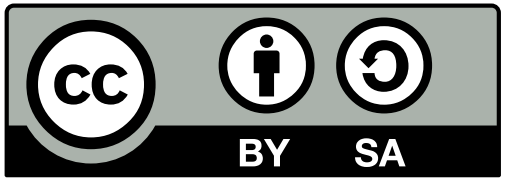
IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Introduction to Git

ILSD – Git
2022-2023



You are **free**:

- to **use**, to **copy**, to **distribute** and to **transmit** this creation to the public;
- to **adapt** this work.

Under the following terms:

- **Attribution** (**BY**): you must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **ShareAlike** (**SA**): if you modify, transform, alter, adapt or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

[1] `http://creativecommons.org/licenses/by-sa/2.0/`.

Important notes

- If you do not understand something, please ask your questions. *We cannot answer the questions you do not ask. . .*
- If you disagree with us, please say it (we follow Crocker's rules[2])
- People don't learn computer science by only reading few academic slides: practicing is fundamental

[2] http://sl4.org/crocker.html

- Most of the slides come from the VCS lecture in the DCL TAF

$\Rightarrow$ Student from the DCL TAF should be familiar with Git and should not learn anything with this lecture

- This is not an exhaustive Git lecture (we do not have the time)

$\Rightarrow$ this lecture is NOT sufficient to be fluent with Git

- Consider this lecture as a starting point (or a refreshing of your brain for ex DCL students)
- Understanding and using VCS is mandatory in software development

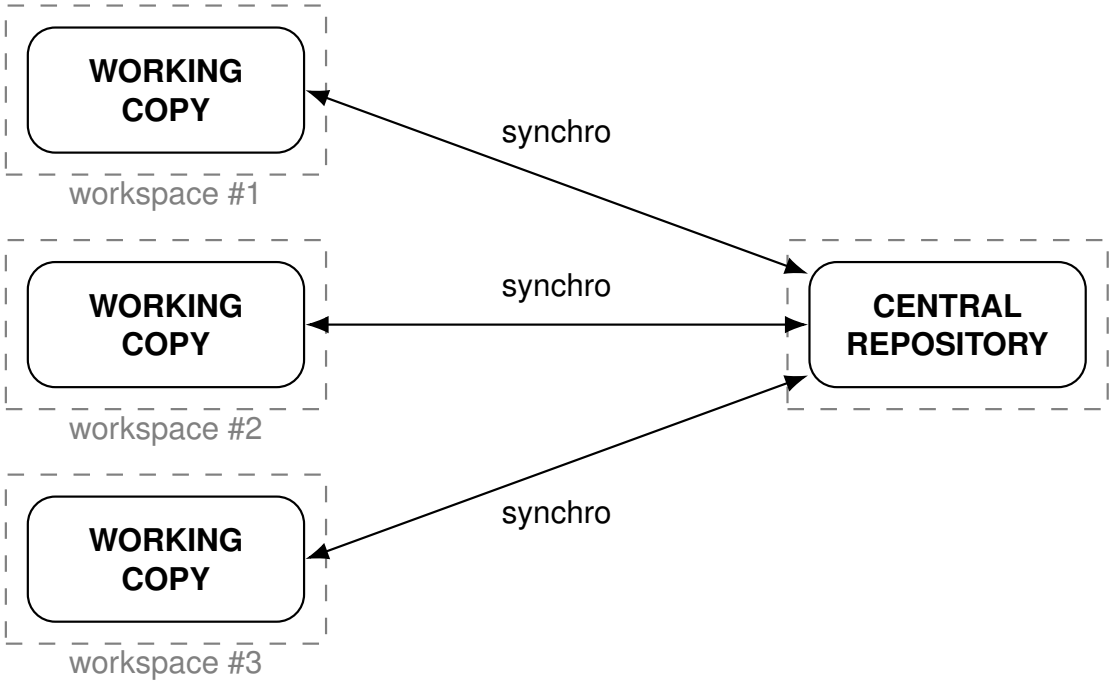
- *Aaaaah! Three months of work lost!*
- *Oops. . . Was this file really important?*
- *Great, everyone has finished! Who integrates all the parts?*
- *Why did I wrote this piece of code?*
- *Great functionality, but I think the last week version was better. Uh. . . which one?*
- *I cannot find the version we have made 6 years ago for BigCustomer Inc., I need it immediately for a new contract!*
- *I have already done this bugfix. . . on my laptop I left at home.*
- *It doesn't work anymore! Who messed up my code?*

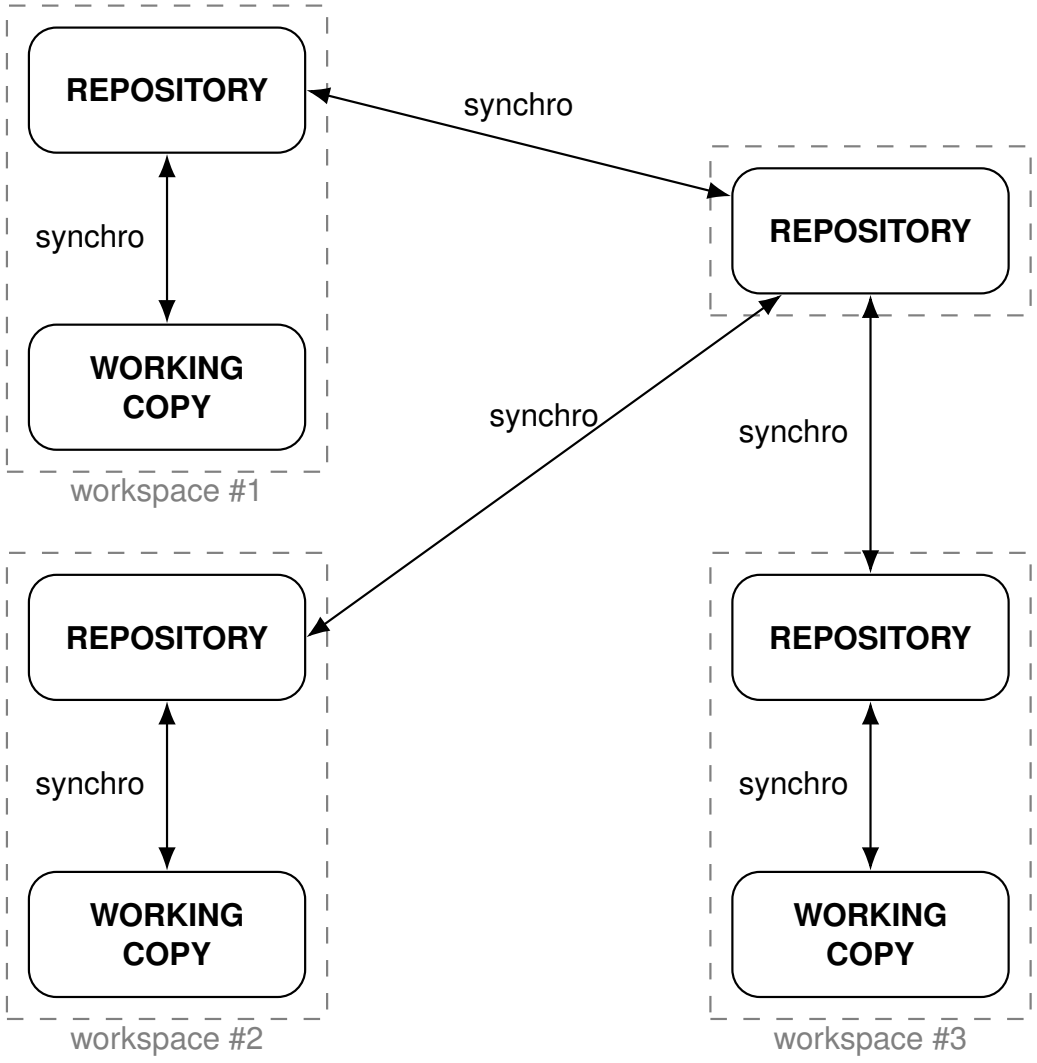
- Software traceability: tracking and documenting changes, retrieving former versions
- Flexibility: feature trials, quick rollbacks
- Parallelism and team work: multi-sites, multi-computers, multi-developers and multi-activities
- Safety: “backup”[3] with history

  $\Rightarrow$ One needs tools to solve these problems

[3] VCS are not (space) efficient backup systems

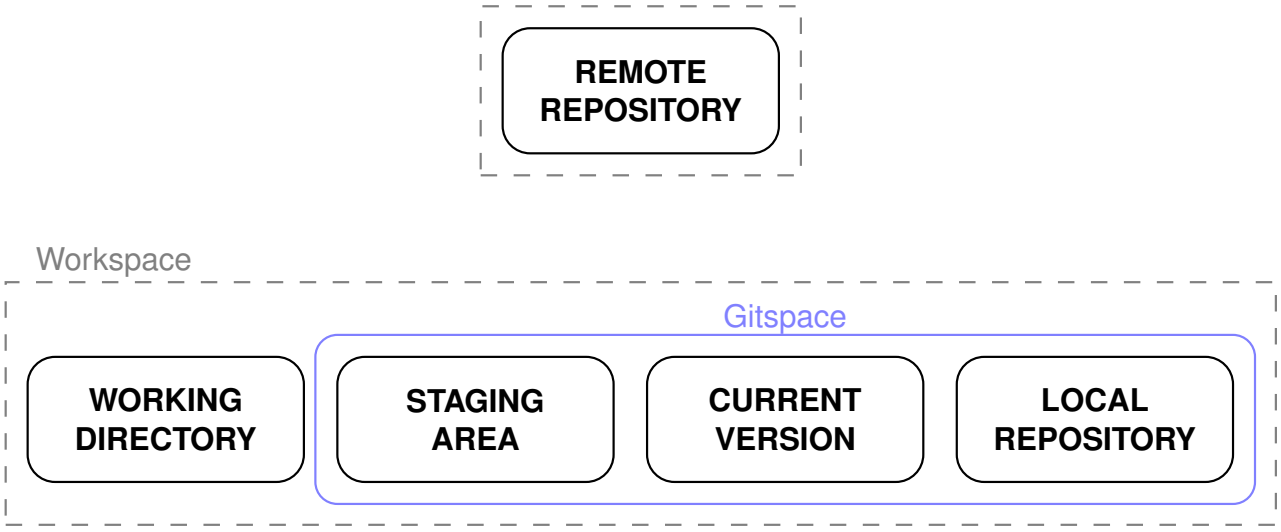
- Used for
  - storing files
  - keeping track of changes on those tracked files
  - sharing
- Each collaborator works on a local copy
- Synchronization with one (or several) remote server(s)
- 2 families of VCS
  - centralised (Subversion, CVS, . . .)
  - distributed (Git, Mercurial, Darcs, . . .)


