

Git

iCub programming course
2014-07-10

Daniele E. Domenichelli

`daniele.domenichelli@iit.it`

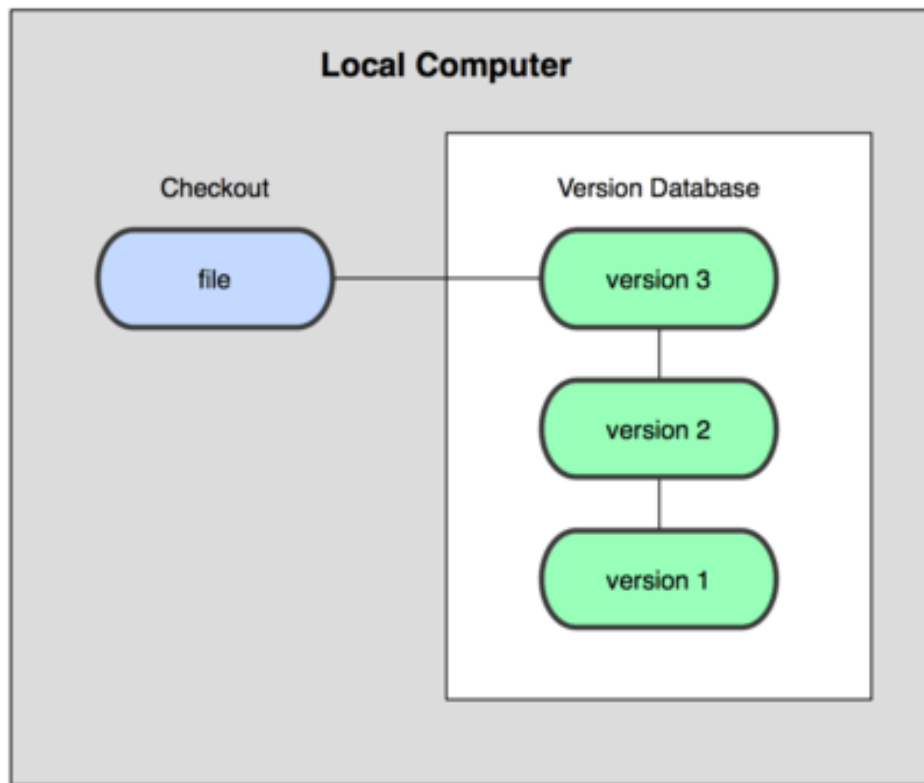
Version Control

- System that records changes to files.
- Allows you to restore a specific versions later.
- A better option than copying files with a different name or into another directory.

Source Control Management

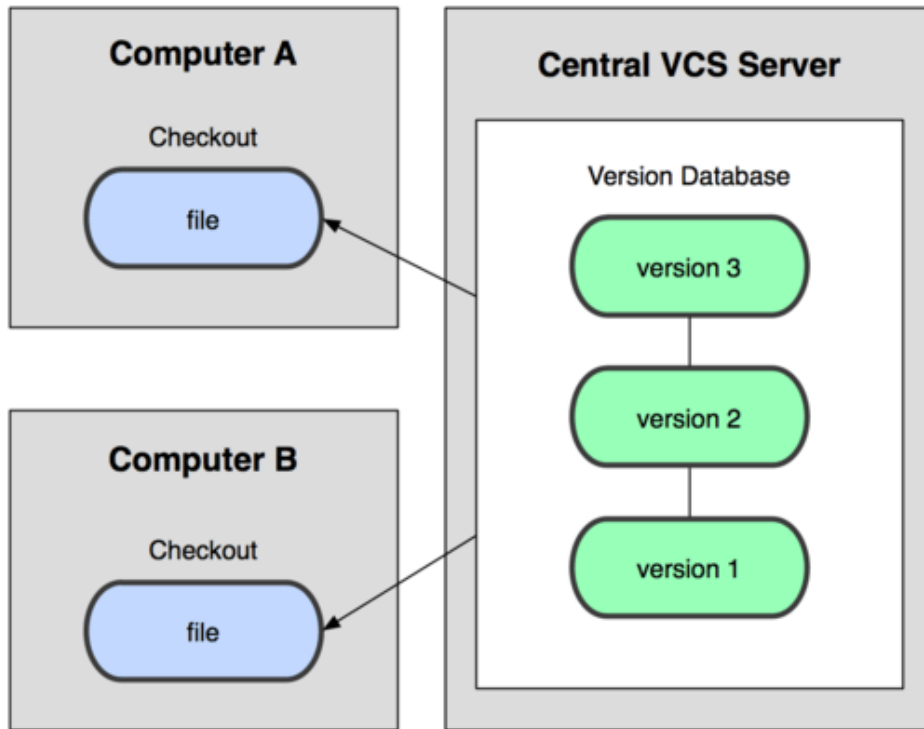
- Working tree
 - Files and directory in your repository, at their current state
- Repository content (database)
 - Files
 - Commits
 - Parent/Child relationship

