Software Engineering

Configuration Management with Git

SoftEng
http://softeng.polito.it

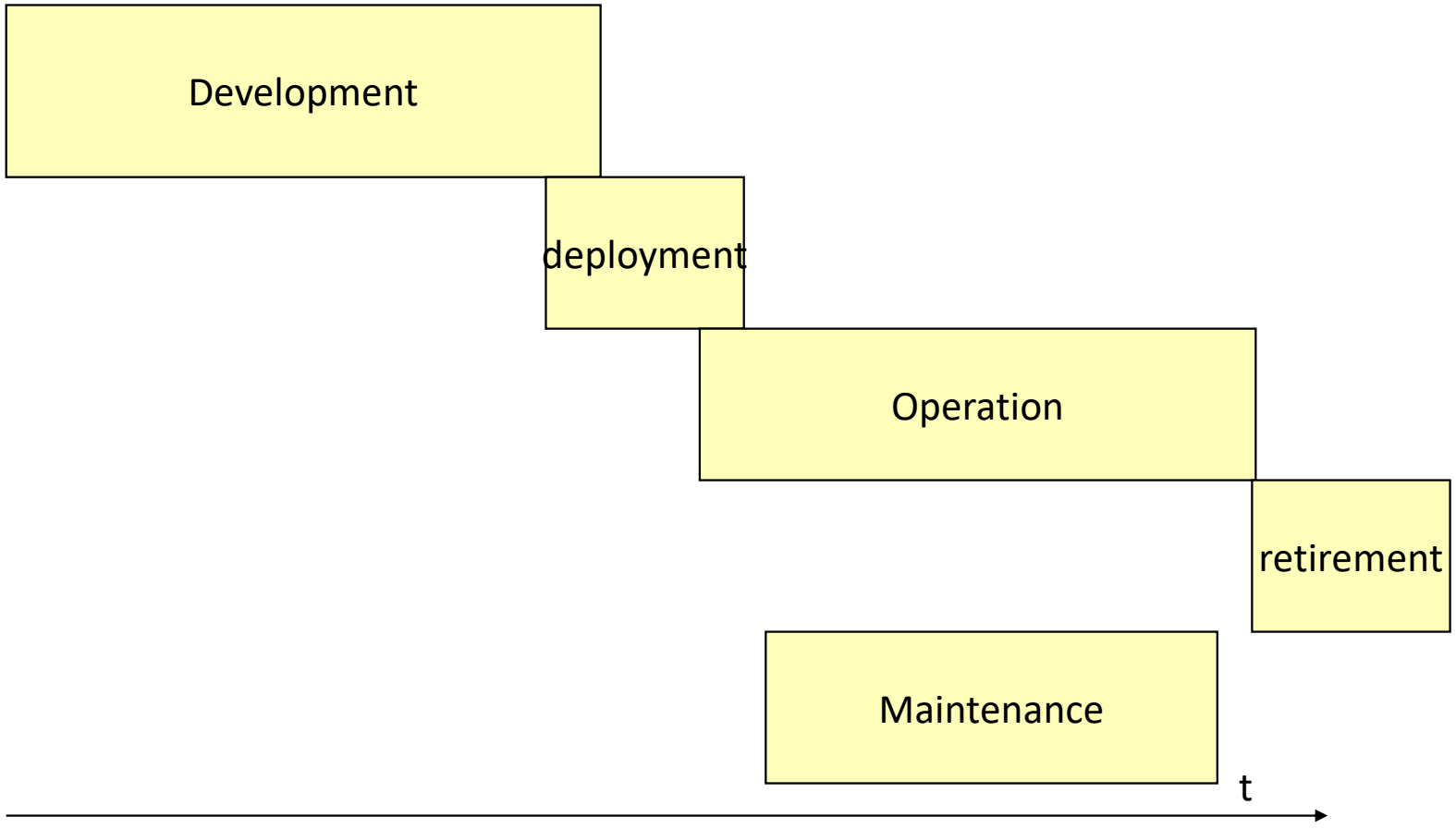
Version 2.0
© Maurizio Morisio, Luca Ardito, 2024

CC BY NC ND

Configuration Management

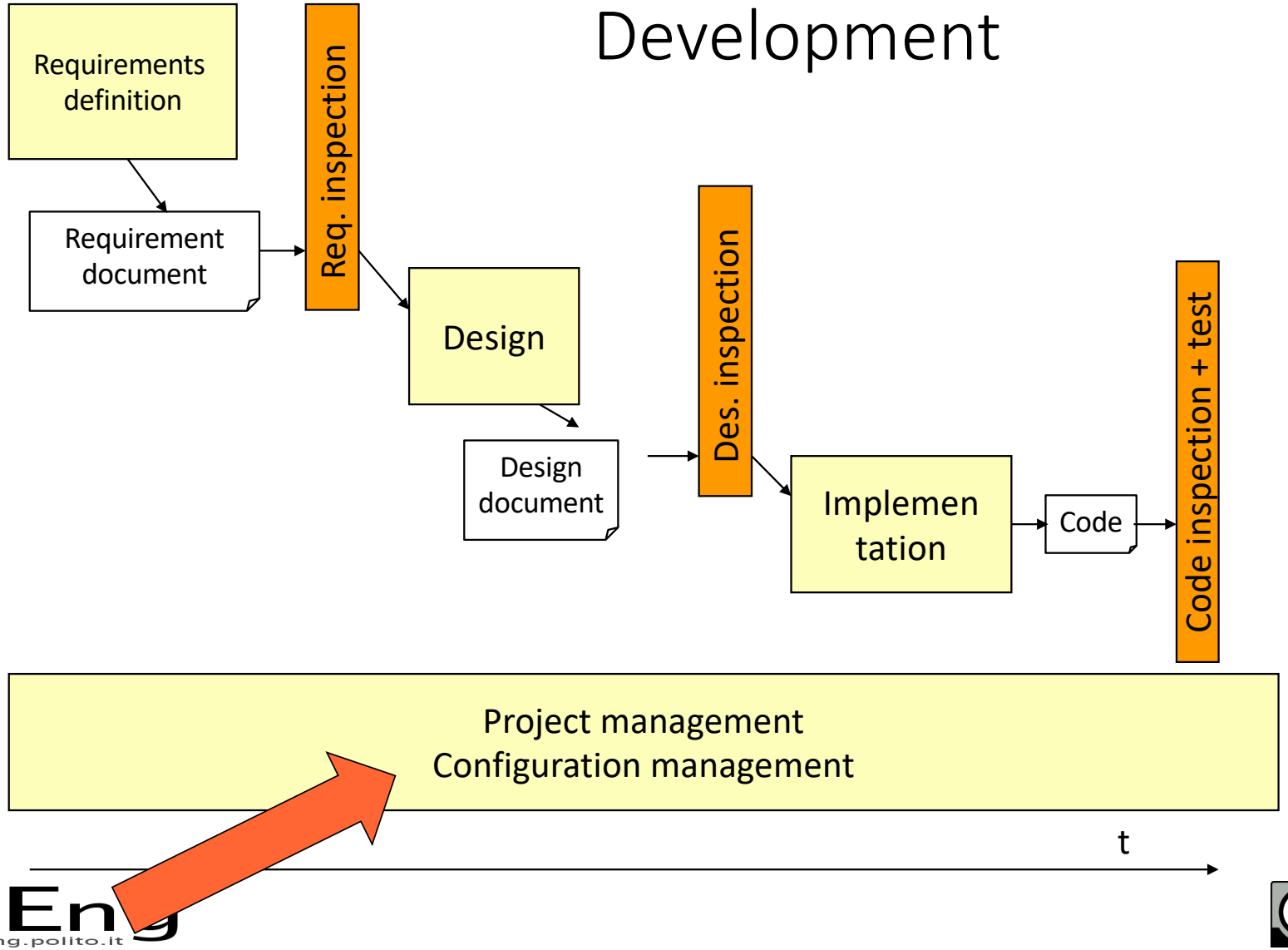
cc BY NC ND

Main Phases



cc BY NC ND

Development



cc BY NC ND

Motivation

- Many persons need to work on the same documents
  - Remember: software made of requirements, design, code, user manual, data,..

  for a long period of time
  - development, operation and maintenance

- Where are the documents?
- Who can change what, when?
  - Concurrent access and modification of documents
- Who did modify what?
- Which was the last working version?

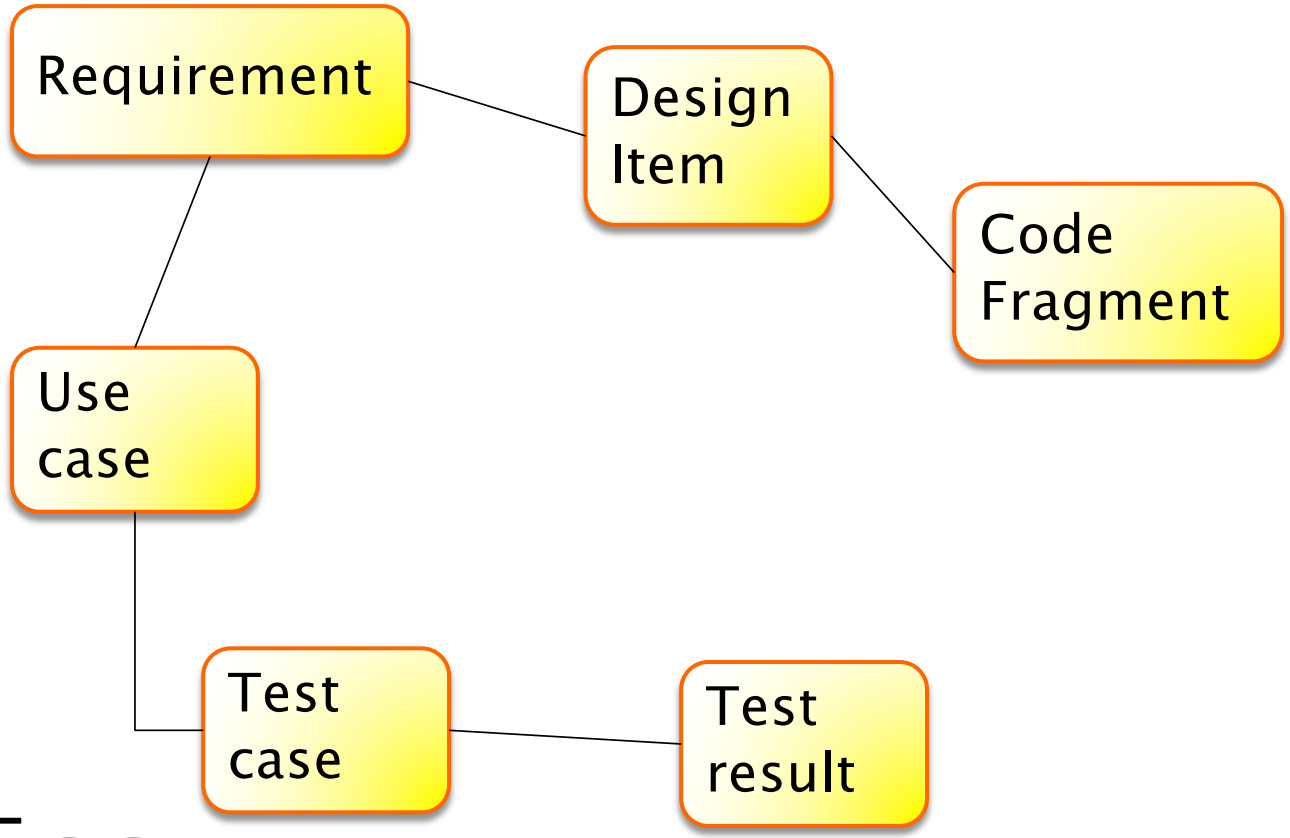
cc BY NC ND

Motivation - 2

- Documents have dependencies, in most cases dependencies are not formalized
  - Ex of formal dependency: ‘import’ class or package in java
  - Ex of non formal dependency: requirement document vs design document vs code vs test cases

cc BY NC ND

Dependencies



cc BY NC ND

CM Concepts

- Configuration Item (CI)
  - Unit under configuration management
    - Can be composed of one (ex requirement document) or more files (C++ class in two files)
- Configuration
  - Set of CIs
    - Can be consistent CIs (CIs where all dependencies are correct) or not
- Repository
  - Logical / physical place where CIs are

CC BY NC ND

CM Concepts

- Versioning
  - Capability of storing / rebuilding all past versions of a CI or configuration
- Change control
  - Capability of granting right of reading / writing a CI to a defined user

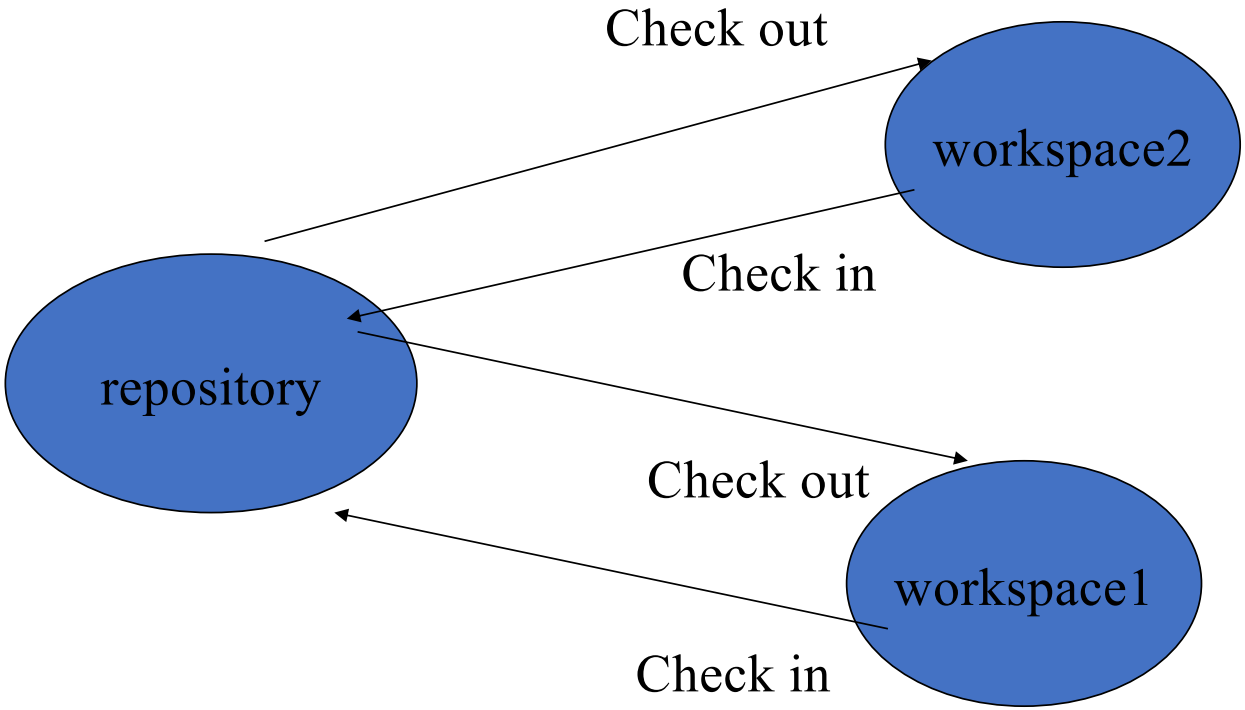
cc BY NC ND

CM concepts

- Check-in Check-out
  - Formal operations made by a user to notify the CMS that a CI is being accessed and modified
  - Checkout: notification that a CIs is being accessed by a certain user
  - Checkin: notification that a CIs is no longer accessed, and has been modified

    (‘check in’ also called ‘commit’)

cc BY NC ND


Check out
workspace2
Check in
repository
Check out
workspace1
Check in

cc BY NC ND

CM Concepts

Change control models

- Lock modify unlock: the first user locks the CIs, no other user can access it until first user unlocks
  - PRO: no concurrent modifications
  - CON: obliges to serialize work, first user often forgets to unlock

