The joy of git

Version control

Literature

- Free book licensed under Creative Commons: Pro Git, find it at https://git-scm.com/book/en/v2
- Google it, stack overflow, and if all else fails, Pro Git

- Graphics are taken from Pro Git, unless otherwise specified

- http://lhcb.github.io/analysis-essentials has a good section on git

Goal

- Teach students how git works
  - Git is very useful if you work with code
  - Makes you more effective developer
  - Redundancy in your backups: harder to lose progress
  - Understanding how it works is a great skill to have in general (many people do not!)
  - (Physics) programming courses do not talk about it
  - Many open source projects use it: contributing

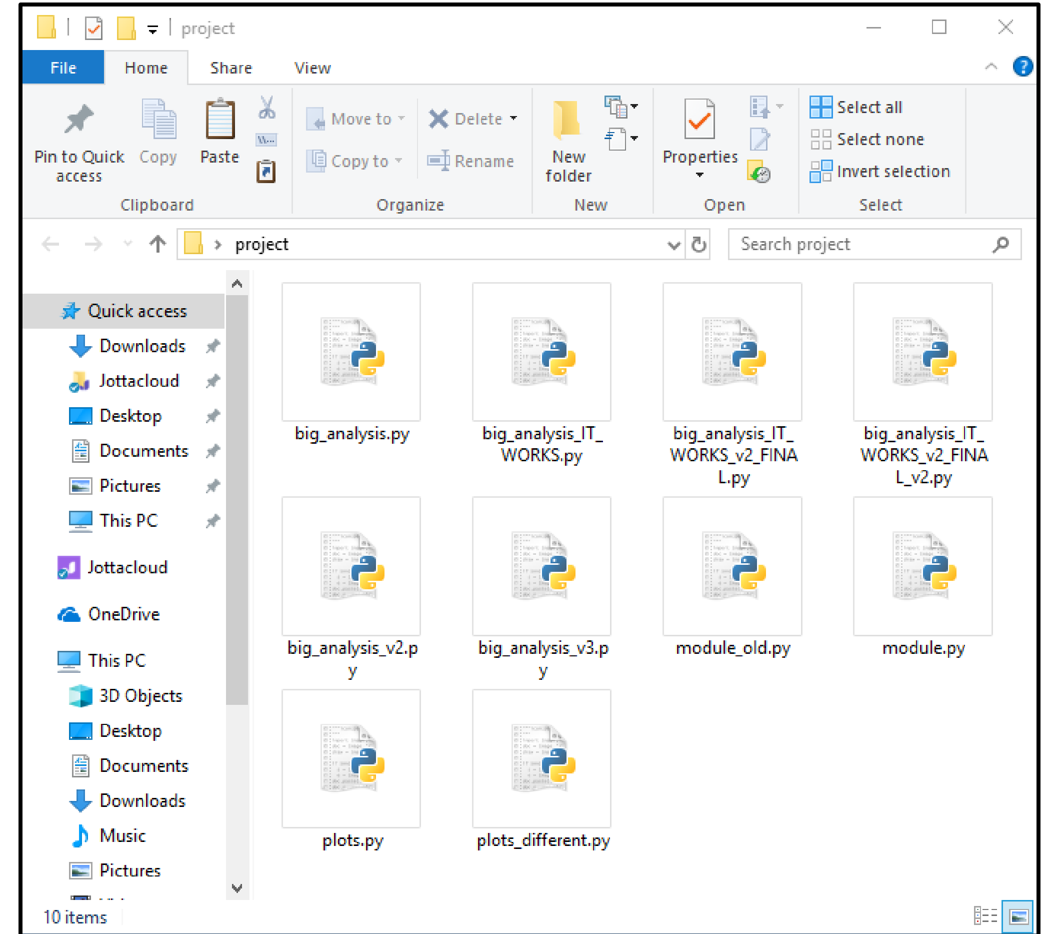
Outlook

- Part I : How does git work
- Part II : Basic commands
- Part III : Branching
- Part IV : Merging
- Part V : The cool stuff

What is version control

- Management of changes in documents
- MS office track changes
- Cloud storage systems track changes

- Working on code
- CVS, Subversion, git



Setting up git

- https://help.github.com/

How does git work (theory)

- Repository (repo) on drive
  - Repo contains the history of your project
  - Meta data: author, date, etc.
  - Data: code and changes
- .git folder
- .gitignore

Jargon

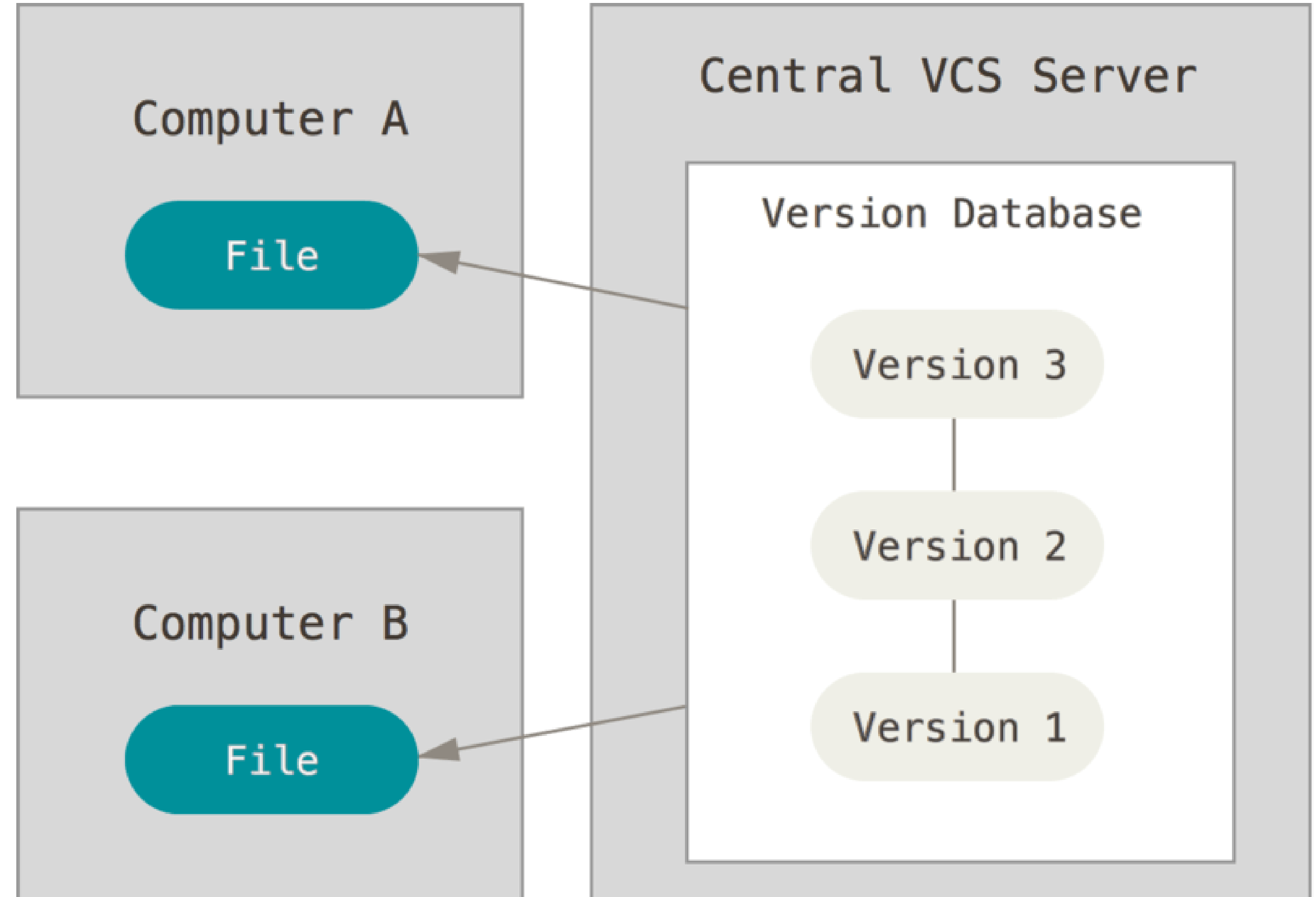
- Repo(sitory): folder containing the code that is tracked by version control
- Remote: server (or network drive) containing a copy of your repo
- Commit: recorded change to your repo
- Hash*: fingerprint of your commit

- Git: version control program
- Github: website where you can host your git repos (owned by MS as of 2018)
- Gitlab: program to run git on your own server (community and enterprise editions)

*Technically a hash is the output of a hash function, a one way function that takes some input and maps it into a fixed length output.