Local Version Control Systems



- RCS (1982)
- Local version database.
- Local checkout

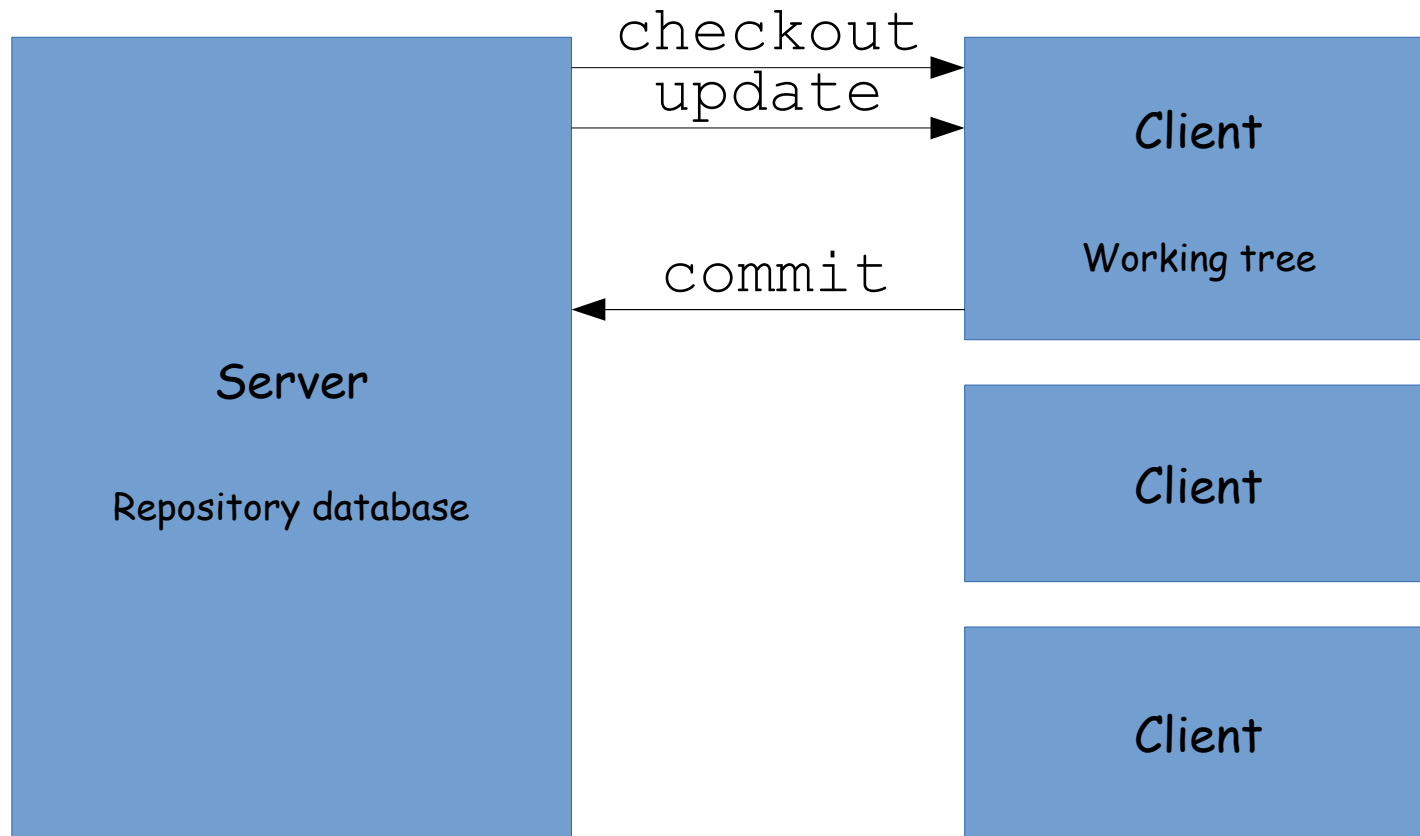
Centralized Version Control Systems



Scott Chacon CC BY-NC-SA 3.0