WORKING COPY
workspace #1
synchro
WORKING COPY
workspace #2
synchro
CENTRAL REPOSITORY
WORKING COPY
workspace #3
synchro


REPOSITORY
synchro
synchro
WORKING COPY
workspace #1
REPOSITORY
synchro
synchro
REPOSITORY
synchro
WORKING COPY
workspace #2
REPOSITORY
synchro
WORKING COPY
workspace #3

Why Git?

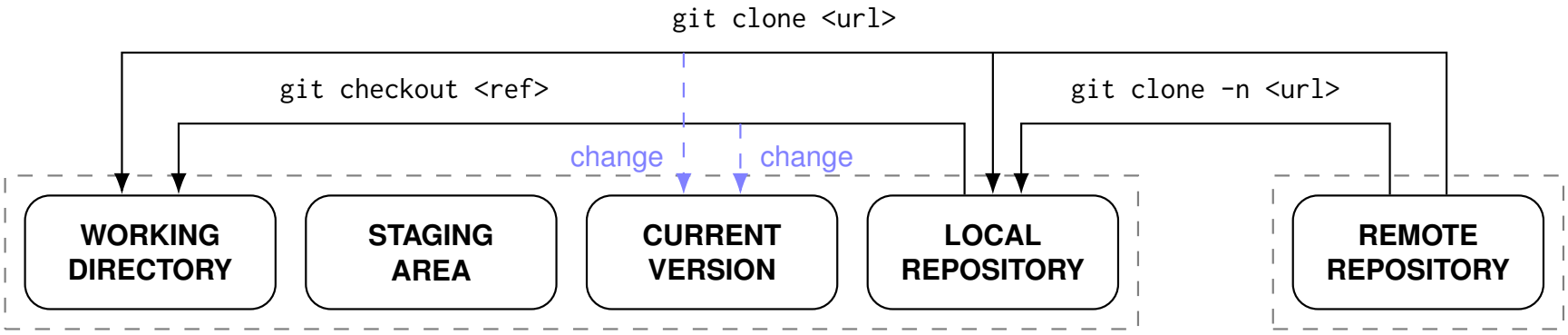
- very popular
- many platforms provide services built on Git (Bitbucket, Gitlab, GitHub)
- a bit less intuitive than other VCS for beginners, therefore if you are able to use Git, you will be able to use other VCS
- . . . and because we had to choose a tool



- working directory = files where changes are made
- staging area = current selected changes
- current version = current reference version
- (remote/local) repository = a database of changes

- Practical use cases in order to learn few commands
  - setting up a new repository (`init`, `remote url`)
  - retrieving a repository (`clone`)
  - making changes in the working repository (`status`)
  - updating the remote environment (`add`, `commit`, `push`)
  - checking differences after changes (`diff`)
  - updating dev environment (`fetch`, `pull`)
  - diverging/branching (`branch`, `merge`, `checkout`)
  - ...
- Non-exhaustive use cases
- Workflows

Let's have a look at the terminal!
(I'll probably forget the slides)



`$> git clone -n <url>`

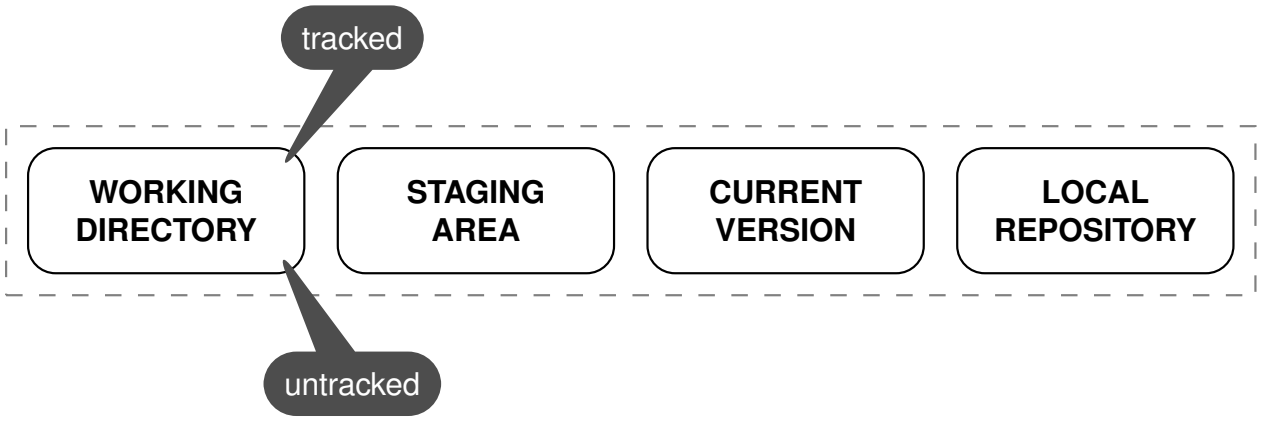
only creates the `.git` directory

`$> git checkout <ref>`

retrieves files from local repository into the working directory

`$> git clone <url>`

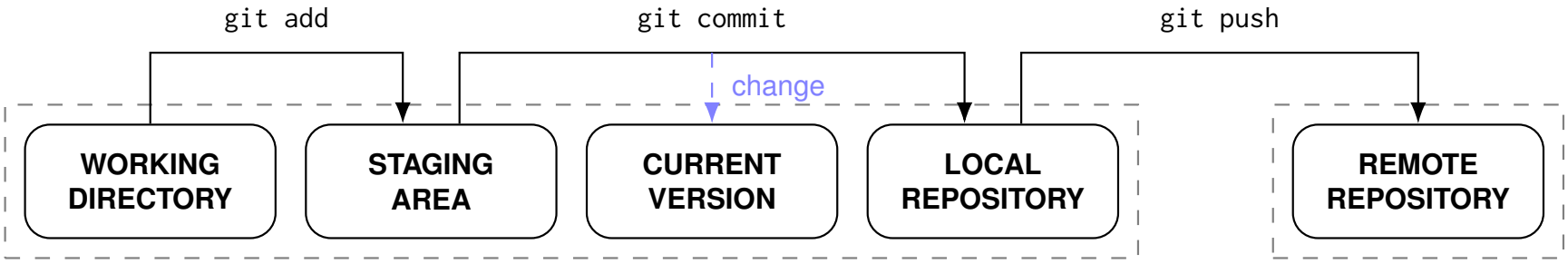
creates the `.git` directory and retrieves files into the working directory; `clone = clone -n + checkout`



Checking the current state

```
$> git status
```

```
On branch master
Your branch is up to date with 'origin/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:   file1
        ...
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file4
        ...
no changes added to commit (use "git add" and/or "git commit -a")
```



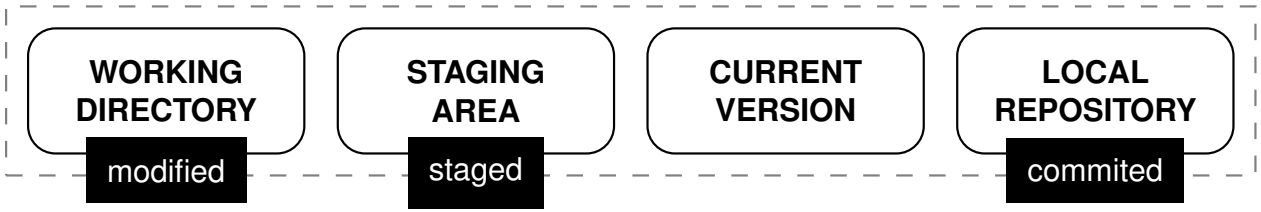
Example

```
$> git add file1 file2 file3 ...
```
add in the index of the staging area

```
$> git commit -m “add my super new feature”
```

```
...
```

```
$> git push
```
push into the remote repository