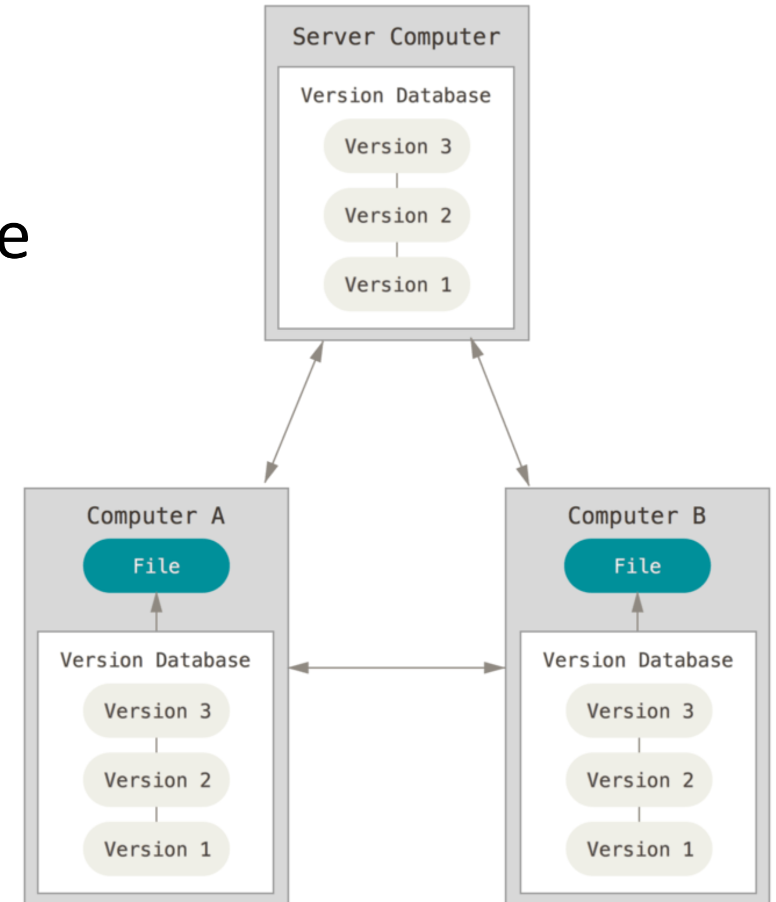
Centralized version control

- One source of truth: The server
- Have to sync every time you want to update your versioning
- My analogy: classical mechanics, only the server is right



Decentralized version control

- Every user has their own truth: their history
- By syncing with the server* you communicate with other users
- You can work locally, f.ex: on a plane
- You can rewrite your own history (or everyones)
- My analogy: relativistic mechanics, all observers (users) are right

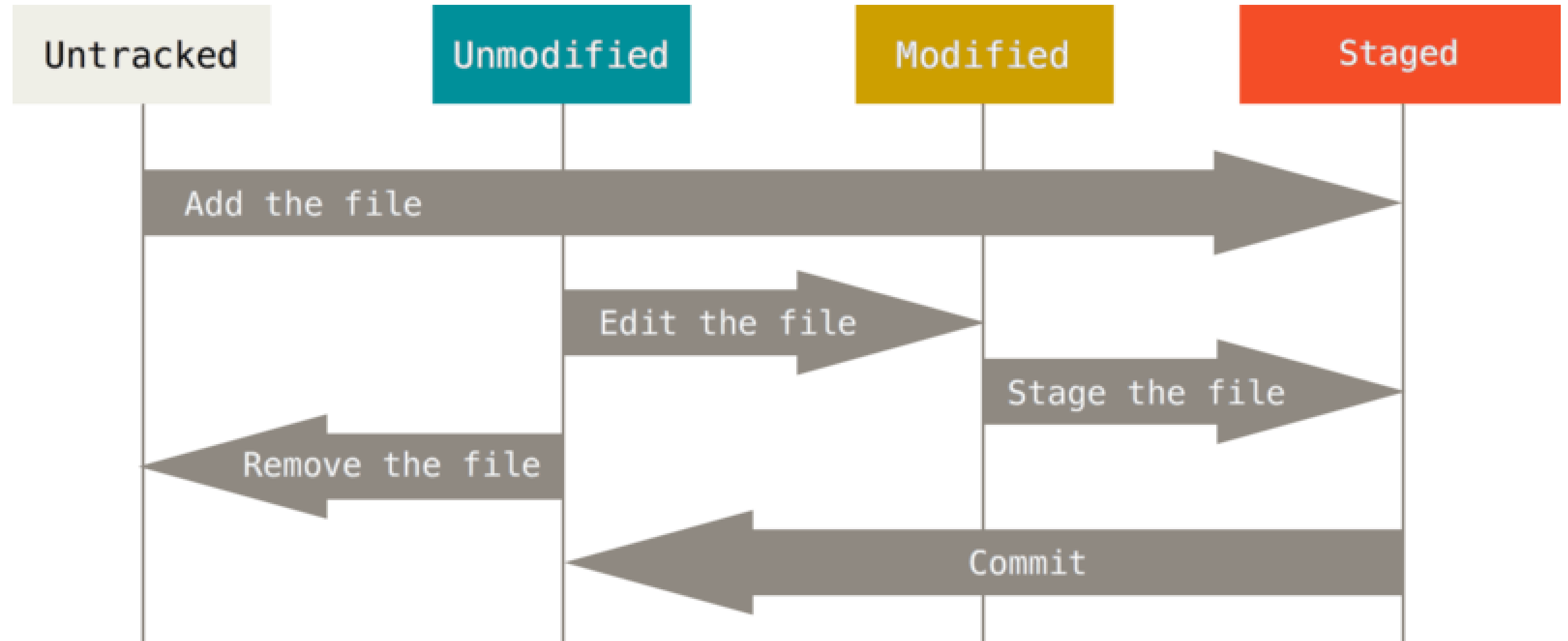


*You do not need a server, you can use git on network drives.

Basic work flow

Normal file system:
All files are untracked

File tracked:
-Unmodified
-Modified
-Staged (about to updated)