<http://www.git-scm.com/figures/18333fig0102-tn.png>

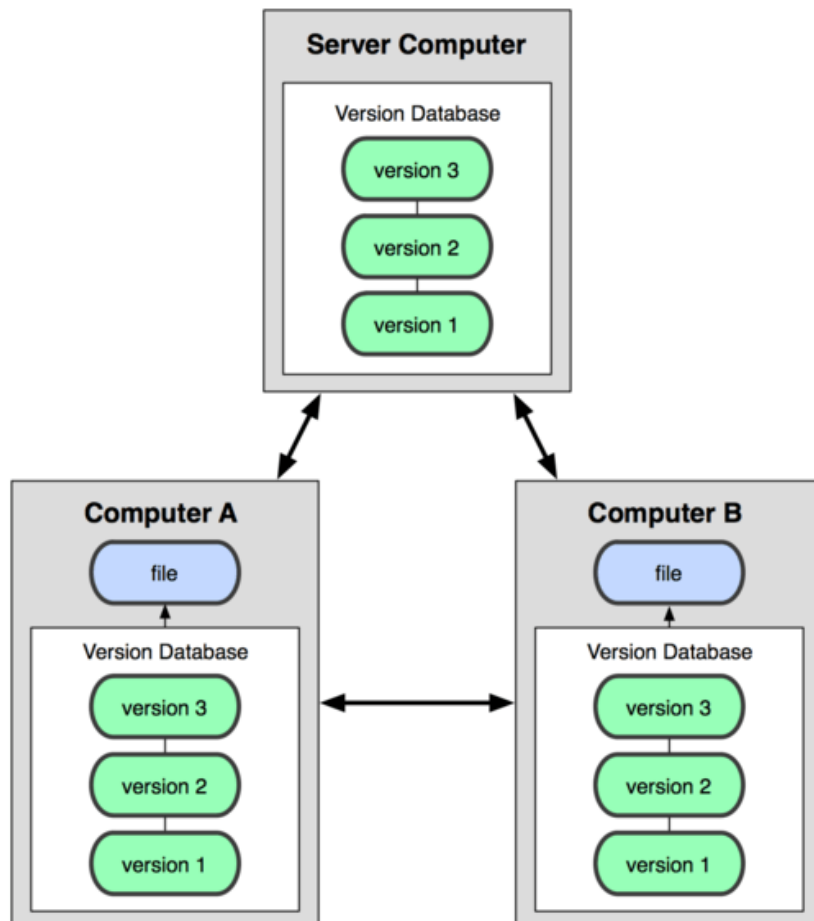
- CVS (1990)
- Perforce (1995)
- Subversion (2000)
- ❏ Central version database (server)
- ❏ Several remote checkouts (client)

