initialize from scratch (`git help init`):

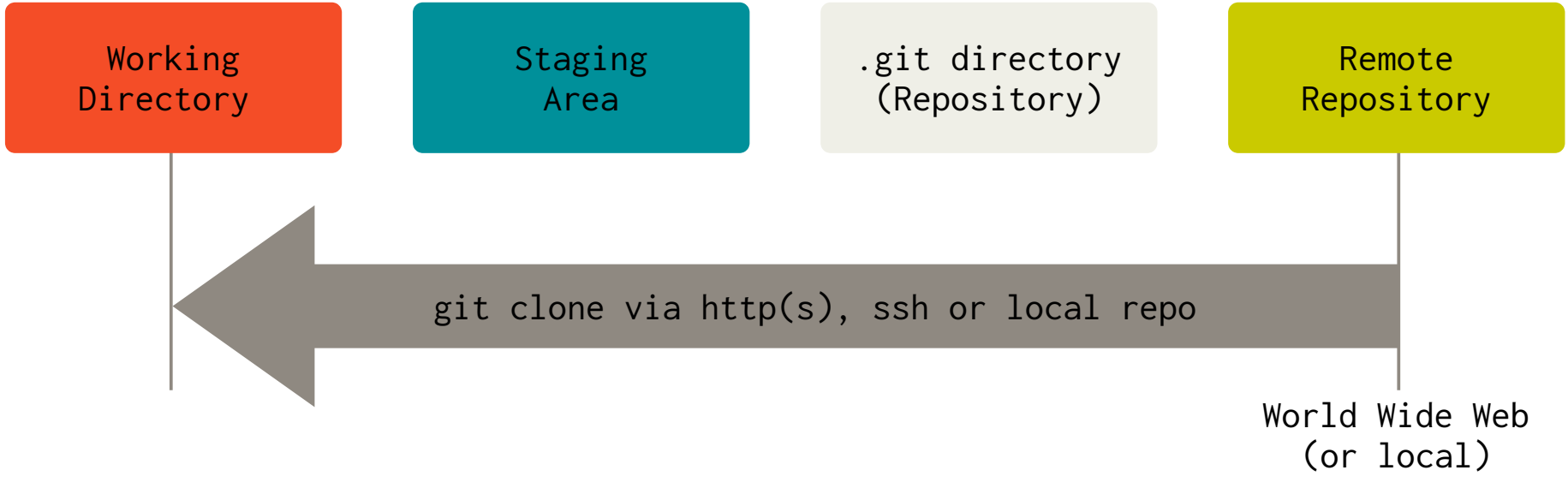
```
$ git init  # you must run this command in the root of your project
Initialized empty Git repository in /home/fabs/myrepo/.git/
$ ls -a  # the git init command above created a .git repository
.  ..  .git
```

- Clone a local project (can be from a remote URL or a local repository):

```
$ cd ..
$ git clone myrepo cloned_repo # local project from above (the warning is expected)
Cloning into 'cloned_repo'...
warning: You appear to have cloned an empty repository.
done.
```

STEP BY STEP GIT EXAMPLE

Step 1: obtain `.git` directory (visually what happens)



STEP BY STEP GIT EXAMPLE

Step 2: check the status (we are in the empty `myrepo` of step 1)

```
$ git status  # inside: /home/fabs/myrepo
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
$ ls .git  # no index yet...
config  HEAD  hooks  objects  refs
```