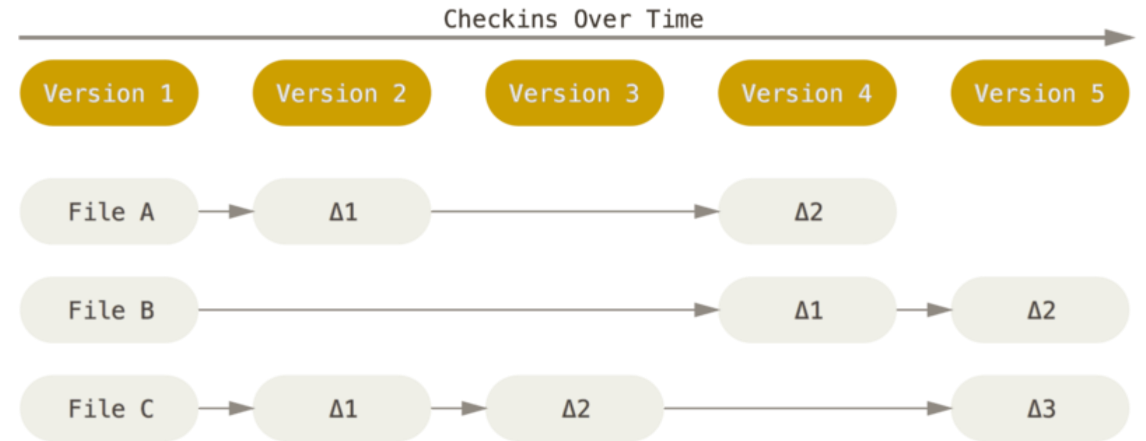


Timelines of versions

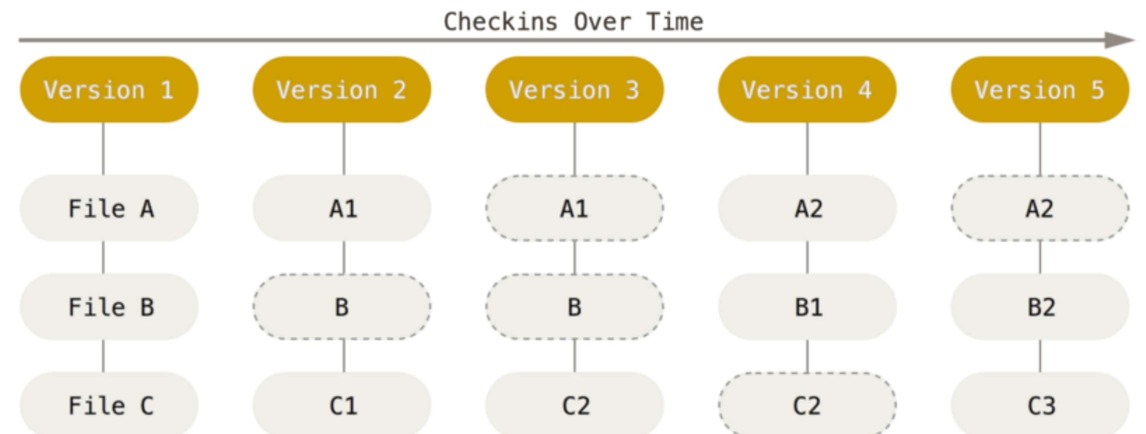
- File history
- File diffs

- 'Just a bunch of pointers' with file changes attached to them

Central server: Saving diffs of files

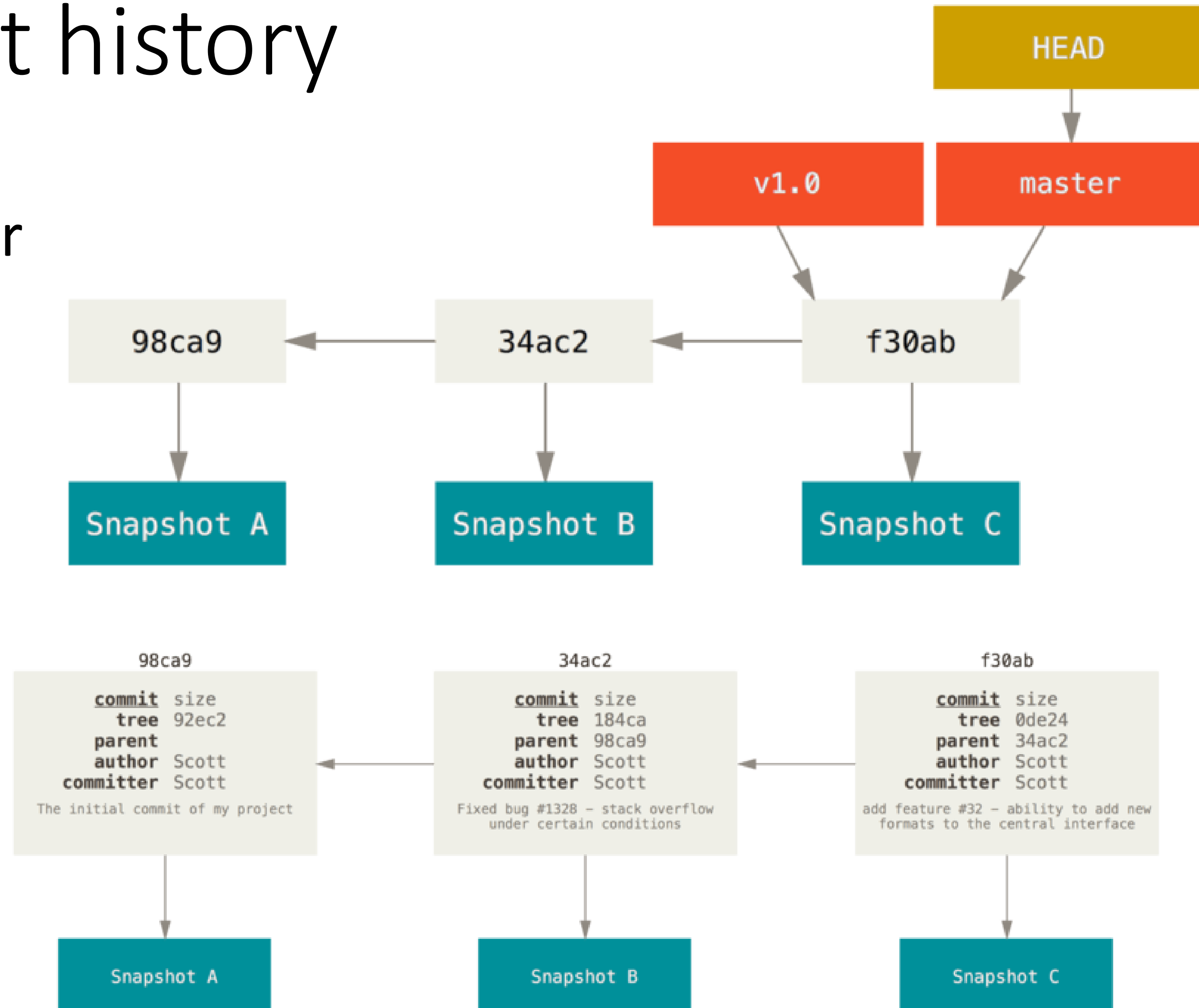


Git: Saving snapshot of all files



Overview of a project history

- Branches with a history: master
- Tag: v1.0
- HEAD: special pointer
- Commit SHA-256: unique identifier based on previous and current commit
- Meta-data:
  author, committer, message, …



Basic commands

Basic git workflow

- Local
  - init: initialize your repository, you only do this once.
  - add: add a new file to track, or add a new change to be committed (recorded).
  - commit: record your changes to the repository.
  - status: get the current status of the repository, like new changes, moved files
  - log: look at the repo history
  - show: look at the *actual* changes of a certain commit
  - checkout: checkout a file/commit/version/branch
  - diff: see the changes made in your folder, or compare with historic changes
- Remote
  - clone: copy a remotely hosted repo into a folder on your drive
  - push: upload your changes to the remote server, now others can see your changes!
  - pull*: get the changes from other users from the remote server, now you can see their changes!
- Git log: look at the commit history
- Unstaging: when you want to remove a file from the commit you are about to make

*Pull is two commands: fetch and merge, more on this later

If you want to mess around while I present:

```
$ ssh USER@login.nikhef.nl
USER@login.nikhef.nl's password:
$ git clone /data/antares/users/ljnauta/test_repo
$ git ...
```

Getting a repo

```
$ mkdir my_repo
$ cd my_repo
$ git init
```

```
$ git clone https://github.com/pandas-dev/pandas
```

Status and history

- What is HEAD?
  HEAD is where you currently are (commit, point in history)

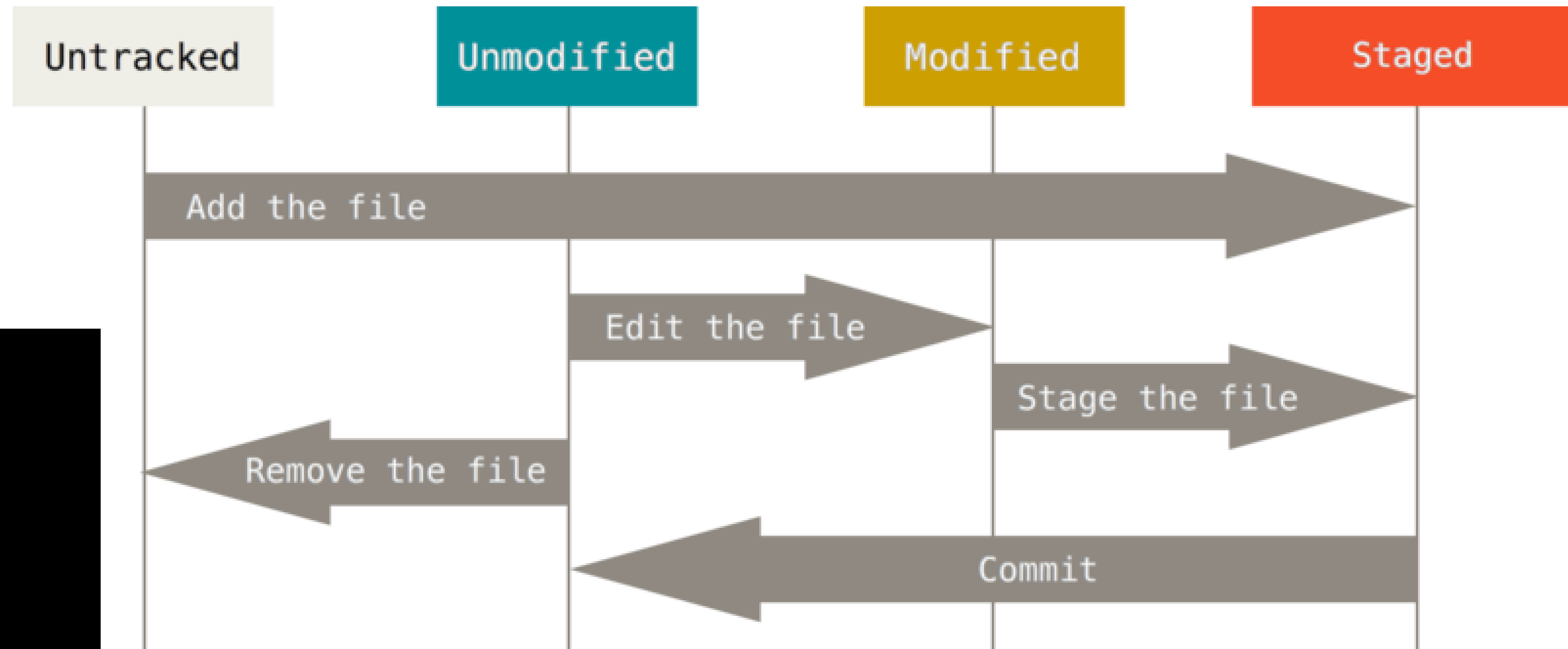
```
$ git status

$ git log
$ git log --graph --pretty
$ git log --graph --pretty=oneline --decorate --abbrev-commit --all
```

Ignoring files

- In general you only track code
- You do not want to track all files:
  - Compiled files: user makes themselves
  - Blobs: not trackable since it's not plaintext, so diffs are meaningless
- Every folder (or root for whole repo) can have a .gitignore file where you can add files to ignore by git
- Ignored files can still be added, but until tracked: ignored by `status`

Basic commands




```
Untracked: file1
$ git add file1
$ git commit
Tracked: file1

Edit file1
$ git add file1
$ git commit -m "changed file1"

$ git status
```

Basic commands

```
$ git show c1efc5e
```

Basic commands

- Git checkout: go through history easily
  - Checkout a changed file (but not committed) to the current version:

```
$ git checkout file1
```

  - Checkout a commit at in the repo history:

```
$ git checkout f34a200b
```

  - Checkout a different branch to work on (you will end at the tip):

```
$ git checkout my_awesome_branch
```

  - Checkout a certain version of the repo (can be in the past on a different branch):

```
$ git checkout v3.14
```

  - Checking out a file deletes all your current work
  - Checking out a commit only works if there are no changes

Undoing your changes and unstaging

- We saw that checkout can fix many things, but depends on commit history, what about our staging area?

```
$ git add file1
$ git status
$ git reset

$ git reset HEAD file1
```

- Oh no, I forgot to add a file! (or made a typo, etc)