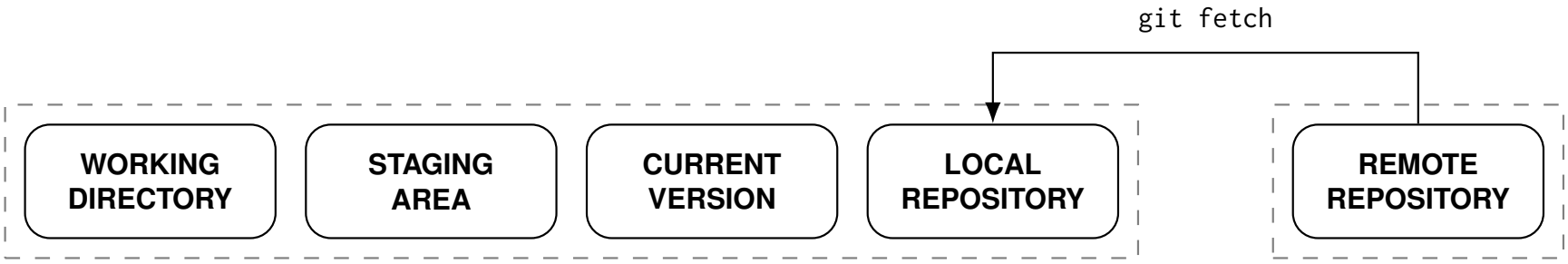


Diff commands

```
$> git diff
$> git diff --staged
```

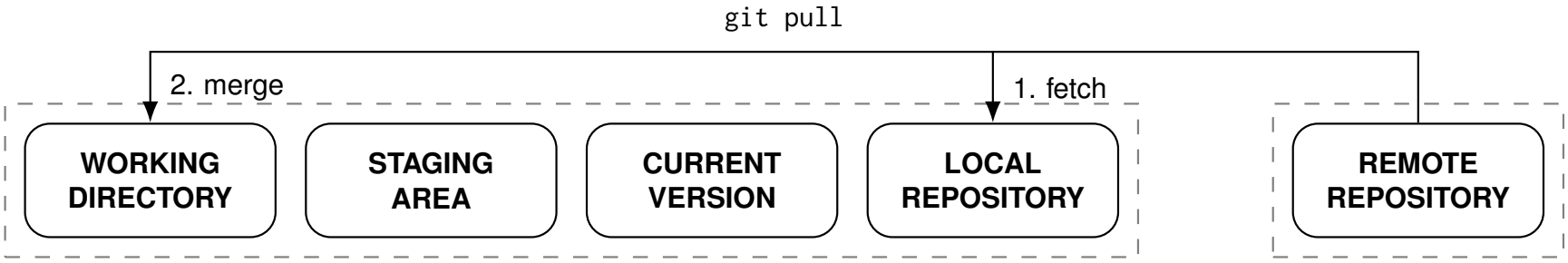
`$> man git-diff` will help you

Updating dev environment (`fetch`)




```
$> git fetch
```

- retrieves updates from the remote repository
- is safe
  - does not affect working directory $\Rightarrow$ cannot lose uncommited changes,
  - no automated merge

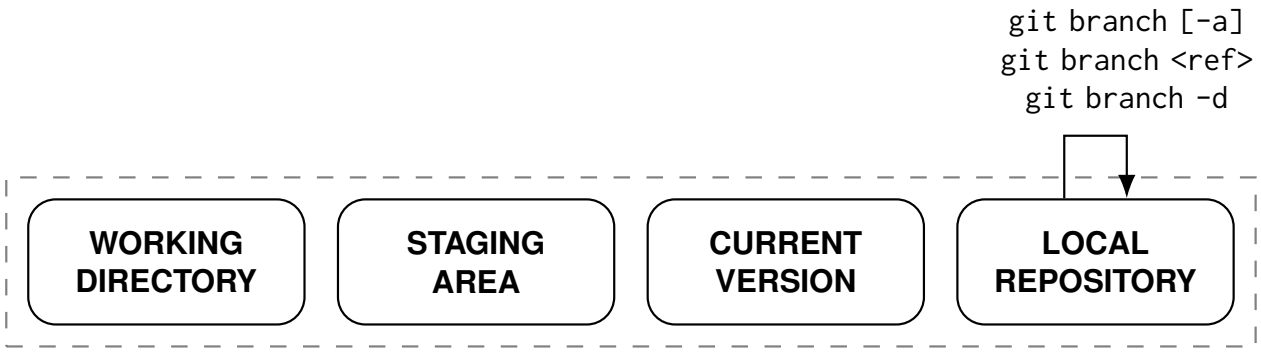


`$> git pull`

retrieves updates from the remote repository and merge them with the working directory

- `git merge`: to be seen few slides later

- a branch = a reference to a version
  - can be seen as a “local checkpoint” (another says like a bookmark)
- branching
  - creating a named reference to a version
  - the common way to work without messing with the main line




```
$> git branch
```

list local branches

```
$> git branch -a
```

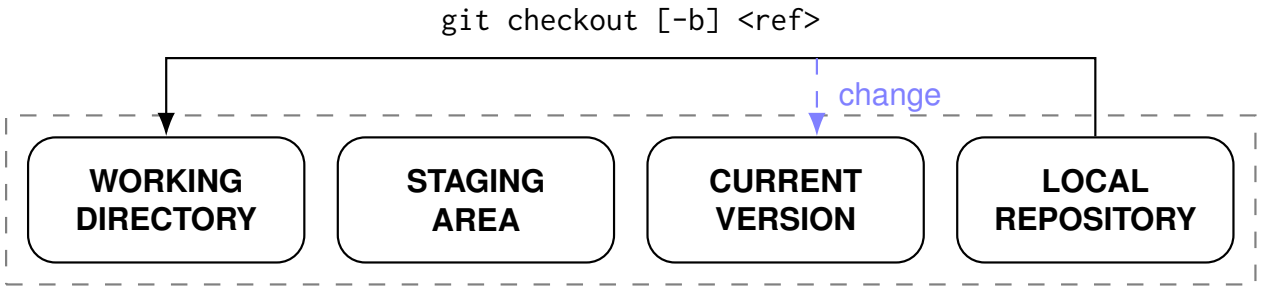
list all (local and remote) branches

```
$> git branch <ref>
```

creates a named branch from the current branch

```
$> git branch -d <ref>
```

deletes a named branch



`$> git checkout <ref>`

changes the current branch

`$> git checkout -b <ref>`

creates a branch from the current branch and changes to it (= `git branch` + `git checkout`)

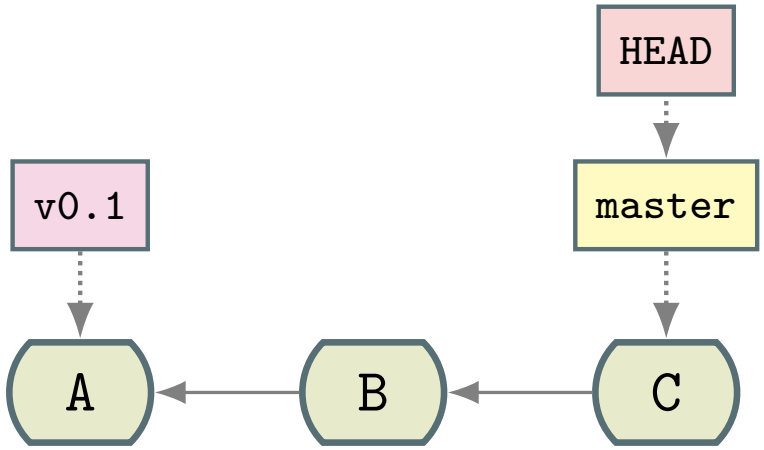
- Starting point: 2 branches (`master` + `newawesomefeature`), `HEAD` points to `master`

```
$> git merge newawesomefeature
```

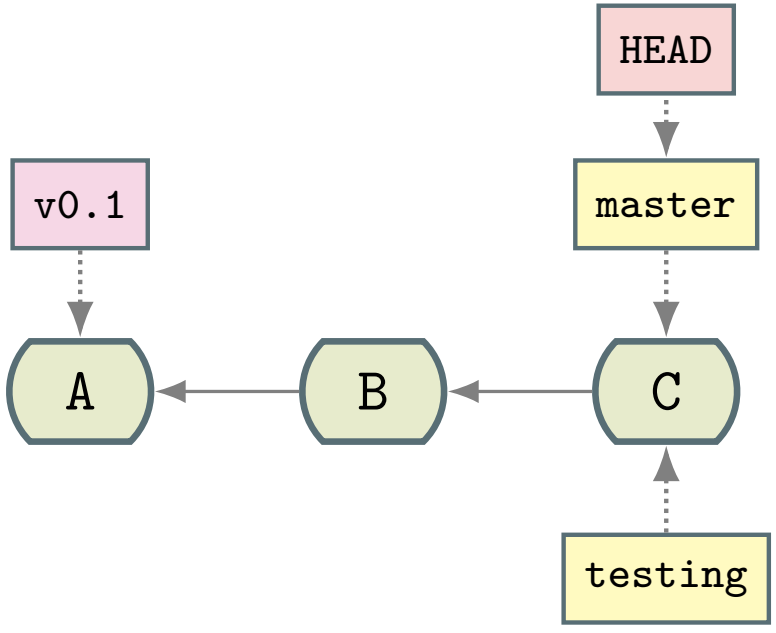
integrate changes from `newawesomefeature` branch into `master`

- Two situations
  - no conflict: changes from `newawesomefeature` are integrated in the main (local) line, time to push...
  - conflicts: resolution needed in order to be able to push
- Conflict resolution:

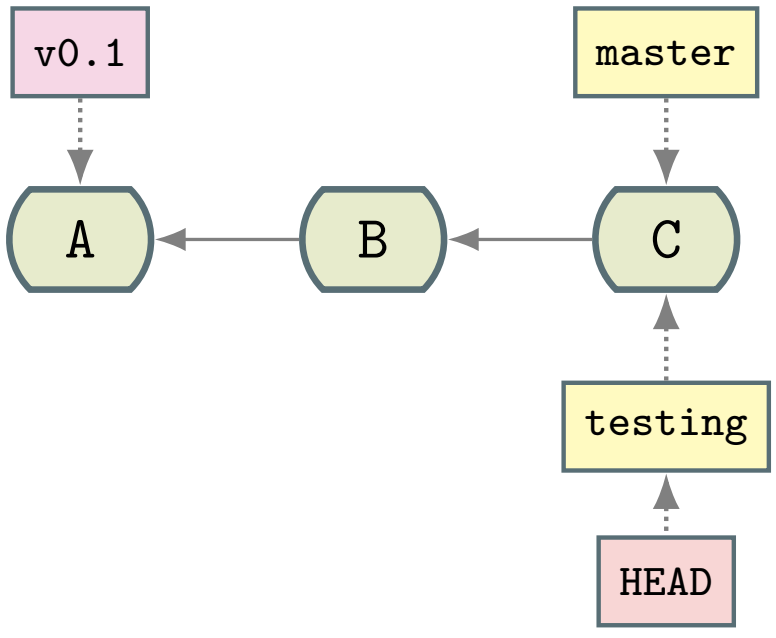
1. fix the conflicts (edit the files, keep/remove stuff)
2. add the changes
3. commit