STEP BY STEP GIT EXAMPLE

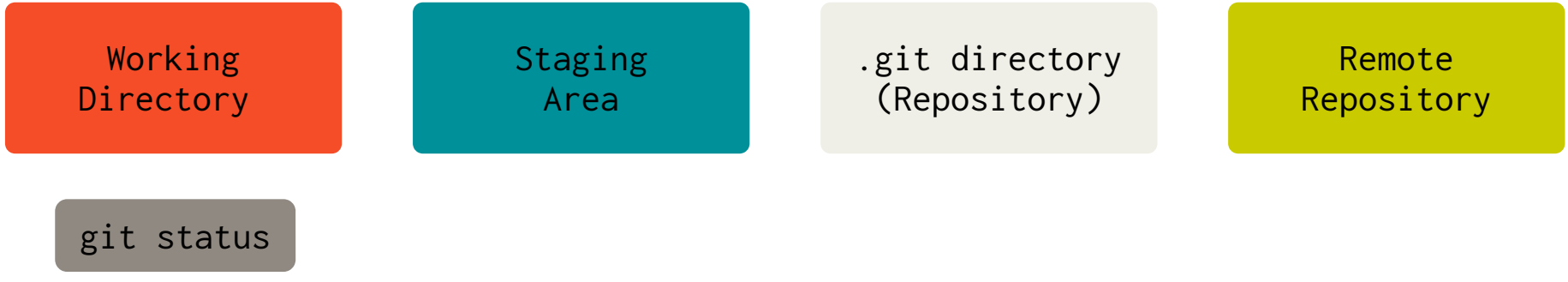
Step 2: check the status (we are in the empty `myrepo` of step 1)

```
$ git status  # inside: /home/fabs/myrepo
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
$ ls .git  # no index yet...
config  HEAD  hooks  objects  refs
$ echo "Hello Git!" > file
$ git status
On branch master

No commits yet

Untracked files:  # files that git is not aware of at this point
  (use "git add ..." to include in what will be committed)
        file      # we have just created this file

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

Using `git status` is important for orientation and you should develop a habit to invoke this command often.

STEP BY STEP GIT EXAMPLE

Step 2: check the status (we are in the empty `myrepo` of step 1)



The status is run in the working directory. It will show the status of the working directory which my include information about the *staging area*.

STEP BY STEP GIT EXAMPLE

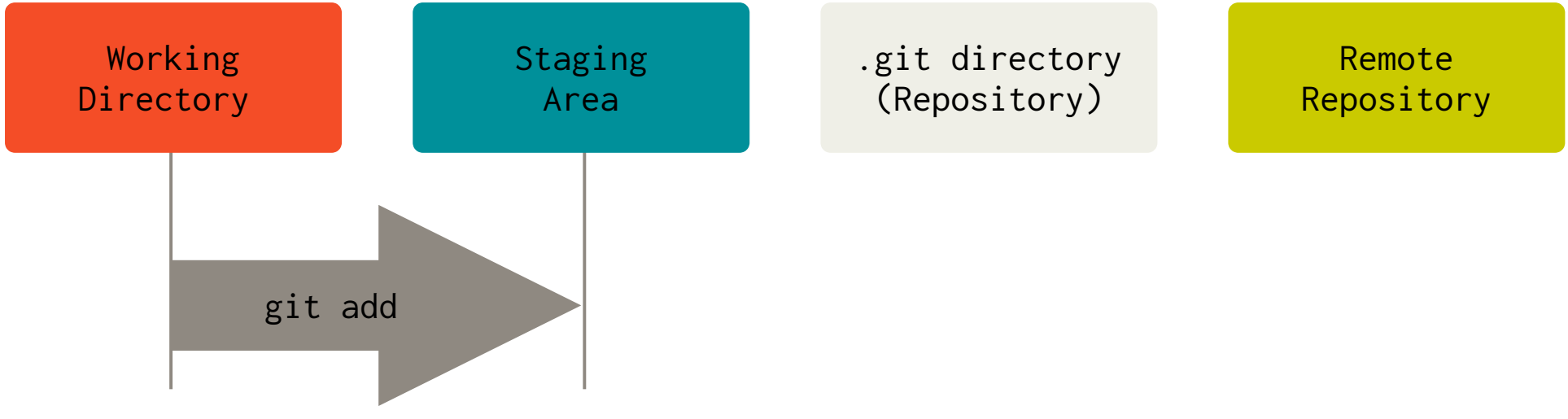
Step 3: add new or modified files to the index

```
$ git add file  # inside: /home/fabs/myrepo
$ git status
On branch master

No commits yet

Changes to be committed: # now git is aware of the file...
  (use "git rm --cached ..." to unstage)
        new file:   file