- Copy modify merge: many users can checkout in parallel and work on copies
  - PRO: allows parallel work
  - CON: conflicts (== inconsistent modifications of the same CI by different users) are possible, merge operation is delicate, time consuming and possibly error prone

cc BY NC ND

CM choices

- What becomes a CI
  - Not all documents / files in a project become CI
  - A CI has advantages (versioning access control etc) but at a cost of an overhead (check in check out)
- Lock model or not
- Time distance between commits
  - Every week? day? hour?
- CM manager role
  - Defined or not
- CMS tool

cc BY NC ND

Configuration Management System (CMS)

- Software application capable of
  - Storing and versioning CIs
  - Storing and versioning configurations
  - Change control

CC BY NC ND

CMS Functionalities

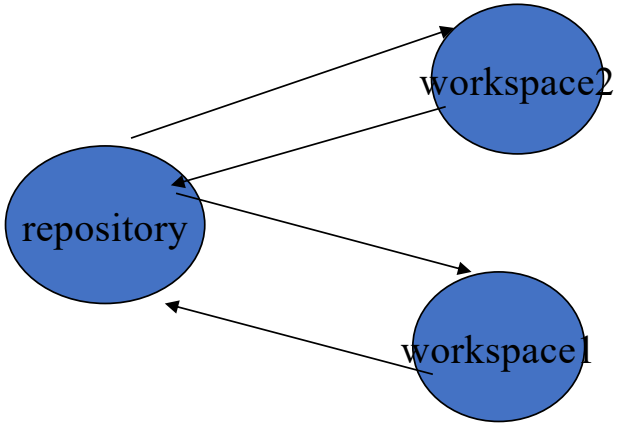
- Revert CI back to a previous version
- Revert configuration back to a previous version
- Compare changes over time
- Control access (who can read / write what Cis)
- Track who modified what when

cc BY NC ND

- Where are the documents?
  - → repository, CIs, configurations
- Who can change what, when?
  - → change control
- Who did modify what?
  - → tracking
- Which was the last working version?
  - → configurations, versioning

cc BY NC ND

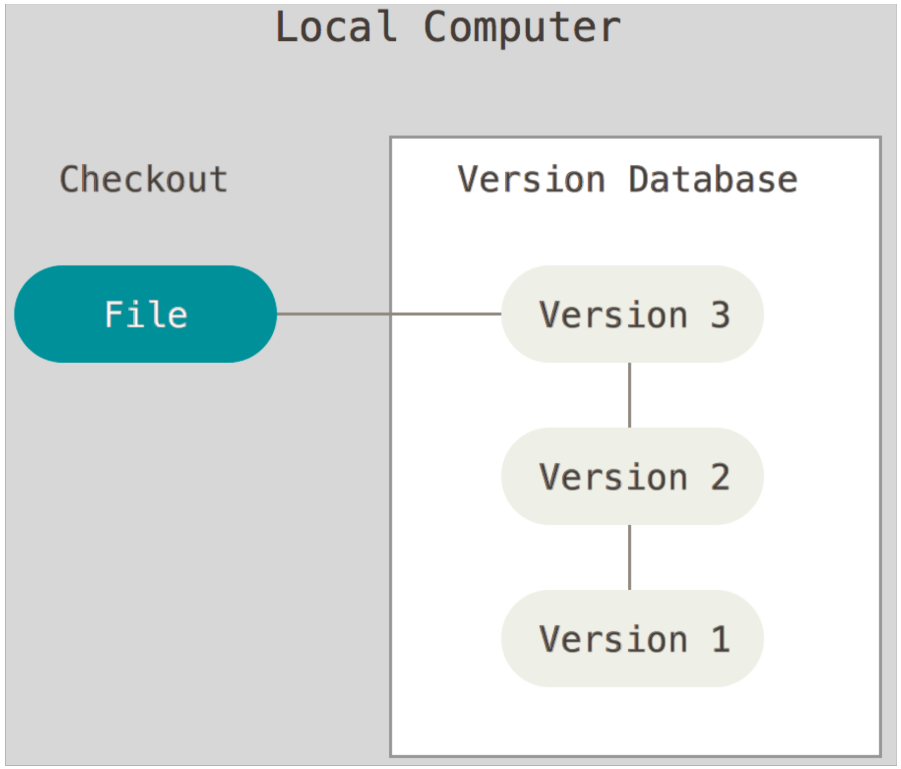
CMS Taxonomy



- Local CMS
  - Only one machine, repository and workspace on the same machine
- Centralized CMS
  - Server contains repository (all CIs all versions), clients contain copy
- Distributed CMS
  - Server contains repository, each client mirrors the repository locally

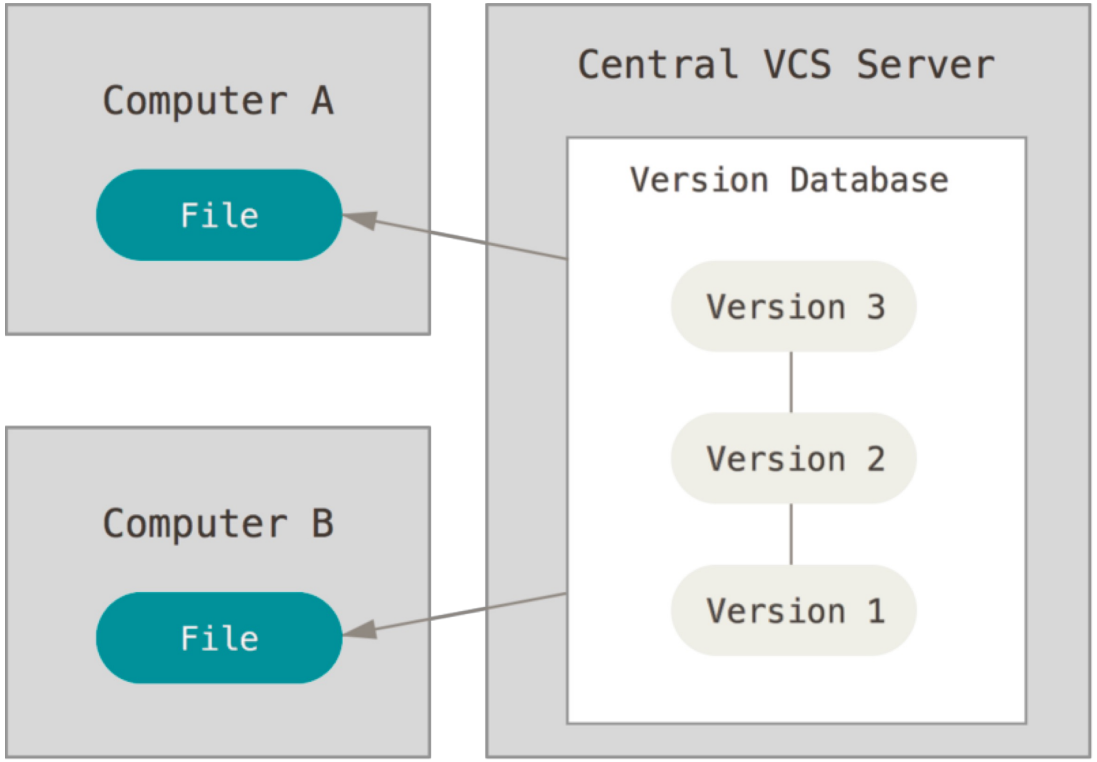
cc BY NC ND

Local CMS



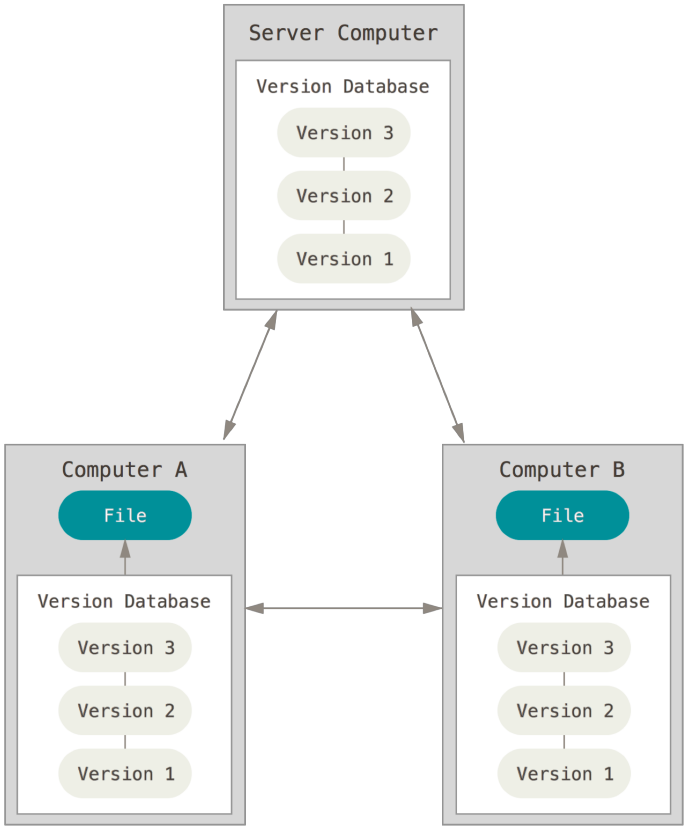
cc BY NC ND

Centralized CMS



CC BY NC ND

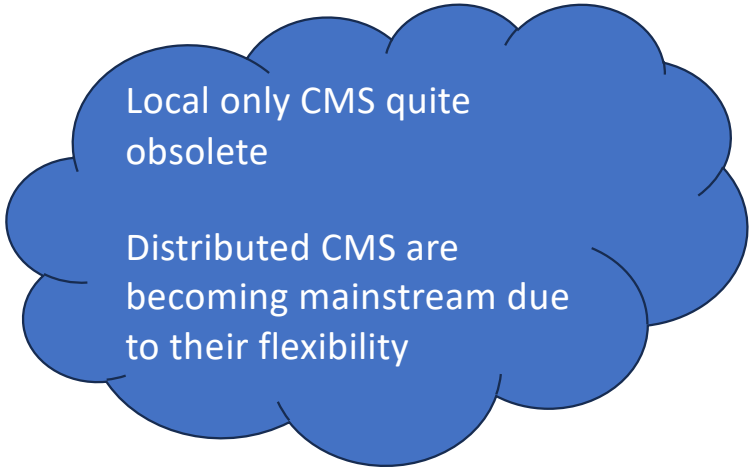
Distributed CMS



cc BY NC ND

Examples

- Local CMS
  - RCS
- Centralized CMS
  - CVS
  - Subversion
  - Perforce
- Distributed CMS
  - Git
  - Mercurial
  - Bazaar or Darcs



cc BY NC ND

Storage Models

- Deltas
  - For a CI the root version is stored, for next versions only the differences are stored

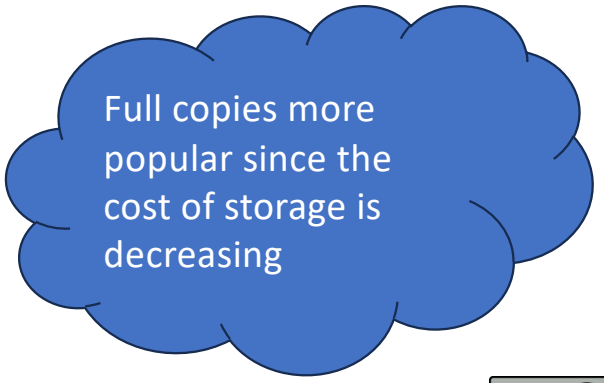
  PRO. Less space is used

  CON. Reconstructing a version can take time

- Full copies
  - For a CI a full copy of each version is stored

  PRO. More space is used

  CON. Each version available in zero time



cc BY NC ND

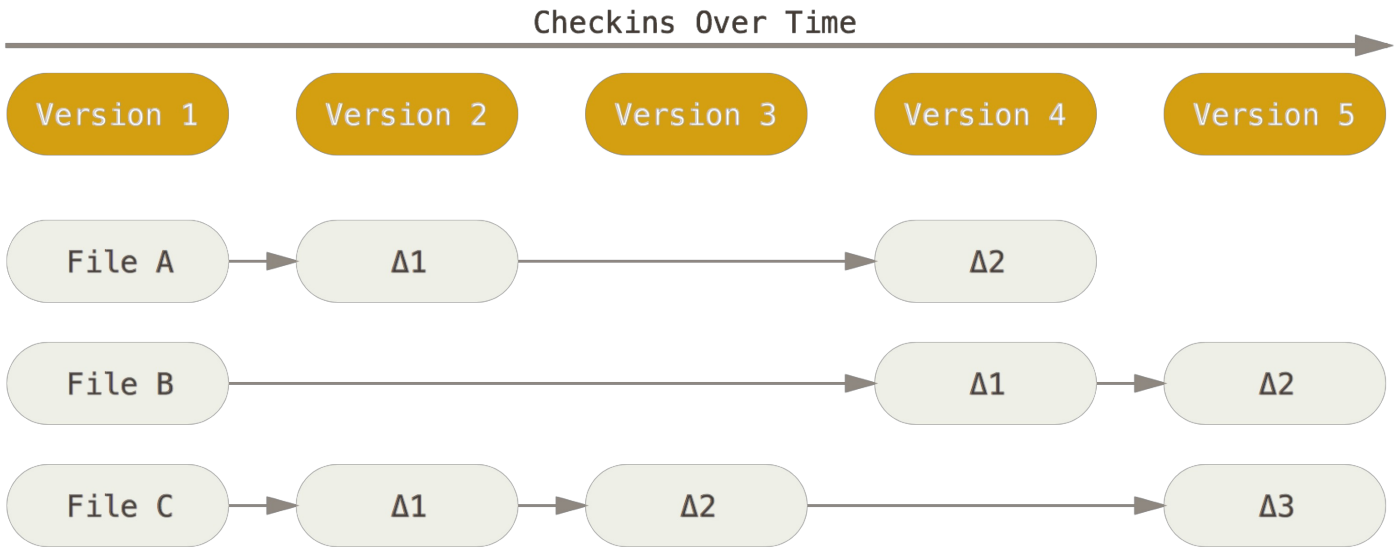
Configuration management models

- Differences (ex svn)
  - A commit by a user changes the version of a CI (only)

- Snapshot (ex git)
  - A commit by a user defines a new snapshot, including all CIs
  - A snapshot is made of all CIs in the project
  - If a CI did not change version, a link to the previous one is used

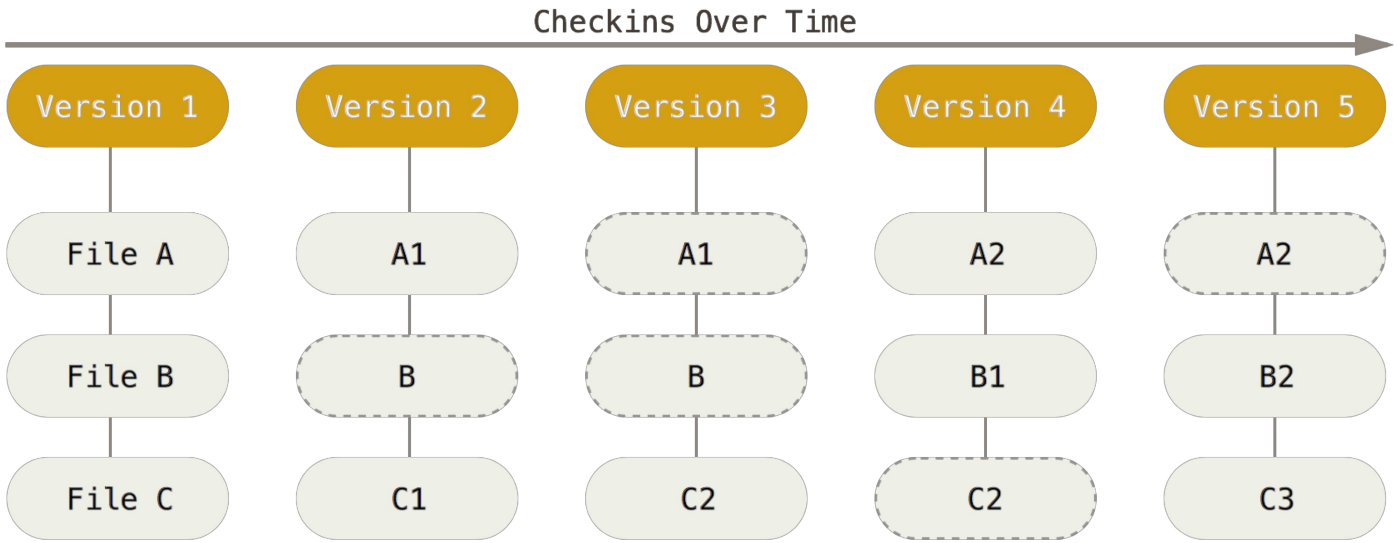
CC BY NC ND

Differences



CC BY NC ND

Snapshots



CC BY NC ND

Introduction to Git

SOftEng
http://softeng.polito.it

CC BY NC ND

Git

- Started by Linus Torvalds in 2005 to support Linux development
- Radically different from previous CMSs
- Focus on
  - Speed
  - Support for large projects, with thousand of branches
  - Support for distributed, non linear development model

cc BY NC ND

Git

- Distributed CMS
- Snapshots as configuration model
  - All files together, not one by one
- Local operations
  - No information is needed from other computer(s) (server or else) → speed
- Integrity
  - Everything is check-summed before it is stored and is referred to by that checksum → any change is recognized, no untracked change of any file or directory
- Additive
  - Information is added, never deleted → No worry about messing up things

cc BY NC ND

GitHub

- Online code hosting service built on top of git
- Mainly used by OSS projects
- Commercial use is growing
  - Used by

