Software Engineering

Configuration Management with Git

SoftEng
http://softeng.polito.it

Version 1.0

Configuration Management

Configuration Management System (CMS)

- System that records changes to a file or set of files over time so that you can recall specific versions later

CMS Functionalities

- Revert files back to a previous state
- Revert the entire project back to a previous state
- Compare changes over time
- Monitor who last modified something that might be causing a problem
- Monitor who introduced an issue and when

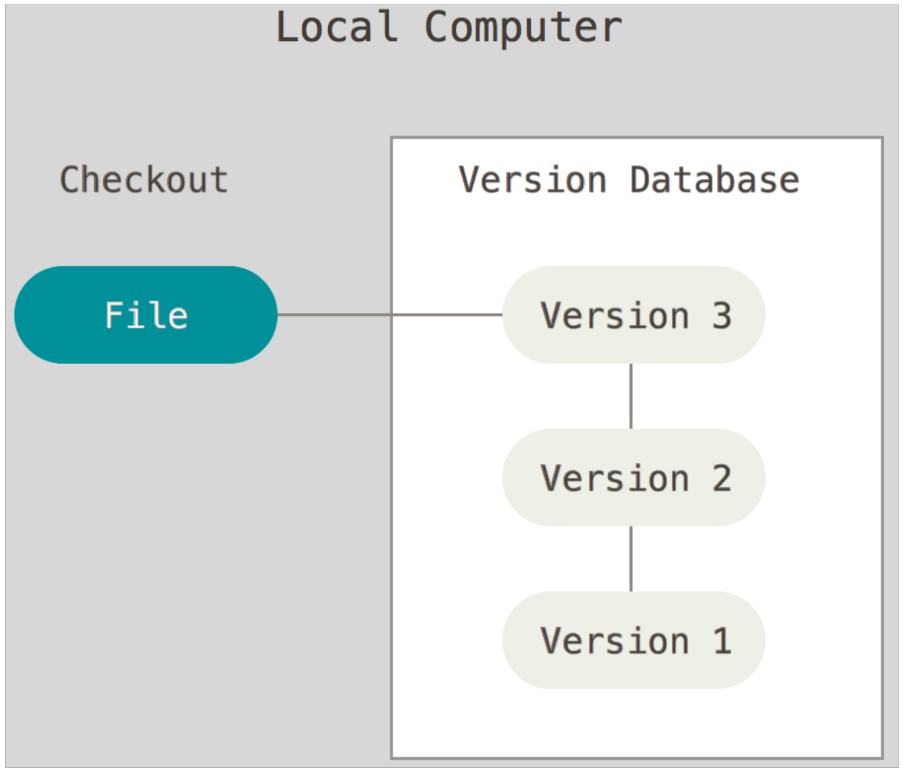
CMS Taxonomy

- Local CMS
  - A simple database that kept all the changes to files under revision control
- Centralized CMS
  - A single server that contains all the versioned files, and several clients that check out files from that central place
- Distributed CMS
  - Clients fully mirror the repository