$ ls .git  # ...and we have an index file (a.k.a. staging area)
config  HEAD  hooks  index  objects  refs
```

STEP BY STEP GIT EXAMPLE

Step 3: add new or modified files to the index

New or changed files are added to the ***index*** also known as the *staging area*. Files in the staging area will go into the next commit.



STEP BY STEP GIT EXAMPLE

Step 4: commit staged changes to the repository

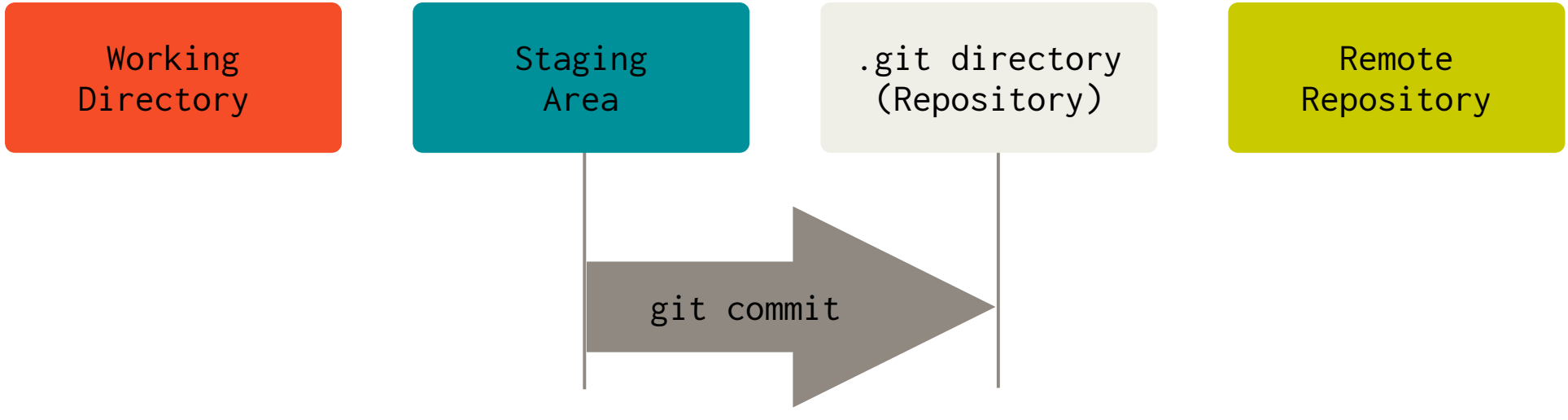
```
$ git commit -m "My short commit message"
[master (root-commit) 5e8adae] My short commit message
 1 file changed, 1 insertion(+)
 create mode 100644 file
```

- The `-m <message string>` option allows for a short commit messages on the command line. Useful if only few changes have been committed
- If you run `git commit` only, your editor specified in `~/.gitconfig` or the `GIT_EDITOR` environment variable will open up.
  See `git help commit` for more details.

STEP BY STEP GIT EXAMPLE

Step 4: commit staged changes to the repository

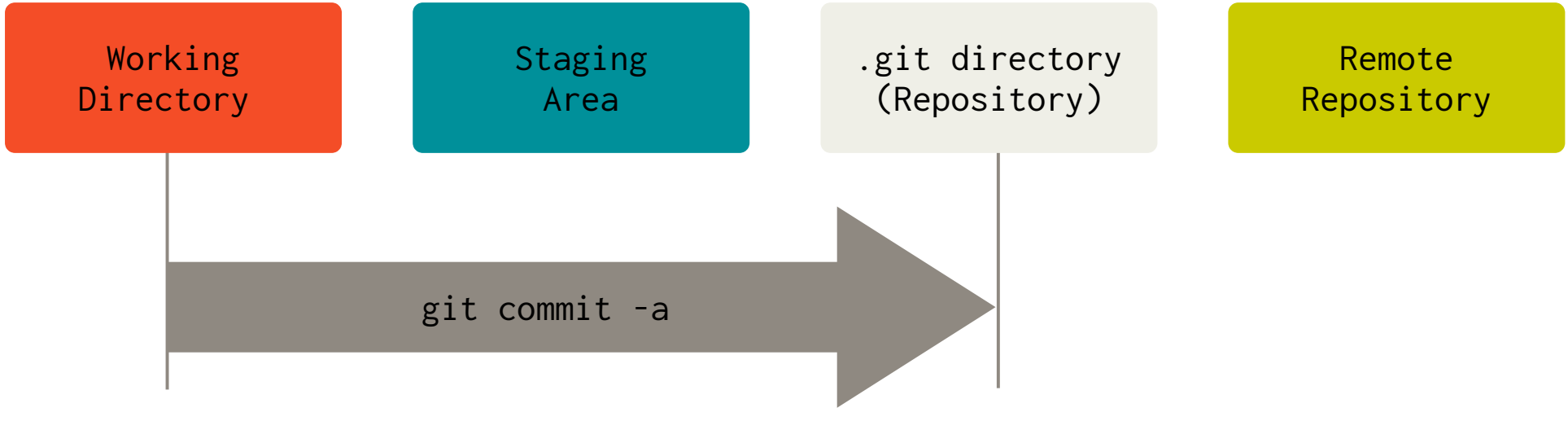
Staged changes are committed to the ***repository***.



STEP BY STEP GIT EXAMPLE

Step 4: commit staged changes to the repository

Shortcut to add tracked files that are *modified* or *deleted* in the working directory and commit right away. This omits the staging step with `git add`.



STEP BY STEP GIT EXAMPLE

Step 5: push commits to the remote repository