BY NC ND

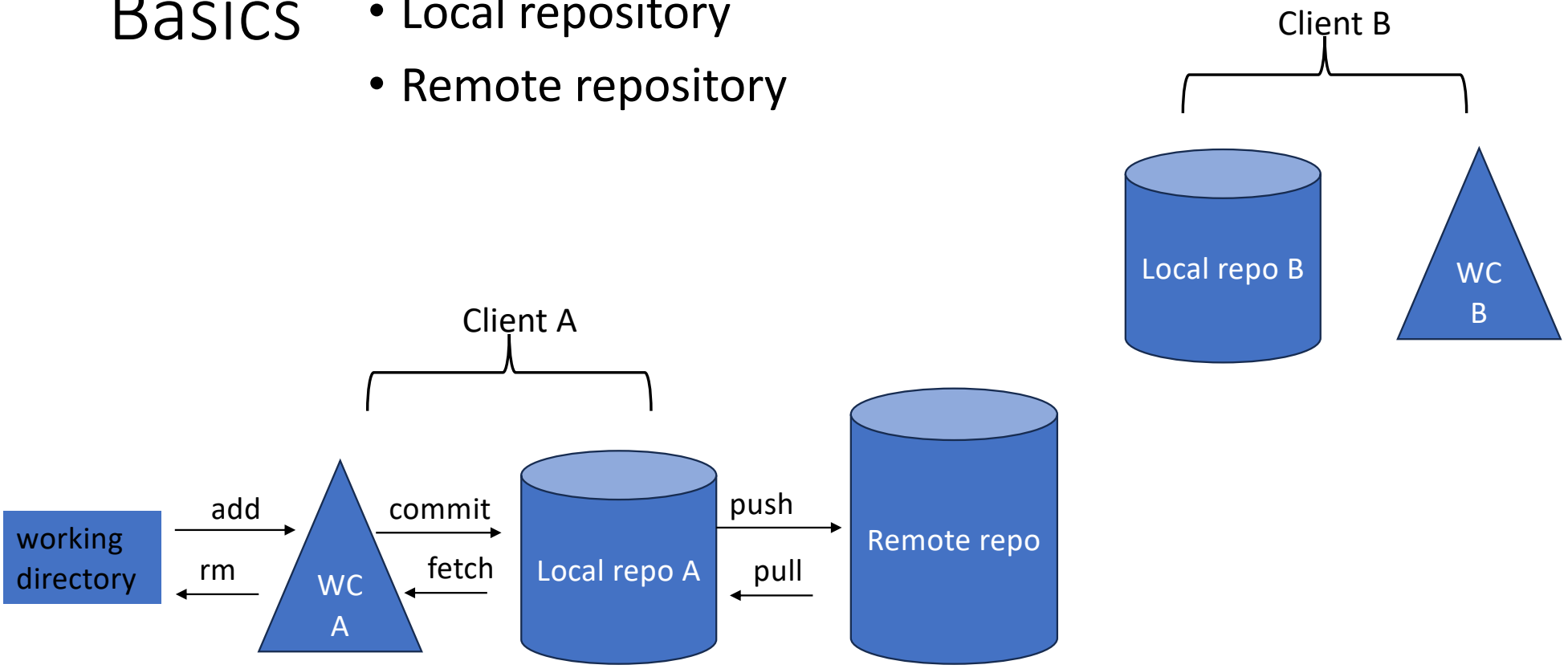
GitLab

- Another Online code hosting service built on top of git

cc BY NC ND

Basics

- Working copy (WC)
- Local repository
- Remote repository



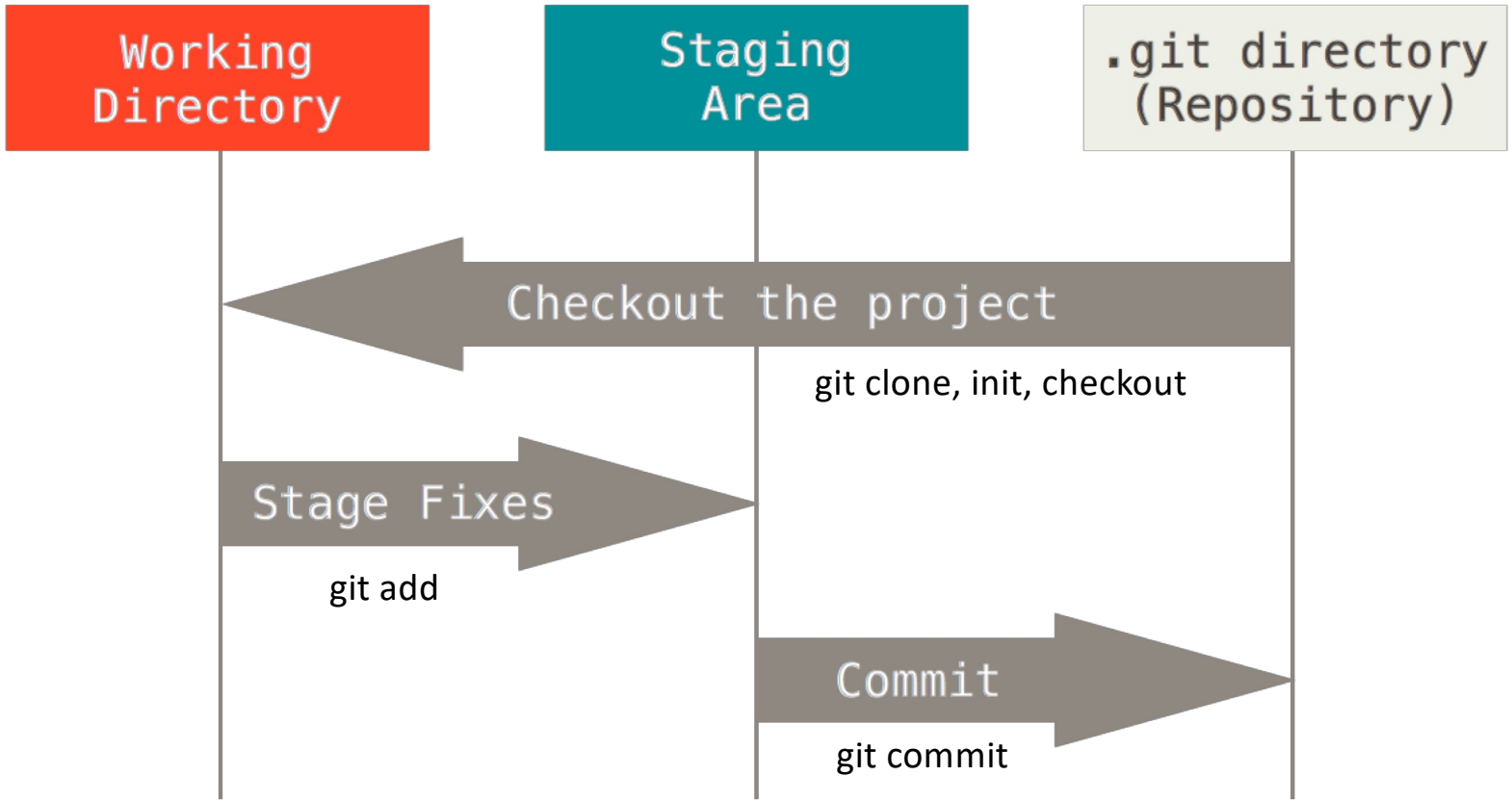
cc BY NC ND

Client side

- Folder and file system, or working directory or working tree
  - Files and folders
- Working copy (WC), or index, or staging area
  - Files and folders to be committed, but not yet versioned
    - Add: enters a file from file system into working copy
    - Rm: removes file from working copy
- Local repository
  - Files and folders, versioned
    - Commit: enters all files (snapshot) from WC to local repository
    - Fetch, rebase: copies snapshot from local repository to WC, merge them

cc BY NC ND

Typical Workflow



cc BY NC ND

Client side: commit

- Modifies the local repository
- Atomic operation
  - Either succeeds completely, or fails completely
    - If many Cis are committed and one fails, no CI is committed
  - The integrity of the repository is assured
- It is mandatory to provide a log message (or comment) with the commit
  - to explain the changes made
  - the message becomes part of the history of the repository

cc BY NC ND

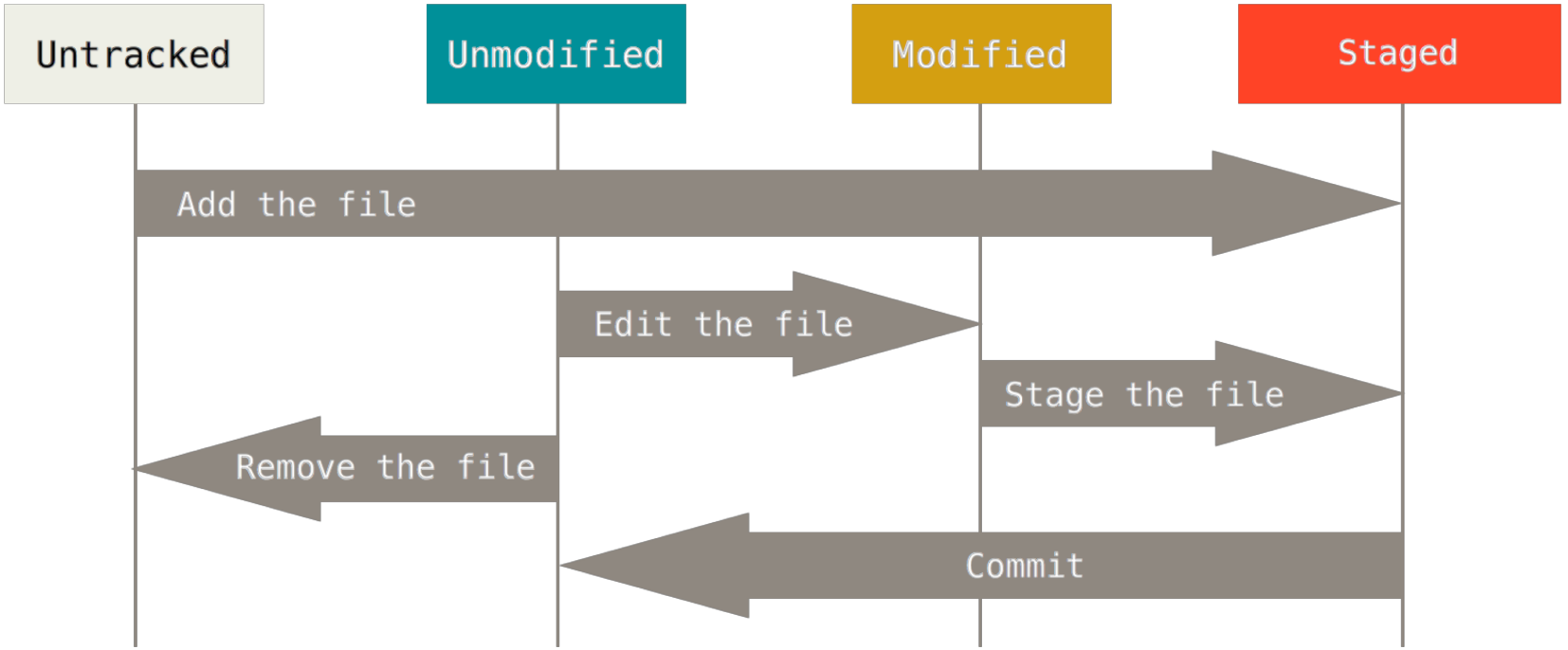
Cls states

- Untracked: in the working folder but not tracked (monitored) by git
- Staged (tracked): in WC (or staged area) - add command
  - Modified
  - Unmodified
- Committed: versioned, in the local repository – commit command

(another option is to explicitely exclude files from version control using the .gitignore file so that such files and folders will not be staged)

CC BY NC ND

Cls States



CC BY NC ND

Snapshot

- Set of all files (CIs) in a repository, in a certain version
- Identified by a unique ID (commit hash) computed at commit time

  Ex long version 8406cb12bff17d95a92ee0ac704d2dc037fa97

  short version 8406cb1

  You get id with

  git log

  git rev-parse HEAD

  git rev-parse –short HEAD

cc BY NC ND

Server side

- Remote repository, shared by all clients
  - Push: uploads a snapshot from local repository to remote repository (sync of repositories)
  - Pull: downloads snapshot from remote repository to local

CC BY NC ND

Summary of commands

- Set up user
  - Config –global user.name «my_name»
  - Config –global user.email «my_email»
- Set up repo
  - Local
    Init : creates a local repo in current folder
  - Remote
    Remote add: connects local repo with remote repo
- Work on local repo
  - Add, rm, mv
  - Commit
  - Checkout <hashcode>
  - Fetch, Rebase
  - Log, status, diff
- Work on remote repo
  - Push, pull

cc BY NC ND

Commands - set up

- `git config –global user.name «my_name»`
- `git config –global user.email «my_email»`

  Define name and email of user – to be done once

- `git init`

  Initializes an empty local repository in the current folder

  creates a .git directory inside it
- `git remote add origin http://server.com/project.git`

  Adds a new remote repository

  Origin is the ‘standard’ name for indicating the principal remote

CC BY NC ND

Commands - understand

- `git status`

  Show which files are in which state

- `git diff`

  Similar to previous, but shows also lines changed

- `git log`

  Show history of what happened in a repository

- `git ls-files`

  Show tracked files

CC BY NC ND

Commands - modify local repo

- `git add <file>`
  - add <file> to the WC (<file> changes will be tracked from now on)
- `git rm <file>`
  - Remove <file> from the WC (changes will not be tracked anymore) and file system
- `git restore <file>`
  - Restore file in file system
- `git mv <file>`
  - Rename a file
  - Git doesn't explicitly tracks renaming. No metadata is stored in Git that tells it the file was renamed
- `git commit`
  - Commit changes from WC to local repo
- `git commit -a`
  - Commits all tracked files, without need to add them to staging area

CC BY NC ND

Basic Concepts: fetch, rebase

- Updates the working copy with respect to the repository
  - Fetch gets changes from the repository
  - merge such changes with the ones you have made to your working copy, if necessary

  - checkout

cc BY NC ND

Commands – work with remote repo

- `git pull`
  - fetch and then merge remote repo on local repo
- `git push`
  - upload local repo on remote repo

cc BY NC ND

Commands – help

- `git help  <command>`
- `git <command> -help`

cc BY NC ND

Common scenarios

Start a project from scratch. Create local repo, work on it, sync on remote repo

- Git init
- … work on files file1 file2 filen
- Add file1, file2, filen
- Commit
- Remote add
- push

cc BY NC ND

Common scenarios

A project has already been created by others, sync locally, work on it, sync remotely