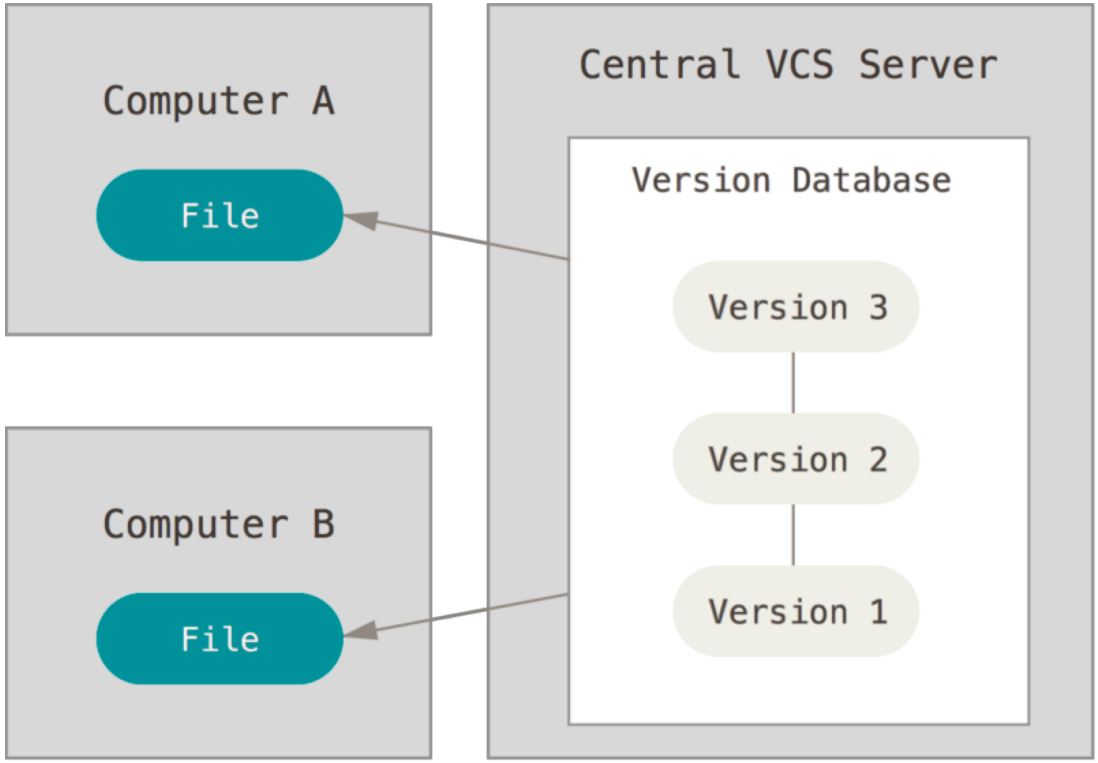
Examples

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

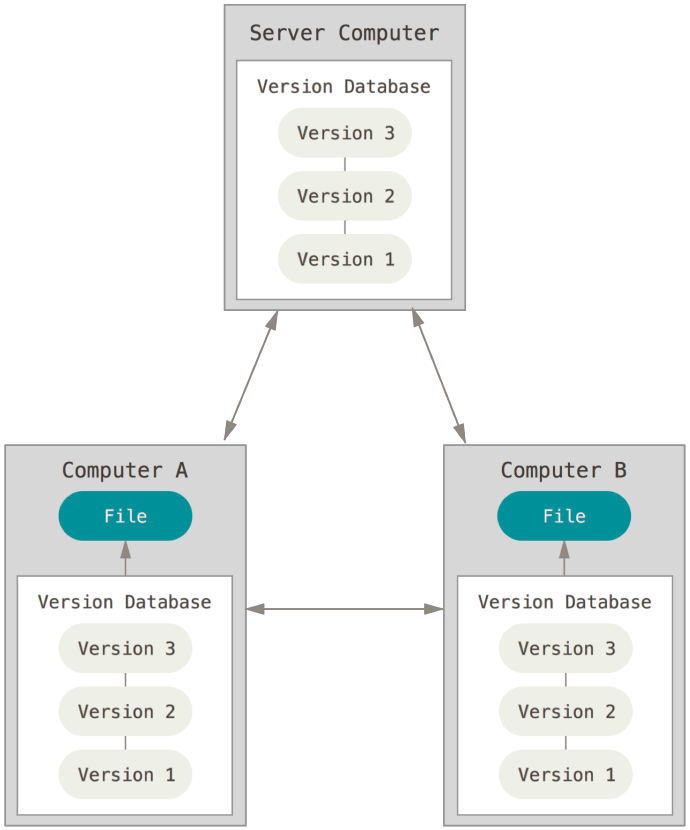
Local CMS



Centralized CMS



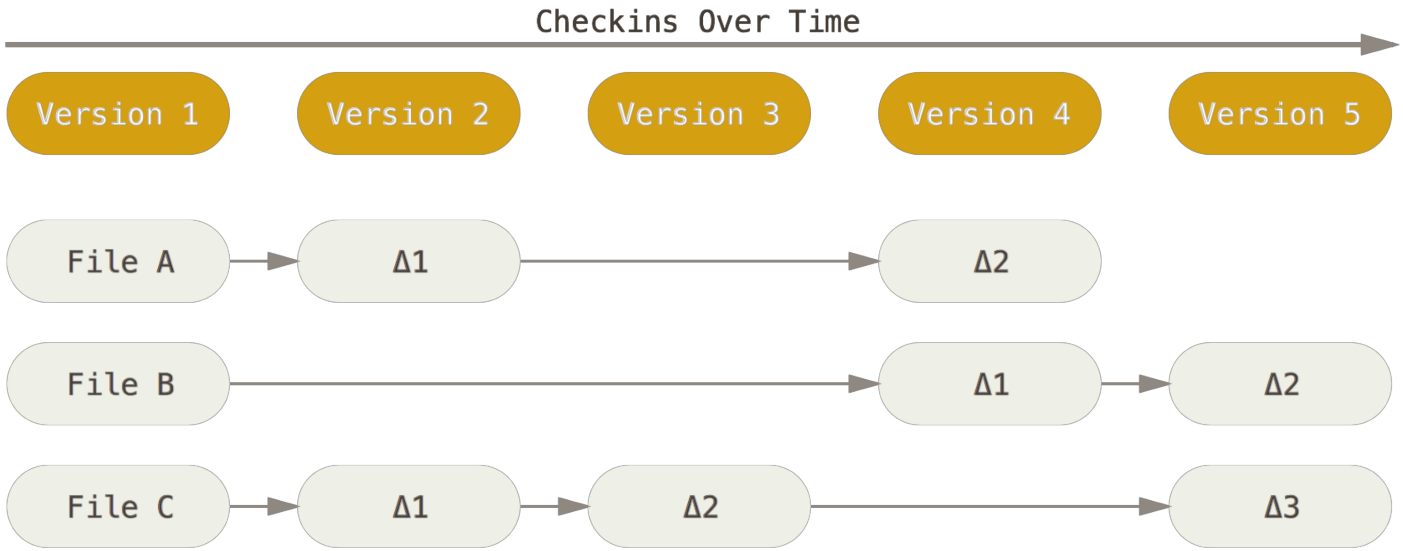
Distributed CMS



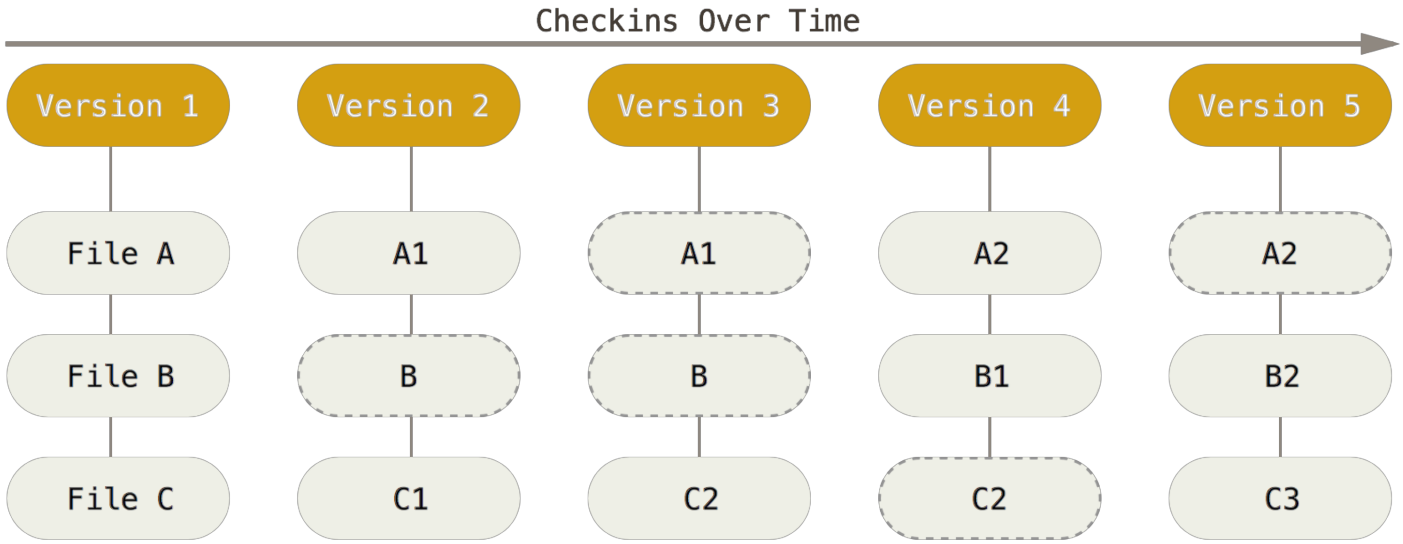
Data Management Models

- Differences
  - Information is kept as a set of files and the changes made to each file over time.
- Snapshots
  - Every commit, a picture of what all your files look like at that moment is taken and the system stores a reference to that snapshot
  - If files have not changed, system stores just a link to the previous identical file

Differences



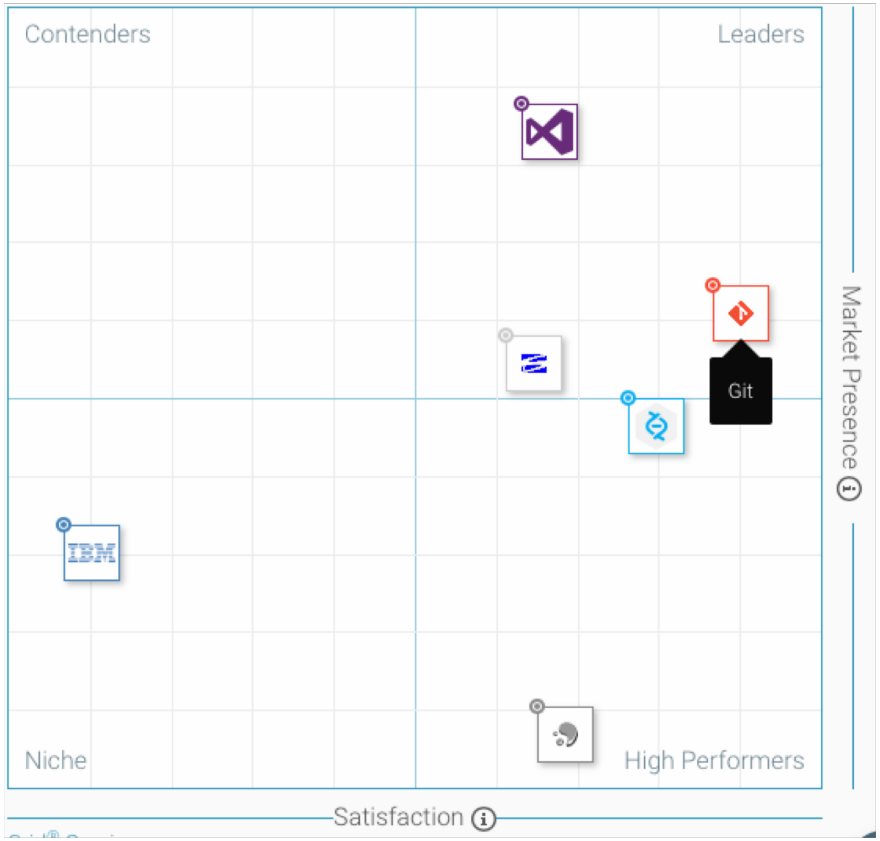
Snapshots



Introduction to Git

SoftEng
http://softeng.polito.it

Why Git?



GitHub

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

GitLab

- GitLab is a web-based DevOps lifecycle tool that provides a Git-repository manager providing wiki, issue-tracking and CI/CD pipeline features, using an open-source license

Git

- Distributed CMS
- Uses snapshots
- Exploits local operations
  - Generally, no information is needed from another computer on your network
- Has integrity
  - Everything is check-summed before it is stored and is referred to by that checksum
  - No untracked change of any file or directory

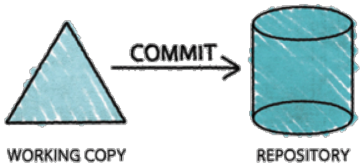
Basic Concepts: Repository

- Place where you store all your work
- It contains every version of your work that has ever existed
  - files
  - directories layout
  - history
- It can be shared by the whole team

Basic Concepts: Working Copy

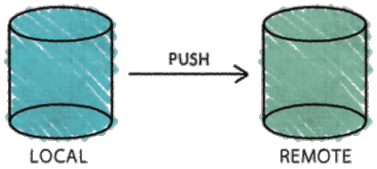
- It is a snapshot of the repository where the changes happen
- It is private, not shared by the team
- It also contains some metadata so that it can keep track of the state of things
  - has a file been modified?
  - is this file new?
  - has a file been deleted?

Basic Concepts: Commit



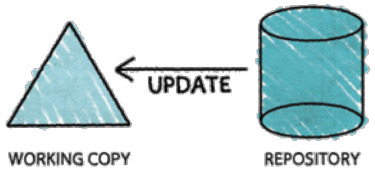
- Modifies the repository
- It is atomically performed by modern version control tools
  - The integrity of the repository is assured
- It is typical to provide a log message (or comment) when you commit
  - to explain the changes you have made
  - the message becomes part of the history of the repository

Basic Concepts: Push



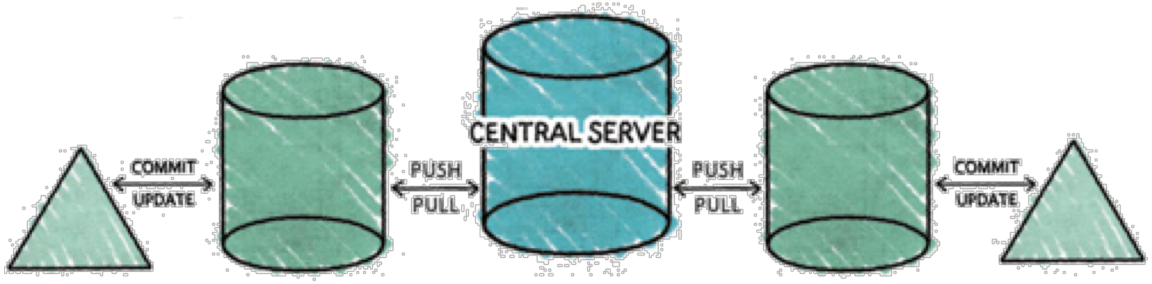
- Pushes copy changesets from a local repository instance to a remote one
  - synchronization between two repository instances

Basic Concepts: Update



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

Example

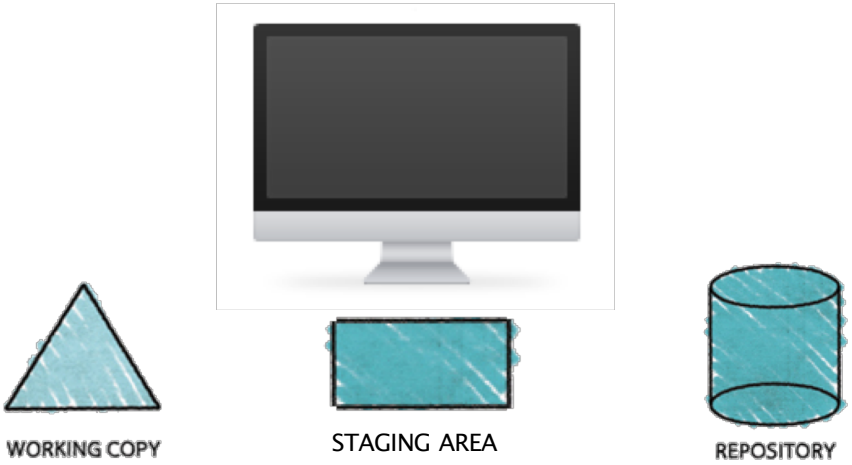


- Marco and Dave must do the Software Engineering course project.
- They use Git for version control.

Git

- Only adds data
- Every time adds a new snapshot
- Three states
- Committed: data is safely stored in your local database
- Modified: changed the file but have not committed it to your database yet
- Staged: a modified file is marked in its current version to go into your next commit

Staging Area



- A sort of loading dock
- It can contain things that are neither in the working copy nor in the repository instance
- Also called “index”

Excluding Files from Version Control

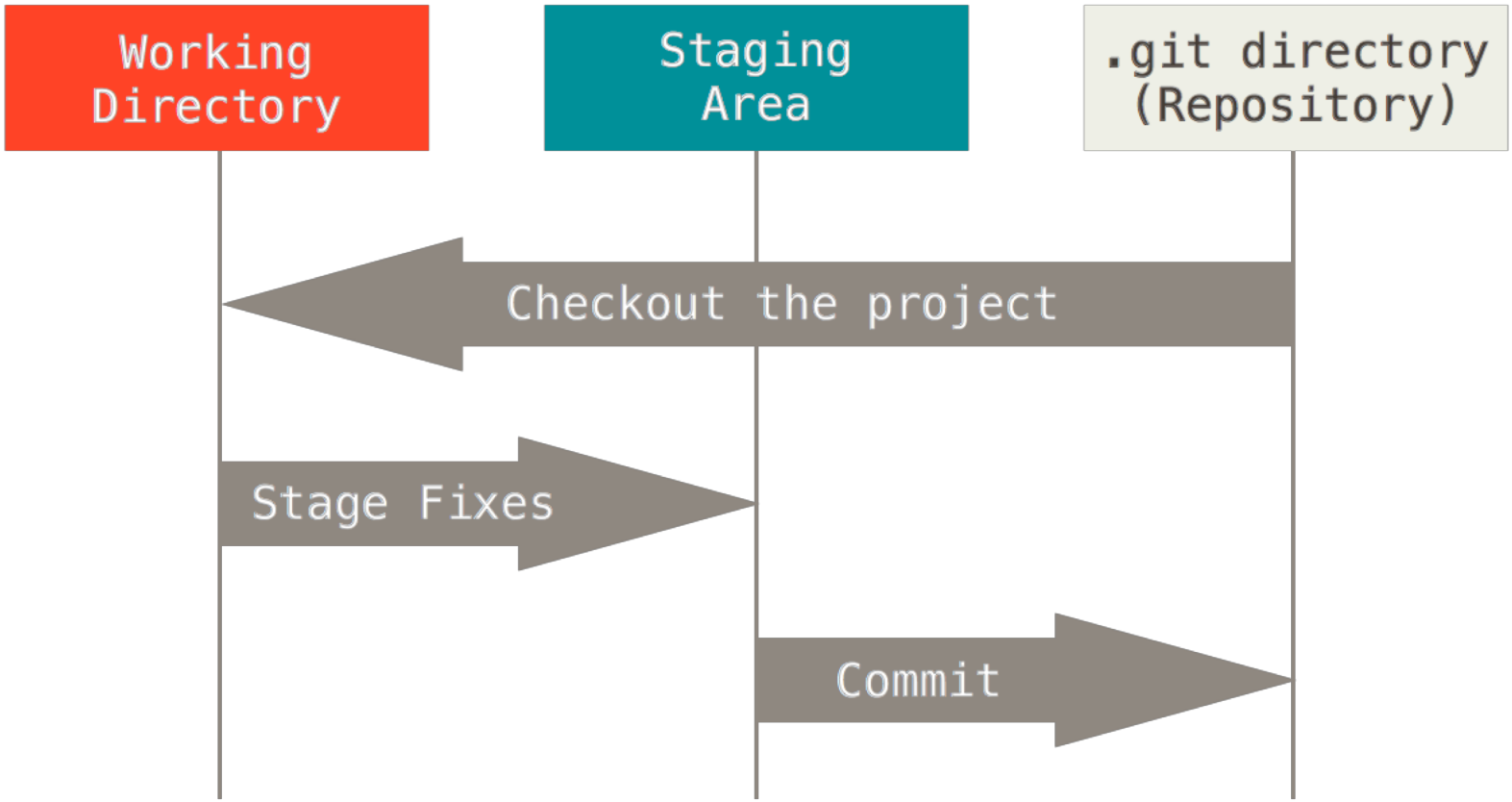
- It is possible to exclude permanently from version control some files in the project folder