Centralized SCM



- Operations (log, diff, blame) require a server

Distributed Version Control Systems

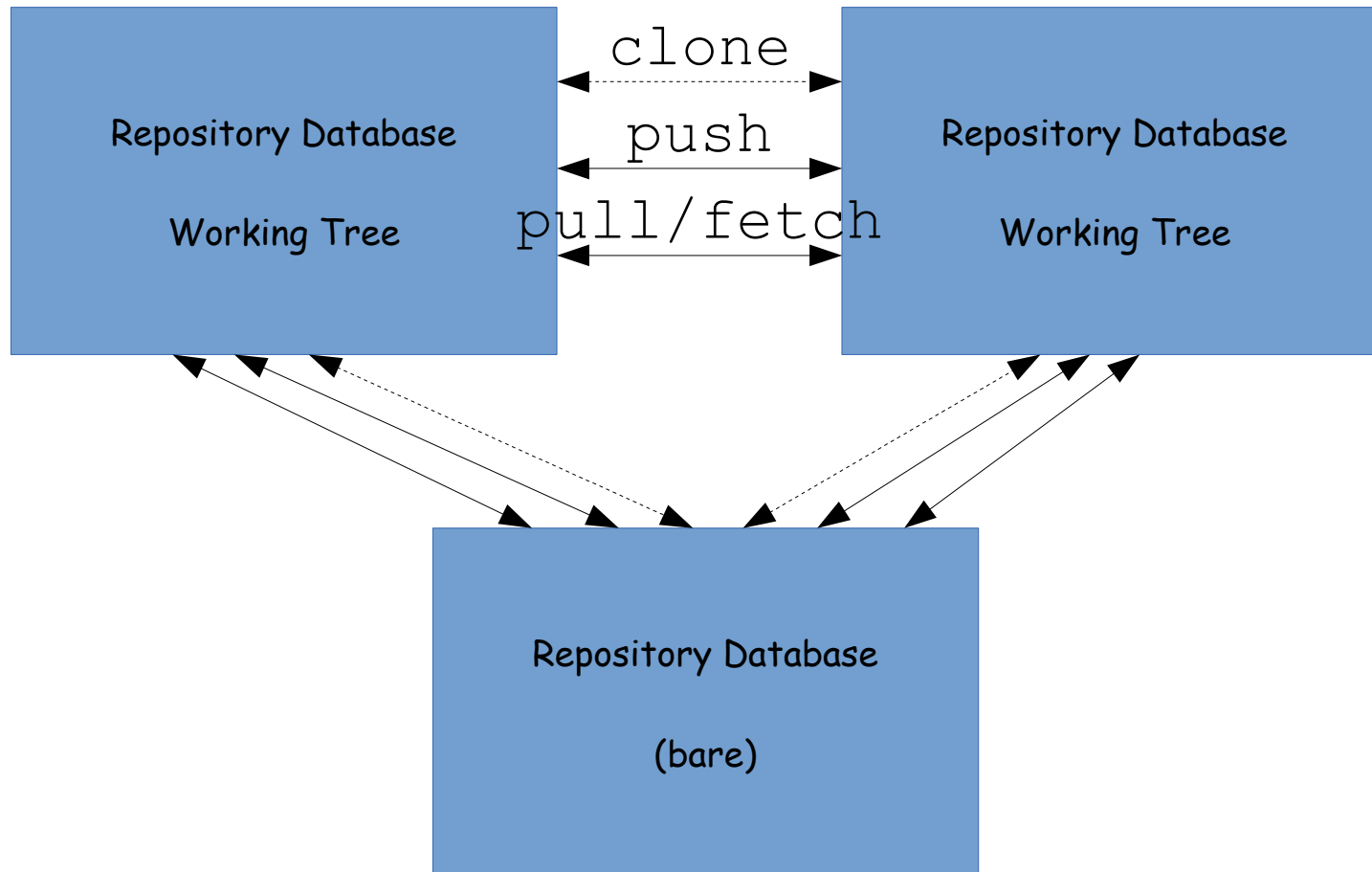


Scott Chacon CC BY-NC-SA 3.0

<http://www.git-scm.com/figures/18333fig0103-tn.png>

- Bitkeeper (proprietary - 1998)
- GNU arch (2001)
- Darcs (2002)
- Bazaar (2005)
- Git (2005)
- Mercurial (2005)
- ❏ Each "clone" can be used as server by other clones, and contains the complete history
- ❏ Each "clone" may or may not have a checkout

Decentralized SCM



- Anyone can be a server
- Operations (log, diff, blame) are performed locally
- "bare" repositories do not have a working tree

Git Brief History

- Linux Kernel
 - 1991-2002 Patches
 - 2002 BitKeeper
 - 2005 Linus Torvalds starts developing Git to host the kernel

Git Design

- Fully distributed
- Fast
- Strong support for non-linear development
- Able to handle large projects
- Cryptographic authentication of history