- Git init
- Git add remote
- Pull
- .. Work on files
- Add
- Commit
- Push

CC BY NC ND

Common scenario

- Modifying a project, build fails, cant understand why. Retrieve previous version that did build
  - Git checkout <commit id>
  - Git checkout –b <new branch> <commit id>

cc BY NC ND

Branching

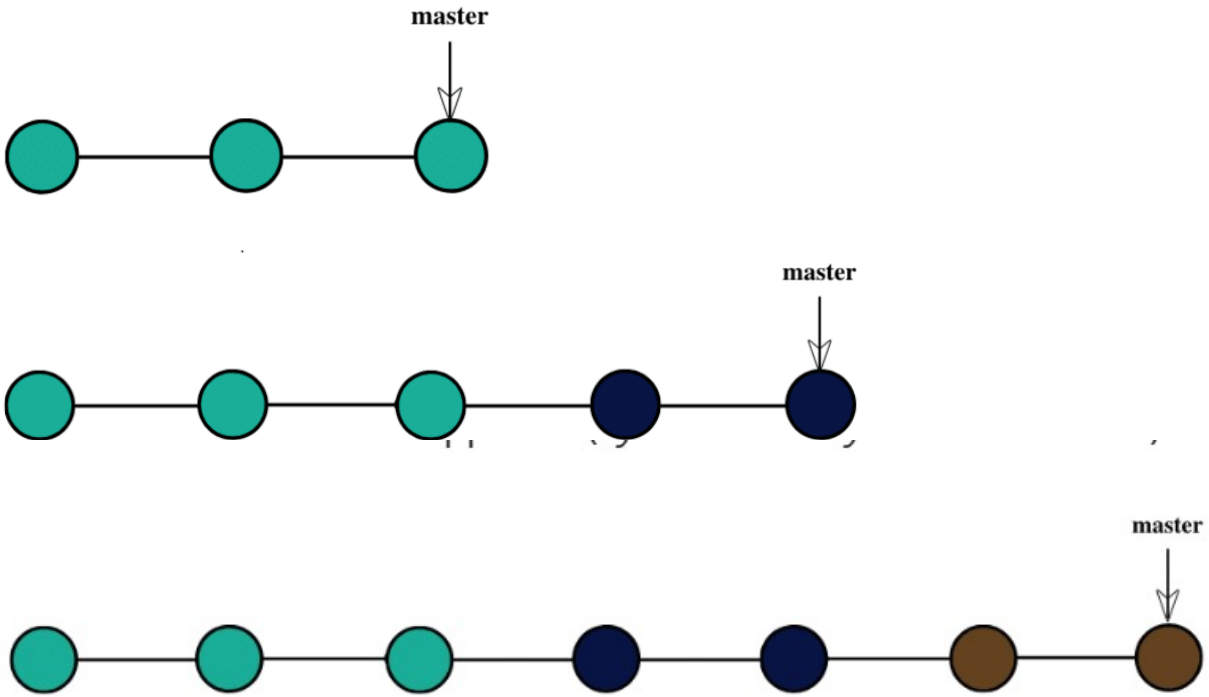
cc BY NC ND

Branching - motivation

- Two modes of development in a project
  - Linear
  - Branching

cc BY NC ND

Linear development

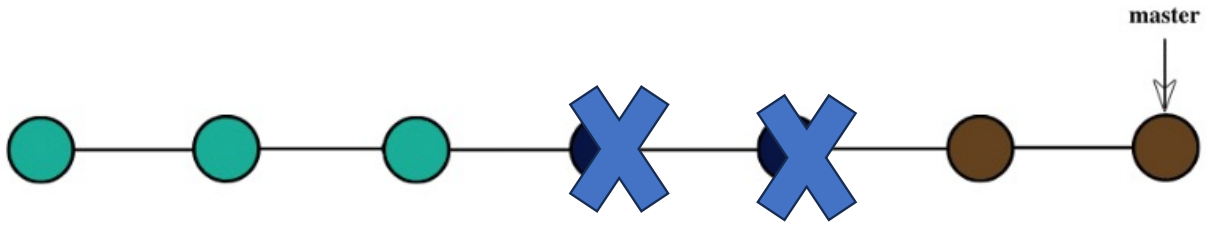


- Initial status of project
- Add feature 1 (blue)
- Add feature 2 (brown)

cc BY NC ND

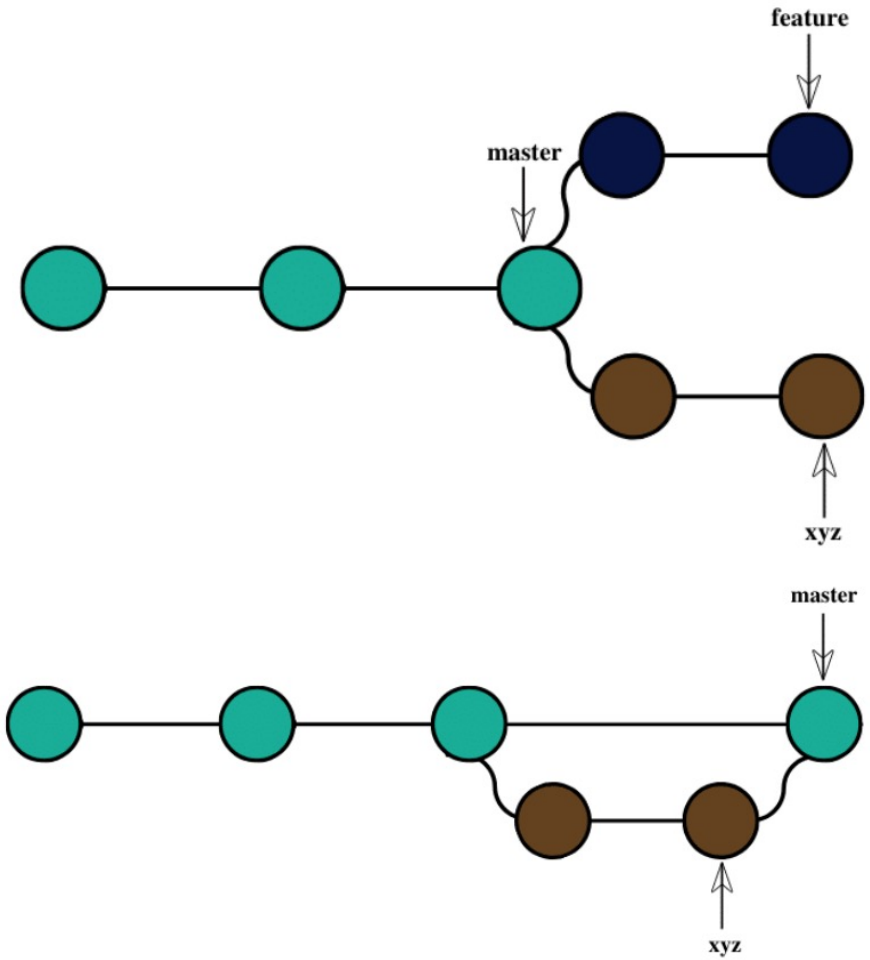
Linear development

- For some reason (ex client rejects feature 1, but wants feature 2) the project needs to delete feature 1 and keep feature 2
- However, changes are intertwined, doing this will be very difficult



cc BY NC ND

Branching



- One branch per feature, master remains unchanged
- To cancel blue feature just delete its branch, and merge brown branch into master

BY NC ND

Git Branching

SoftEng
http://softeng.polito.it

CC BY NC ND

Commands

- Create a new branch
  - git branch <new branch>
  - git checkout -b

cc BY NC ND

Data Storage in Git

- Git stores a commit object that contains
  - a pointer to the snapshot of the content you staged
  - pointers to the commit or commits that directly came before this commit (its parent or parents)
- Staging the files
  - checksums each one
  - stores that version of the file (blobs)
  - adds that checksum to the staging area

CC BY NC ND

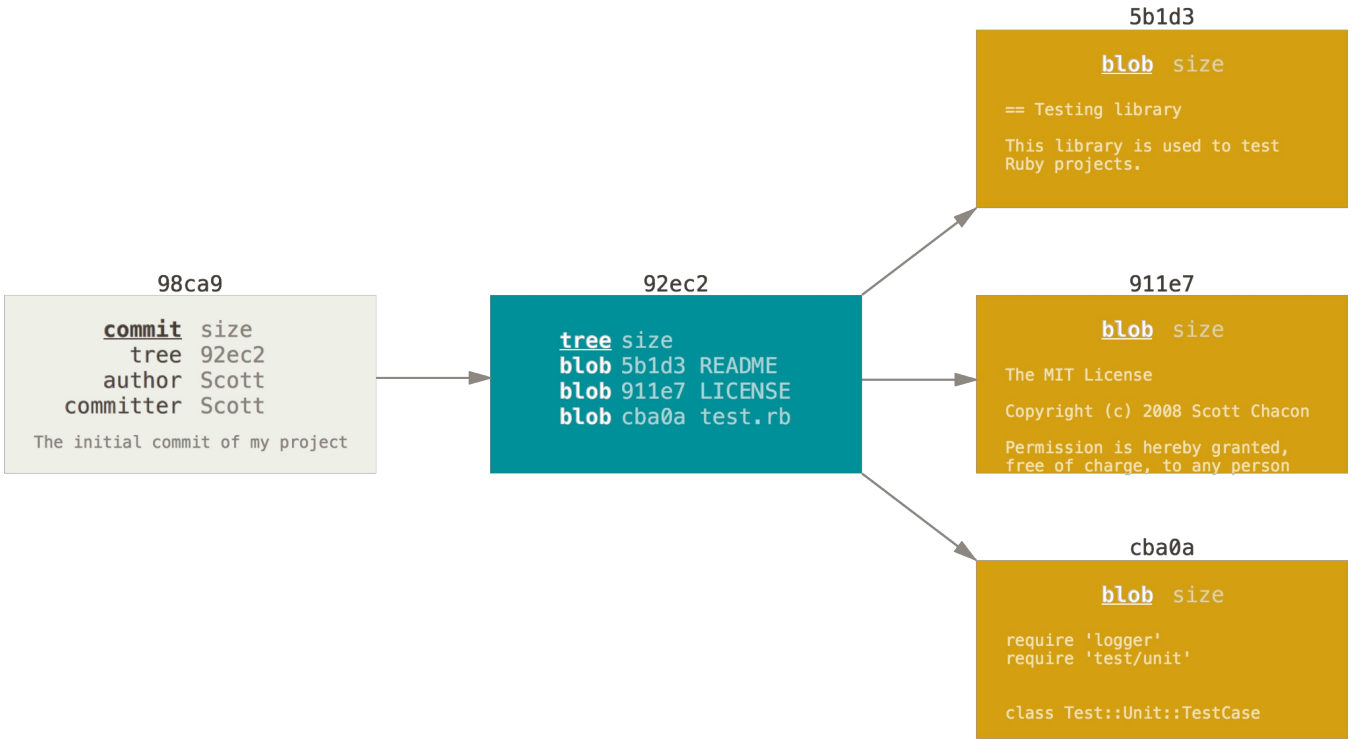
Example

- Given a directory containing three files, stage them all and commit

- Git repository now contains five objects
  - one blob for the contents of each of the three files
  - one tree that lists the contents of the directory and specifies which file names are stored as which blobs
  - one commit with the pointer to that root tree and all the commit metadata

CC BY NC ND

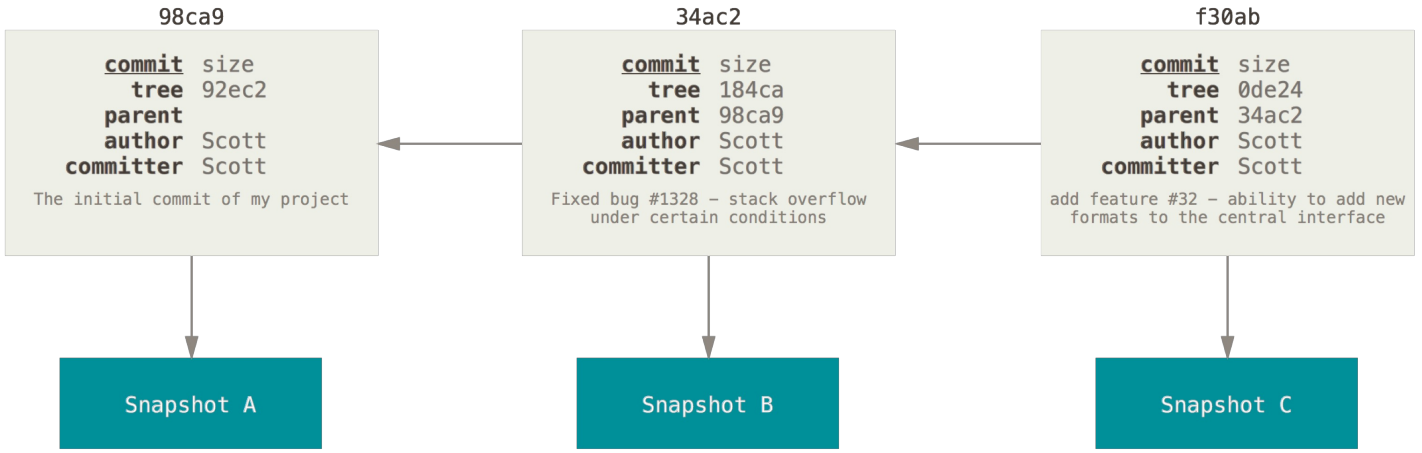
Example



cc BY NC ND

Example

- The next commit stores a pointer to the commit that came immediately before it



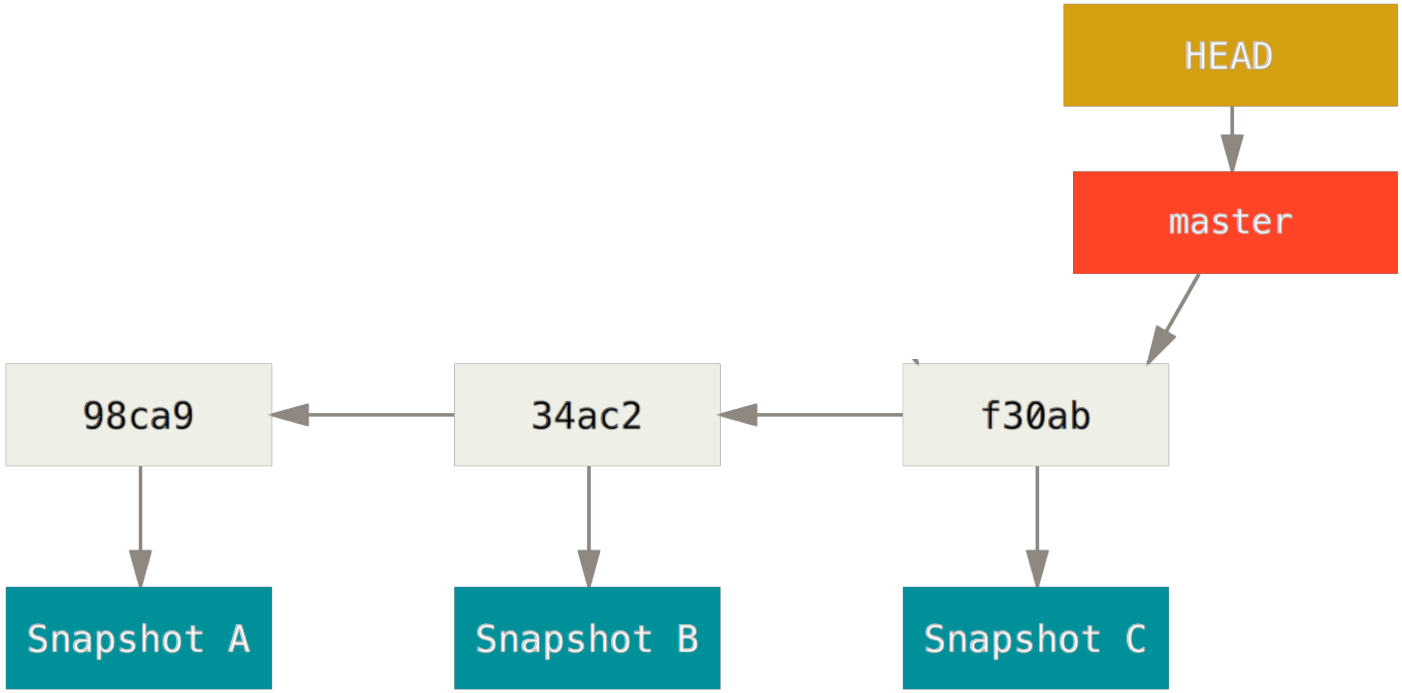
BY NC ND

Branches

- A branch in Git is simply a lightweight movable pointer to one of these commits
  - Git branch <name> // creates a branch, ie creates a pointer
- HEAD points to the current branch
  - The default branch name in Git is master
- Every commit, HEAD moves forward automatically
- HEAD can be moved in other ways too
  - Git checkout <name> // HEAD now points to <name>
  - Git checkout –b <new branch> <commit id> // creates branch and moves HEAD