- These files have to be listed in the .gitignore file so that such files and folders will not be staged

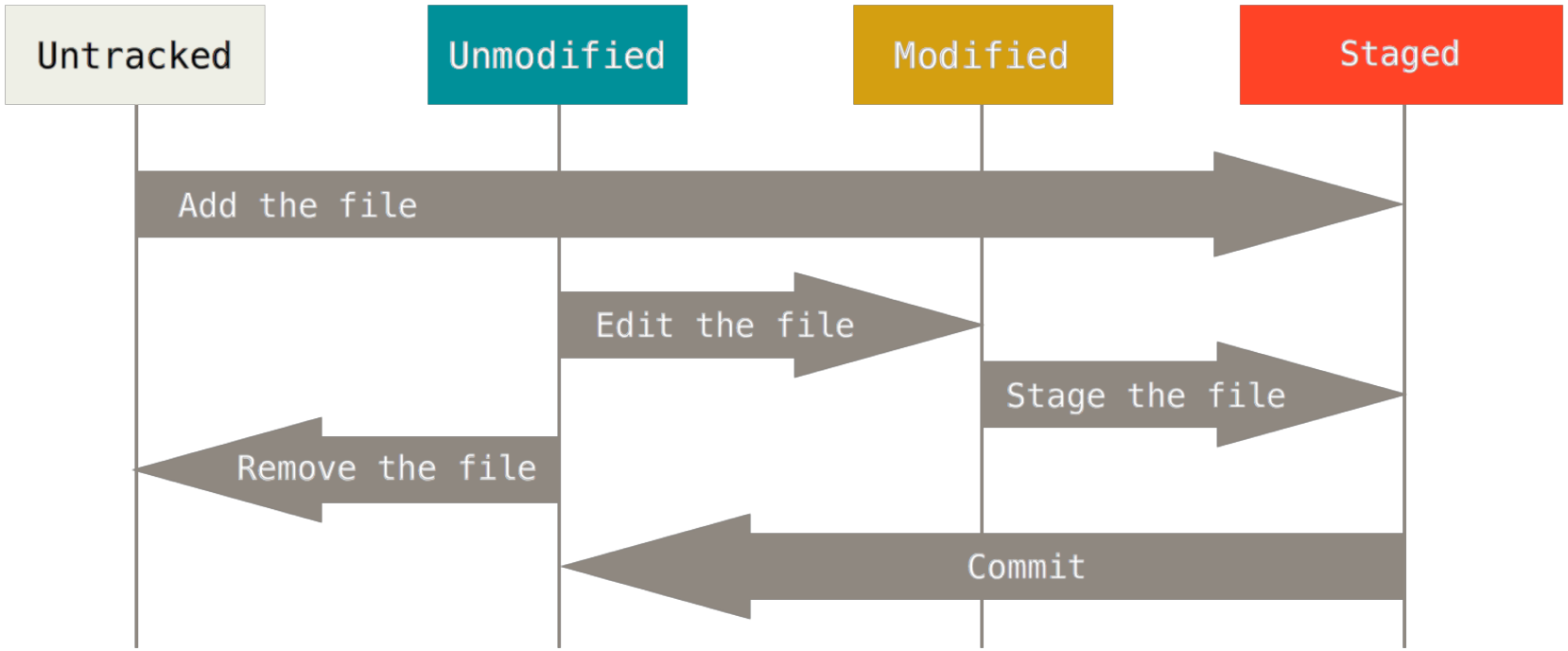
Typical Workflow

1. Modify files in your working directory
2. Stage the files, adding snapshots of them to your staging area
3. Do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

Typical Workflow



Files Lifecycle



The Narrative Metaphor

- In many respects the commit history we create with Git is a narrative that tells us (and others) how the code evolved to its current state.
- Committing changes has a lot in common with telling a story, and that story can be interesting or boring, it can be presented in a logical way or totally confusing, even if the final code in both cases is the same.
- While there are different styles of telling the story well, a badly told narrative will make us all suffer. So please think about the logical order in which your changes make most sense and formulate and format your log messages appropriately

Atomicity

- Git commands tend to be focused on one task.
- Therefore, what the user perceives as one logical step may require two or three consecutive command calls.
- This helps in understanding what you are doing, and when something goes wrong you know where exactly the problem occurred.

Git Basic Commands

Basic Commands

- `git init`
  - Initializes an empty Git repository in the current folder
  - It 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

Recording Changes

- `git status`
  - Determine which files are in which state
- `git add`
  - Track a new file
- `git diff`
  - Know exactly what you changed
- `git commit`
  - Commit changes

Recording Changes

- `git rm`
  - Remove a file from your tracked files and also from your working directory
- `git mv`
  - Rename a file
  - Git doesn't explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file

Viewing the Commit History

- `git log`
  - look back to see what has happened in a repository

Working with Remotes

- `git pull`
  - Automatically fetch and then merge a remote branch into your current branch
- `git push`
  - Share the changes, i.e. transfer them on the server

In practice

```
echo "# tmp" >> README.md
git init
git add README.md
git commit –m "first commit"
git remote add origin git@github.com:user/tmp.git
git push –u origin master
```

Git Branching

Data Storage in Git

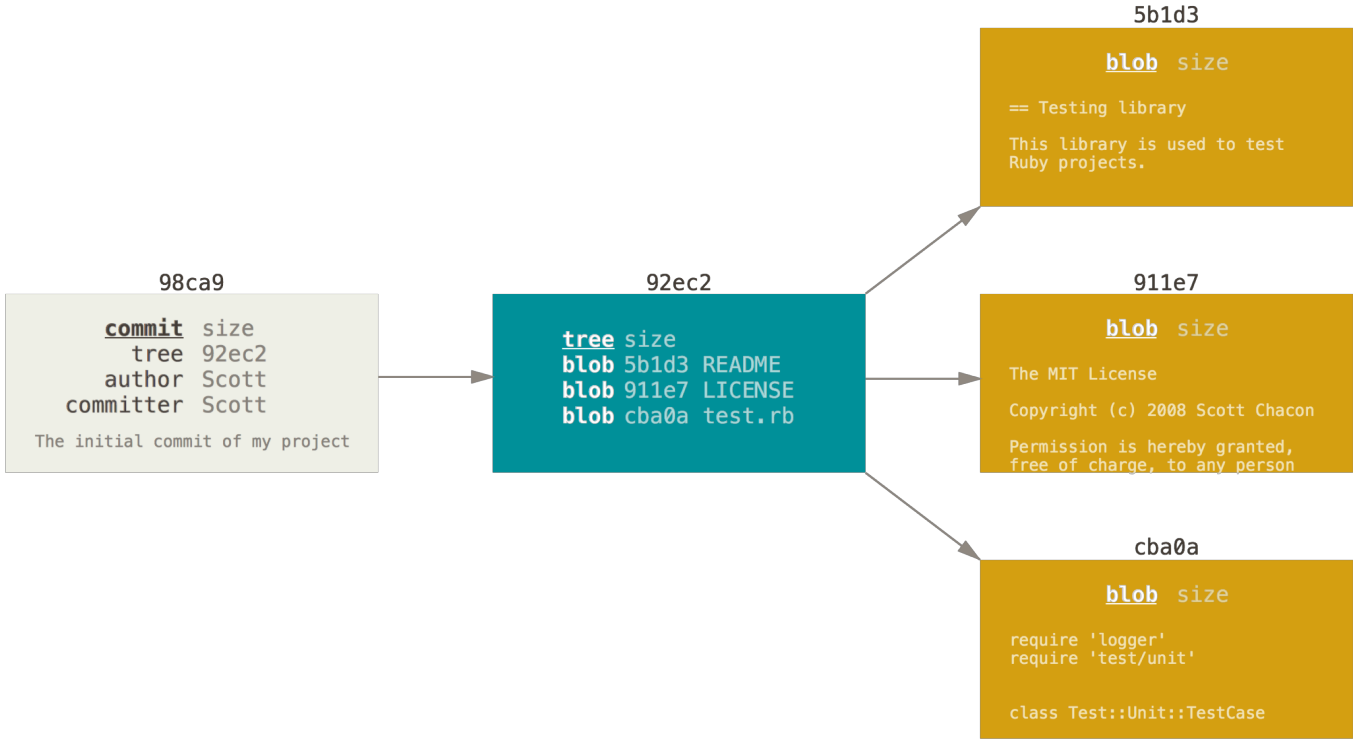
- Git stores a commit object that contains
  - a pointer to the snapshot of the content you staged
  - pointers to the commit or commits that directly came before this commit (its parent or parents)
- Staging the files
  - checksums each one
  - stores that version of the file (blobs)
  - adds that checksum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

