DIFFER

Introduction to git

For SVN users and complete version control beginners

K.L. van de Plassche[1], D. van Vugt

[1] *DIFFER, PO Box 6336, 5600 HH Eindhoven, The Netherlands*

June 11, 2020

DIFFER is part of NWO

What is git?

Git is a version control system with the following design features:

- Non-linear: Branching and merging are core features
- Distributed: Each developer has a local copy
- Multi-developer: Designed for projects with hundreds of developers

And the following advantages:

- Popular: Many guides available
- Well-supported: Many (free) tools available (e.g. GitLab, GitHub)
- Flexible: Configurable and extendable by configs and scripts

BUT

These features make it a complicated yet powerful tool!

Outline

- The design of git
- Using git: Basics
- Using git: Branching
- Using git: Multi-user collaboration

Outline

- The design of git
- Using git: Basics
- Using git: Branching
- Using git: Multi-user collaboration

A commit is a repository snapshot in time

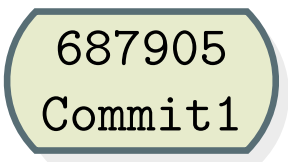
- Versions are managed in git as snapshots of your files
- These are called **commits** Similar to SVN revisions
- Commits are made manually
  - When some small change is 'finished'
  - Commit message describing the change

Opposed to SVN

- The concept of 'commit' does not imply sending data to a server!
- Paradigm: Commit the smallest possible feature, and commit often

Each commit is a node

Think in graphs! Each commit is a node, and history is a Directed Acyclic Graph; A chain of commits

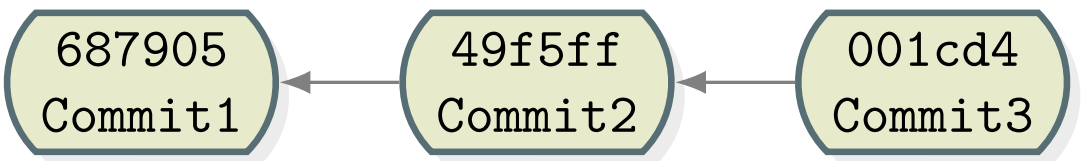


Showing:

- A unique identifier identifying the commit, the "short git hash"
- A short message, the "oneline git log"

The commit history forms a graph

Think in graphs! Each commit is a node, and history is a Directed Acyclic Graph; A chain of commits



A commit stores state information in itself

- A pointer to the state of the files at this time
- A pointer to the commit representing the previous version
- An author
- A message explaining the changes

A commit stores state information in itself

- A pointer to the state of the files at this time
- A pointer to the commit representing the previous version
- An author
- A message explaining the changes

Using an internal git command, no need to ever use this:

```
$ git cat-file commit 687905
tree 60703b283d7979f741457416039aa9a935301c13
author Karel-van-de-Plassche <karelvandeplassche@gmail.com> 1591687060 +0200
committer Karel-van-de-Plassche <karelvandeplassche@gmail.com> 1591687060 +0200
20191113_differ_core_meeting CC
```

Branches, tags: Just pointers to a commit

For convenience we can use different names to point to a commit:

- Full SHA: `687905c31cd36b711aa5c275f0295bfa5939e2bc`
- Short SHA: `687905` (any amount of uniquely identifying characters works)

Branches, tags: Just pointers to a commit

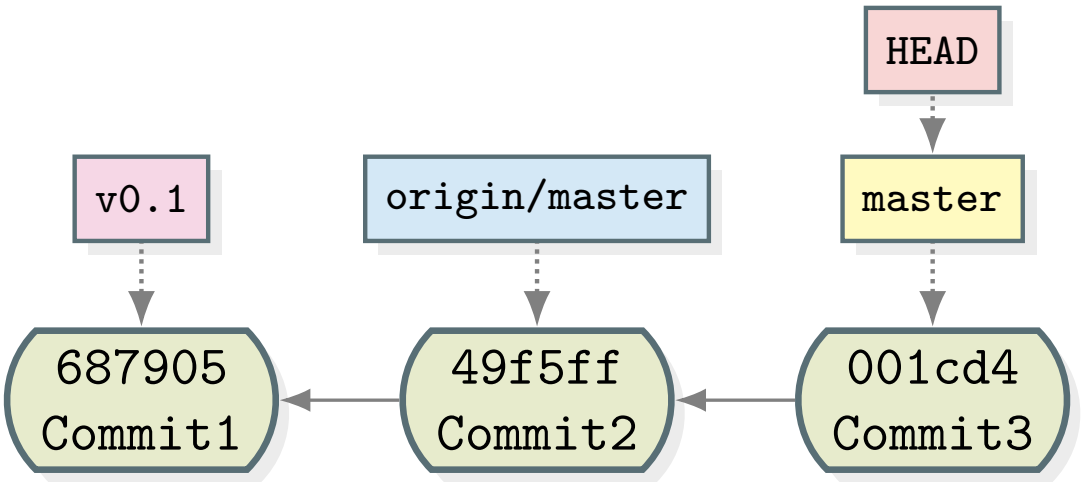
For convenience we can use different names to point to a commit:

- Full SHA: `687905c31cd36b711aa5c275f0295bfa5939e2bc`
- Short SHA: `687905` (any amount of uniquely identifying characters works)

Some common ones you find often (explained later in this presentation)

- Working copy: HEAD
  - Currently 'checked out' version, the 'working copy'
- Tags: v0.1, Karels-special-version
  - Human-friendly names for specific commits
- Branches: 'master', 'develop' (part 3)
  - Compare to SVN 'trunk'
- Remote branches: 'origin/master', 'origin/develop' (part 4)
  - Tracked separately from the local repository (remember, distributed?)