CC BY NC ND

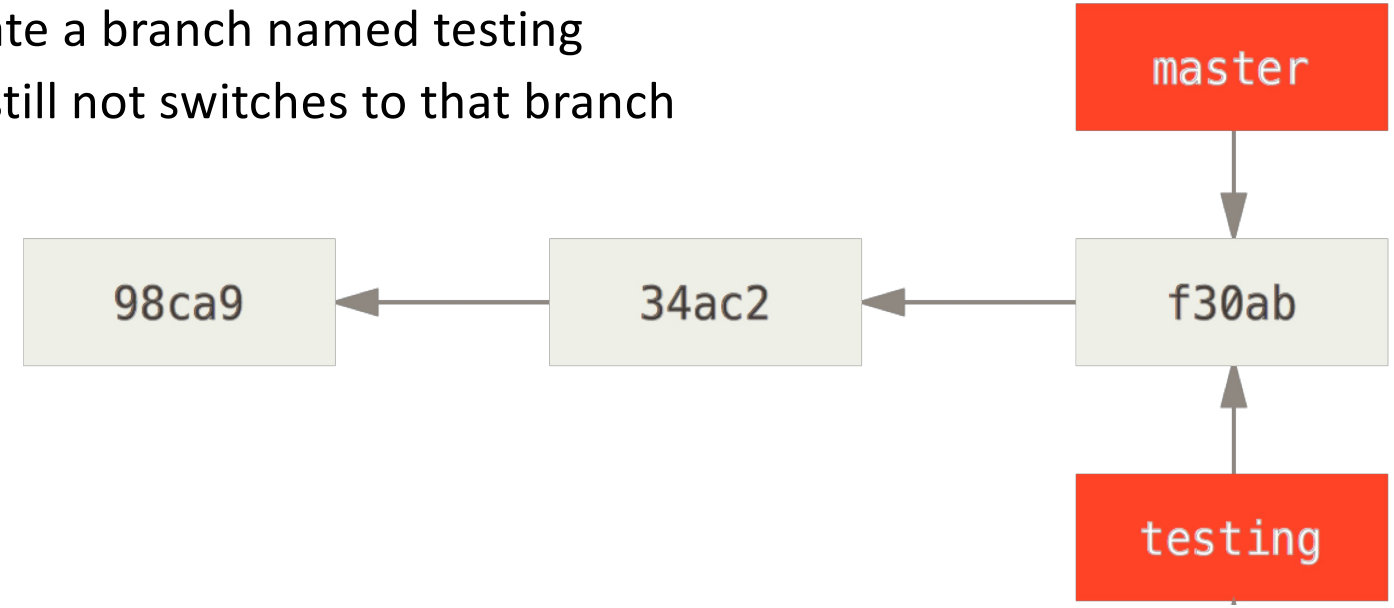
Branches



cc BY NC ND

Creating a Branch

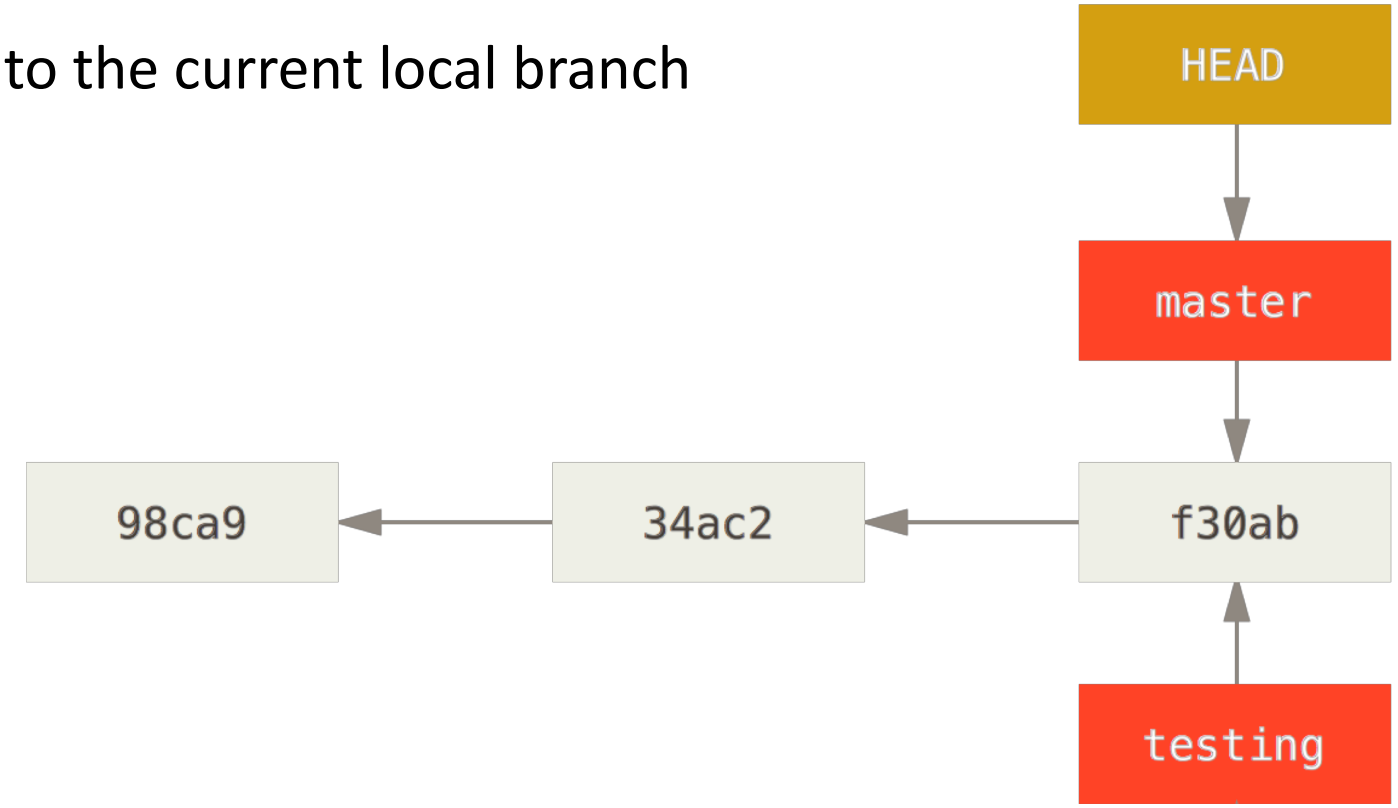
- `git branch testing`
  - Create a branch named testing
  - Git still not switches to that branch