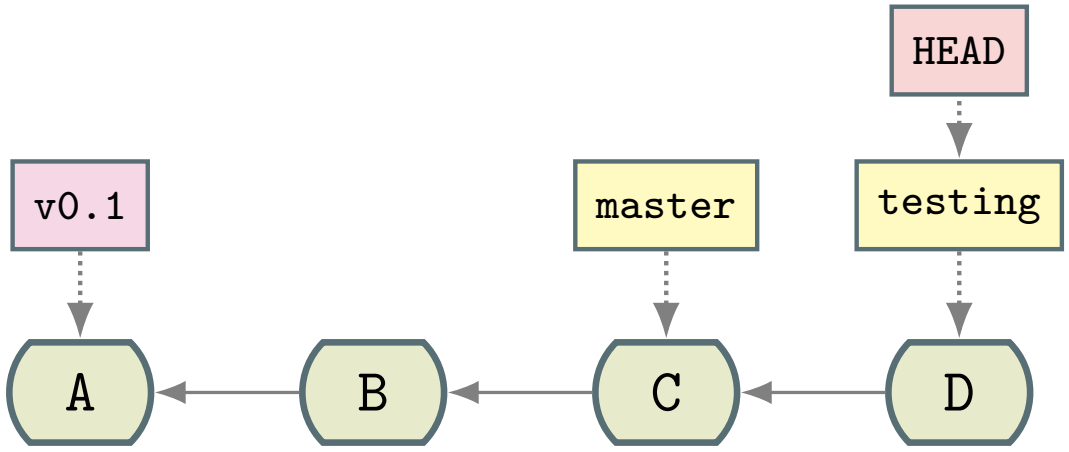
Initial situation



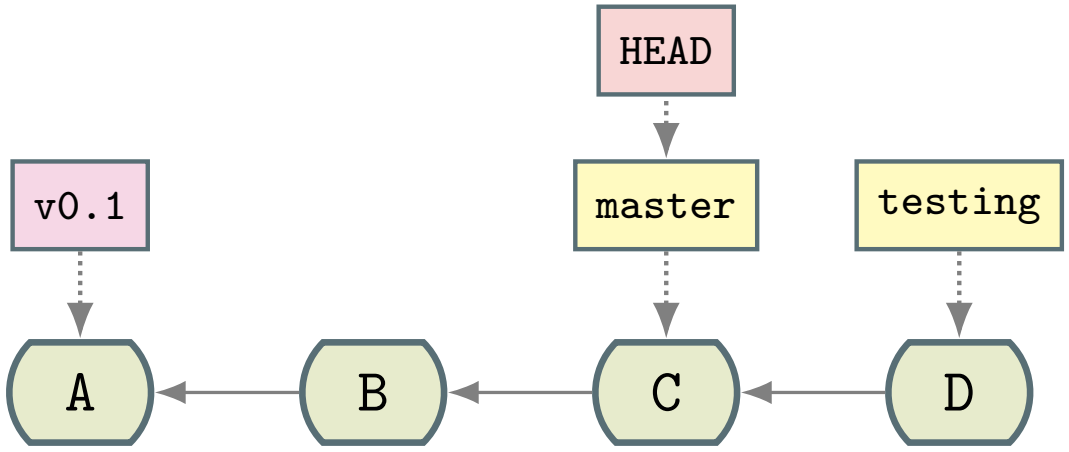

```
$> git branch testing
```



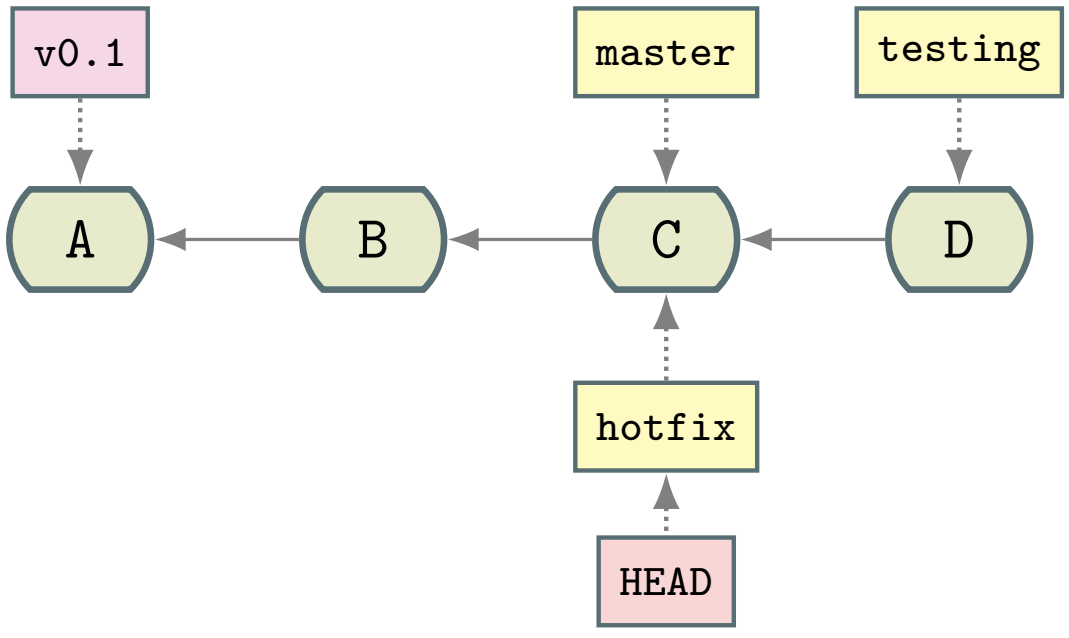

```
$> git checkout testing
```



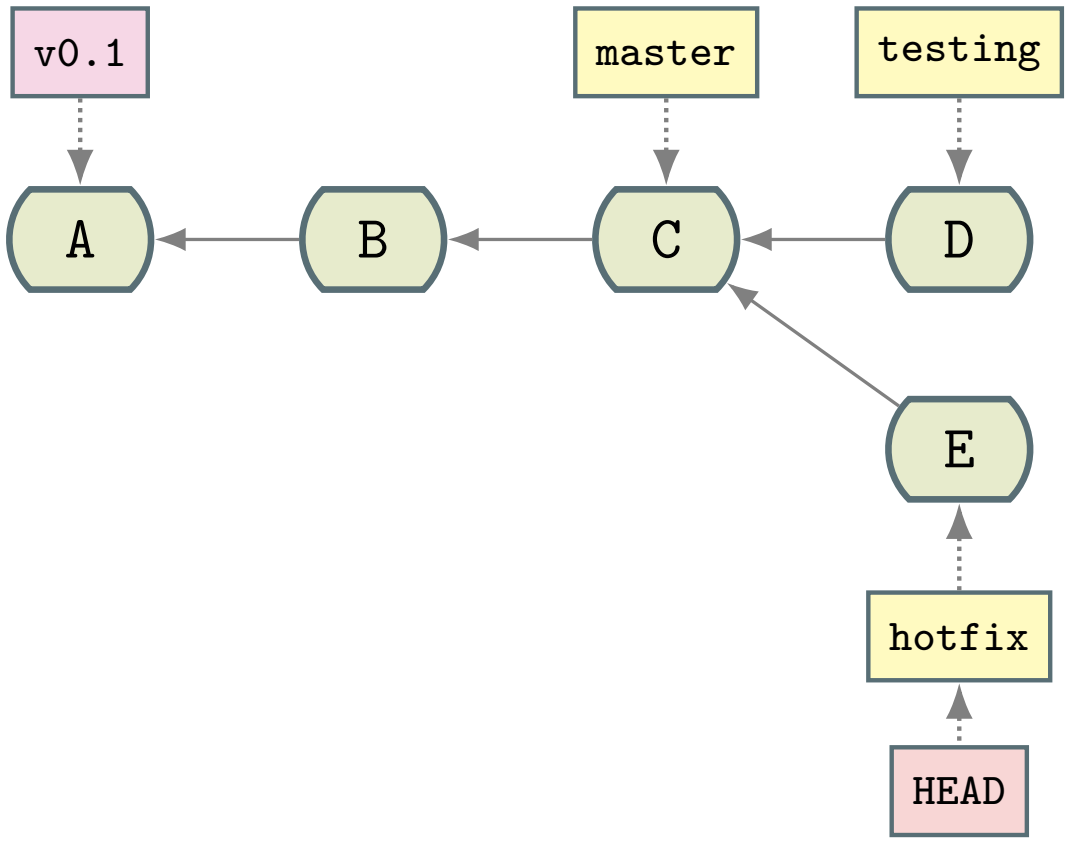
One commit later