Git Repository Structure

Repository
Database

The History

Index

Staging

Working Tree

Files you can edit

Git "Bare" Repository Structure



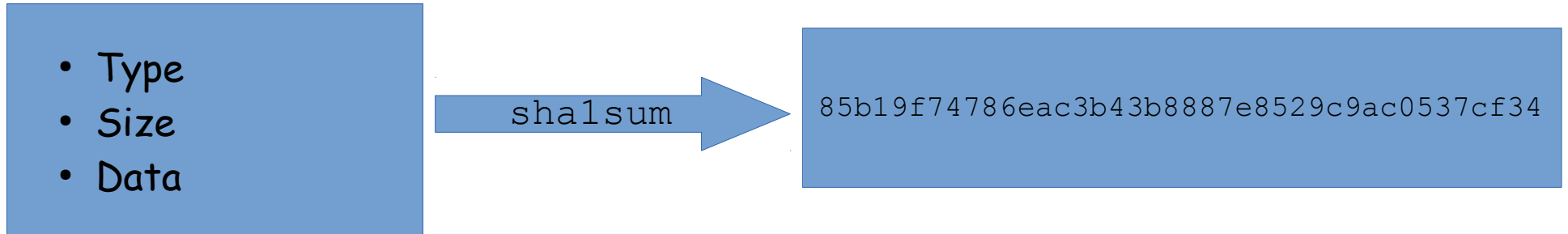
Repository
Database

The History

Repository

- Contained in the `.git` directory (or in the main directory for "bare" repositories)
- Files in the `.git` directory are never staged or committed
- `.git/config`: configuration of your repository
- `.git/info/exclude`: local patterns to ignore
- `.git/objects`: objects in your repository
- `.git/refs`: references (branches, tags, etc.)

Objects



- Each object is "compressed" and stored using the hash as name
- From the hash each object is addressable
- The hash also ensures that the content of the object cannot be manipulated
- Immutable (instead of modifying an object, a new one is created)