Example

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

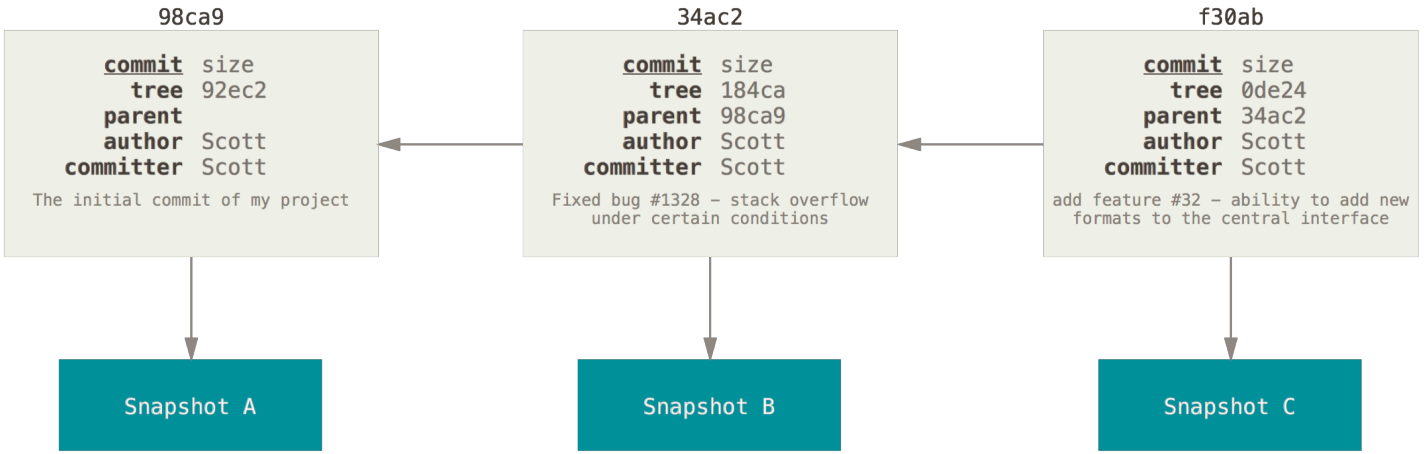
- Git repository now contains five objects
  - one blob for the contents of each of the three files
  - one tree that lists the contents of the directory and specifies which file names are stored as which blobs
  - one commit with the pointer to that root tree and all the commit metadata

Example



Example

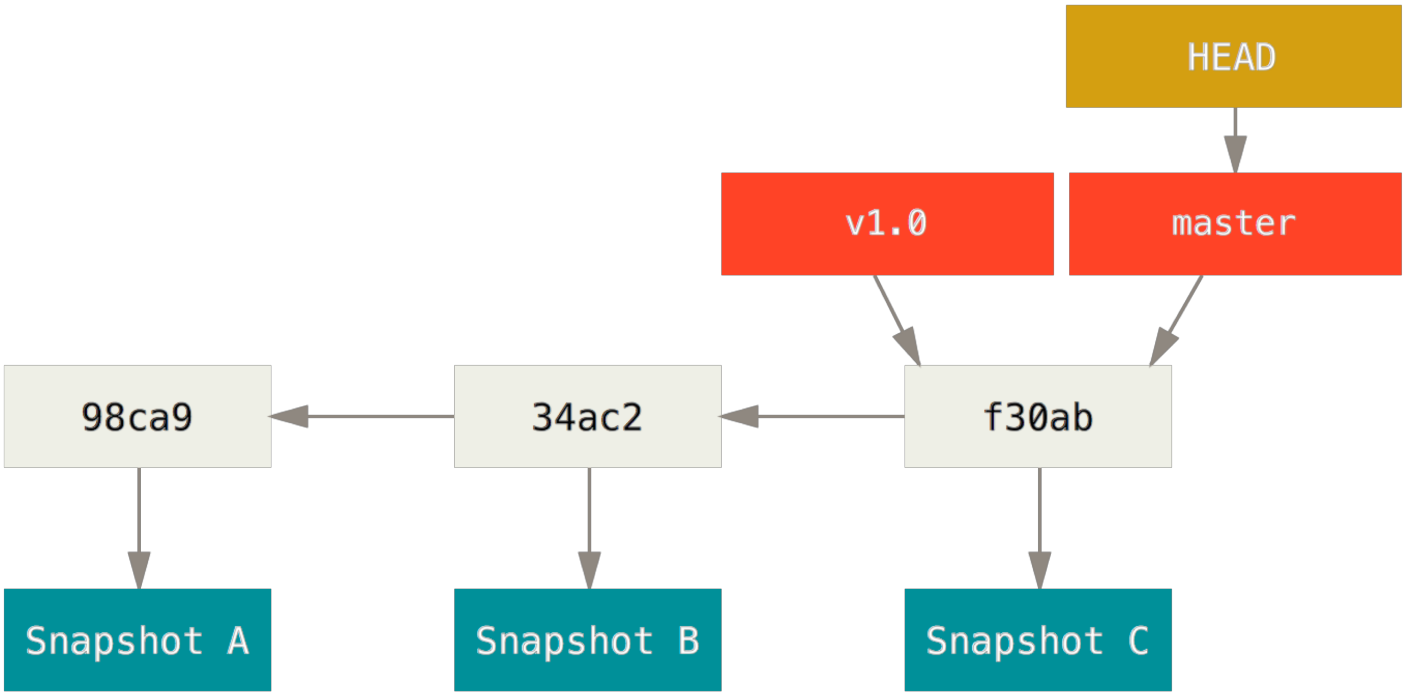
- The next commit stores a pointer to the commit that came immediately before it



Branches

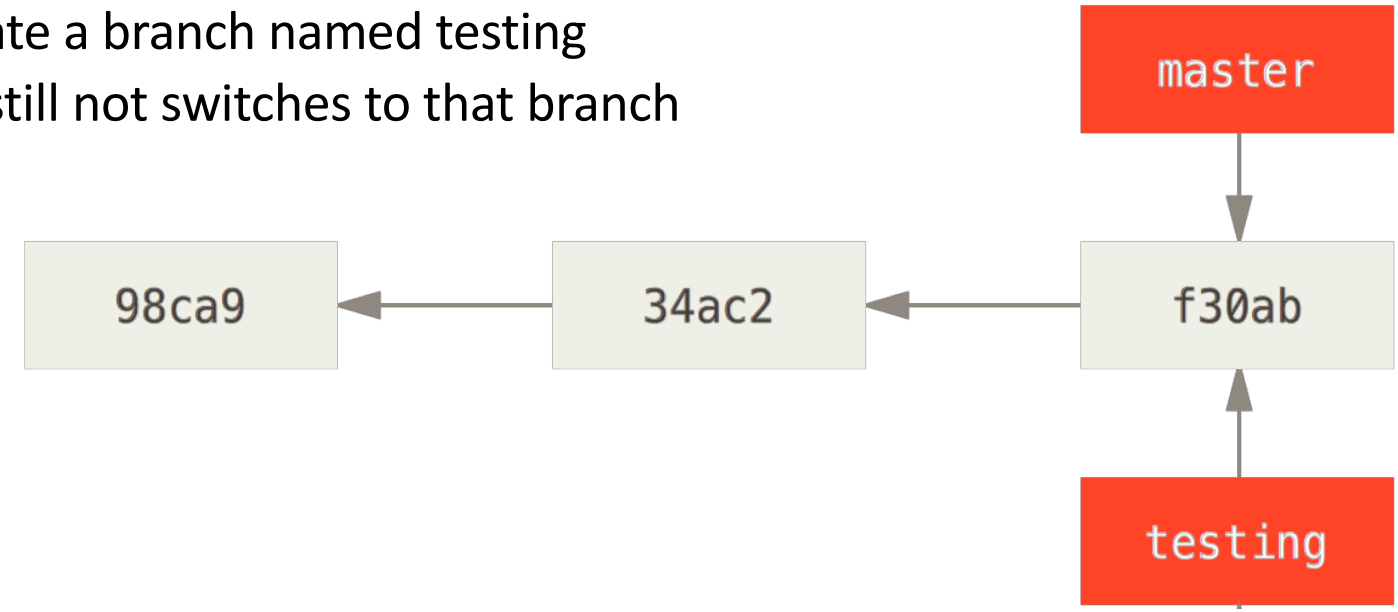
- A branch in Git is simply a lightweight movable pointer to one of these commits
- The default branch name in Git is master
- Every commit, it moves forward automatically