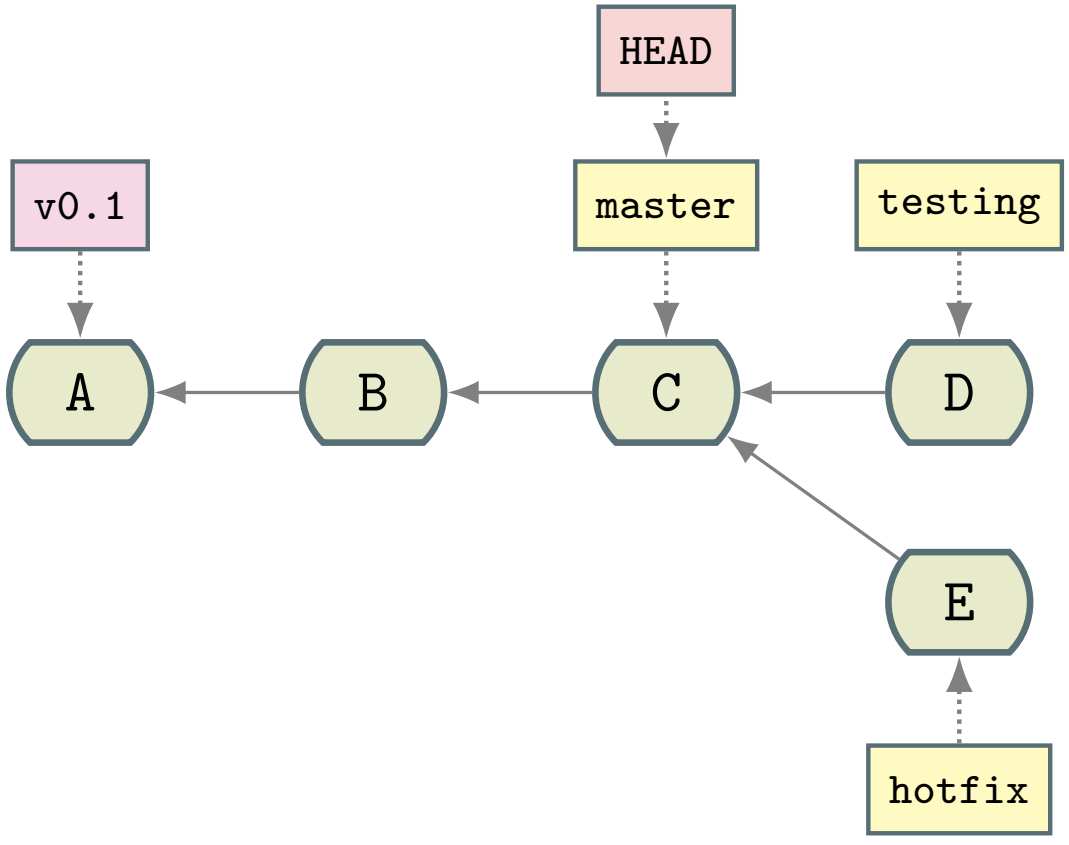

```
$> git checkout master
```



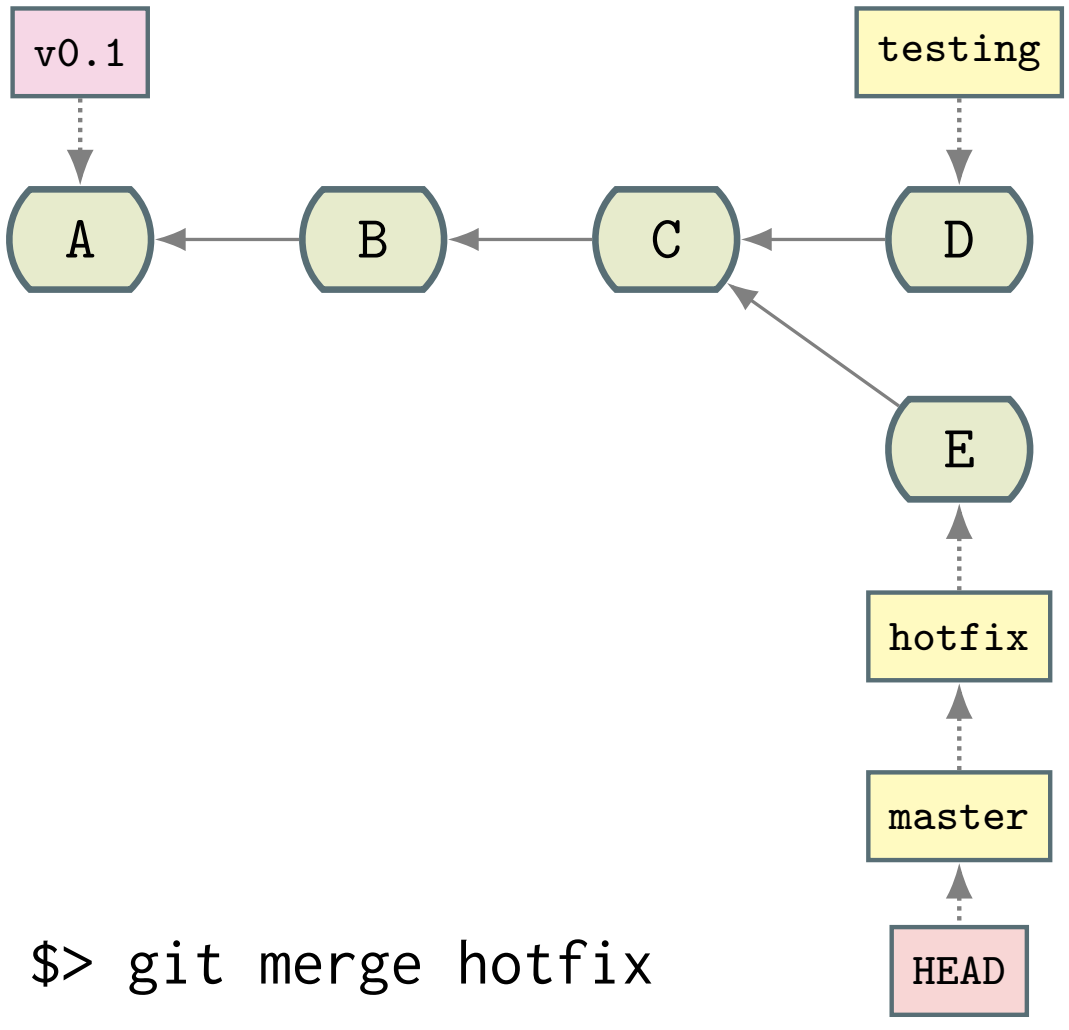

```
$> git checkout -b hotfix
```



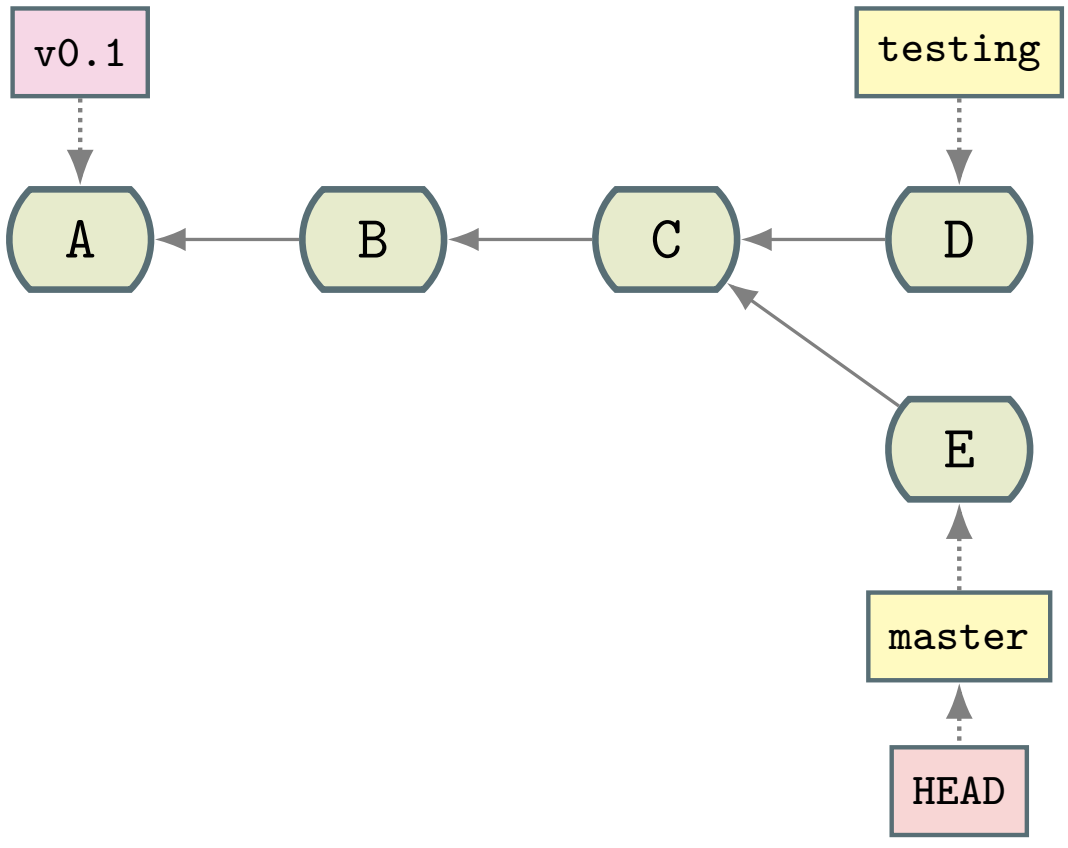
One commit later



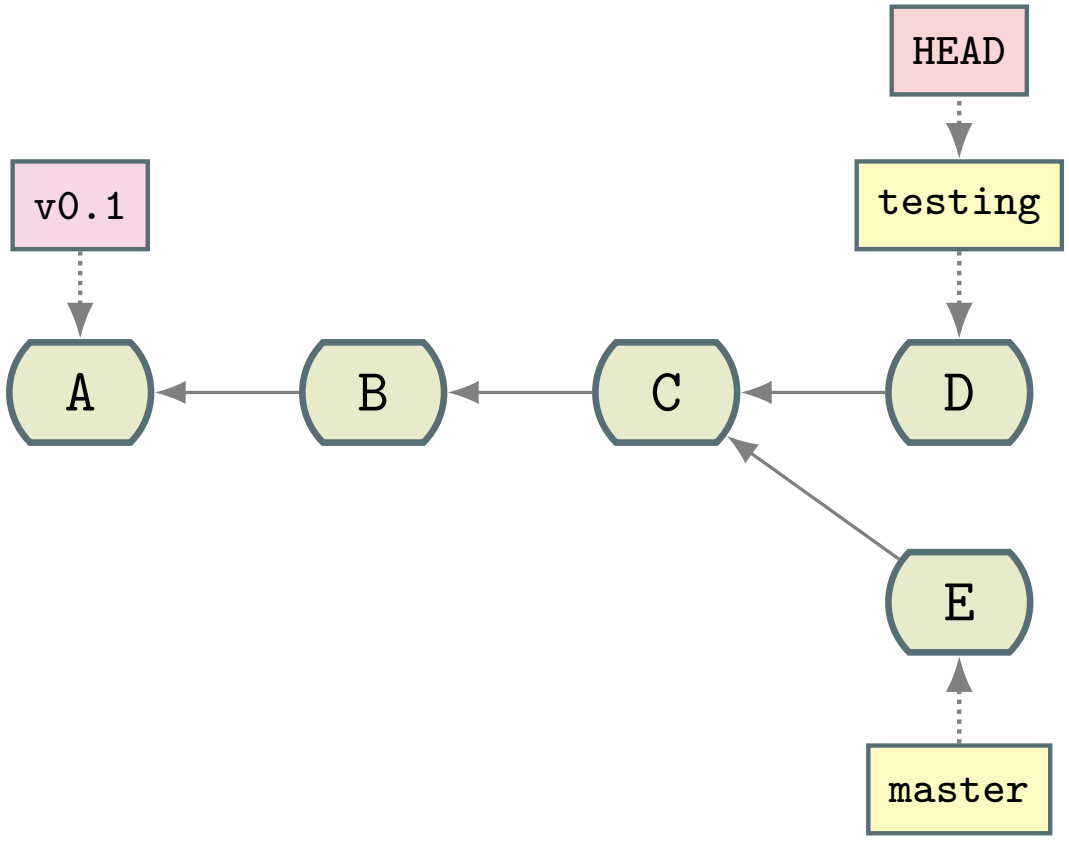

```
$> git checkout master