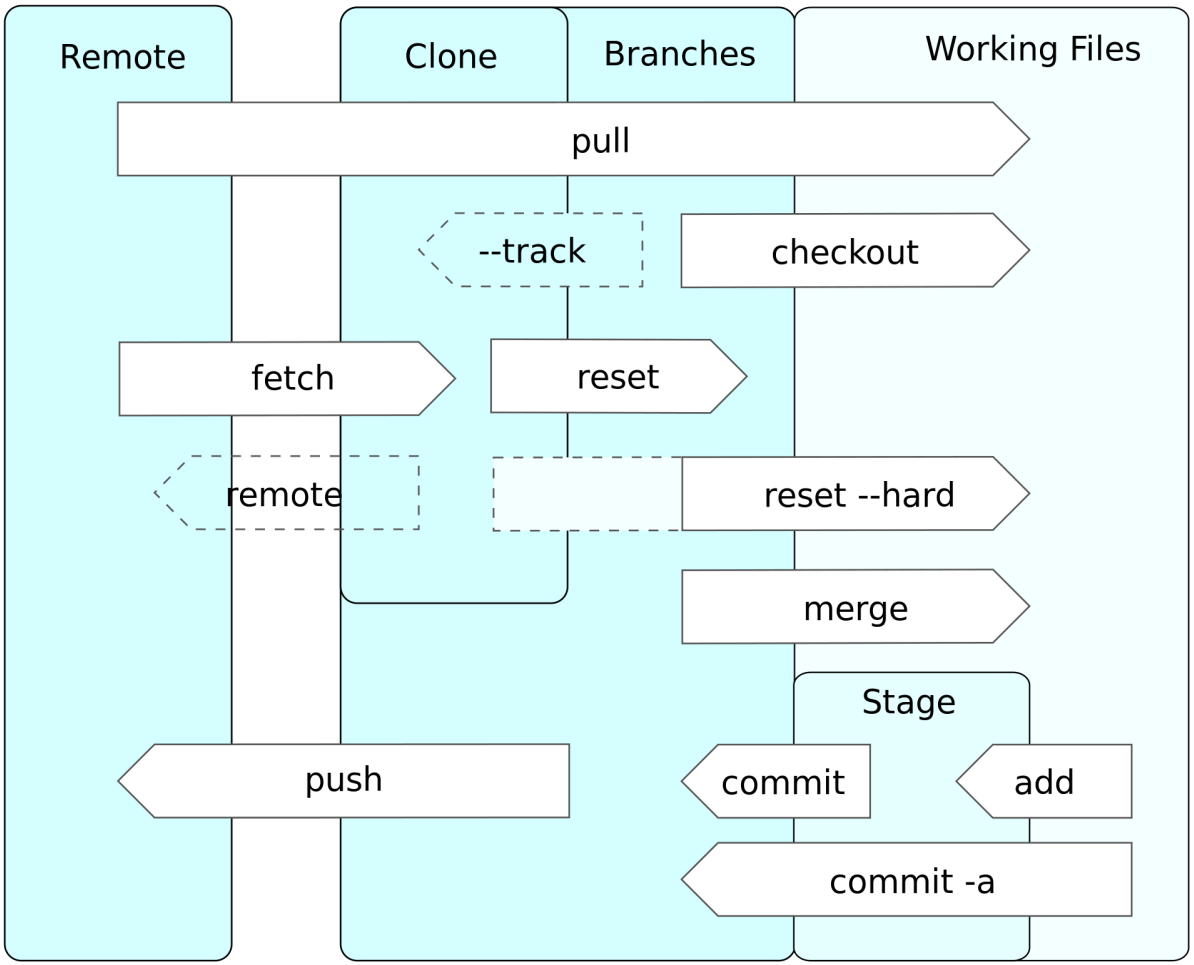
A git repository is a graph with pointers



So that all! Git is a graph with pointers, simple, right?

Distributed development, not so simple!

Or maybe not so simple..



Outline

- The design of git
- Using git: Basics
- Using git: Branching
- Using git: Multi-user collaboration

Getting a repository: init and clone

Initialize an empty repository in current folder

```
$ git init .
```

Clone an existing repository

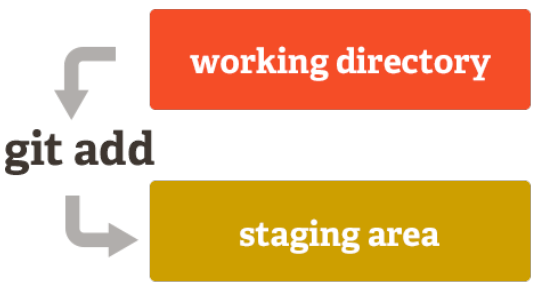
```
$ git clone git@gitlab.com:qualikiz-group/QuaLiKiz.git
$ cd QuaLiKiz
```

Similar to `svn checkout`, but:

- Distributed: Full copy of server in `.git` folder
- Split in two git concepts: `git clone` $\approx$ copy server and `git checkout` default branch

Preparing for a commit: git add

- `git add` adds a file to the staging area
- Staging area stores all changes meant to go into a commit
- Allows reviewing exactly what will be commited