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

Remote

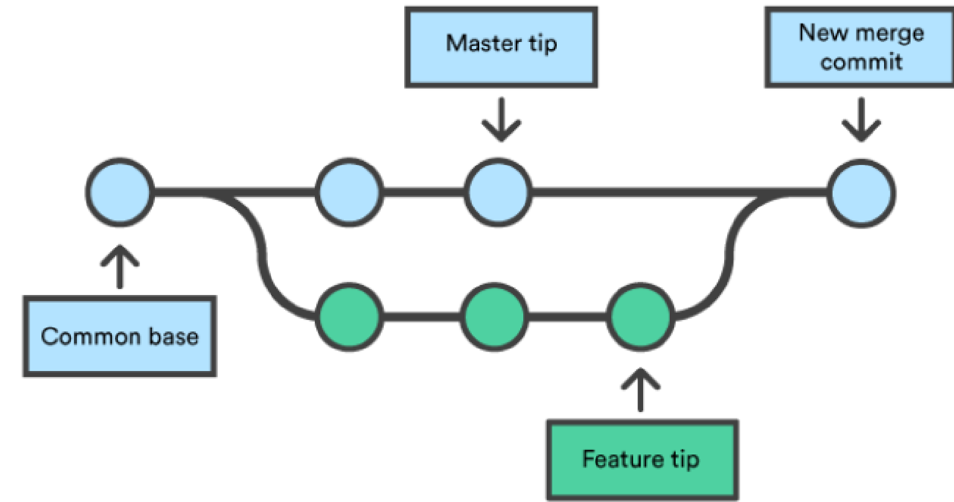
- In folder of test repo:

```
$ git remote -v
$ git pull
$ git push
```

- More on pulling in section Merging

Branching

Branching

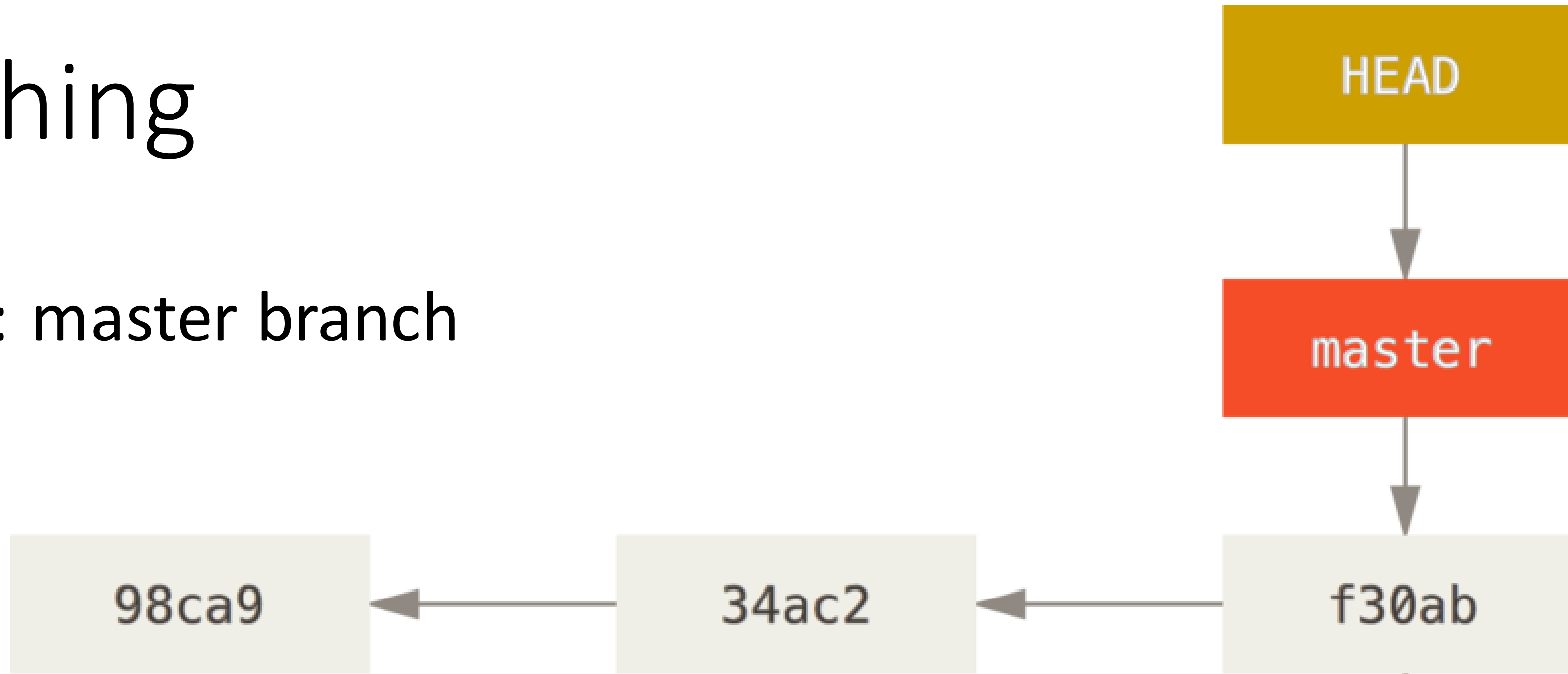


https://www.atlassian.com/git/tutorials/using-branches/git-merge

- What is branching?
  Branching is way to split up your history to work on different subjects in parallel.
- Branches emerge when collaborating because histories diverge between users.
- Why branch as a user?
  To separate different subjects/fixes/ideas in the history and to make changes modular (“bug fix 56” is separated from “functionality 12”)