```



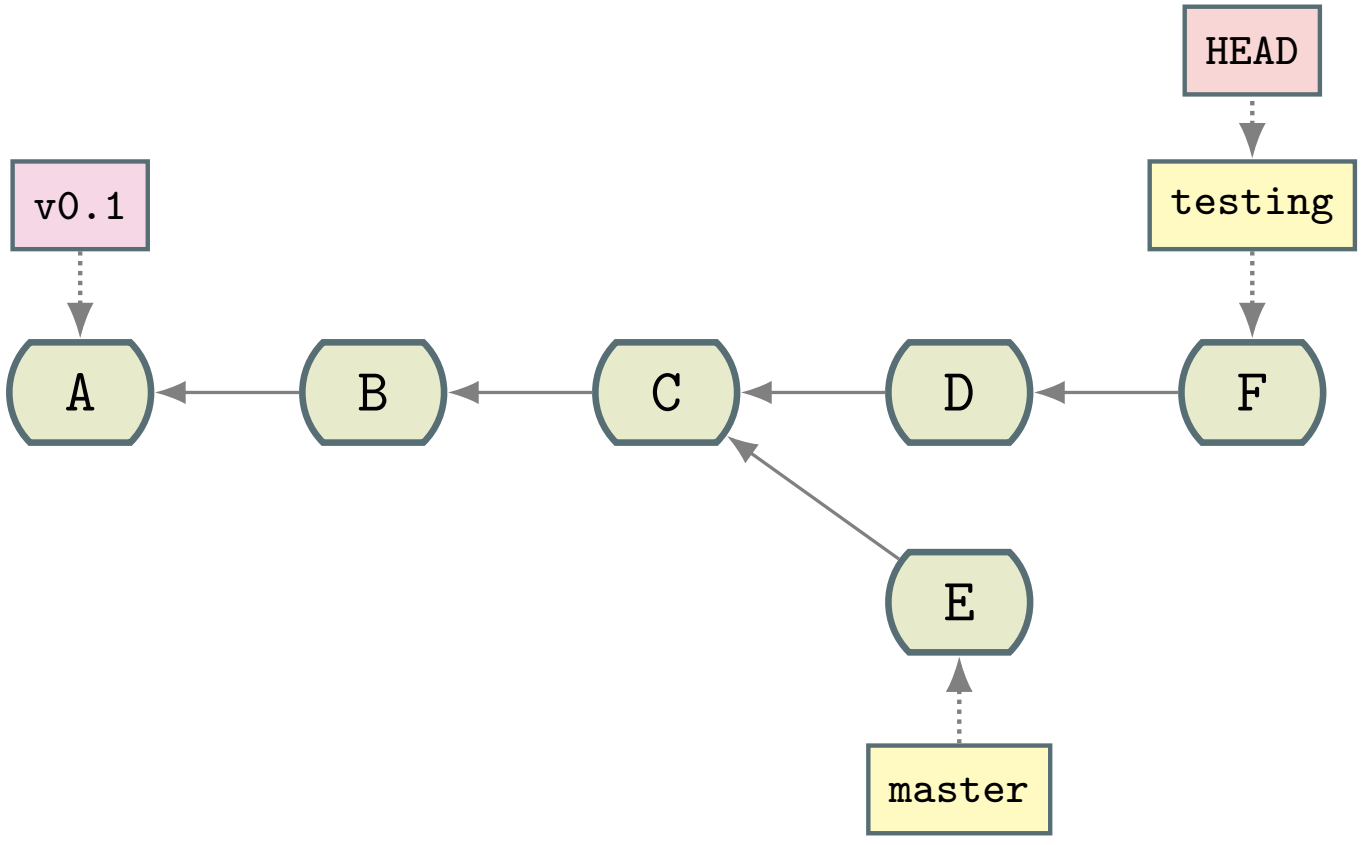

```
$> git merge hotfix
```



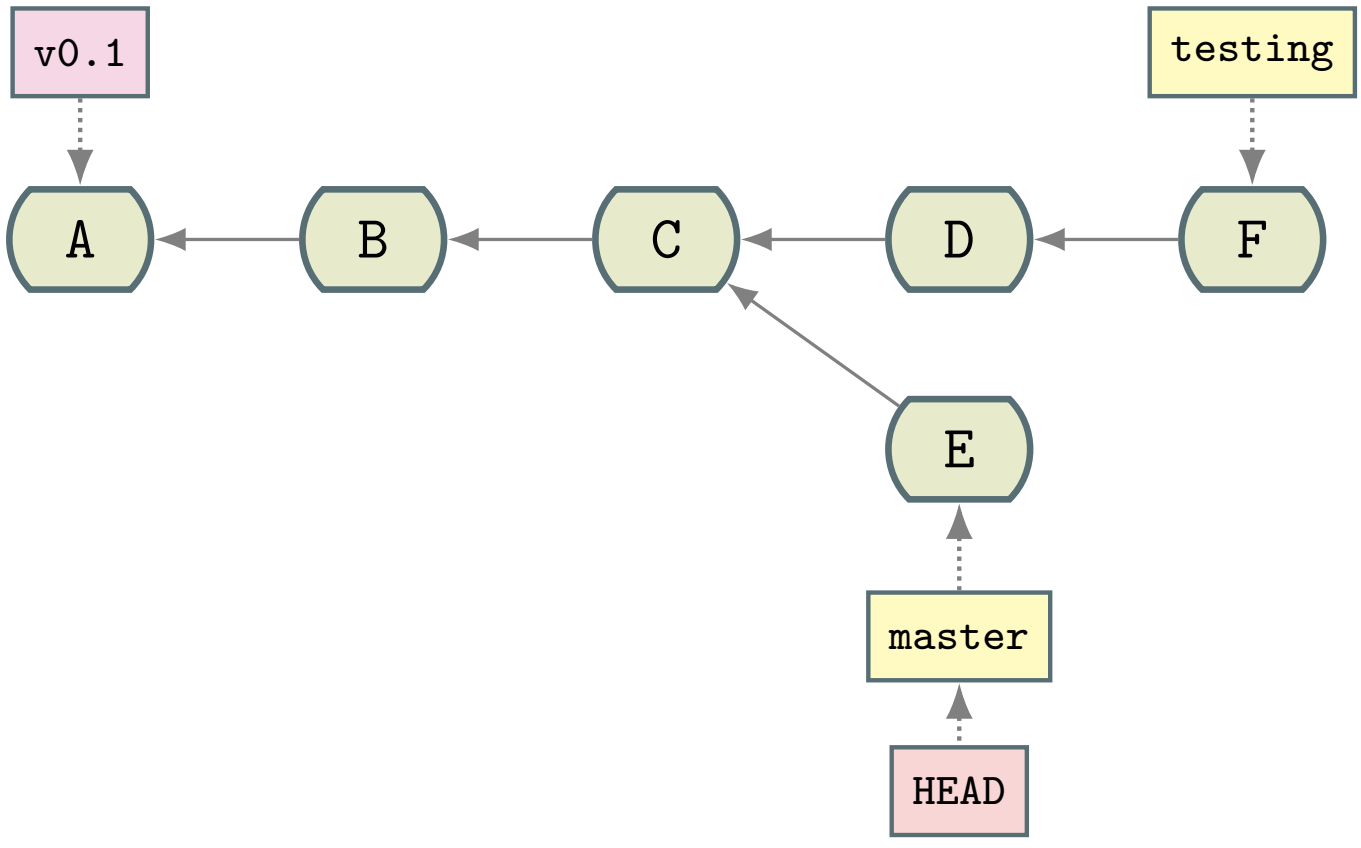

```
$> git branch -d hotfix
```



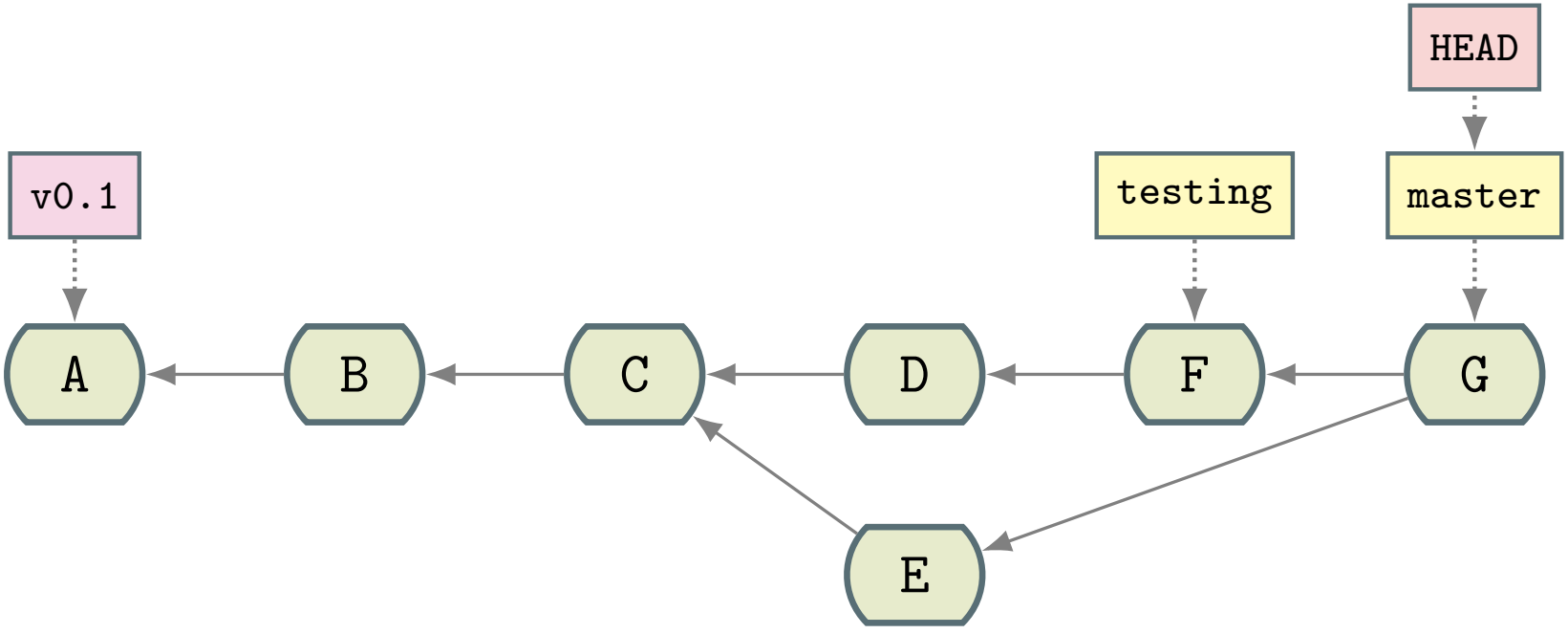

```
$> git checkout testing
```



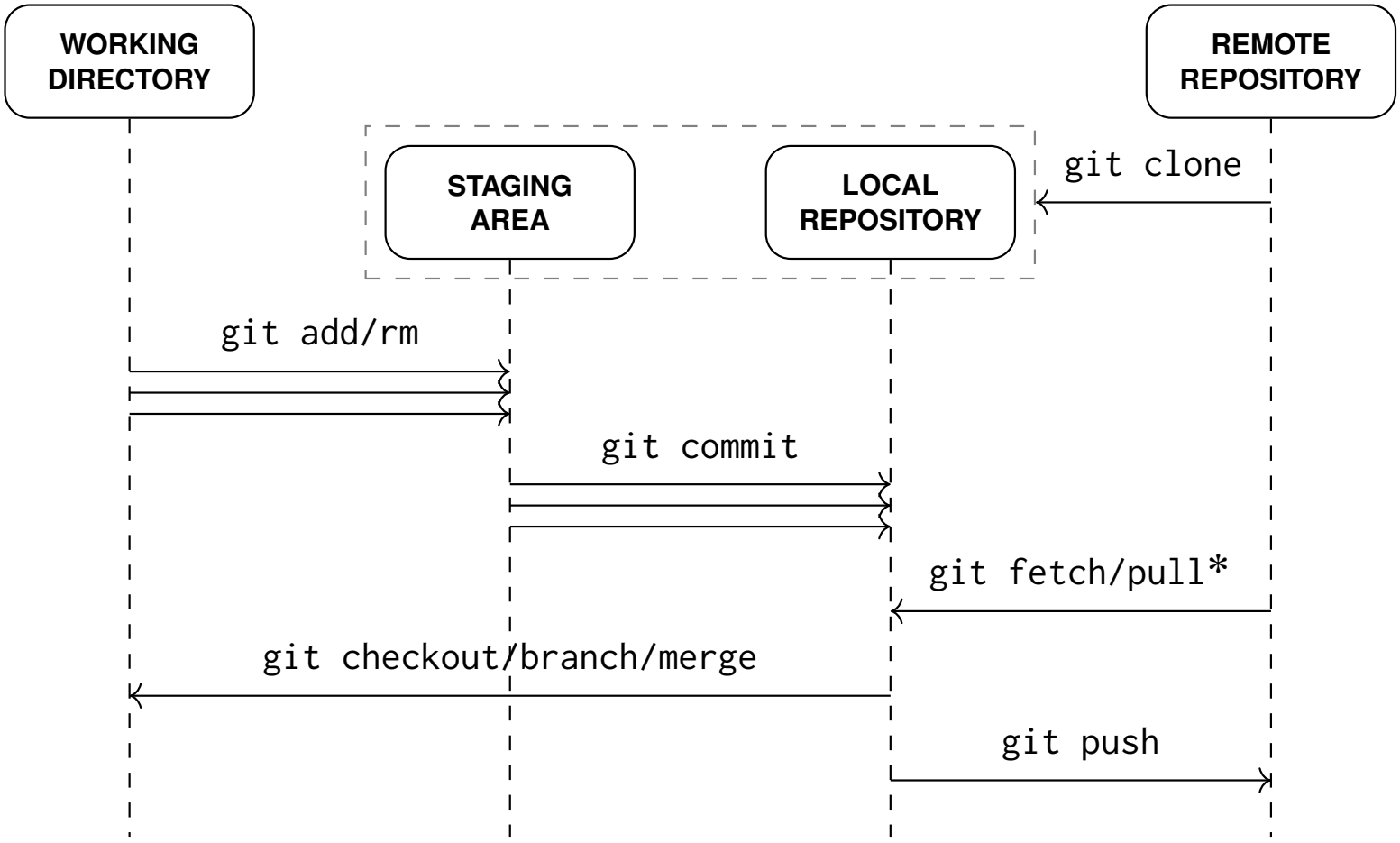