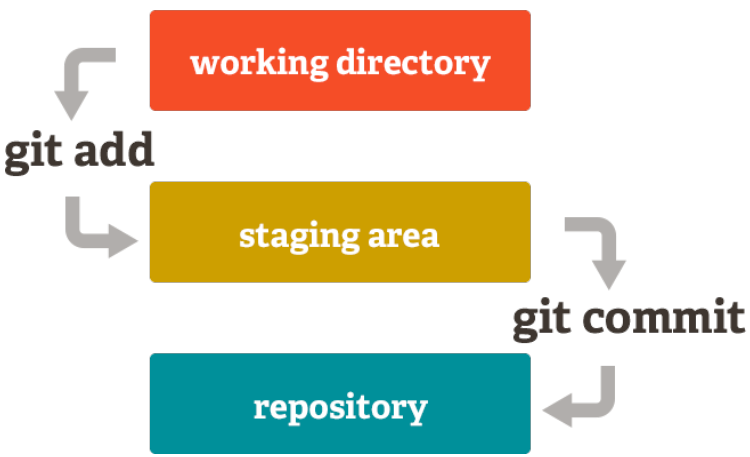



```
$ echo "version 1" > test.txt
$ git add test.txt # Stage file
$ git diff --cached # Review staged files
```

Commiting: git commit

- `git commit` commits all staged files to the **local** repository
- Needs a commit message, either:
  - No flags: git opens a text-editor (by default vim, exit with `<ESC>:q!`)
  - Pass on command line with `git commit -m "helpful message"`

```
$ git commit # Commit all staged files
```



Intermezzo: commit messages

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

Intermezzo: what is a good commit message?

- Describe what was changed and why
- Insert a blank line after first line
- Wrap subsequent lines at 72 characters

Model commit message[1]

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body. The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug"
or "Fixes bug."

- Bullet points are okay, too
```

[1] No, I don't always do this myself

Finding your way: git status

Running `git status` shows:

- Staged changes
  - Changes to add-ed files
- Unstaged changes
  - Changes in non-add-ed files
- Untracked files
  - Files in the working directory but not in the repository

On our fresh pre-commit repository:

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
  new file:   test.txt
```

Finding your way: git status (2)

Now we fiddled around a bit:

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
      new file:   newfile.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
      modified:   test.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
      uselessfile.txt
```

We see:

- We have staged a new file for commit: `newfile.txt`
- We have modified an old file, but are not commiting that: `test.txt`
- We have a file that is not under version control: `uselessfile.txt`

Finding your way: git status (3)

Fiddle some more

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   newfile.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   test.txt
        modified:   newfile.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        uselessfile.txt
```

`newfile.txt` was changed *after* staging it, that change will not be committed

Finding your way in time: git log

- Shows information about previous commits
- Many options:
  - Show for specific folder
  - Per date
  - Per author
  - Draw a graph

Some examples:

```
$ git log # Basic 'flattened' log
$ git log -3 # last 3 commits
$ git log --after='yesterday' # logs since yesterday
$ git log --before="2014-01-01" # logs before a date
$ git log --oneline --decorate --graph # logs with a graph for current branch
$ git log --oneline --decorate --graph --all # logs with a graph for all branches
```

Finding your way (advanced): git describe

- ‘git describe‘ describes the current state of the repository
  - describes the state relative to so called `annotated tags`
  - `annotated tags` are tags with a message, e.g. `git tag -a`
  - Most releases made with external tools are annotated tags
- Gives one-line description instead of full log
- You might be asked you to run this by people trying to figure out where you are in git history

Finding your way (advanced): git describe how-to

Let's see where we are:

```
$ git describe
fatal: No annotated tags can describe '13ac1c701afdd21626401b7ee237b116a17566f3
However, there were unannotated tags: try --tags.
```

No annotated tags were there, let's also use unannotated tags

```
$ git describe --tags
v2.6.2-219-g15245d4
```

- Closest (unannotated) tag was `v2.6.2`
- We are the 219th child of that commit
- Our child commit is called `15245d4`

Maybe also check if we do have uncommitted changes

```
$ git describe --tags --dirty
v2.6.2-219-g15245d4-dirty
```

Yes! We have a `dirty` repository. If someone is remote-debugging for us, we should `git stash`

Cleaning the repository; Do not hard reset!

Warning! This you see in guides often, but **non-recoverably discards your changes**. I advice not to run this

```
$ git reset --hard # You will lose data!
```

Outline

- The design of git
- Using git: Basics
- Using git: Branching
- Using git: Multi-user collaboration

The power of non-linear: branches

- Before we had a single linear history of changes: the master branch
- Git allows multiple parallel branches
- A 'git branch' is an automatically updated pointer to a commit
- Allows for multiple versions at the same time