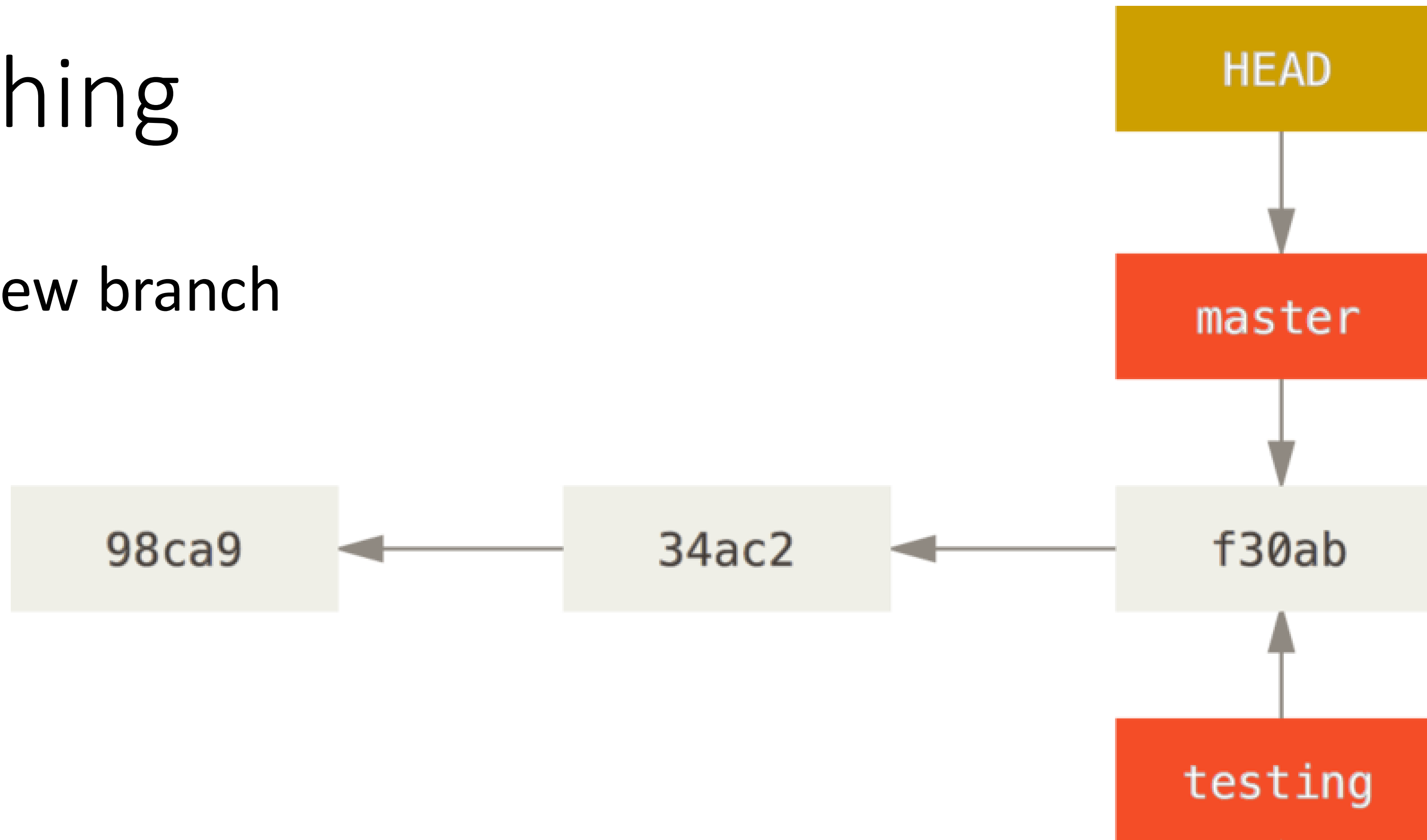
Branching

- Default: master branch



Branching

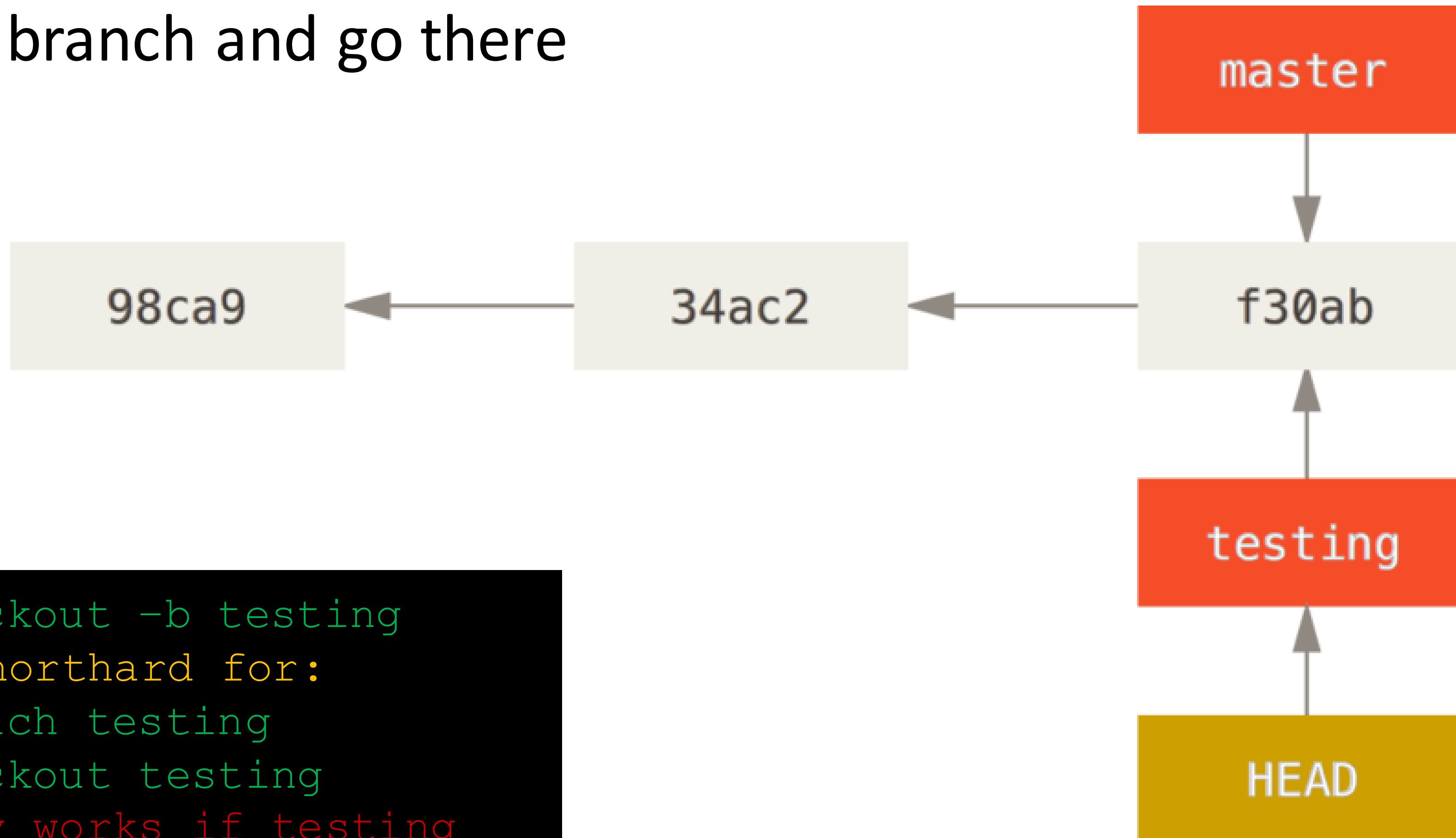
- Make new branch




```
$ git branch testing
```

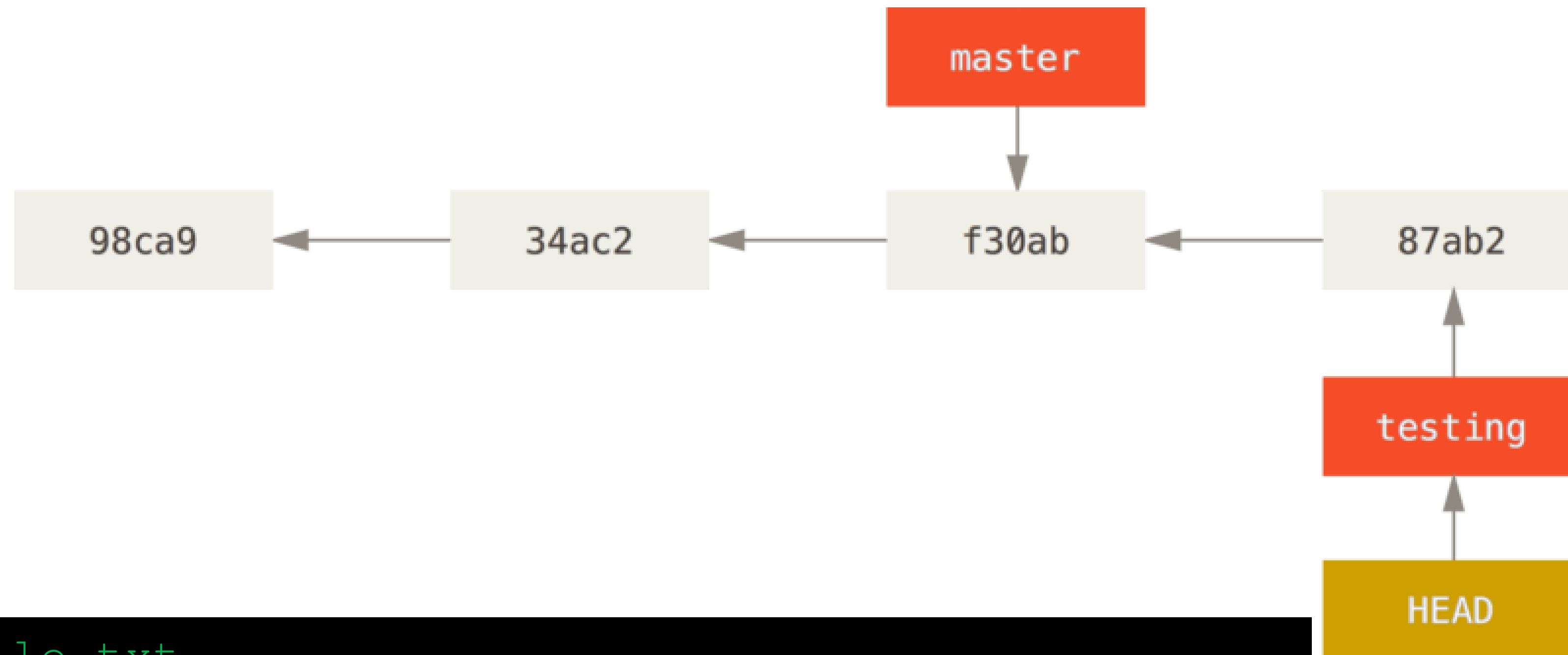
Branching

- Make a branch and go there




```
$ git checkout -b testing
          Shorthard for:
$ git branch testing
$ git checkout testing
Note: only works if testing
does not exist
```

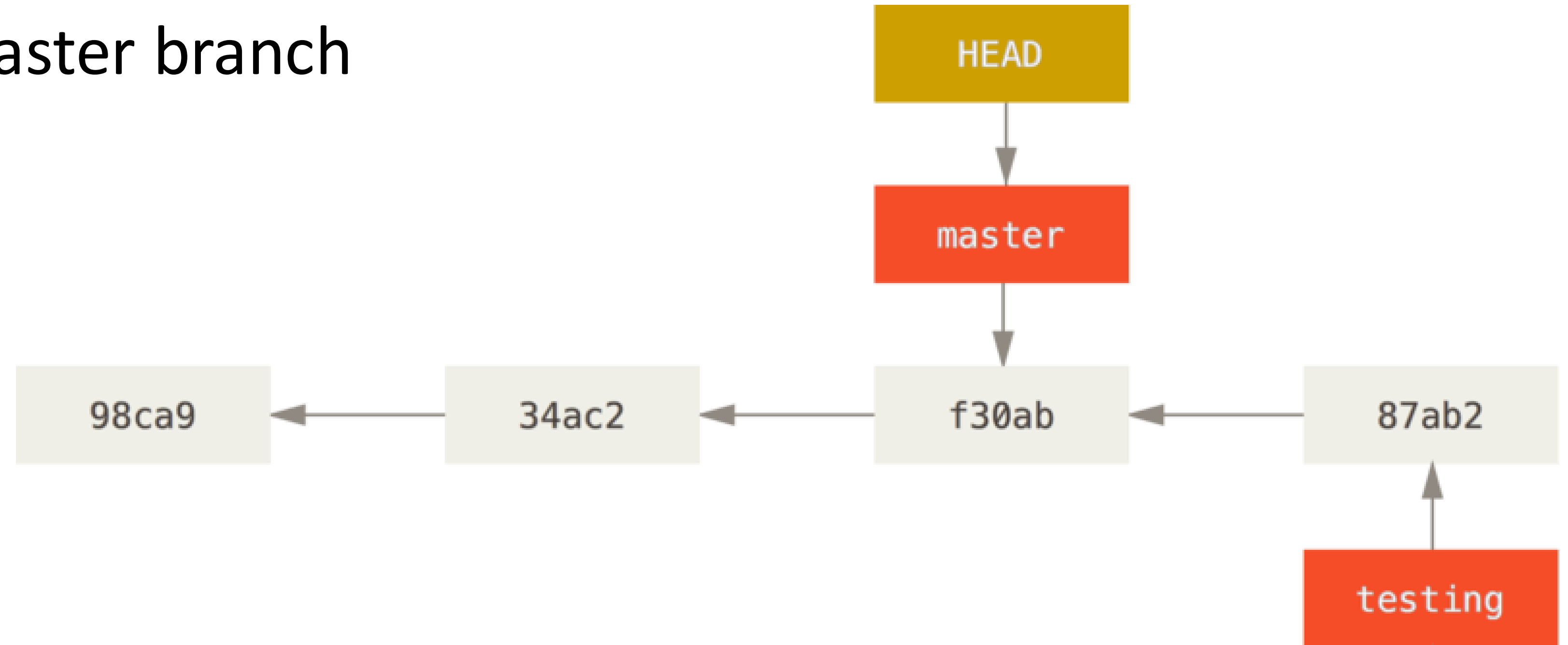
Branching




```
$ git add newfile.txt
$ git commit -m "Add newfile with awesome functionality"
Note: we have not pushed/pulled, this is only local!
```