One commit later




```
$> git checkout master
```

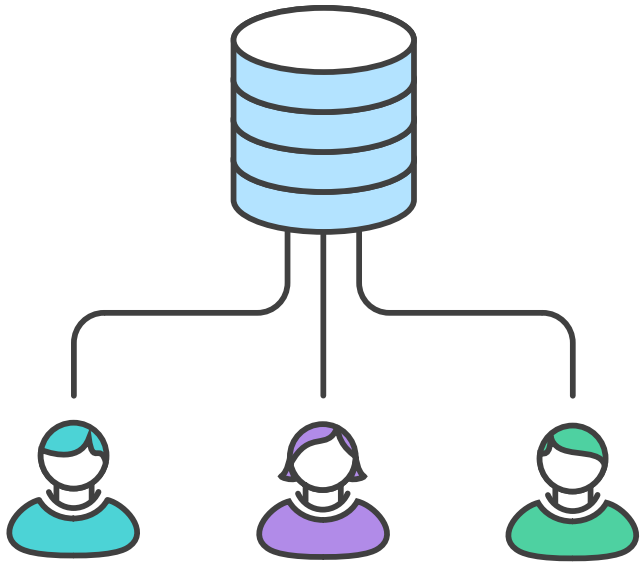



```
$> git merge testing
```


WORKING DIRECTORY
STAGING AREA
LOCAL REPOSITORY
REMOTE REPOSITORY
git clone
git add/rm
git commit
git fetch/pull*
git checkout/branch/merge
git push