```
$ git push -u origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 264 bytes | 264.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To <remote path or URL>
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'
```

- The option `-u` is only needed if you push for the first time and you would like Git to remember that subsequent pushes from the local `master` branch are meant for the `origin/master` branch on the remote called `origin`.
- The name `origin` is a conventional default name for the default remote repository in Git.

STEP BY STEP GIT EXAMPLE

Step 5: push commits to the remote repository

STEP BY STEP GIT EXAMPLE

Step 6a: synchronizing with the remote (fetch)

```
$ git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 247 bytes | 247.00 KiB/s, done.
From ../project_remote
   5e8adae..7b53cec  master     -> origin/master
```

- You only need `git clone` at the very beginning to get the `.git` directory
- Once you have a `.git` directory, you obtain new commits from the remote using `git fetch` (or `git pull`)
- You may specify a remote to fetch from

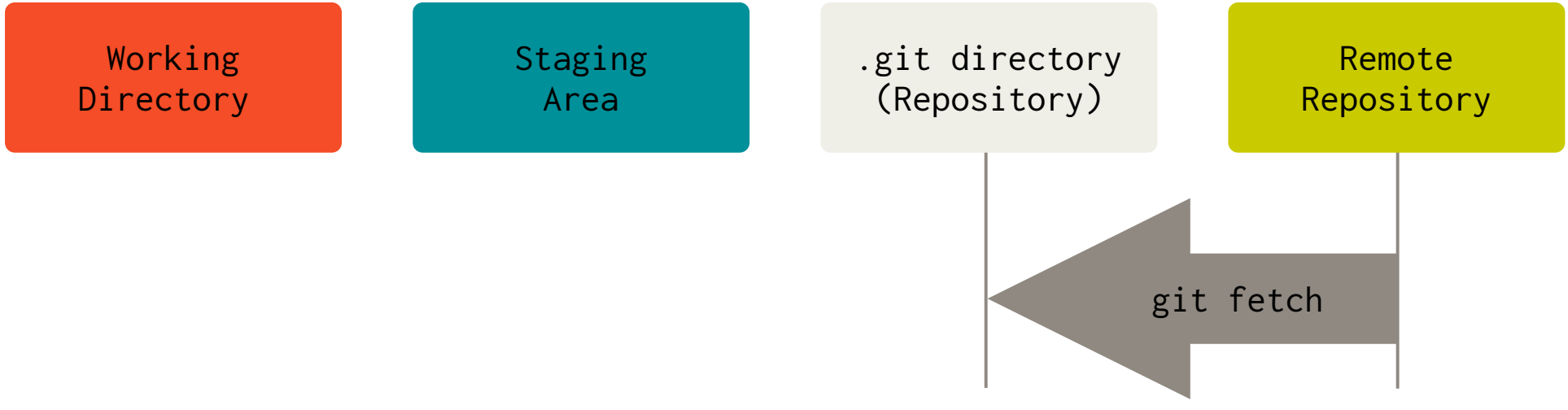
```
$ git fetch origin
```

or fetch from all the configured remotes