Branching

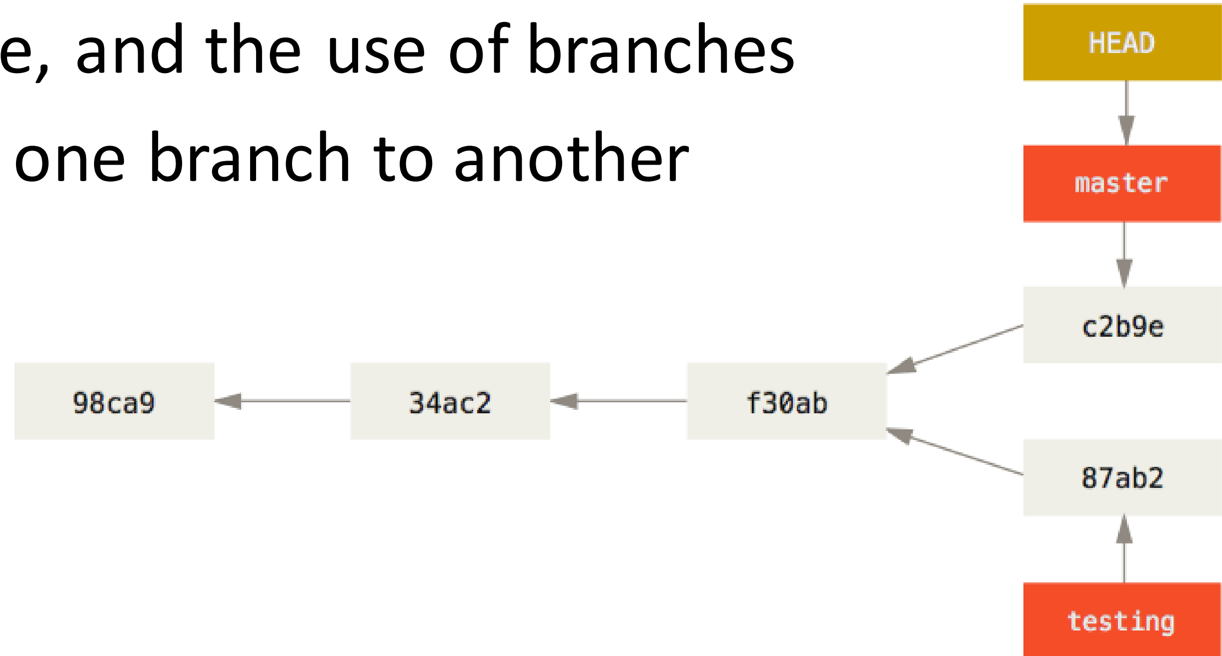
- Moving back to master branch




```
$ git checkout master
```

Branching

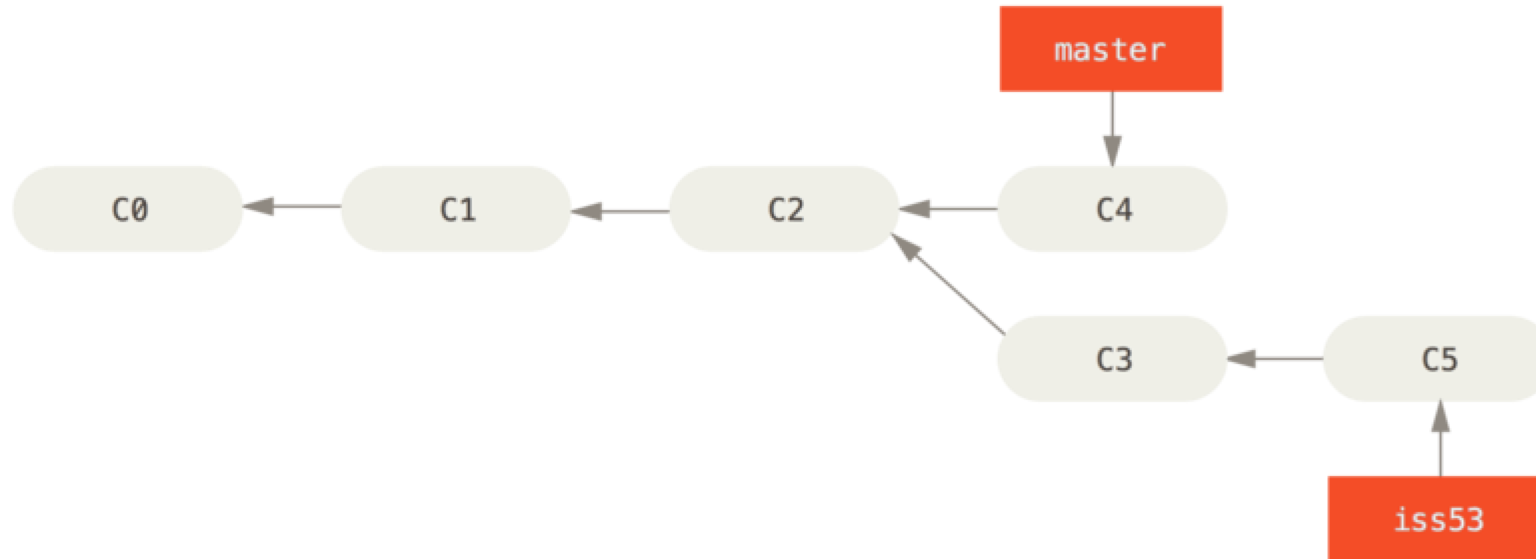
- Now you see histories diverge, and the use of branches
- We need to move code from one branch to another somehow…
- Next section: merging



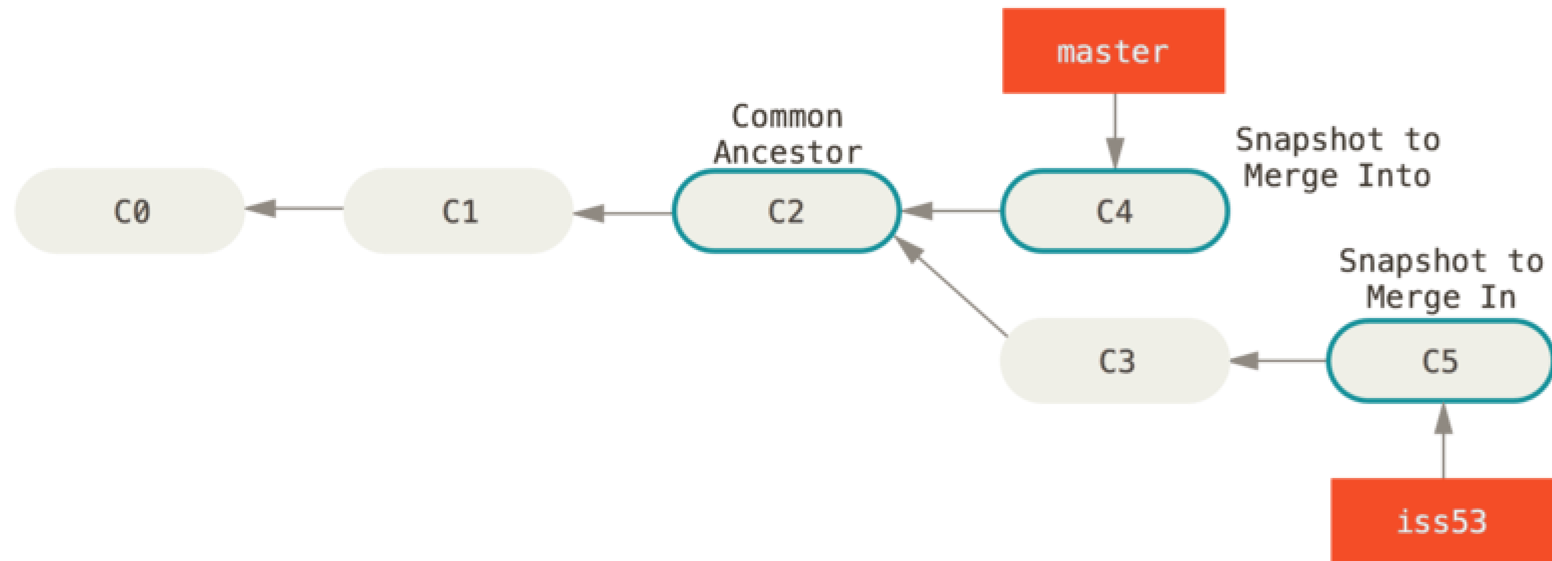

```
$ git add oldfile.txt
$ git commit -m "Change oldfile because of bug Y"
```