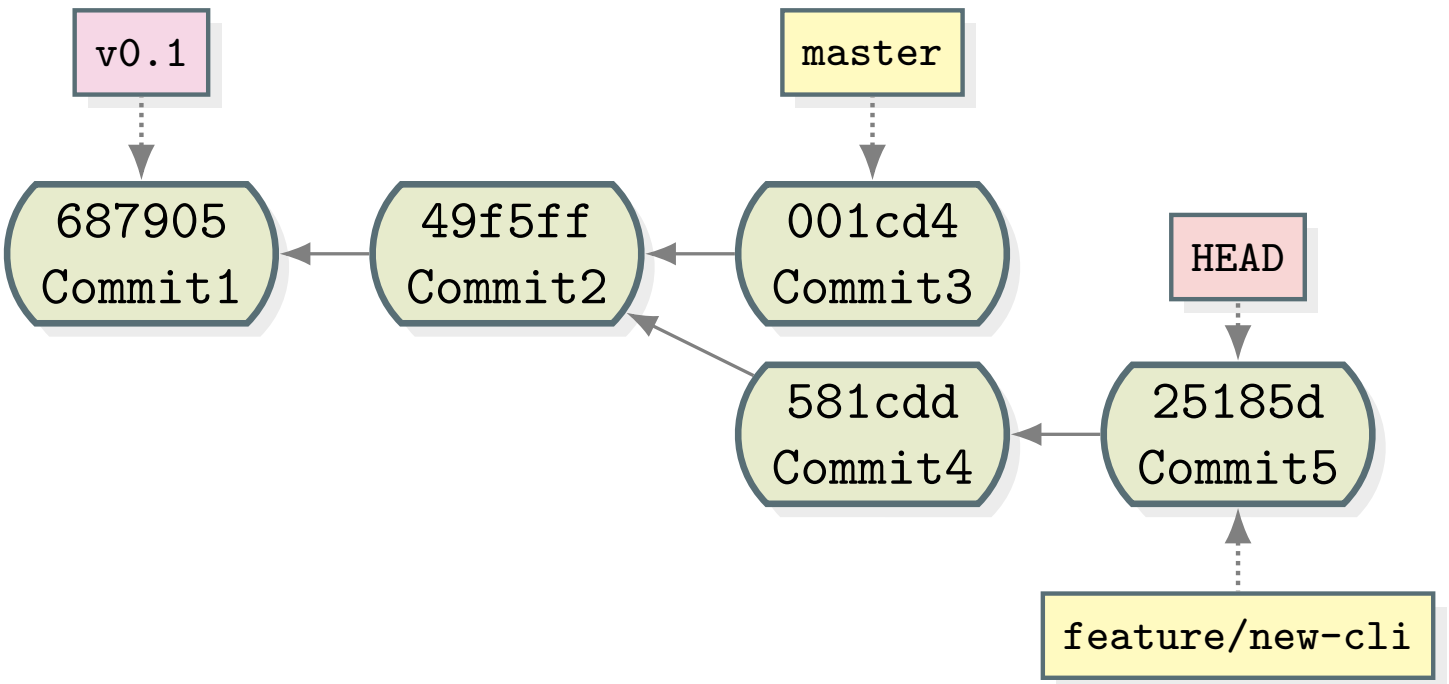
Compare to SVN branches:

- Git stores everything as diff to its parent commit
- In a single repository, all branches share the same ancestor commit, the root commit[2]
- Branches are *virtual*; a link to a commit, not a folder in a repository

[2] Not counting subtrees

Common workflow: Have a feature branch

- Master should be as stable as possible at all times
- Feature branch contains a single, self-contained feature