Branches



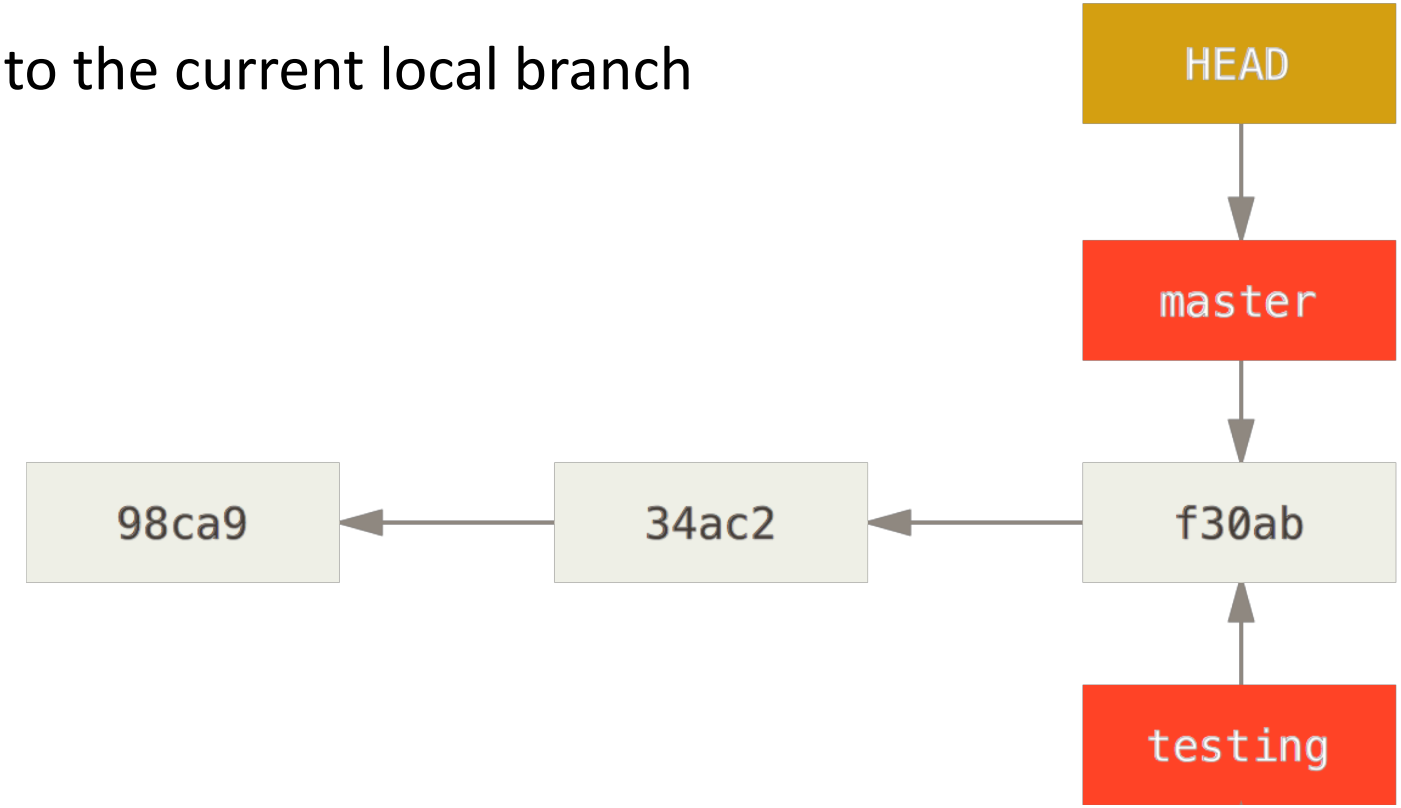
Creating a Branch

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



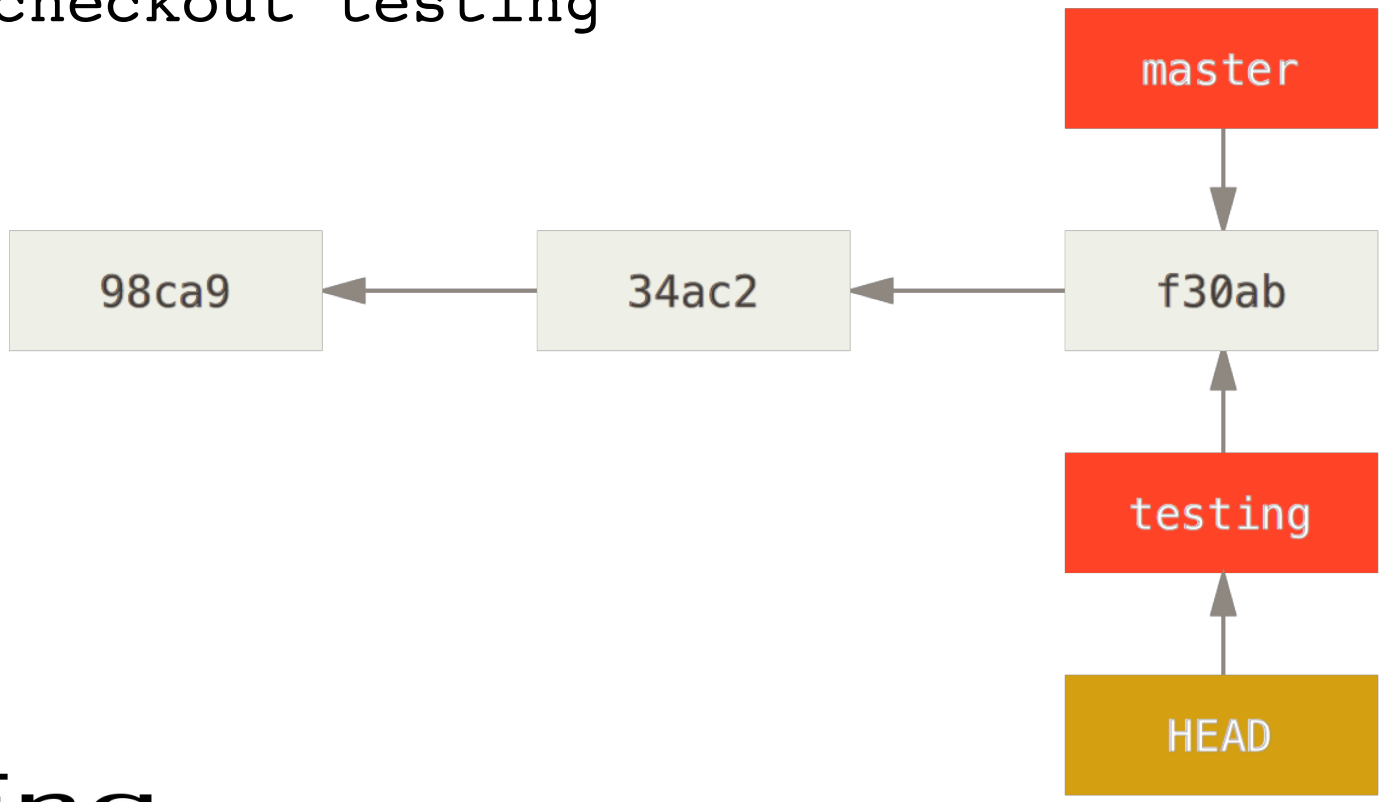
Head Pointer

- Pointer to the current local branch



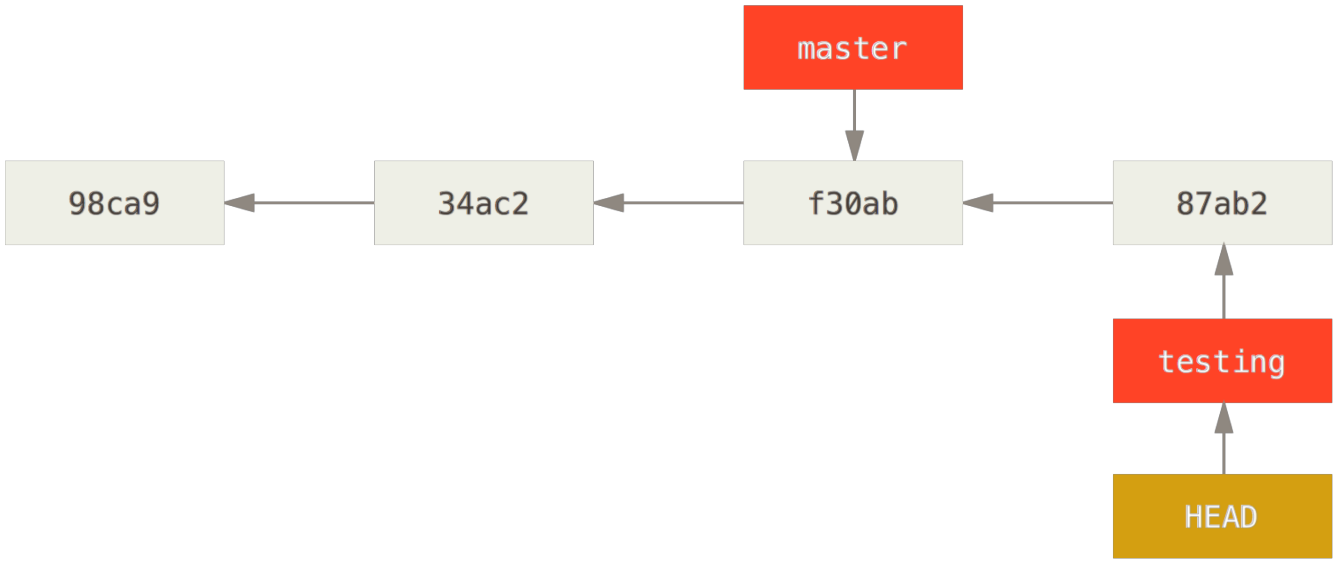
Switching Branches

- `git checkout testing`



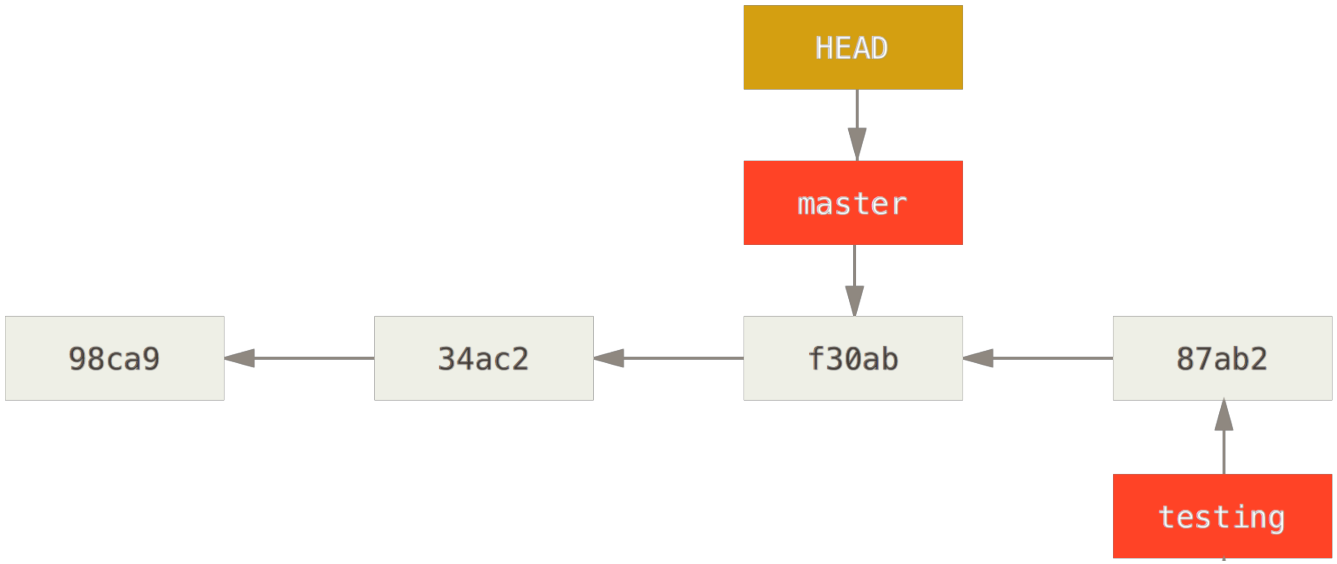
Switching Branches

- After another commit



Switching Branches

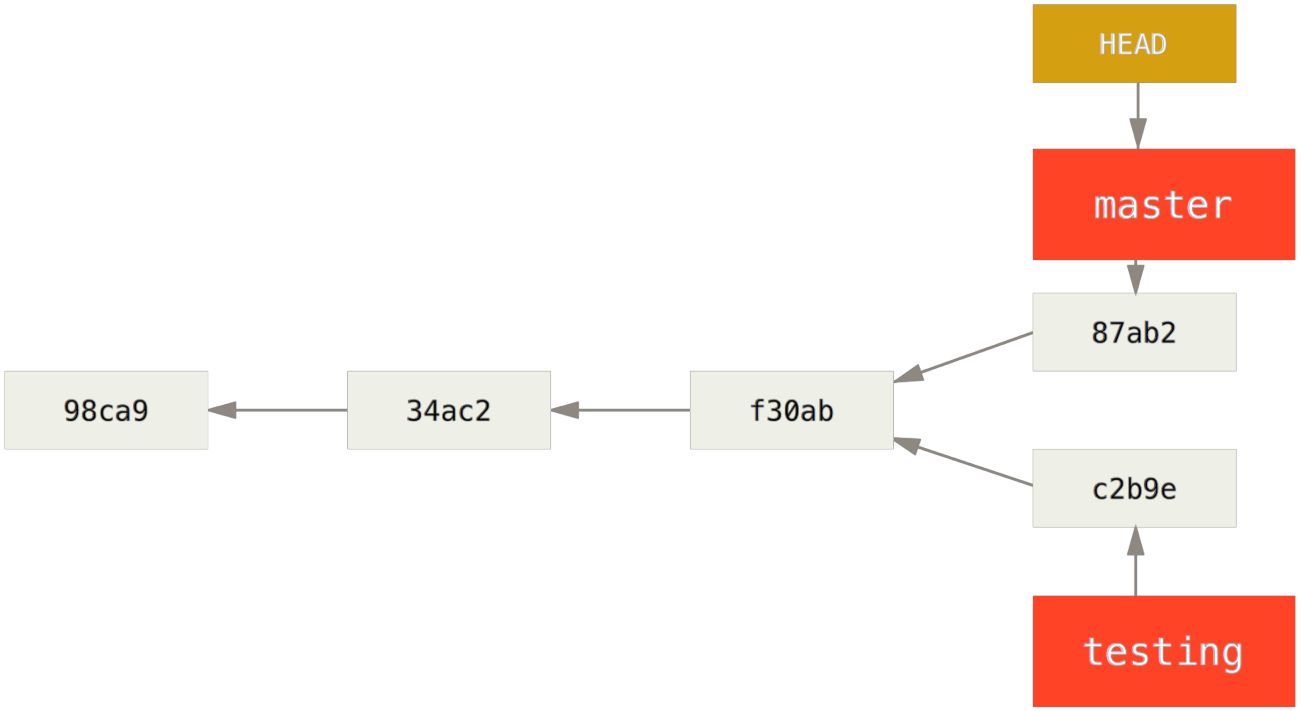