```
$ git fetch --all
```

STEP BY STEP GIT EXAMPLE

Step 6a: synchronizing with the remote (fetch)



STEP BY STEP GIT EXAMPLE

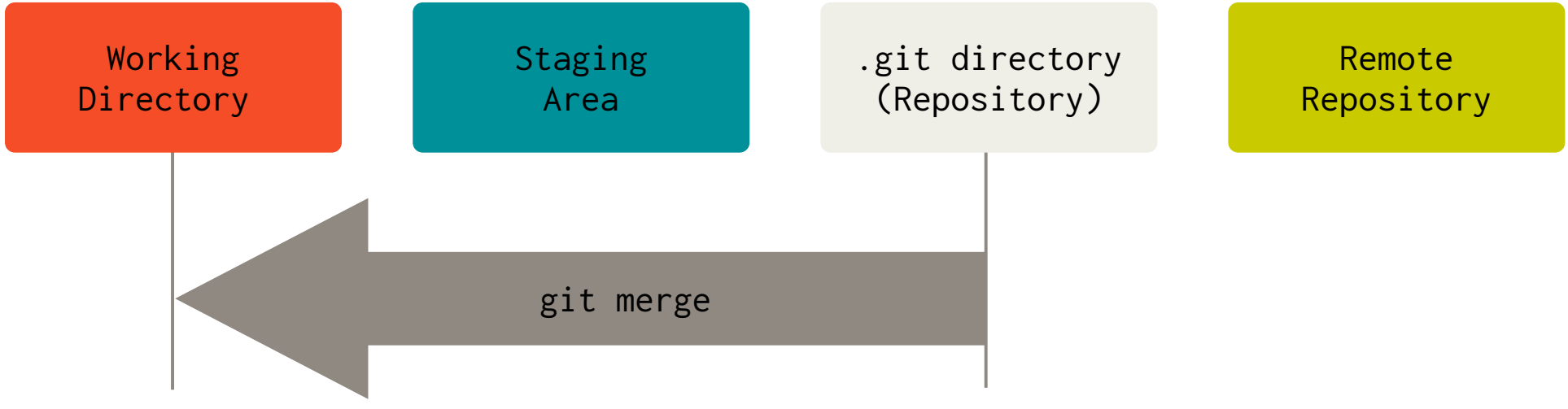
Step 6b: synchronizing with the remote (merge)

```
$ git merge
Updating 5e8adae..7b53cec
Fast-forward
 file | 1 +
 1 file changed, 1 insertion(+)
```

- Fetching from a remote only updates the remotes in the `.git` repository.
- To update your *working directory*, you need to merge the commits you just *fetched*.

STEP BY STEP GIT EXAMPLE

Step 6b: synchronizing with the remote (merge)



STEP BY STEP GIT EXAMPLE

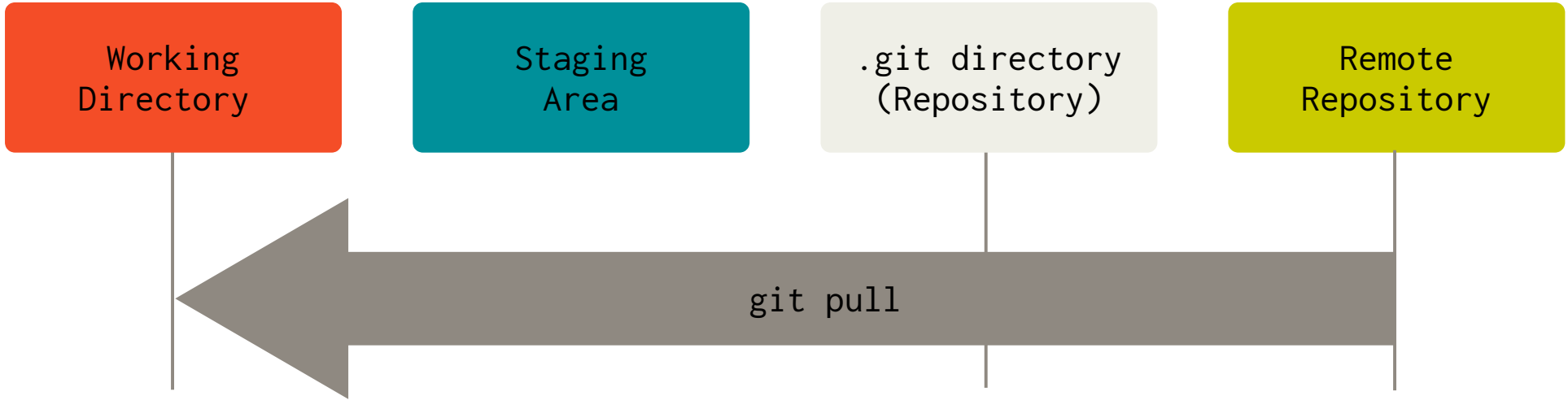
Step 6c: synchronizing with the remote (pull)