cc BY NC ND

Head Pointer

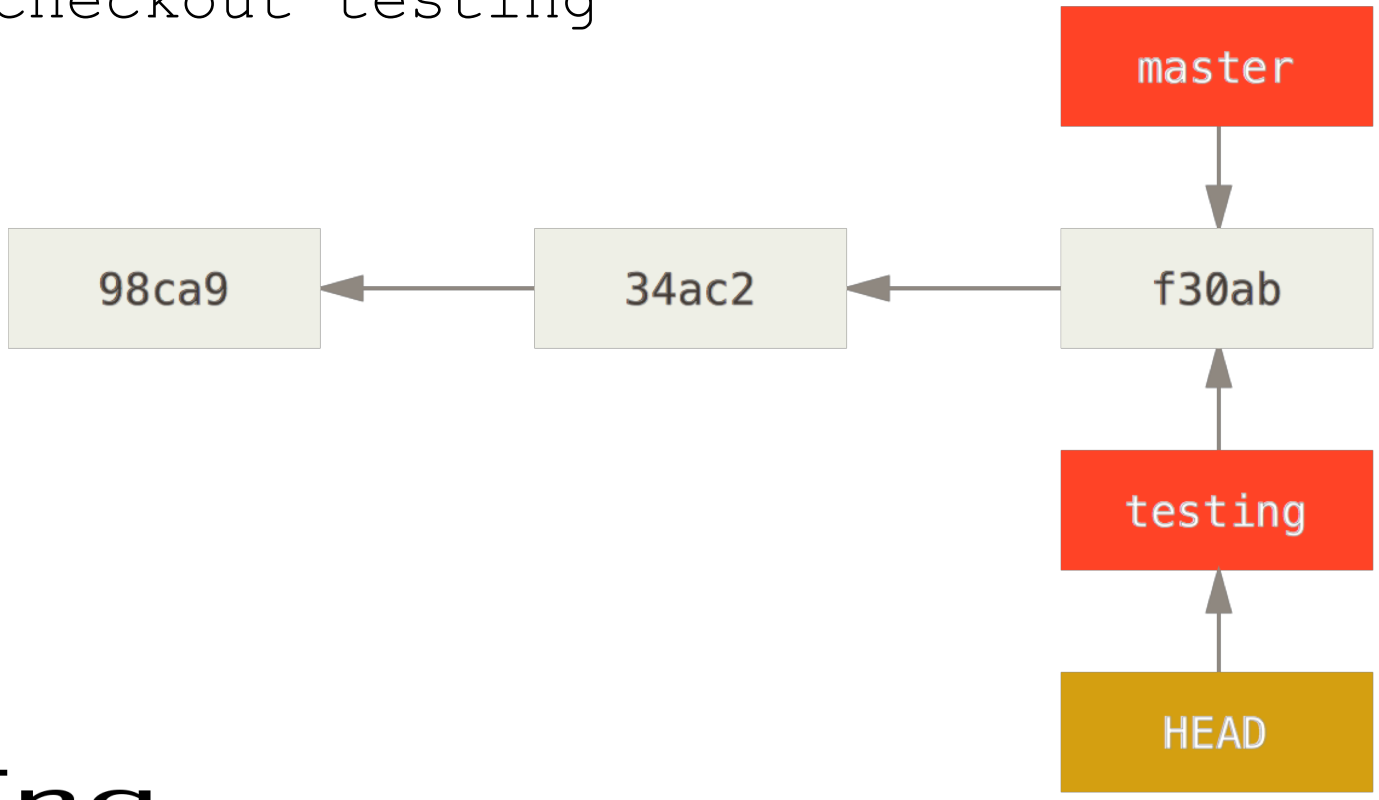
- Pointer to the current local branch



cc BY NC ND

Switching Branches

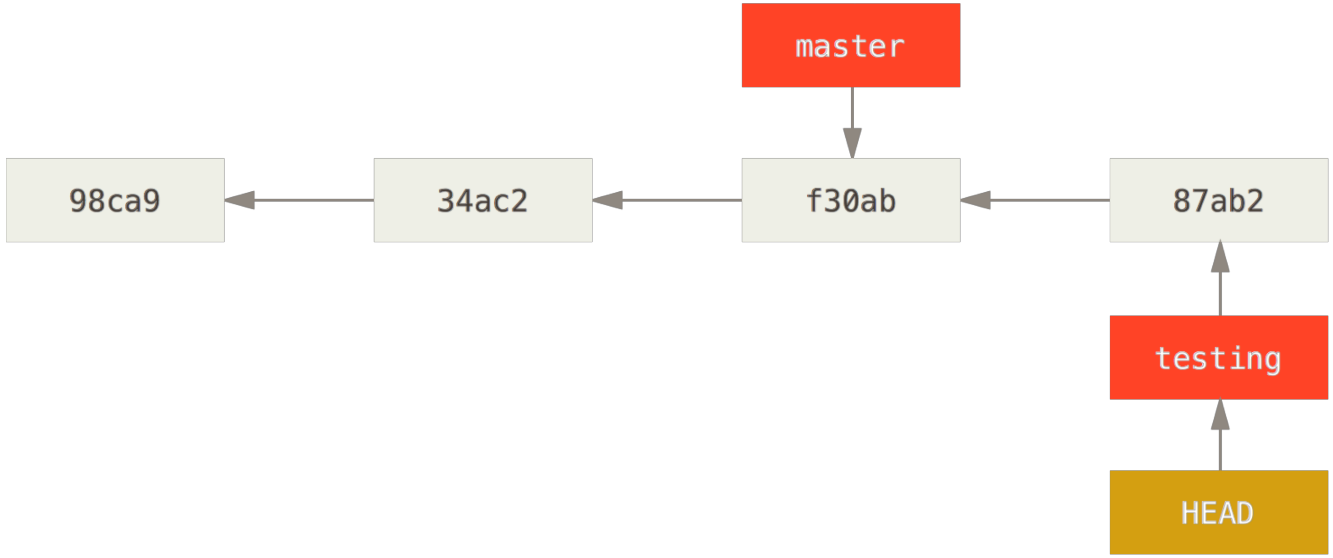
- `git checkout testing`



cc BY NC ND

Switching Branches

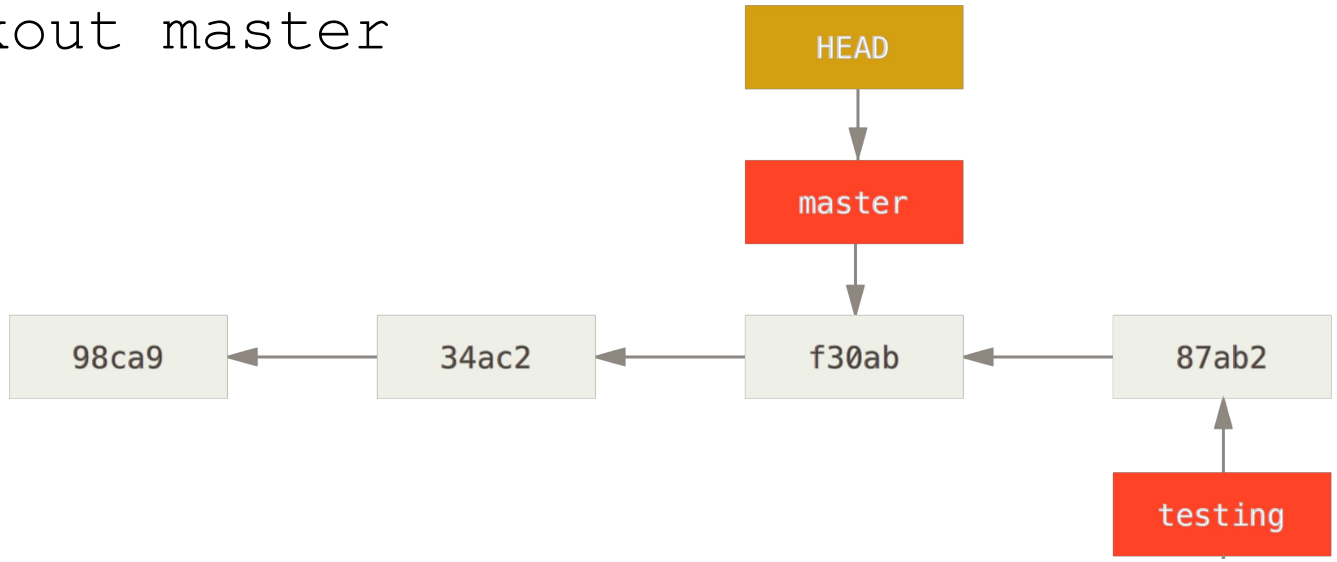
- After another commit



cc BY NC ND

Switching Branches

- `git checkout master`

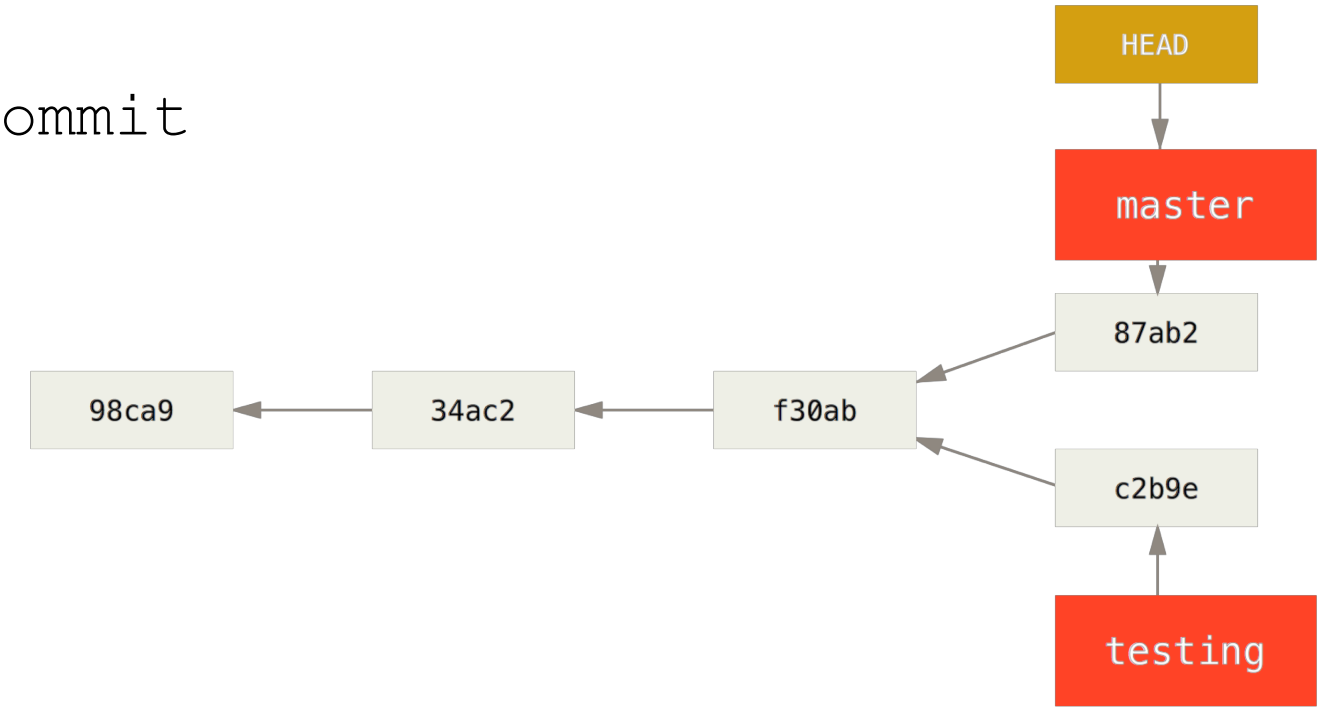


- Not only moves HEAD, also switches WC (files in .git folder now are the files of snapshot f30ab)
-

CC BY NC ND

Switching Branches

- `git commit`



- commits from this point on will diverge (‘divergent history’)

cc BY NC ND

Switching – git log

- Git log
  - Will show only one flow of commits (master)
- Git log testing
  - Will show flow of commits for ‘testing’ branch
- Git log –all
  - Will show all flows

cc BY NC ND

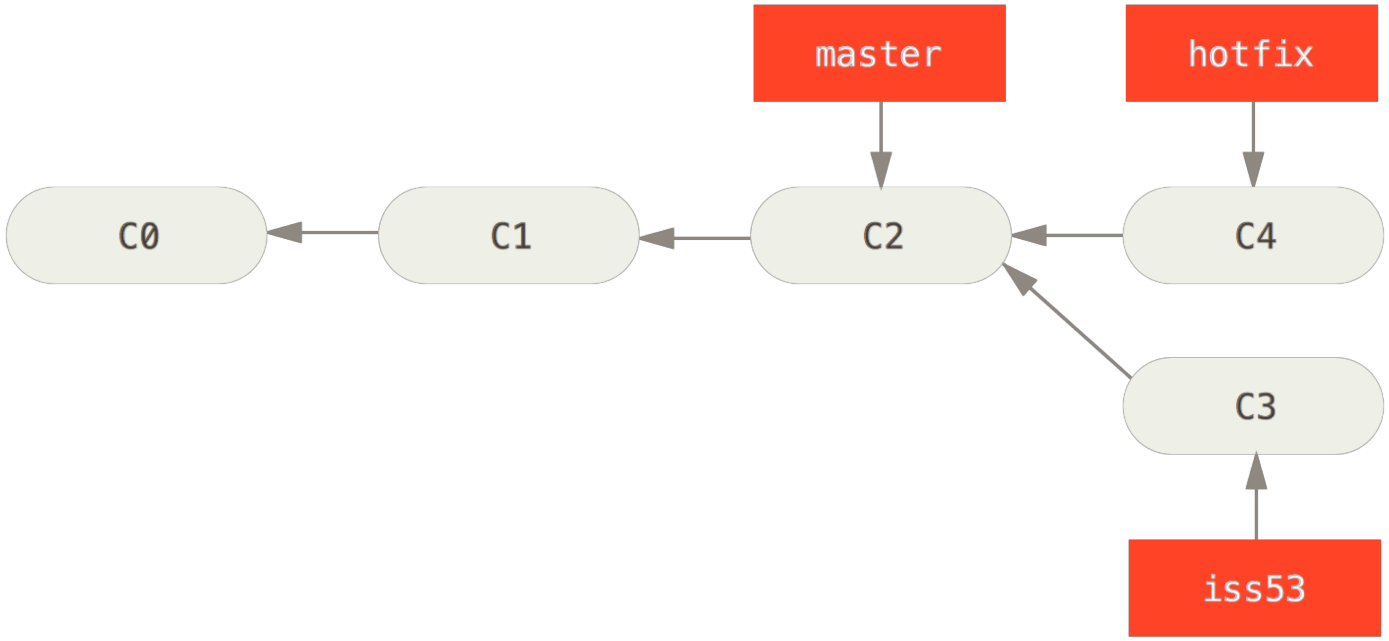
Merging branches

- Without divergent history
- With divergent history

cc BY NC ND

Merging Branches – no divergence

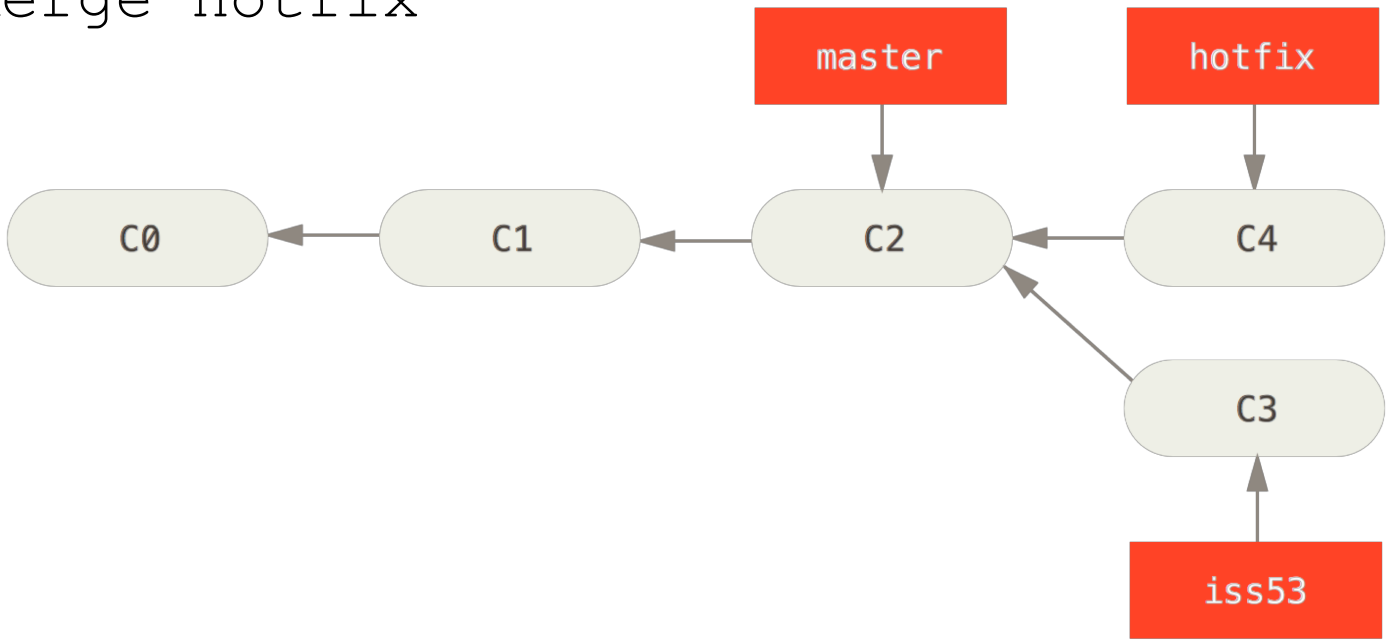
- Initial situation (HEAD → hotfix)



cc BY NC ND

Merging Branches – no divergence

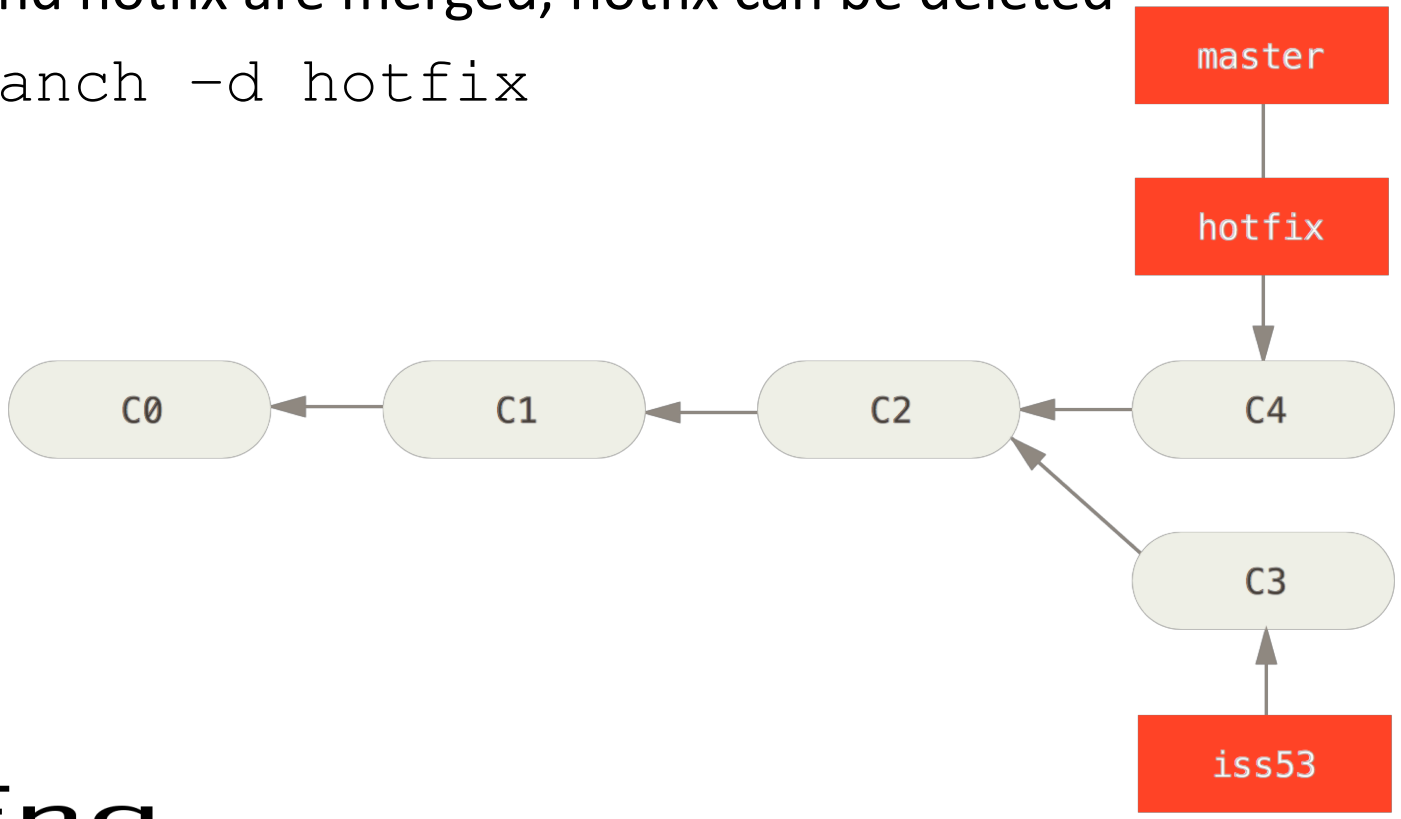
- Merging hotfix with master:
- `git merge hotfix`



cc BY NC ND

Merging Branches – no divergence

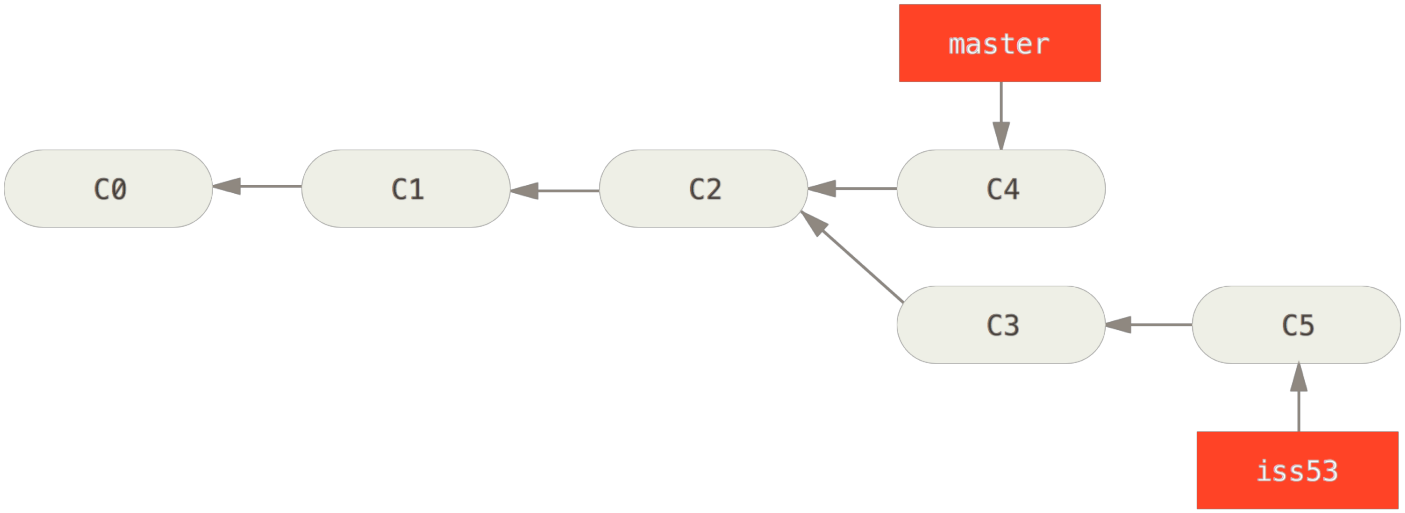
- Master and hotfix are merged, hotfix can be deleted
- `git branch -d hotfix`