- `git checkout master`



Switching Branches

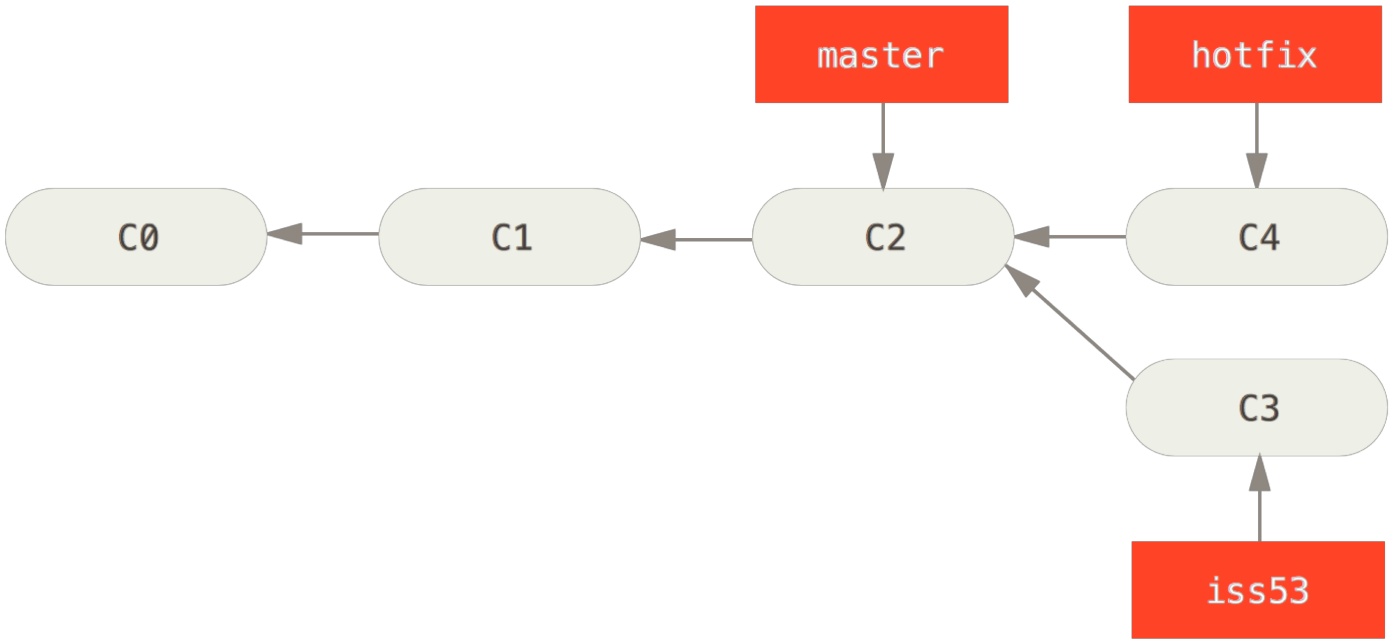
- Reverts the files in working directory back to the snapshot that master points to
- The changes made from this point forward will diverge from an older version of the project
- Files in your working directory changes

Switching Branches



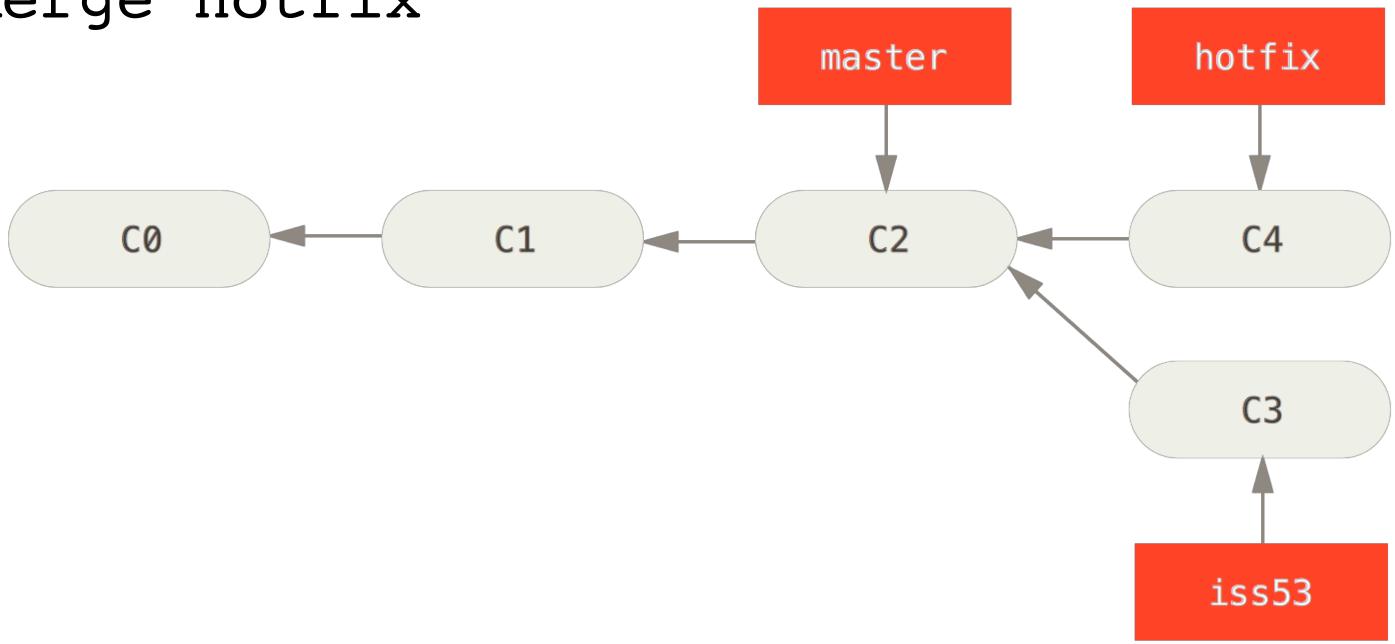
Merging Branches

- Initial situation (current branch is hotfix)