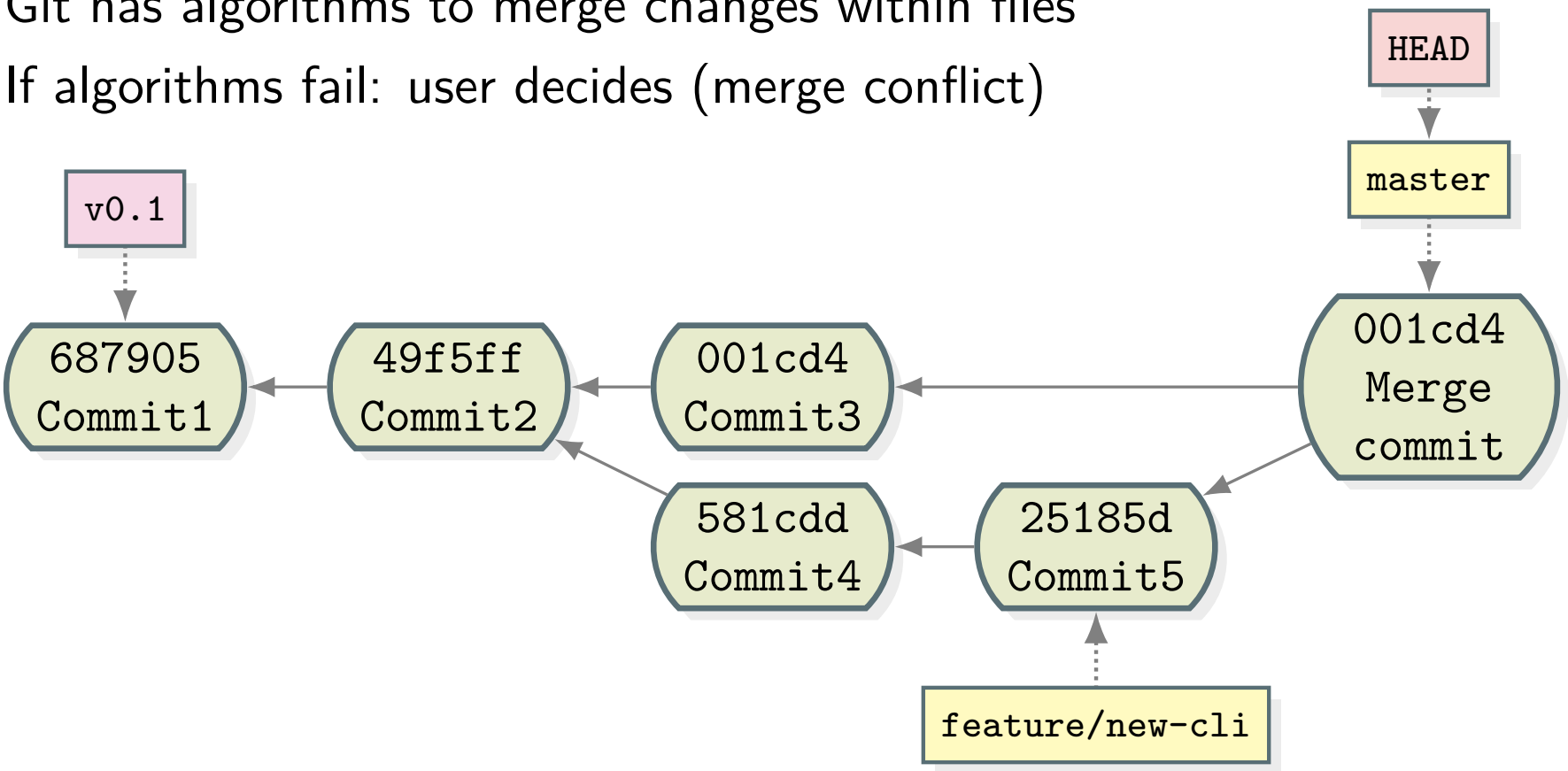


Manipulating branches

```
$ git branch # list branches
$ git branch testing # create branch testing
$ git branch -d testing # delete branch testing
$ git checkout testing # switch to branch testing
$ git checkout -b new_test # create a branch new_test and switch to it
```

Done developing: git merge

- Create a new commit with **two** parents (remember, DAG?)
  - For the rest, just a regular commit
- Git has algorithms to merge changes within files
- If algorithms fail: user decides (merge conflict)




```
$ git checkout master # move back to master branch
$ git merge feature/new-cli # merge feature branch into master
```

User input: Solving merge collisions

- Happens when both parent change the same line
  - Both parents stand on *equal ground* for git
  - No automatic way to determine precedence[3]
- Solution: Let the user solve it

```
$ git merge feature/new-cli # merge feature branch into master
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html # Whelp! A merge conflict!
Automatic merge failed; fix conflicts and then commit the result.
```

[3] Not counting CLI flags

User input: Solving merge collisions

- Happens when both parent change the same line
  - Both parents stand on *equal ground* for git
  - No automatic way to determine precedence[3]
- Solution: Let the user solve it

```
$ git merge feature/new-cli # merge feature branch into master
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html # Whelp! A merge conflict!
Automatic merge failed; fix conflicts and then commit the result.
```

Git stores the result, and a copy of the two parents:

```
$ cat index.html # What did git do?
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

[3] Not counting CLI flags

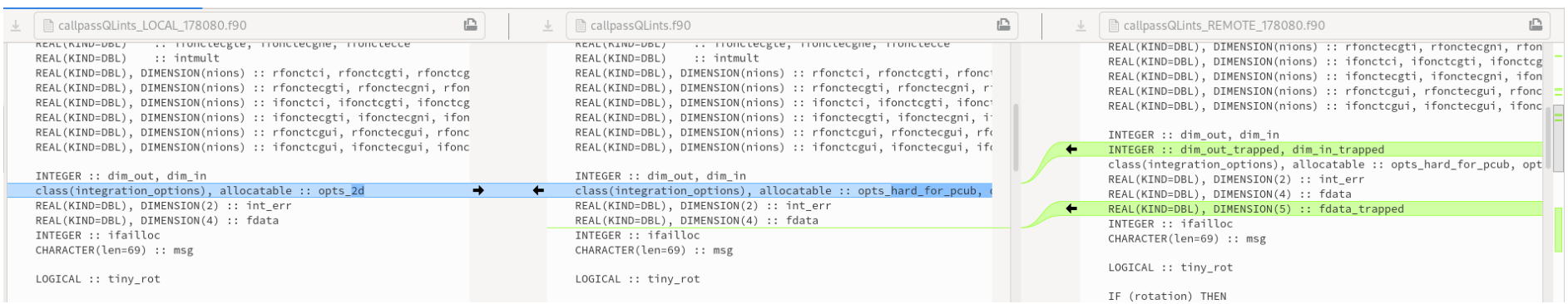
Solving merge collisions

Please do not

- Solve conflict by hand opening the file, look for `<<<<<<<`

But use `git mergetool`

- Most git merges are *three way merges*: The two parents, and the resulting child
- Many tools avaiable to solve this, by default use `kdiff3` (not for the faint of heart)
- Tip: Use meld `git mergetool -t meld` (or set your git default)



Most git commands take overloaded arguments

Most `git` commands take both path-like and version-like arguments

```
$ git log text.txt # View history of text.txt
$ git log develop # View history of develop branch
$ git log develop -- text.txt # View history of test.txt on develop branch
$ git checkout develop # Check out the develop branch
```

Checkout specific versions of files

```
$ git checkout 687905 -- index.html # Check out index.html of a specific commit
$ git checkout v0.1 -- index.html # Check out a specific tag
```

Go into the dreaded 'detached HEAD' state[4]