Merging

Merging a branch into master

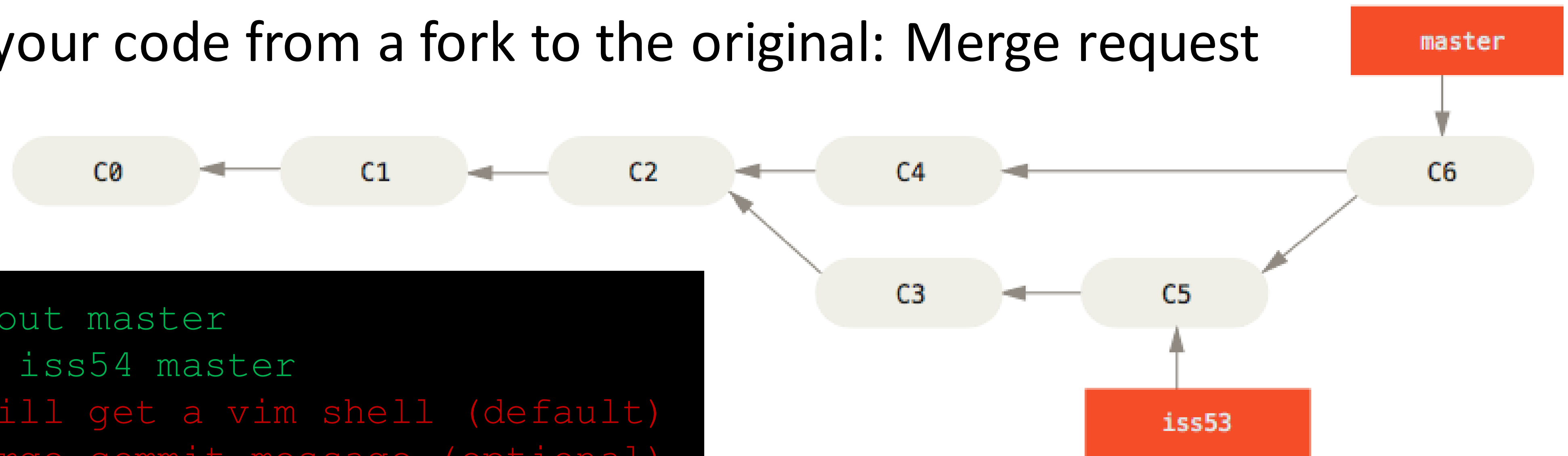


Merging a branch into master



Merging a branch into master

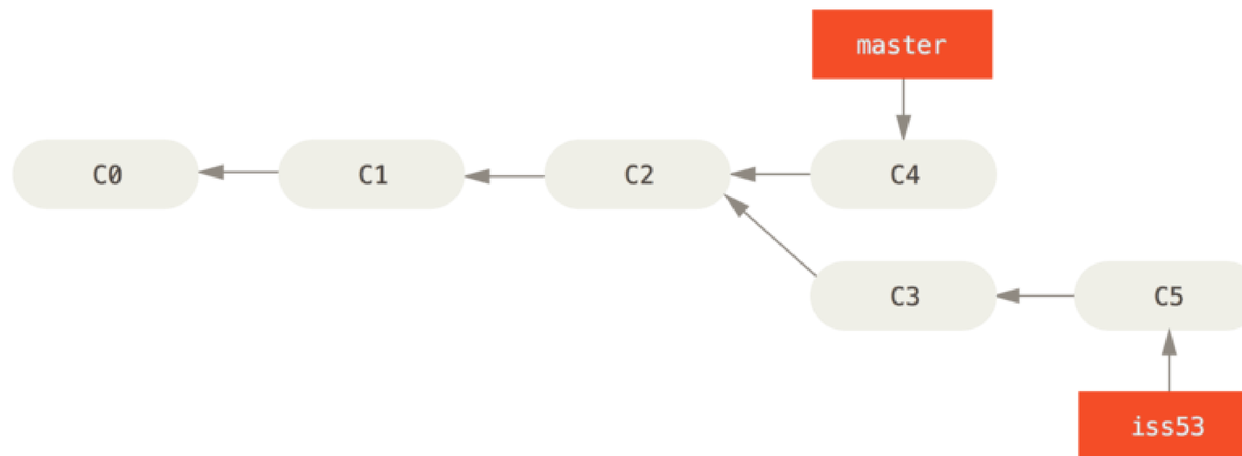
- Explicit merge: Merging a branch into master (or any other branch)
- Automatic merge: Merging by pulling from remote --> merge commit
- Pulling the original repo into your forked code
- Sending your code from a fork to the original: Merge request

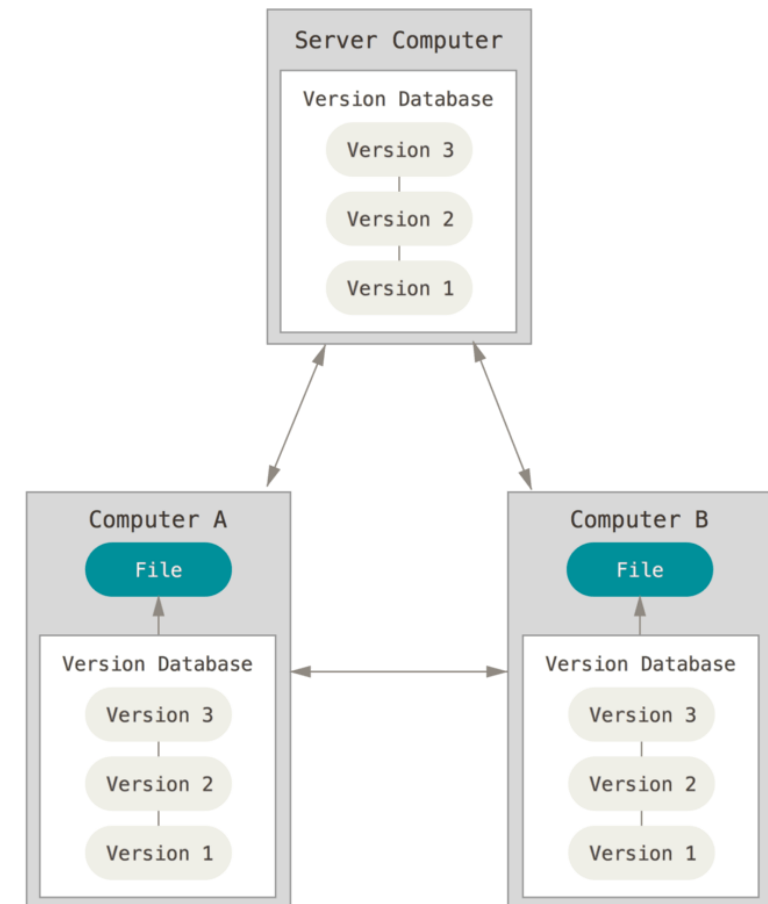



```
$ git checkout master
$ git merge iss54 master
Note: You will get a vim shell (default)
to write merge commit message (optional)
```

Merging remote changes: git pull

- Pulling is actually a combination of two commands:
  - Fetch: download the changes from remote
  - Merge: insert changes into your working directory
- Pull can be confusing: it’s better to use fetch and merge separately
  (Although the author is very hypocritical about this)





Conflicts

- When you pull but someone merged into the file you're working on
- When the diffs in your file touch the same code and there is not one clear solution

- Who is right? Git can not decide: CONFLICT

```
1  <<<<<<< HEAD
2
3  Here is the original change.
4  =======
5  Here is the modified change.
6  >>>>>>> 58326c301d09b58f3ac23d616e73f7b478424cc5
7
```

https://stackoverflow.com/questions/34814837/git-marks-and-comments-like-head-and

Don't do this:

https://xkcd.com/1597/

The cool stuff

Online dashboard

http://git.km3net.de

Stashing

- You made changes but are not ready to commit
- You want to checkout a certain commit: git wants you to commit your changes
- Save the changes in a stash that can be later be reapplied to your working directory

```
$ git stash

$ git pop
```

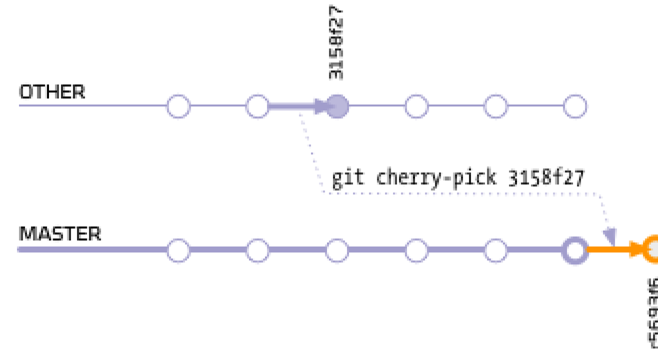
Rebase

- Rebase can change the history of your repository
  - To change ordering of commits (cluster certain changes)
  - To move commits to other branches (cherry-picking, next slide)
  - To merge commits into one larger commit
- You can get an interactive shell to make changes
- Changes are applied to your repository
- WARNING: Since you are changing history, do not use this on pushed changes! If someone has pulled and continues working on it, this will cause conflicts and even lost work

```
$ git rebase -i HEAD~3
```

More commands

- Cherry pick:



- Force: overwrite whatever the history is in remote [DANGER ZONE]