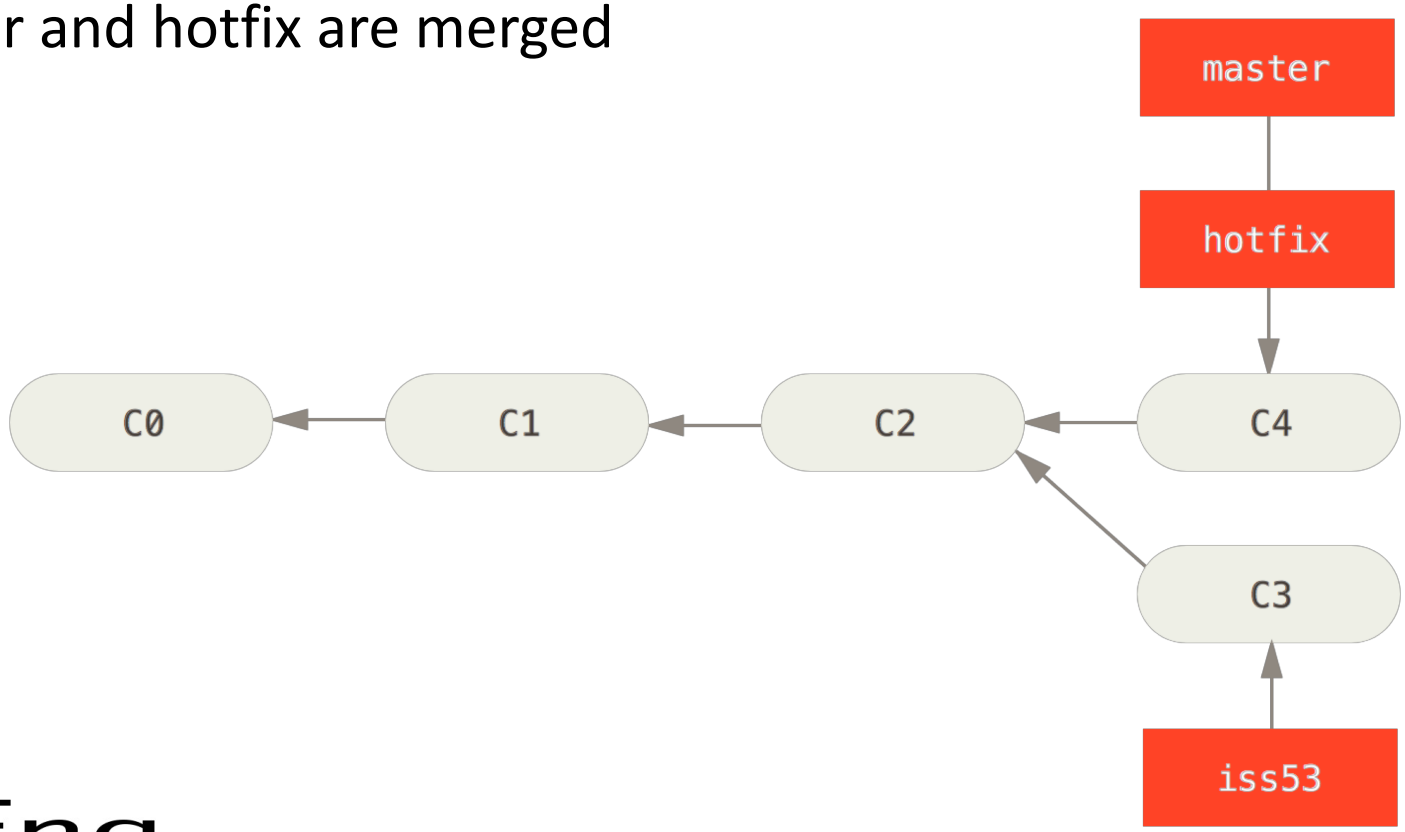
Merging Branches

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



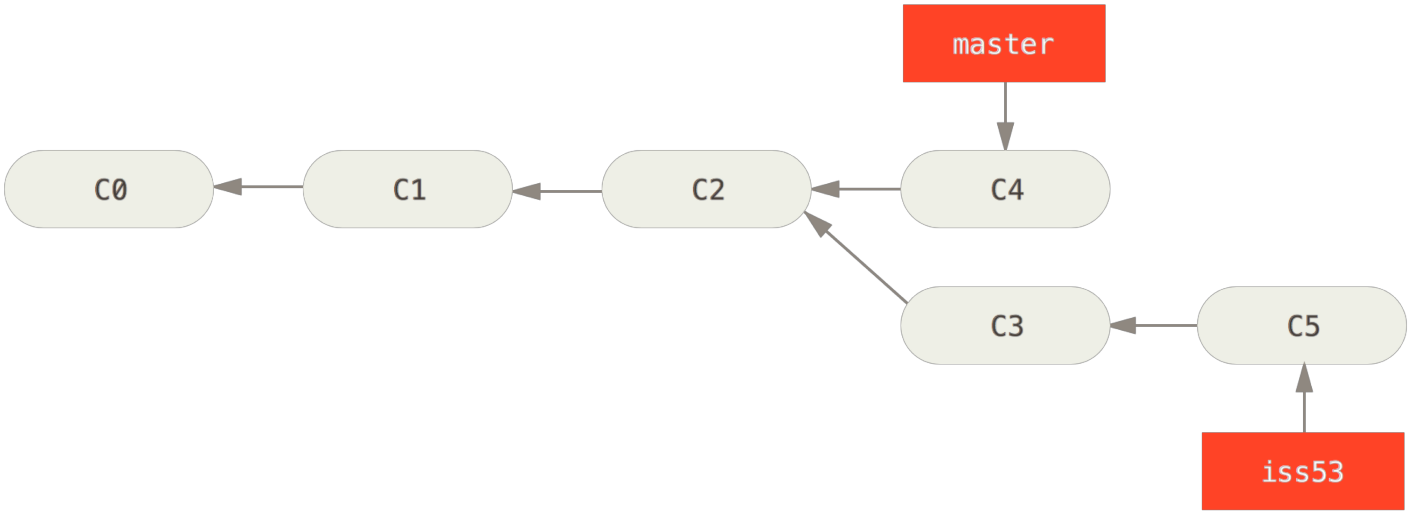
Merging Branches

- Master and hotfix are merged



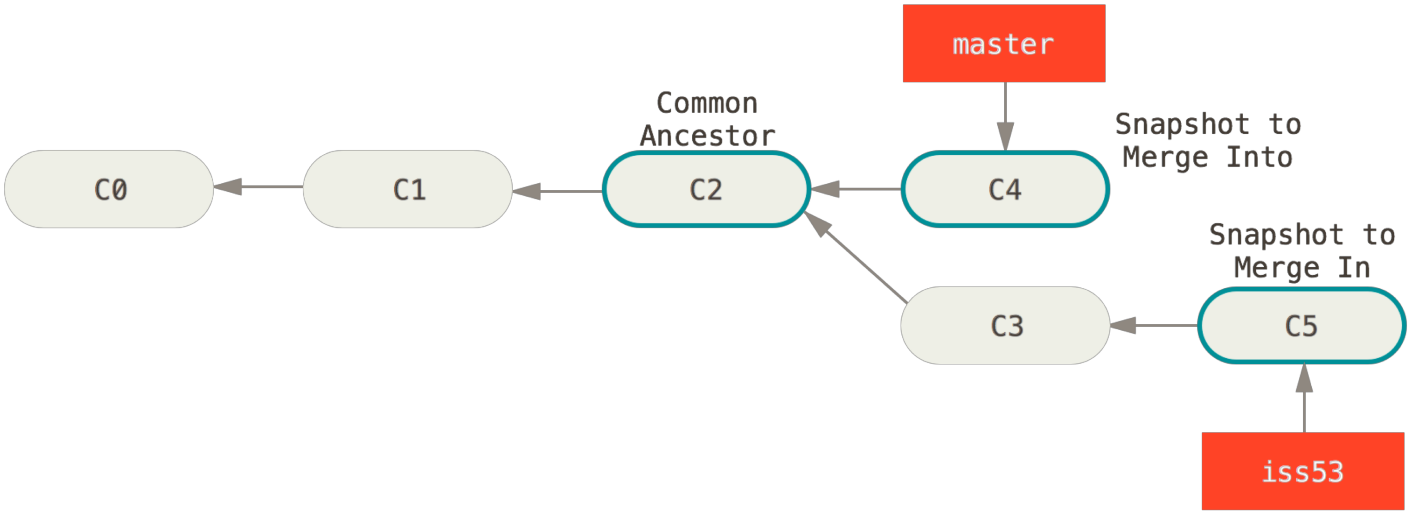
Merging Branches

- New commit on iss53



Merging Branches

- Merging iss53 with master

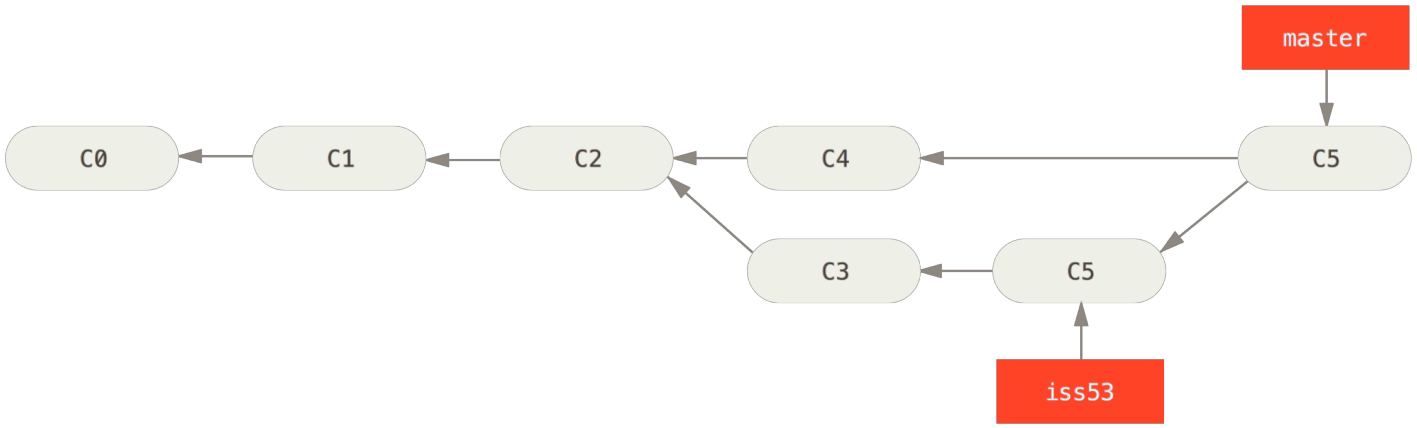


Merging Branches

- 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
- 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
- Conflicts are possible, in this case merging not done until conflicts are managed (as for commits)

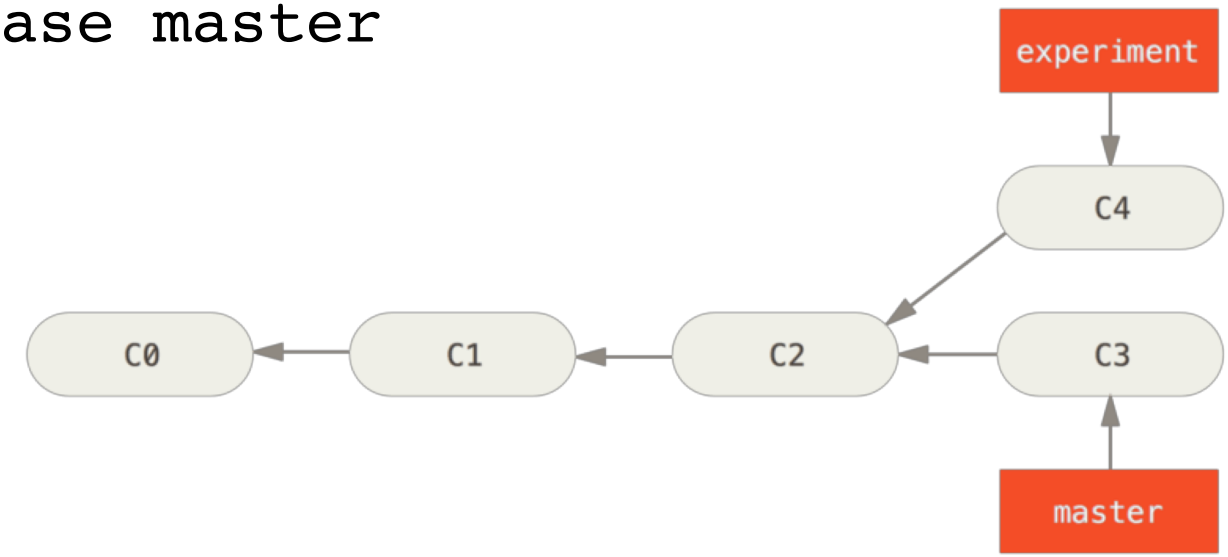
Merging Branches

- Final result:



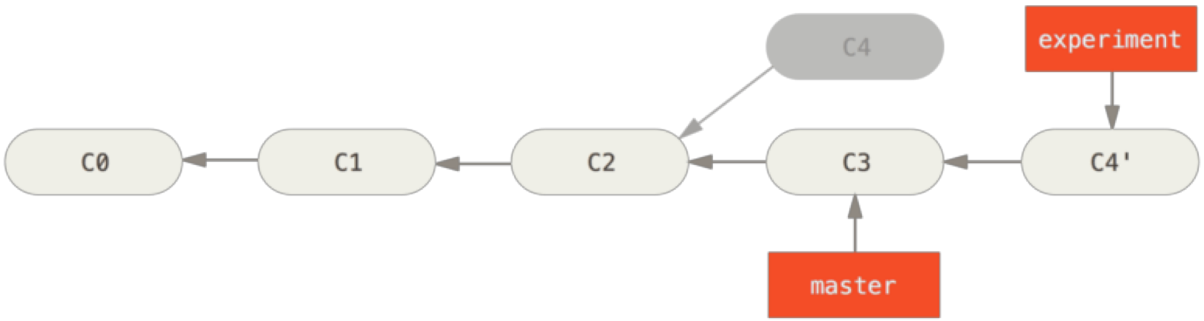
Rebase

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



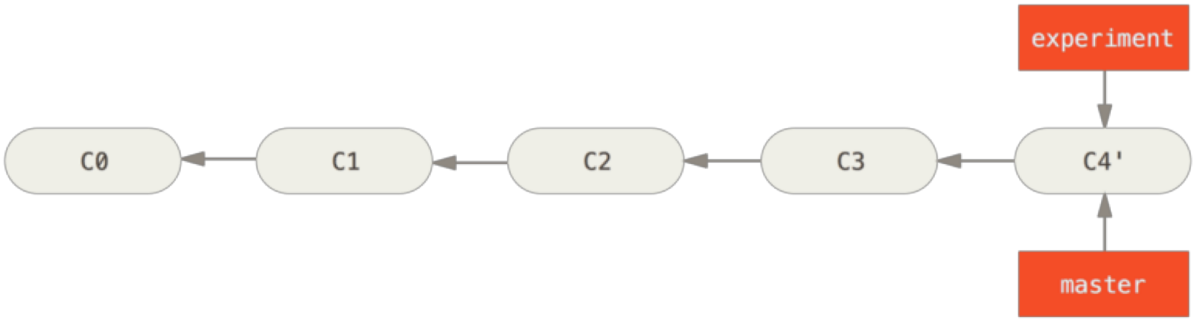
Rebase

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



Rebase

- Result



- Be careful, rebase rewrites the repository history!