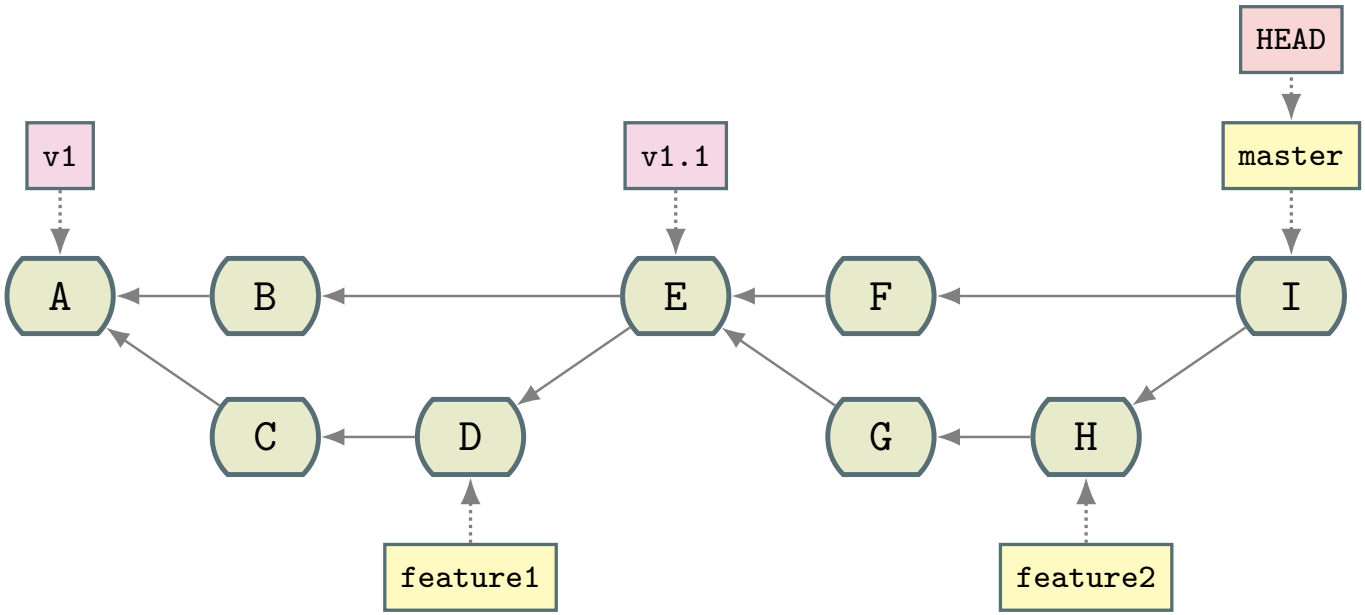
- one tool, many usages
- tools alone do not solve development problems
- need of a process that fits the team
- many possible Git workflows (examples later)
  - centralised workflow
  - feature branch workflow
  - gitflow workflow
  - forking workflow
  - . . .

- one central repository, one branch (master)
- common when coming from centralised systems like Subversion
- common for small size teams
- easy to understand for a newcomer



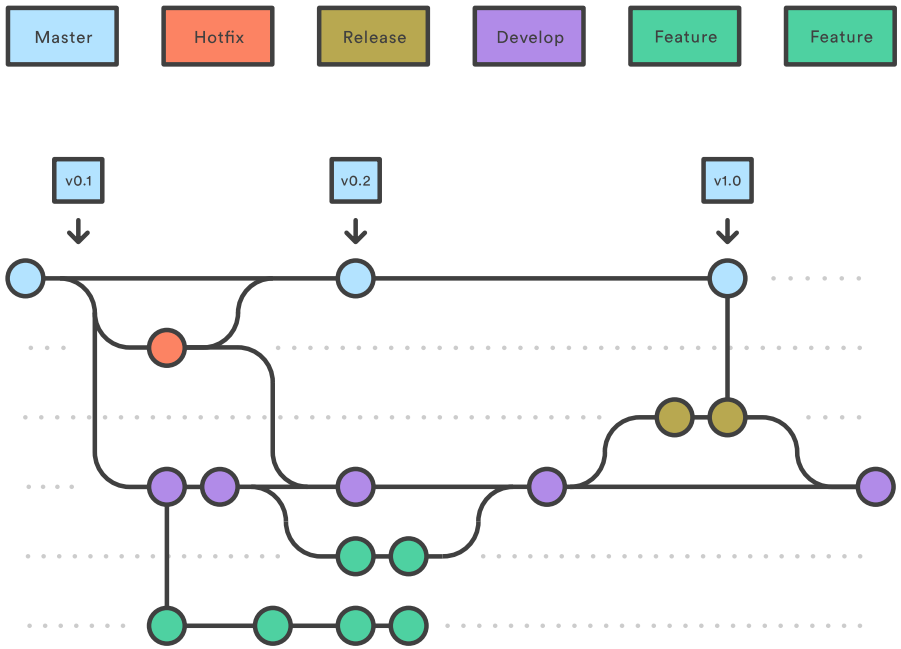
Source: Atlassian

- central repository + master branch = official project history
- one branch per feature: no direct commit on the master branch
- feature branches are pushed to the central repository
- branches are then merged (after pull requests, feedbacks, conflict resolutions)



Gitflow workflow[4]

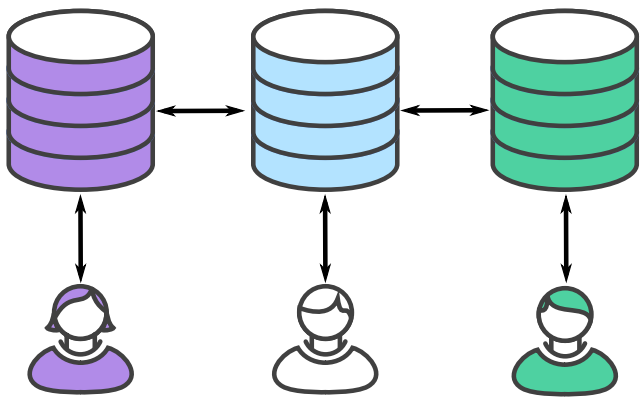
- strict branching model designed around the project release
- well-suited for large projects with deadlines (releases)
- one branch one role, workflow defines their interactions
- can be combined with feature branch workflow
- project history = master (the releases) + development branch



Source: Atlassian

[4] A famous example:
https://nvie.com/posts/a-successful-git-branching-model/

- one serverside repository per developer
- each developer manages her repository and make pull requests to the reference repository
- typical model when contributing to a FLOSS[5] project hosted on GitHub: “Fork us on GitHub”



Source: Atlassian

[5] https://www.gnu.org/philosophy/floss-and-foss.en.html

Workflows: summary

- chosen workflow depends on the team's concerns and organisation
  - no one-size-fits-all Git workflow
- feature workflow: business domain oriented
- forking and gitflow workflows: repository oriented
- what is a good workflow?
  - enhance or limit team efficiency?
  - scale with team size?
  - easy to undo mistakes and errors?
  - impose any new unnecessary cognitive overhead to the team?
  - does it limit conflicts?

- Git
  - useful and powerful tool
  - . . . but a tool alone does not solve all problems. It can also create ones

  $\Rightarrow$ developers do not only need tools, but also working processes

- Good practices
  - formalizing the process/workflow
  - coordinating with co-workers
  - testing before sending changes
  - updating before sending a change
  - commiting meaningful changes
  - commiting often
  - adding meaningful messages for commits
  - not commiting generated files
  - short-lived branches

- By practicing
  - at home
  - during every lab sessions, even in non-CS context
  - ILSD UEs: FIAB, CAD, PROCOM

- One usually needs a server to host repositories[6]
- Some questions to ask before chosing
  - do you want to make your project public?
  - is there any security, privacy or IP problems with the project?
  - is your project a cornerstone of your business?
- Your answers should drive your choices of VCS hosting
  - simple and free non-professional account on an open platform
  - paid service on a platform
  - installation of your own VCS server

[6] . . . but it is not mandatory: you can use Git in serverless mode! See later

- IMTA infrastructure for academic projects and for learning:
  - Gitlab: `https://gitlab.imt-atlantique.fr/`[7]
  - Redmine: `https://redmine-df.telecom-bretagne.eu/`
- Many platforms can be used without any fee:
  - Gitlab: `https://about.gitlab.com/`
  - GitHub: `https://github.com/`
  - Bitbucket: `https://bitbucket.org/`
  - Assembla: `https://www.assembla.com/`
  - Sourcehut: `https://sourcehut.org/`
  - . . . and probably many other
- . . . but you can also install your own server!

[7] prefer it rather than redmine-df which will probably die in a near future

- Git can also be used without any other host

1. `$> mkdir mycode`
2. `$> cd mycode`
3. `$> git init`
   initialize a new Git repository

- that type of Git repository can be shared
  - as every folder (copy/paste on an USB key, ...)
  - or using a Git command to add a remote repository (it has to exist)

`$> git remote add <name> <url>`

- Tendency to confuse VCS and Git
- Basic principles of VCS
  - basic principles
  - two main families: centralised vs decentralised
  - tools diversity
- Some good practices for VCS/Git usage
- Importance of a workflow
  - should be simple
  - should enhance the team productivity
  - should be oriented by business requirements
- VCS usage should be an habit, not a constraint

- VCS
  - https://homes.cs.washington.edu/~mernst/advice/version-control.html
  - https://betterexplained.com/articles/a-visual-guide-to-version-control/
  - https://betterexplained.com/articles/intro-to-distributed-version-control-illustrated/
- Git
  - https://git-scm.com/
  - https://git-scm.com/book/en/v2/ (Pro Git book)
  - http://justinhileman.info/article/git-pretty/
  - https://betterexplained.com/articles/aha-moments-when-learning-git/
  - https://rachelcarmena.github.io/2018/12/12/how-to-teach-git.html
- Subversion: http://svnbook.red-bean.com/
- Mercurial: https://www.mercurial-scm.org/

⚠ Important notes

- if you do not understand something, please ask your questions. *We cannot answer the questions you do not ask. . .*
- if you disagree with us, please say it (we follow Crocker's rules[8])
- people don't learn computer science by only reading few academic slides: practicing is fundamental

[8] `http://sl4.org/crocker.html`