```
$ git checkout 687905 # Check out working copy to a specific commit
$ git checkout v0.1 # Check out working copy to a specific tag
```

[4] Do not panic, it just means you are not on a branch

Outline

- The design of git
- Using git: Basics
- Using git: Branching
- Using git: Multi-user collaboration

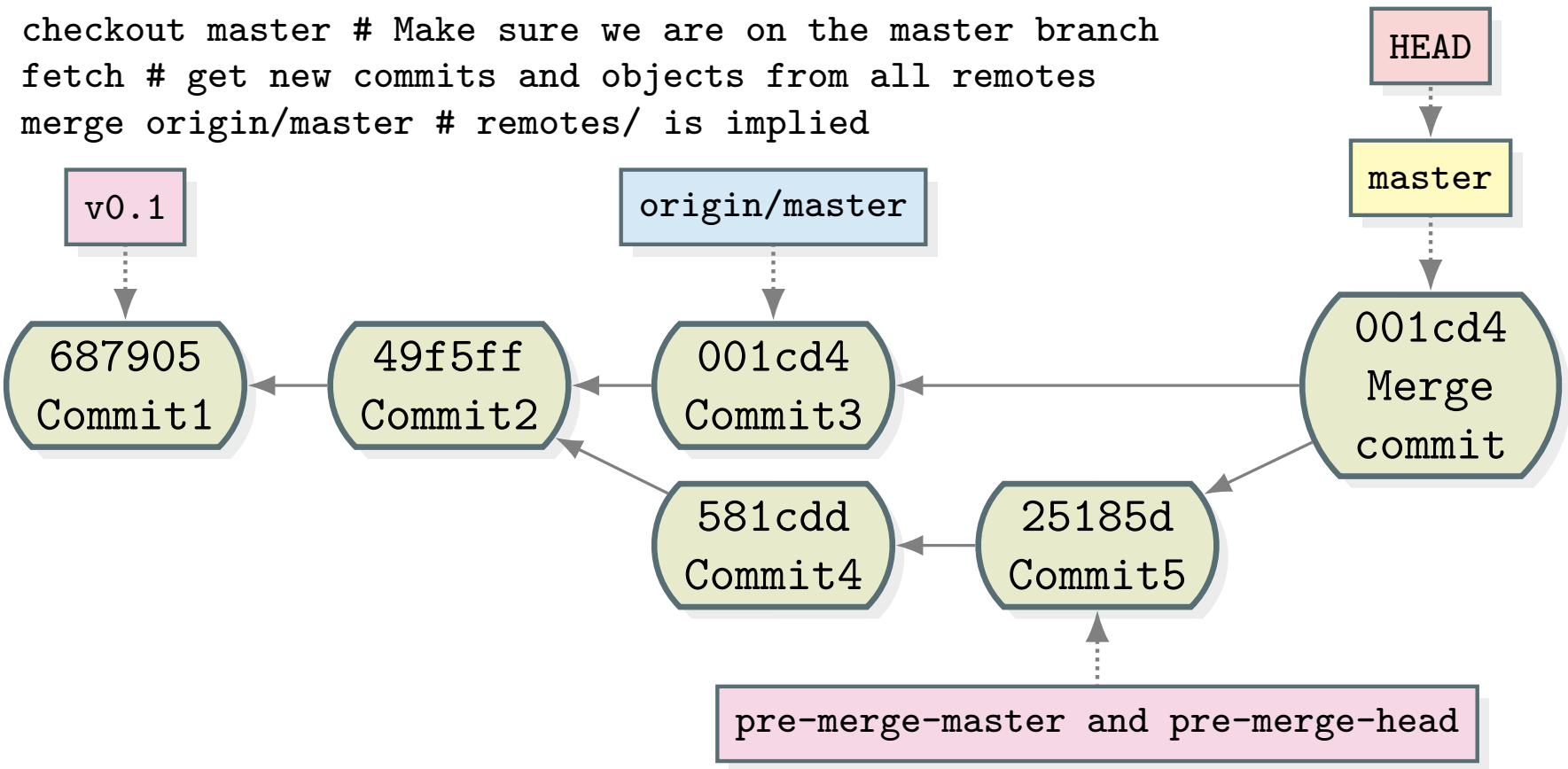
Sharing repositories

- Commonly a central server hosts a repository
- Users push new commits to this, and fetch new changes
- git clone sets up a `remote` called `origin` for you
- Branch `$branch` on the remote is called `remotes/origin/$branch`
  - Treated as any other branch
  - Common confusion: This means it just **tracks a commit on the remote repository**
  - Common confusion 2: If the server gets new updates, **your local copy will not be updated automatically**

Getting updates from the server

- First `fetch` the new changes, updating all our references `remotes/origin/$branch`
- Merge those changes into our local copy

```
$ git checkout master # Make sure we are on the master branch
$ git fetch # get new commits and objects from all remotes
$ git merge origin/master # remotes/ is implied
```



Not grokking git: git pull

Why git is complicated:

- Most new users start on someone elses repository
- This is a relatively complicated git case
- `git pull` is mentioned in pretty much all guides, but:
- `git pull` is a **git fetch + git merge!**
- As seen before, if the remote repository was updated, and we were working on the master branch, we get a **merge collision**

My tip for new users, never `git pull` blindly! Instead