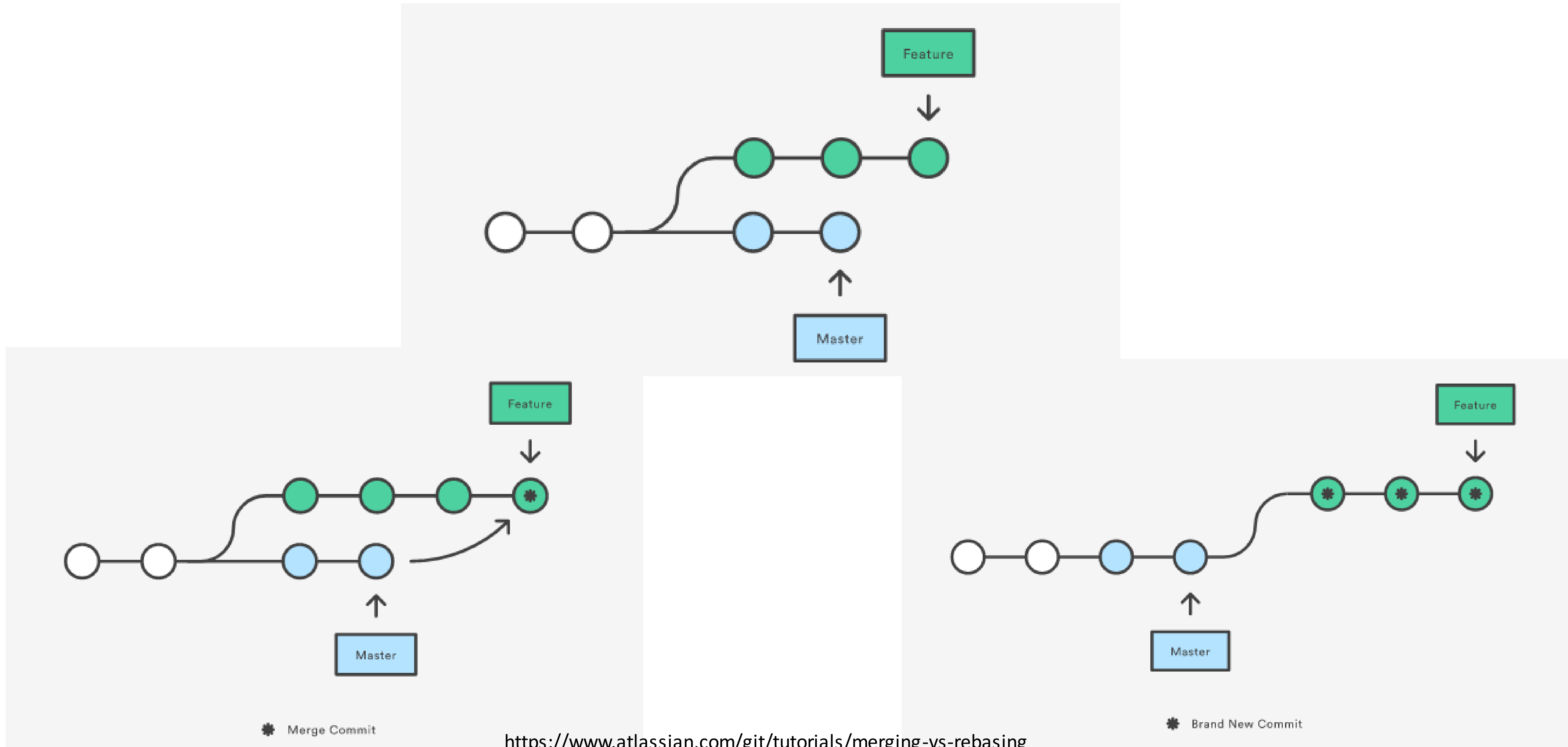
```
$ git push origin master --force
```

- Blame: see who wrote which line

```
$ git blame file1
```

https://www.ralfebert.de/git/cherry-pick/

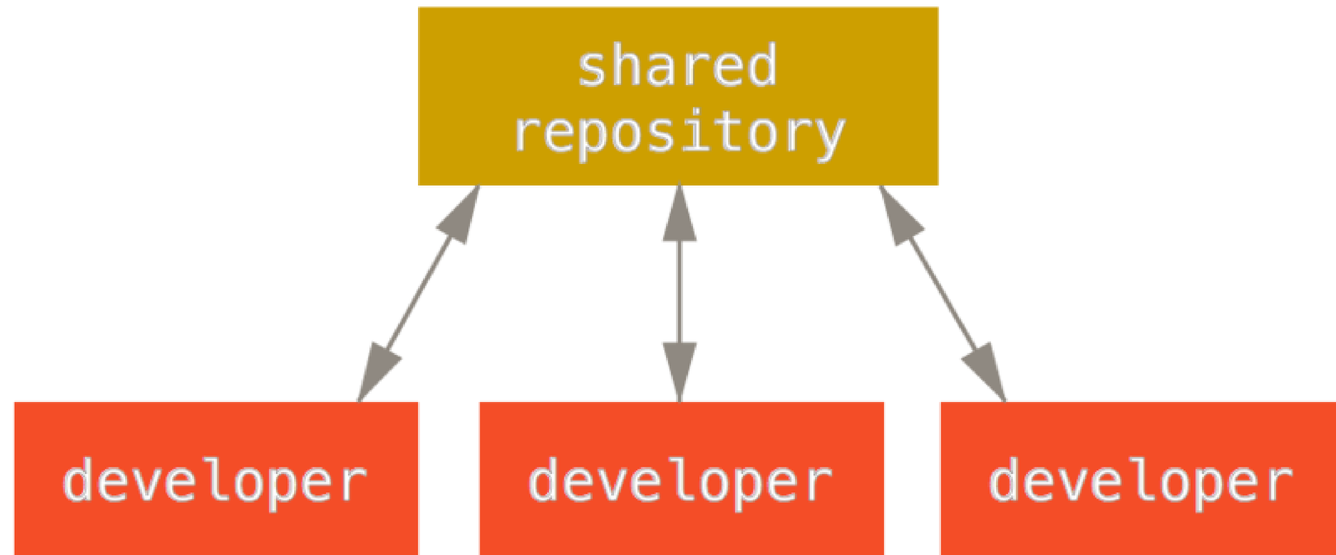
Backup: different way of keeping history



https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Backup: development models

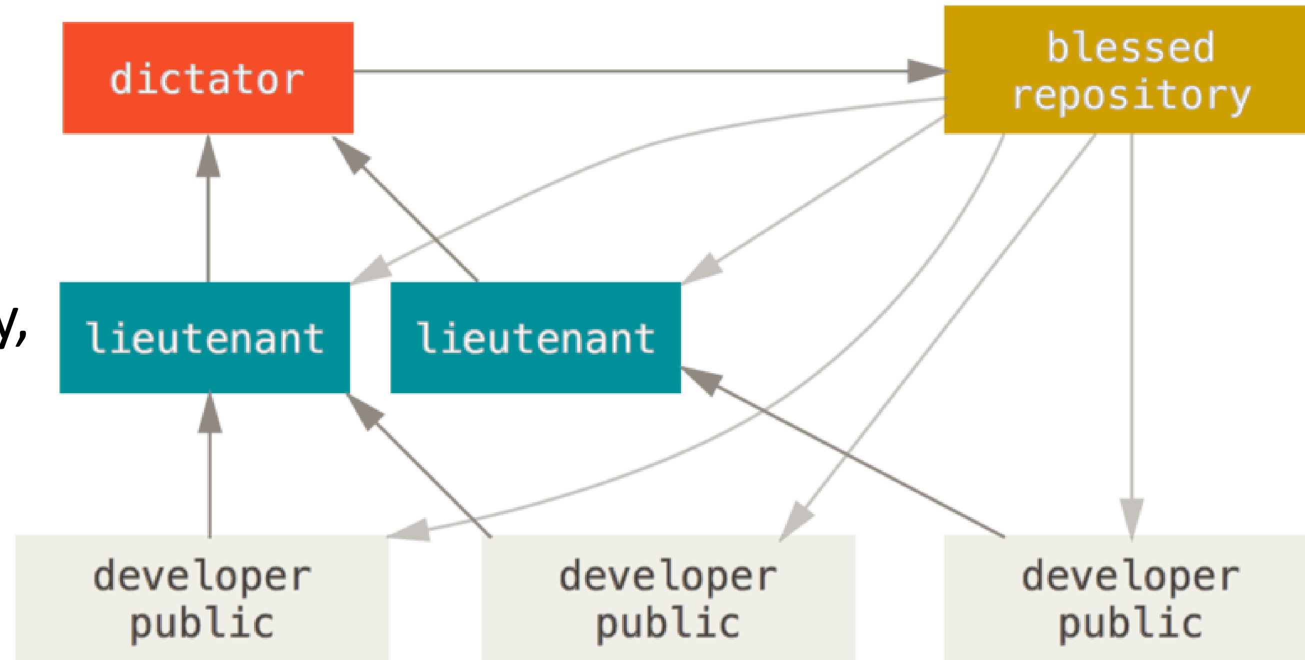
- Default git model:
  - Everyone can pull/push
  - Good (enough) for small projects/not open source
  - Can become chaotic because of bad git practices (bad messages, generally shit commits)



Backup: development models

- Used in large projects: linux kernel, pandas, professional software development
  - High quality commits, clear history, no bad/useless commits

  - Requires good commit messages
  - Takes long to merge changes
  - Owners of repository spend most of their time reading code



Backup: development models

- Gitlab model:
  - Work privately
  - Push to public repository
  - Send merge request (pull request on github) to repo manager