cc BY NC ND

Merging Branches – with divergence

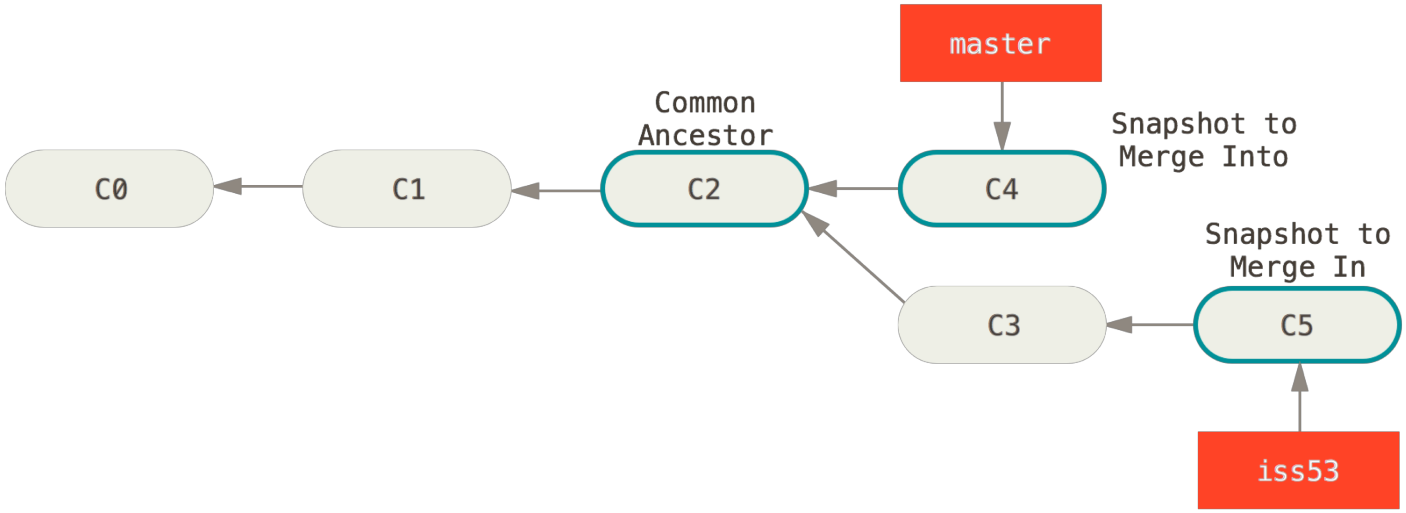
- New commit on iss53



cc BY NC ND

Merging Branches – with divergence

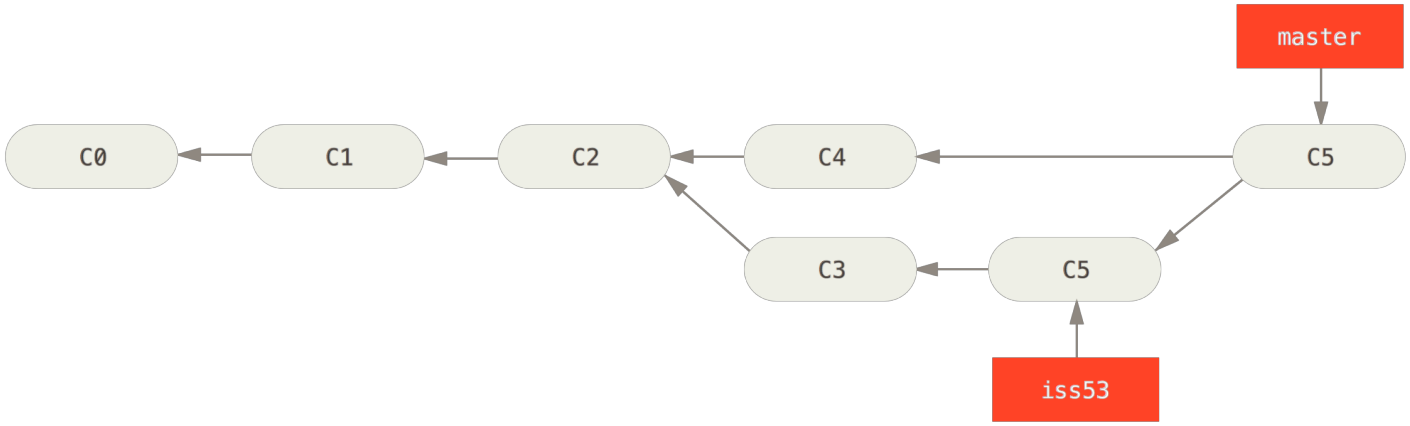
- `git checkout master`
- `git merge iss53`



cc BY NC ND

Merging Branches – with divergence

- Final result:



cc BY NC ND

Merging Branches – with divergence

- Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it
  - Three way: C4, C5, C2 (common ancestor)
- This is referred to as a merge commit, and is special in that it has more than one parent
- Git determines the best common ancestor to use for its merge base

cc BY NC ND

Merging - conflicts

- Conflicts are possible,
  - Ex same file modified in same part in C4 and C5
- in this case merging is not done until conflicts are managed

- `git mergetool`

  Helps in managing conflicts

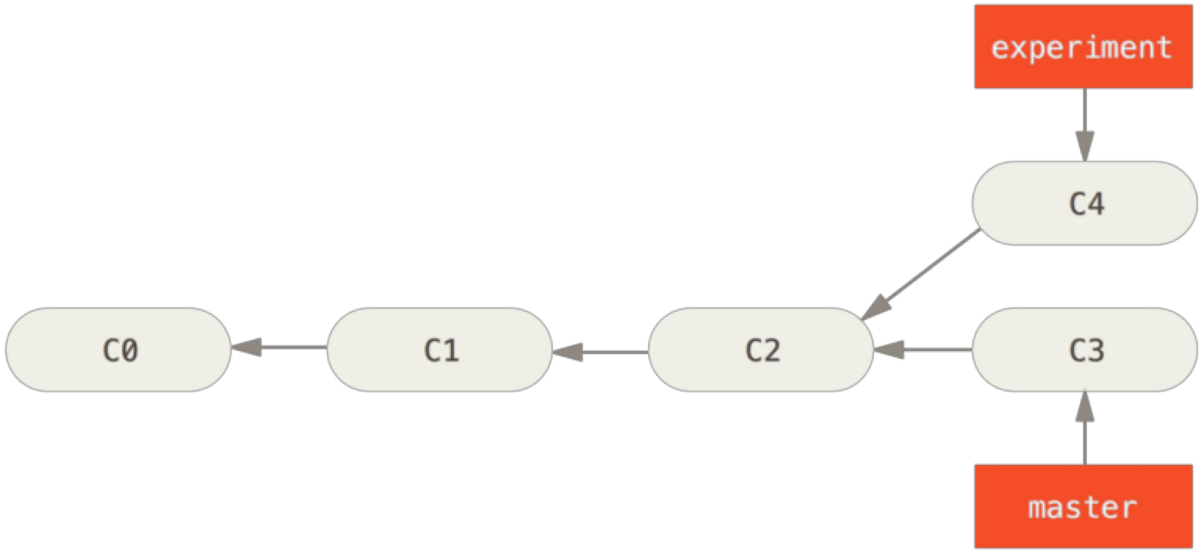
cc BY NC ND

Rebase

- Another way of merging
- Rebase applies changes of one snapshot history to another

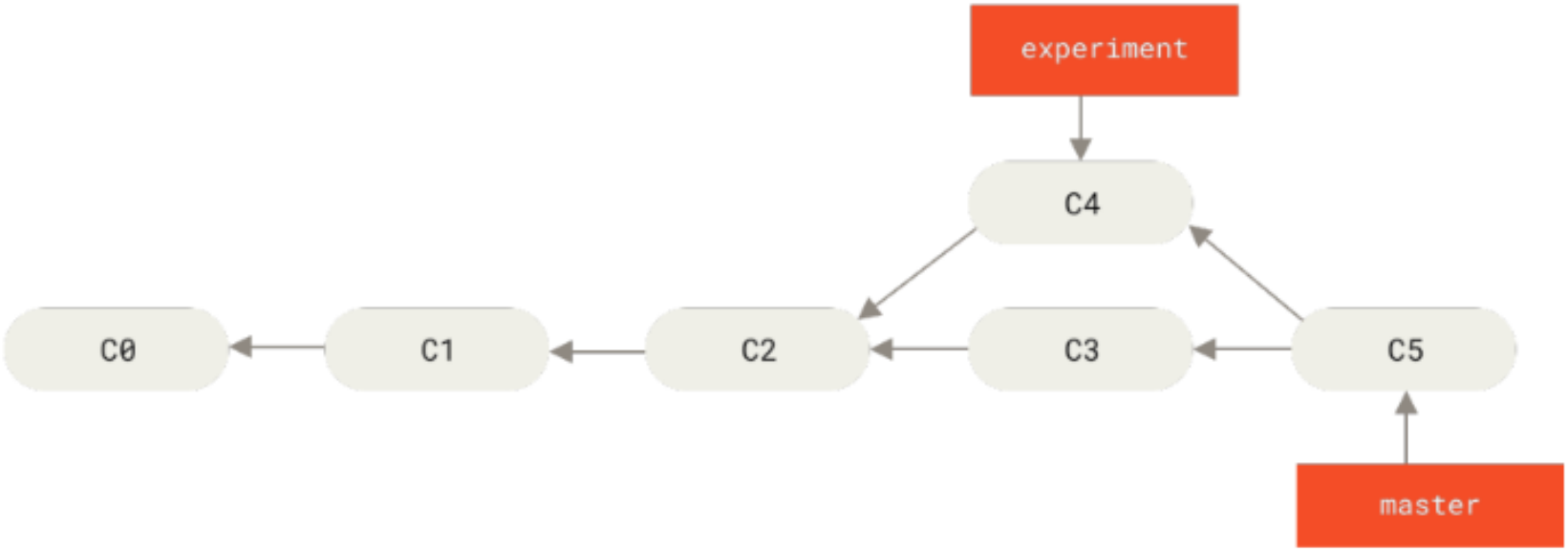
CC BY NC ND

- Starting point



CC BY NC ND

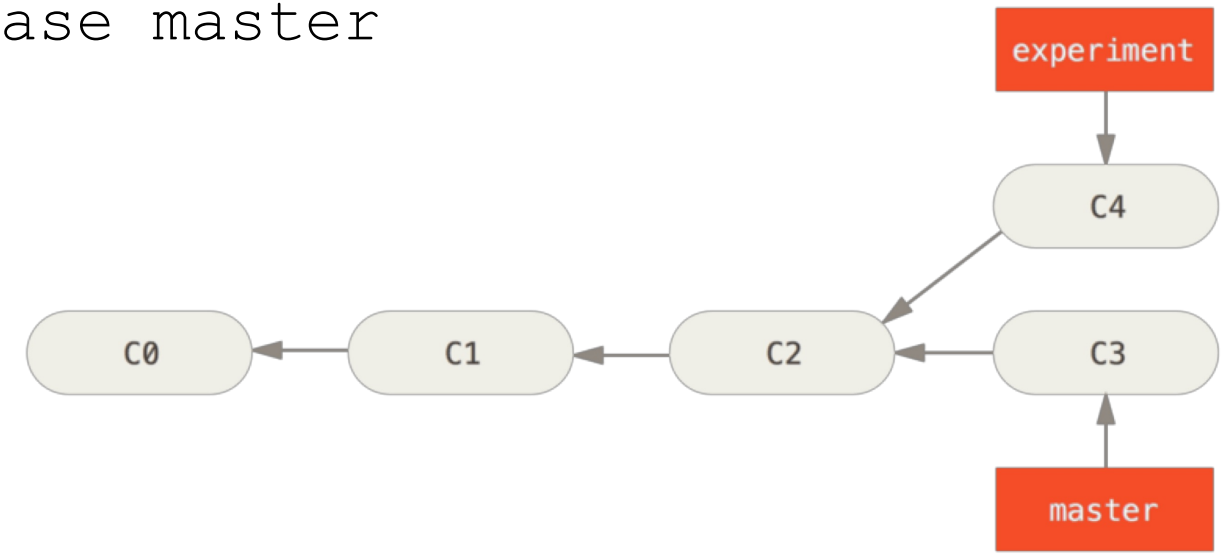
- Result
- `git checkout master`
- `git merge experiment`



cc BY NC ND

Rebase

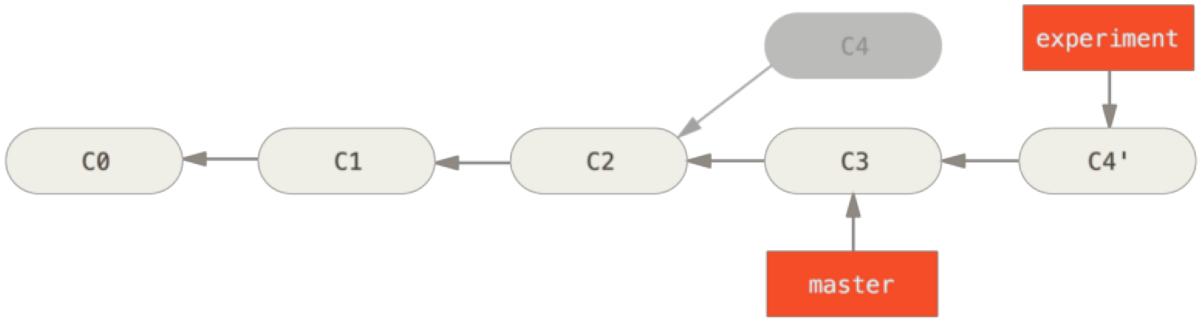
- `git checkout experiment`
- `git rebase master`



cc BY NC ND

Rebase

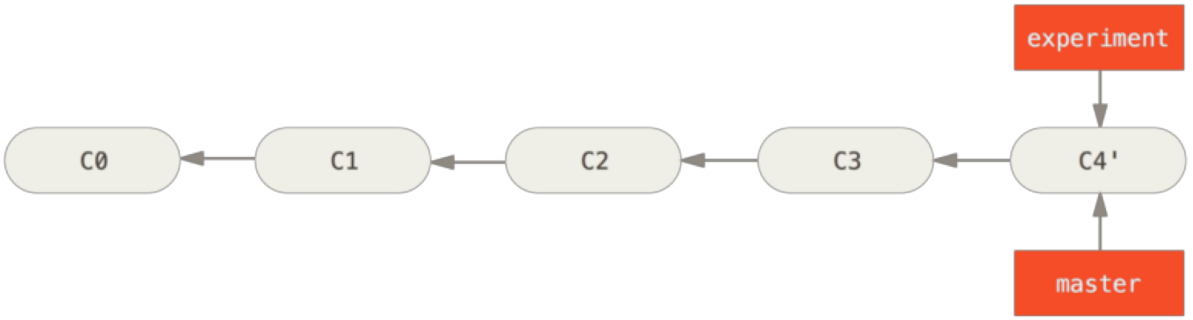
- `git checkout master`
- `git merge experiment`



cc BY NC ND

Rebase

- Result



- Be careful, rebase rewrites the repository history!

cc BY NC ND

Rebase

- It is also possible to organize/edit your commits
- Some useful cases:
  - the commit message is wrong, or it does not make sense.
  - the order of the commits is not nice regarding to git history.
  - there are more than one commit which make similar changes (or even the same thing).
  - a commit grouped a lot of different code, and it makes sense divide it in smaller commits.

cc BY NC ND

Rebase

- `git rebase -i HEAD~4`
  - `-i => interactive mode`
  - `~4 =>number of commits we want to target.`

cc BY NC ND

Changing Commit Order with Rebase

- `git rebase -i HEAD~4`

- `pick 0a0cf97 document`
- `pick d09e470` **`add paragraph`**
- `pick 59b2309` **`add second document`**
- `pick 16013c6 change title`

- Change the order of these lines (cut and paste)

CC BY NC ND

Changing Commit Order with Rebase

- `git rebase -i HEAD~4`

- `pick 0a0cf97 document`
- `pick 59b2309` **`add second document`**
- `pick d09e470` **`add paragraph`**
- `pick 16013c6 change title`

CC BY NC ND

Edit Commit Messages

- `git rebase -i HEAD~4`
- `pick 0a0cf97 document`
- `pick 59b2309 add second document`
- **`reword`** `d09e470 add paragraph`
- `pick 16013c6 change title`

- Change pick with reword for editing the commit message

CC BY NC ND

Merge Commits

- `git rebase -i HEAD~4`
- `pick 0a0cf97 document`
- **`squash`** `59b2309 add second document`
- `pick d09e470 add paragraph`
- `pick 16013c6 change title`

- Change **pick** with **squash** for merging the second commit with the first

CC BY NC ND

Merge Commits

- The result will be:

```
# This is a combination of 2 commits.  # The first commit's message is:
document
# This is the 2nd commit message:
add second document
# Please enter the commit message for your changes.
```

- Delete all this lines and write a message for the new commit

cc BY NC ND

Rebase vs. Merge

- The main differences between rebase and merge are:
  - merge combines the commit history of two branches, while rebase creates a linear commit history.
  - merge creates a new merge commit, while rebase rewrites the commit history of the feature branch.
  - merge is better suited for integrating changes from different branches, while rebase is better suited for keeping a cleaner commit history.

cc BY NC ND

Rebase vs. Merge

- merge is generally considered safer than rebase because it preserves the original commit history and creates a new merge commit, which makes it easier to undo the merge if necessary.
- Graph structure: Every merge commit increases the connectivity of the commit graph by one. A rebase, by contrast, does not change the connectivity and leads to a more linear history

CC BY NC ND

Models for Collaborative Development

- People collaborate on GIT in two ways:
  - Shared repository
  - Fork and pull

cc BY NC ND

Shared Repository

- With a shared repository, individuals and teams are explicitly designated as contributors with read, write, or administrator access.