```
$ git fetch # get new commits and objects from all remotes
$ git status # will we get into trouble if we pull/merge this
$ git merge origin/master # remotes/ is implied
$ git pull # If you prefer this over a merge
```

Not grokking git: git pull, why?

It is mentioned in guides because git tries to be smart. Guides might assume:

- No two users use the same branch[5]
- > No remote changes are done while working
- > Git can do its special fast-forward merge "`--ff`"

Meaning

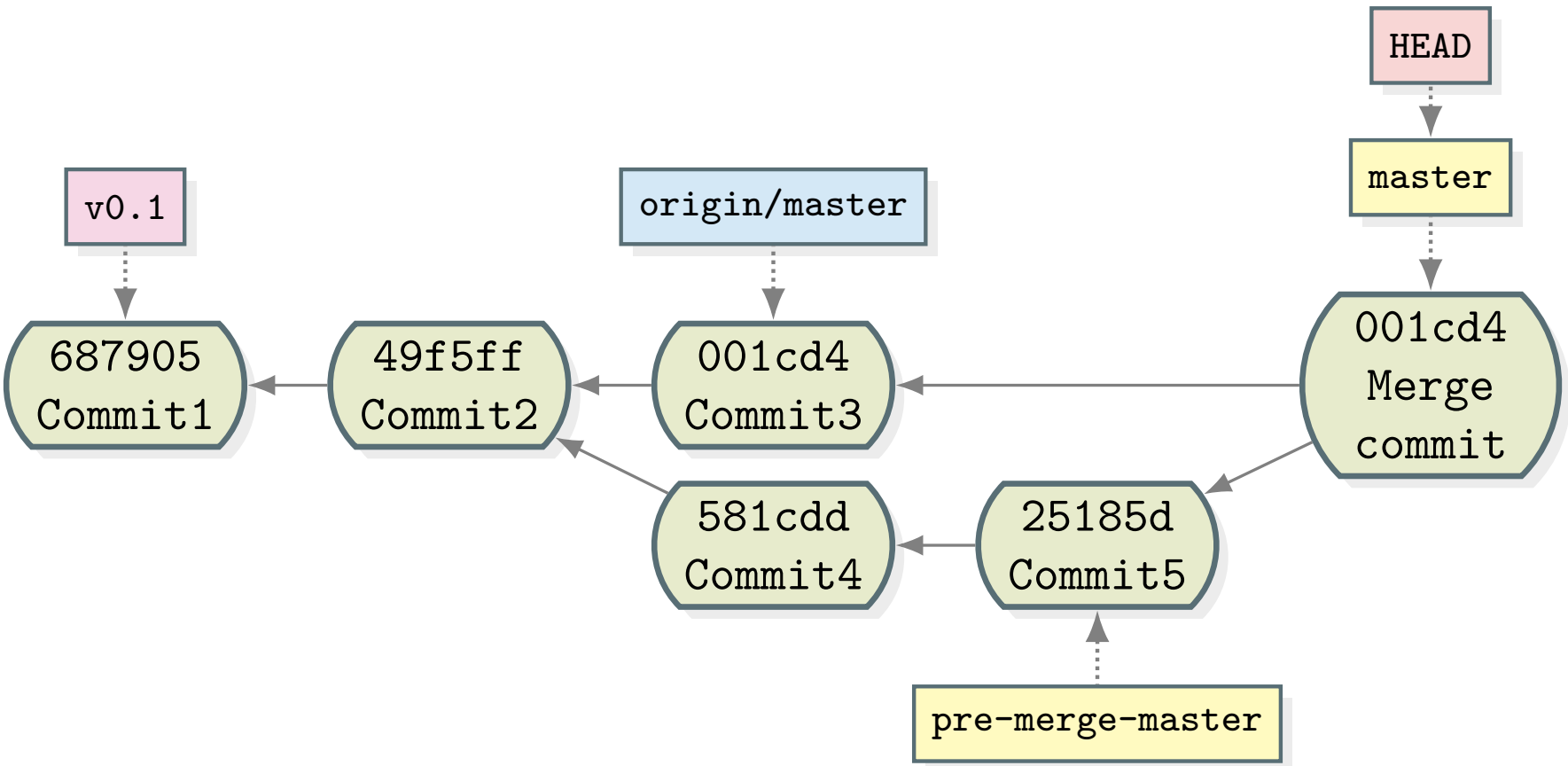
- As there are is no other work done
- And the remote and local branches represent the same thing
- > Do not create a two-parent merge commit, but a single parent regular commit

By default, git prefers to fast-forward if possible, dropping to a regular merge if needed

[5] Or are git proficient if they do

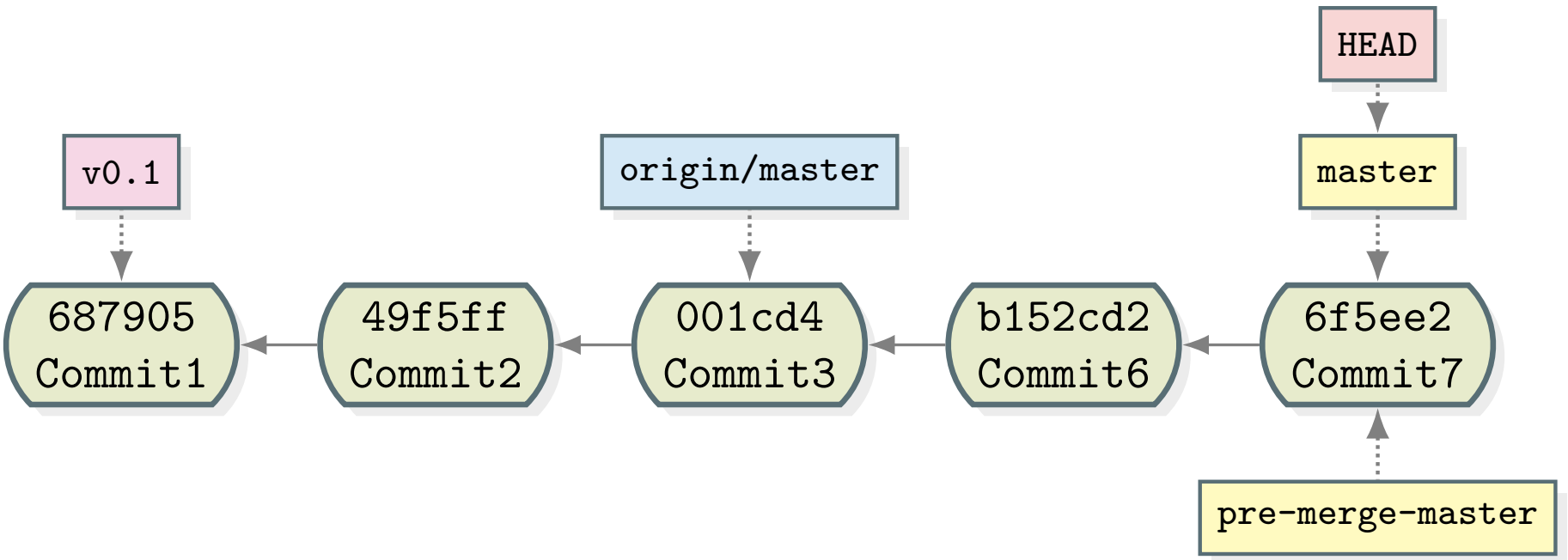
Merge where remote updated: regular merge