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.

Rebase

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

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)

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`

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

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

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

Rebase vs. Merge

Criterion	Merge	Rebase
Resulting graph structure*	More Complex	Simpler
History	Preserved	Modified
Safety	Safer	Less Safe

- 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

Models for Collaborative Development

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

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.

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.

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.

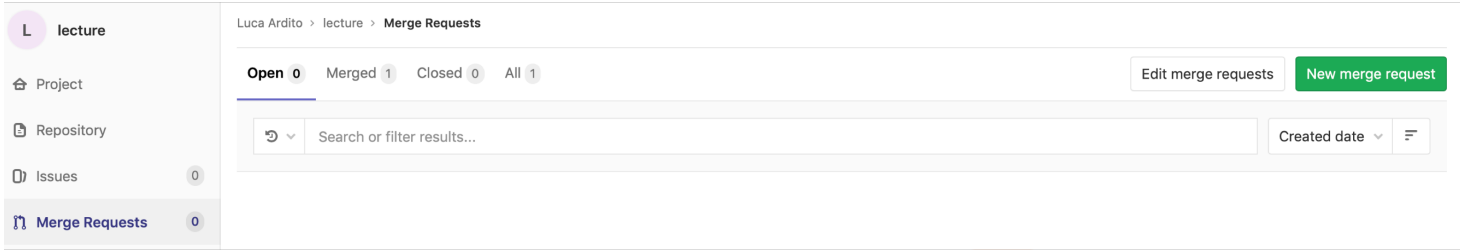
Pull (GitHub) or Merge (GitLab) requests

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

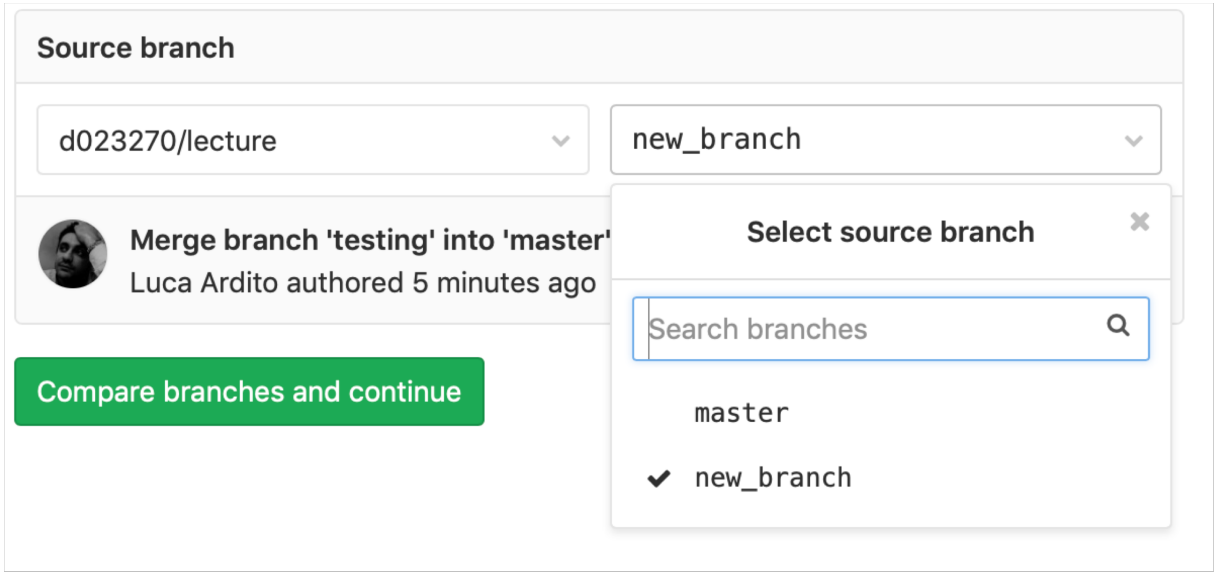
- Once a pull (or merge) request is opened, the developer can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.

Merge Requests

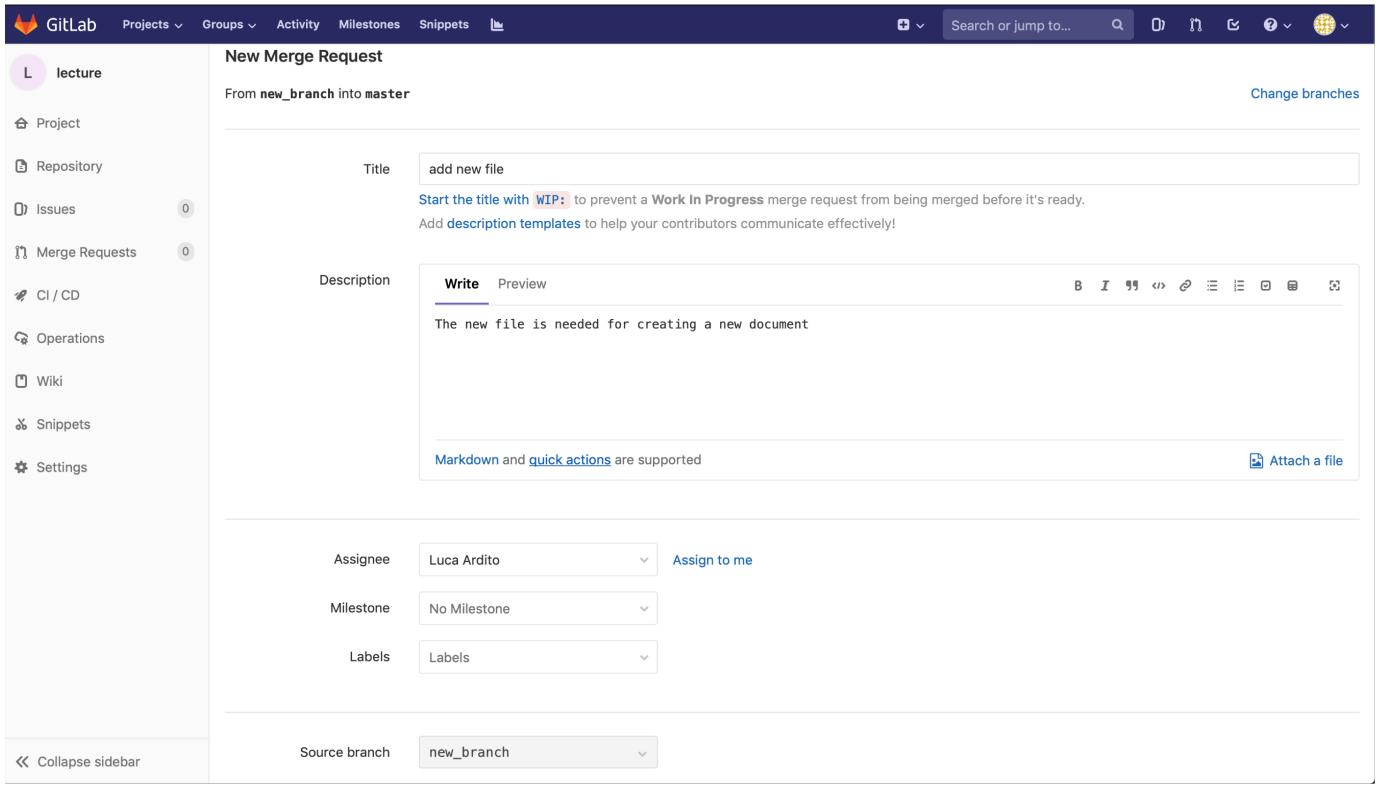
- Create a new branch
- Work on the new branch
- Commit and push your work
- Open the project on Gitlab and create a new merge request



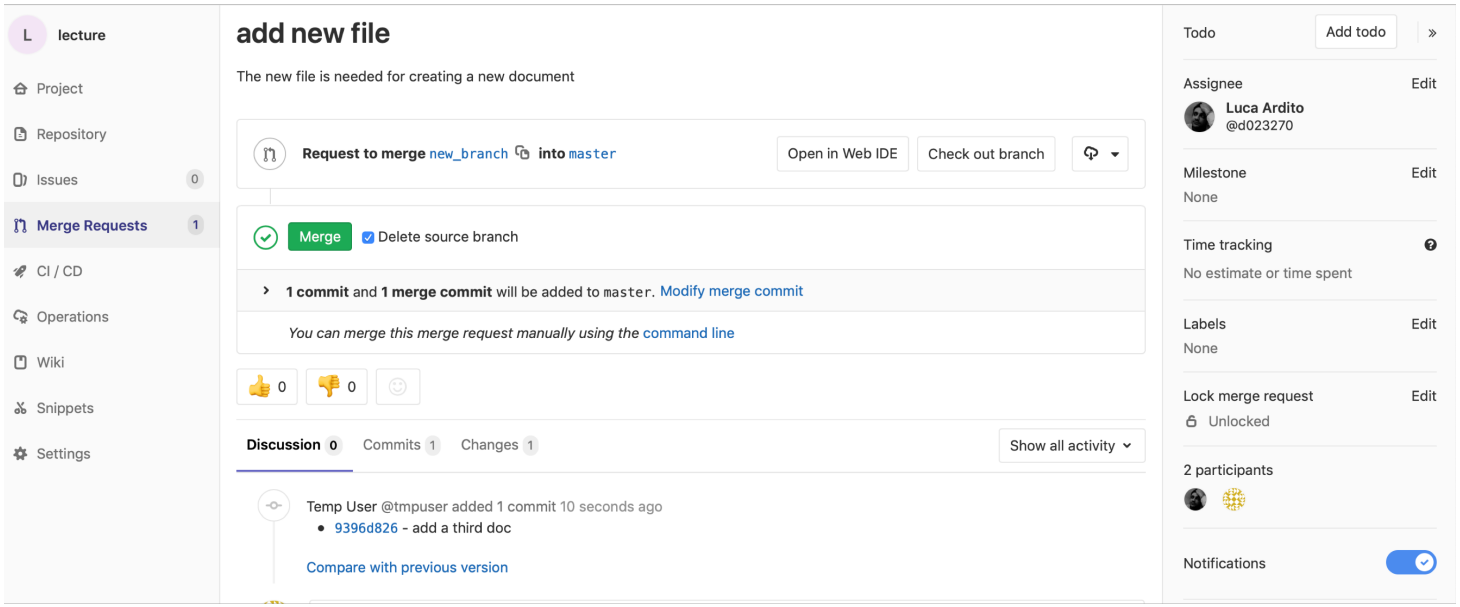
Merge Requests



Merge Requests



Merge Requests



SoftEng
http://softeng.polito.it

What to do when you think you are lost

DON’T PANIC!

`git reflog` is your friend!

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

Recorded Examples

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

Git Additional Resources

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