```
$ git pull  # combine git fetch and git merge in one go
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 248 bytes | 248.00 KiB/s, done.
From ../project_remote
   7b53cec..6a2d2af  master     -> origin/master
Updating 7b53cec..6a2d2af
Fast-forward
 file | 1 +
 1 file changed, 1 insertion(+)
```

- Fetching from a remote is often immediately followed by the merge.
- Git provides a convenience command to combine `git fetch` and `git merge` in one go using `git pull`.

STEP BY STEP GIT EXAMPLE

Step 6b: synchronizing with the remote (pull)



A `git pull` does update the remotes in the `.git` repository as well.

STEP BY STEP GIT EXAMPLE

Step 7: inspect the history of commits contributed by you and others

```
$ git log
commit 7b53cec71d375b4e570af1113d5a76059de3c1d1 (HEAD -> master, origin/master)
Author: Fabian Wermelinger <fabianw@seas.harvard.edu>
Date:   Fri Sep 3 20:13:33 2021 -0400

    Quick fix

commit 5e8adae9f07b34fca41334b7997750e13bbe6c92
Author: Fabian Wermelinger <fabianw@seas.harvard.edu>
Date:   Fri Sep 3 19:20:42 2021 -0400

    My short commit message

    You can be more verbose in the body of the commit.
```

Or compact in one line per commit:

```
$ git log --oneline
7b53cec (HEAD -> master, origin/master) Quick fix
5e8adae My short commit message
```

HOW OFTEN SHOULD YOU COMMIT CHANGES?

- ***Remember:*** in your local repo, you can be messy and clean up later if you are not sure yet where it is going.
- If you are working on something complex, it may make sense to commit very frequently, such that you can keep track of the impact of your changes (micro commits)
- You can always ***reorder*** and ***combine*** local commits *(before you push them to a remote where others have access too)*
- Committing once or twice a day is too few. Your commits will be bulky and it will be difficult to ***bisect*** if you introduced a bug.
- Commits that you intend to push should ***always build the code correctly*** (if you work with a compiled language) and ***pass all your unit tests***. This will help you avoid trouble in the long run. We will see in later lectures how this process is automated.

INTERACTIVE GIT REBASE

- When you have created some commits in your local repository, you may want to clean them up a bit before you push them to a remote.
- You can do this with *interactive* rebasing in your local commit history.
- The command for this is `git rebase -i`
- Your editor will open and it will allow you arbitrarily shuffle your local commits, squash them together, drop them all together or reword your commit message, for example.
- Interactive rebases are a great tool to tidy up after a tough code session.
- Use the `git_interactive_rebase` example in the lecture codes to practice!

RECAP

- Introduction to version control systems
- Centralized and distributed approaches
- Fundamental objects in Git: blobs, trees, commits
- Git working tree, staging area, the repository and remote repositories
- Interactive rebasing of local commits

Further reading:

- Chapters 1 and 2 in Scott Chacon and Ben Straub, *"Pro Git book"*, open access https://git-scm.com/book/en/v2
- The fundamental inner workings of Git: https://jwiegley.github.io/git-from-the-bottom-up