```
$ git fetch # get new commits and objects from all remotes
$ git status # will we get into trouble if we pull/merge this
$ git merge origin/master # triggers a 'regular' merge
```



Merge where remote did not update: feed-forward merge

```
$ git fetch # get new commits and objects from all remotes
$ git status # will we get into trouble if we pull/merge this
$ git merge origin/master # triggers a 'feed-forward' merge
```



Sharing code with repository: git push

- Code development is done, share with main repo `git push`
- As with `git pull`, this is implicitly two commands:
  - Send you new commits to the server
  - Update the servers' branch pointer to the new latest commit
- HOWEVER! As the server has no human behind it, it
  - Needs to be a direct continuation of the servers' branch
- Solve by:
  - Merging remote (updated) branch into yours again
  - Do as git does, and act like nothing happened `git rebase` (advanced)
  - Ask a human for help

```
$ git fetch # preempt any problems
$ git status # will we get into trouble if we push this
$ git push # Implies 'git push <branch> origin/<branch>'
```

Sharing code with a human: pull requests

A common model is for a repository to have *maintainers*. They:

- Make sure issues are delegated to the right people
- Review, or assign reviewers, to code contributions
- Keeps general overview of the software project

Most git front-ends have the concept of 'pull requests' (GitHub/BitBucket) or 'merge requests' (GitLab). Allows for:

- Arbitrary people to request "pull/merge my code into the servers'"
  - Only the maintainer can merge into master
  - Developers can work on feature branches in the repo
  - General audience can create GitLab/GitHub 'forks'
- Discussion about code being contributed, instead of concepts
  - Concepts are discussed in the issues/slack/skype/mailing list

Summarizing

- Single branch, single user
  - Repository manipulation: `git init`, `git commit`, `git stash`
  - Finding your way: `git status`, `git log`, `git describe`
- Multi branch, single user
  - Branch manipulation: `git branch`
  - Changing working copy: `git checkout`
  - Merging: `git merge`, `git mergetool`
- Multi branch, multi user
  - Getting code: `git fetch`
  - Merging locally: `git merge`, `git pull`
  - Sharing code: `git push`
- We didn't touch on
  - Git frontends: Gitlab/GitHub/BitBucket/Gforge
  - Changing history: `git rebase`, `git commit --amend`
  - Multiple trees: `git subtree`
  - External dependencies: `git submodule`

Resources

- This presentation sources
  - git-for-scientists
  - grokking-git
- Atlassian: Very clear and visual guides
  - intro to VC
  - intro to git (as in this presentation)
- GitLab: Very clean intro to GitLab, intro to git is a bit low-level
  - GitLab basics
  - Using git on the CLI
- GitHub intro to git ('reading' and 'doing' tutorials)
  - try.github.io

Backup

Intermezzo: Why bother with this complexity

- Allows for easy recovery of older code
  - Working on different problems in parallel? Parallel branches!
  - Give up on the feature? Delete branch!
  - Have a CI system that tests on commit only? Dummy commits on dummy branch!
  - Messed up your local copy? Just use `git status` to figure out what to remove
- Essential for complicated workflows
  - Worry about stability?
    - Have a master-develop-feature branches git flow e.g. RAPTOR, JINTRAC
    - Have master-feature-release branches gitlab flow e.g. QuaLiKiz
  - Multi-user workflows (up next!)

Cleaning the repository

Sometimes you do not want to commit, i.e.

- You want to throw away what you have
- You made a change that broke something, and want to go back

We can always check out older versions of files (later in presentation), but in case we want to save what we have. This is `git stash`.

Cleaning the repository: git stash

Run the following command in your dirty repository

```
$ git stash
Saved working directory and index state WIP on
master: 15245d4 Update Python version of docgen image
```

So these uncommited changes were when working on 15245d4

Cleaning the repository: git stash (2)

Run the following command in your dirty repository

```
$ git stash
Saved working directory and index state WIP on
master: 15245d4 Update Python version of docgen image
```

So these uncommited changes were when working on 15245d4

We do our magic, and come back later. List what we had stashed

```
$ git stash list
stash@0: WIP on master: 15245d4 Update Python version of docgen image
```

And apply our saved changes again. As it's a diff, only works automatically if we are cleanly on 15245d4!

```
$ git stash pop # Implies popping stash 0
$ git stash pop 0 # Explicitly pop stash 0
```

Pop implies

```
$ git stash apply # Apply the path. 0 implied
$ git stash drop # Drop the stash if apply worked. 0 implied
```