- This simple permission structure, combined with features like protected branches and Marketplace, helps teams progress quickly when they adopt GitHub.

CC BY NC ND

Fork and Pull

- For projects to which anyone can contribute, managing individual permissions can be challenging, but a fork and pull model allows anyone who can view the project to contribute.
- A fork is a copy of a project under a developer's personal account.
- Every developer has full control of their fork and is free to implement a fix or new feature.
- Work completed in forks is either kept separate or is surfaced back to the original project via a pull request.

CC BY NC ND

Fork

- A fork is a copy of a repository that a developer manages.
- Forks let the developer make changes to a project without affecting the original repository.
- The developer can fetch updates from or submit changes to the original repository with pull (or merge) requests.

CC BY NC ND

Pull (GitHub) or Merge (GitLab) requests

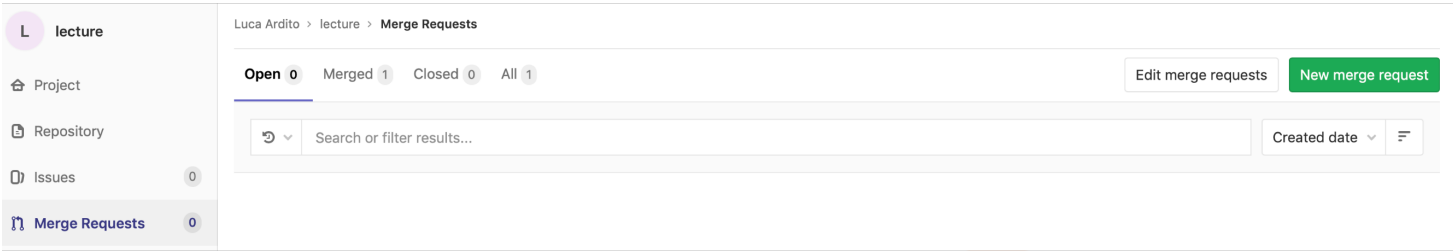
- Pull requests let the developer tell others about changes he or she has pushed to a branch in a repository on git.

- Once a pull (or merge) request is opened, the developer can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.

CC BY NC ND

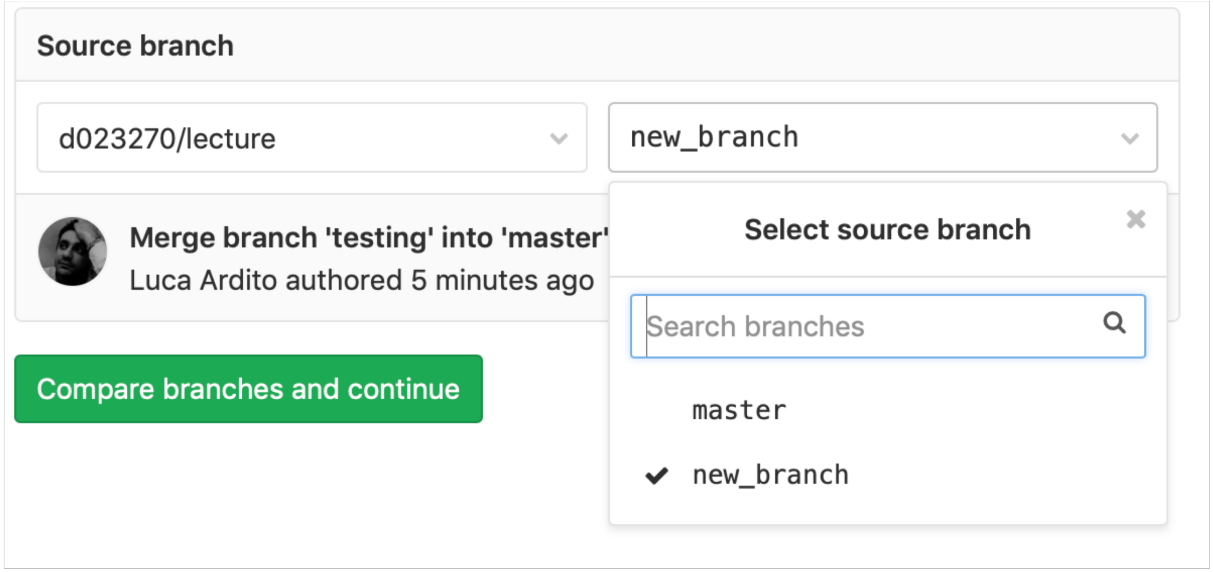
Merge Requests

- Create a new branch
- Work on the new branch
- Commit and push your work
- Open the project on Gitlab and create a new merge request



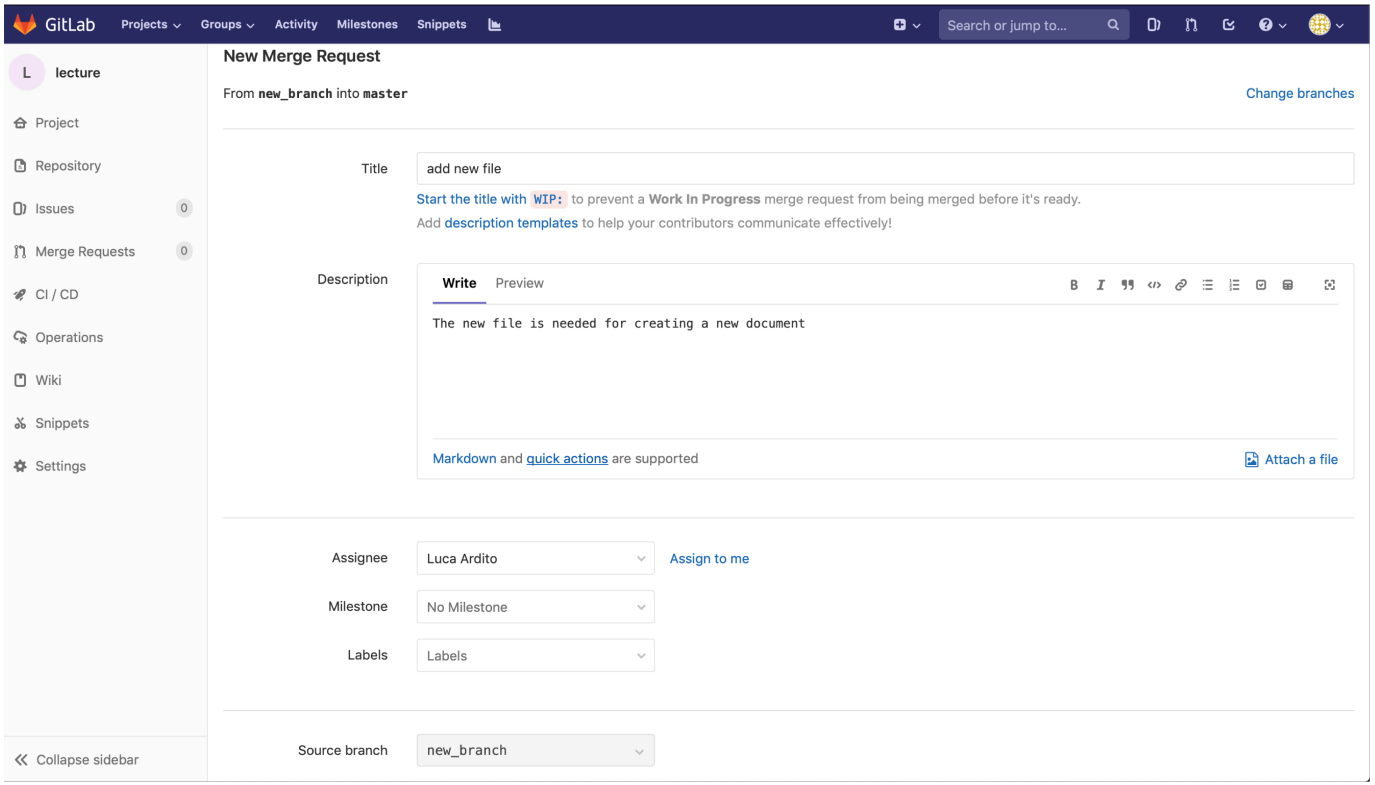
cc BY NC ND

Merge Requests



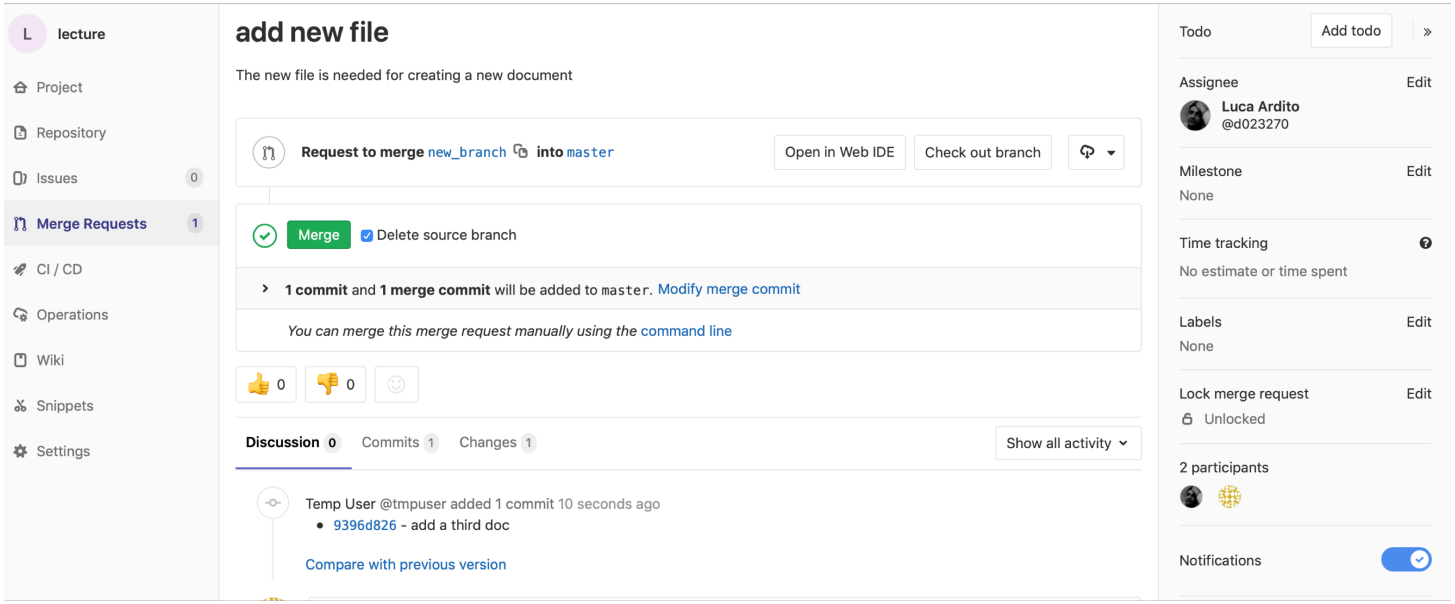
CC BY NC ND

Merge Requests



BY NC ND

Merge Requests



BY NC ND

What to do when you think you are lost

DON’T PANIC!

`git reflog` is your friend!

CC BY NC ND

What to do when you think you are lost

- Git will try to preserve your changes.

- Uncommitted changes to git-controlled-files will only get overwritten if running one of the commands:
  - `git checkout <file-or-directory>`
  - `git reset --hard`
- And of course any non-git commands that change files

- Files unknown to Git will only get lost with:
  - `git clean`
  - any non-git commands that change files

CC BY NC ND

Conventional Commits

- Lightweight convention on top of commit messages
- Provides an easy set of rules for creating an explicit commit history
  - `feat: allow provided config object to extend other configs`
  - `feat!: send an email to the customer when a product is shipped`
  - `BREAKING CHANGE: \`extends\` key in config file is now used for extending other config files`
- https://www.conventionalcommits.org/en/v1.0.0/#specification

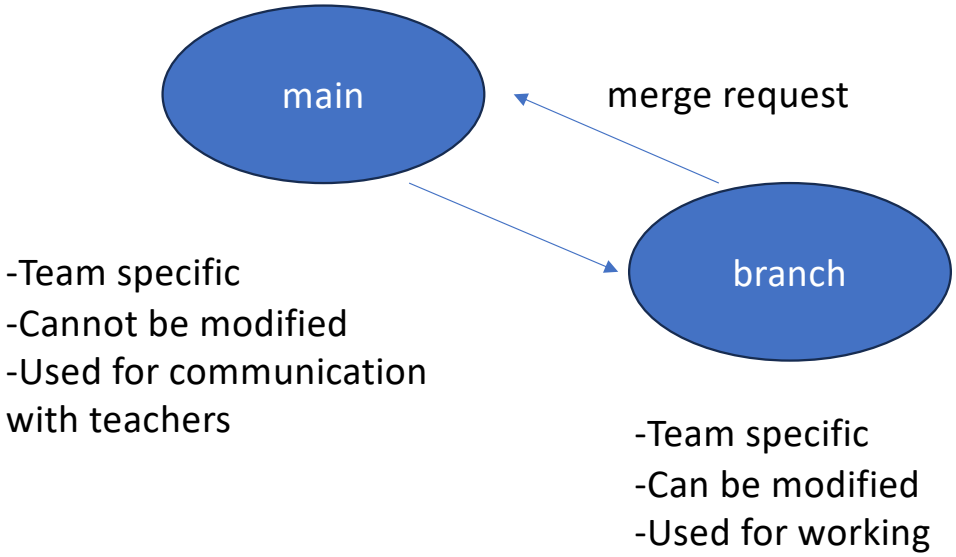
CC BY NC ND

Recorded Examples

https://www.youtube.com/watch?v=UMcXoJbBGRI

cc BY NC ND

How to use git in project



- Each team has a repo
- ‘Main’ branch, ‘branch’

- Team works in ‘branch’
- Deliveries are made via merge requests vs ‘main’
- To be accepted by teachers

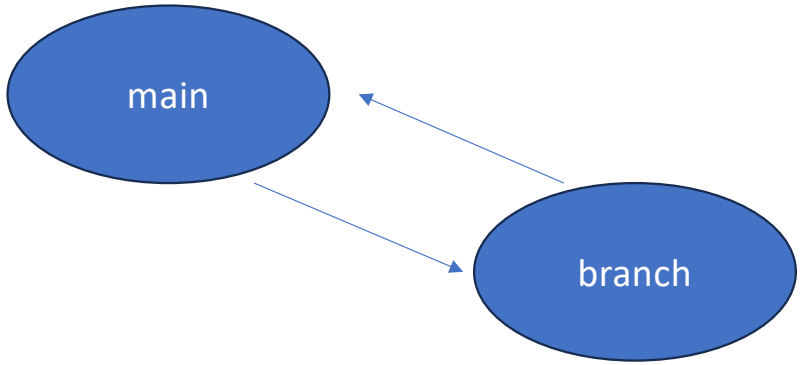
CC BY NC ND

Rules

- No changes to main
- Project deliveries must refer to one branch only
  - If you define many branches below ‘branch’ merge them before doing merge request

cc BY NC ND

How to use git



- Options in your branch
  - Do other branches
    - Per person
    - Per task
  - No other branches

cc BY NC ND

How to use git

- Branching should be used wisely
  - Need to merge at reasonable intervals to check consistency
- Suggestions
  - Single documents (ex requirements): no branches, all team members work on same file
  - Group of related files (ex .js files of code part, test part, GUI prototype):
    - Either no branches, each member works on a different file
    - Or, branch for each team member, then merge the branches

CC BY NC ND

Git Additional Resources

- Reference guide
  - http://git-scm.com/doc
- Book
  - https://git-scm.com/book/en/v2
- GUI tools for git
  - https://git-scm.com/downloads/guis

CC BY NC ND