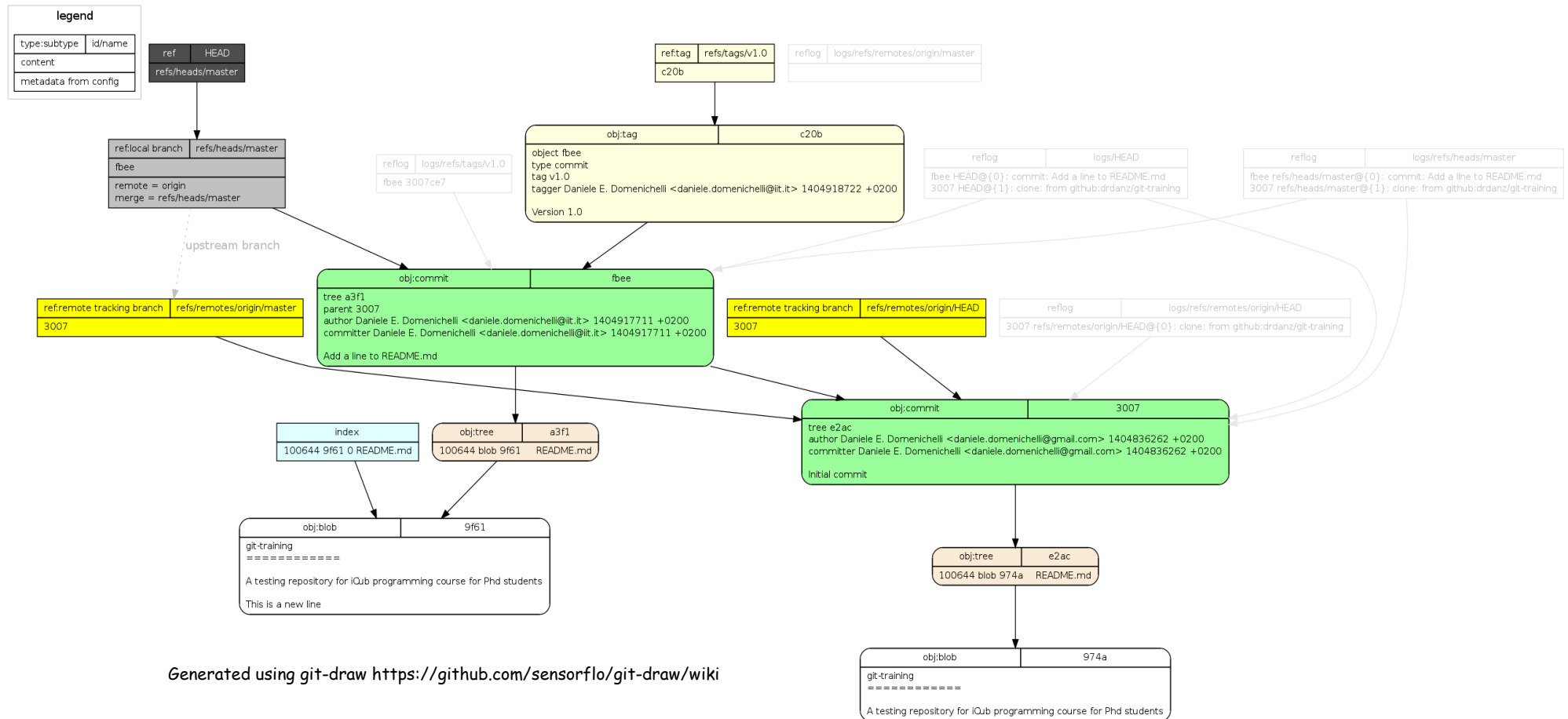
Objects Type

- **"blobs"** (file data)
- **"trees"** (directory) - For each linked object it includes
 - type (tree or blob)
 - file permissions
 - hash
 - name
- **"commits"**
 - Links to a tree
 - Include metadata about the commit (author and committer, time, commit message)
- **"tags"**
 - Links to a commit

Object References

- Full hash (85b19f74786eac3b43b8887e8529c9ac0537cf34)
- Short hash (85b19f74)
- Current checkout (HEAD)
- Branch (local) (master or refs/heads/master)
- Branch (remote) (origin/master or refs/remotes/origin/master)
- Previous branch (-)
- Tag (v2.3.21 or refs/tags/v2.3.21)
- One commit before HEAD (HEAD~ or HEAD~1)
- Some commits before master (master~5)
- The first parent of the current commit (HEAD^ or HEAD^1)
- The second parent of the current commit (for a merge commit) (HEAD^2)
- Two commits before head, then the second parent at the merge, then again three commits before, and the first parent at the merge (HEAD~2^2~3^1)
- Date, commit message...

Git Objects Example



- "Rounded corners" boxes are objects, the other boxes are references.
- White objects are **blobs**
- Pink objects are **trees**
- Green objects are **commits**
- Light Yellow objects are **tags**

Getting Help

- `git help`
 - List of common commands
- `git <command> --help`
- `man git-command`
 - man page for <command>
- `git <command> -h`
 - brief help for <command>
- Search the internet (**stackoverflow** has most of the answers, you just have to figure out the right question)
- Pro Git (Scott Chacon)
 - <http://www.git-scm.com/book>
- Some more useful links can be found here:
 - http://wiki.icub.org/wiki/Learning_more_about_git

Interactive Tutorials

- You should try them, even if you already know how to use git, they can teach you some new tricks.
- <http://try.github.com/>
- <http://pcottle.github.io/learnGitBranching/>

Disclaimer

- Git has thousands options, and new options are added in every release. Some of the described options might not be available on your git version.
- There are several ways with git to achieve exactly the same results.
- The following slides will introduce some of the most commonly used commands, you can learn a lot of other uses just by reading the man pages.
- I will not show how to use GUIs.

GETTING READY

Install Git

- Debian/Ubuntu

- `sudo apt-get install git`

- Windows

- **Git for Windows:** <https://msysgit.github.io/>
 - **TortoiseGIT:** <http://code.google.com/p/tortoisegit/>
 - **Atlassian SourceTree:** <http://www.sourcetreeapp.com/>

- OS X

- **Git-osx-installer:** <http://sourceforge.net/projects/git-osx-installer/>
 - Using MacPorts
 - Installed with XCode

- <https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools>

Global Configuration

- Command line

```
git config --global section.param "value"
```

- Edit your `~/.gitconfig` file or "git config --global -e"

```
[section1]
    param1 = value
    param2 = value

[section2]
    param1 = value
```

- You should do this in each computer you will use to work with git (just copying the `~/.gitconfig` file in all your computers will do)

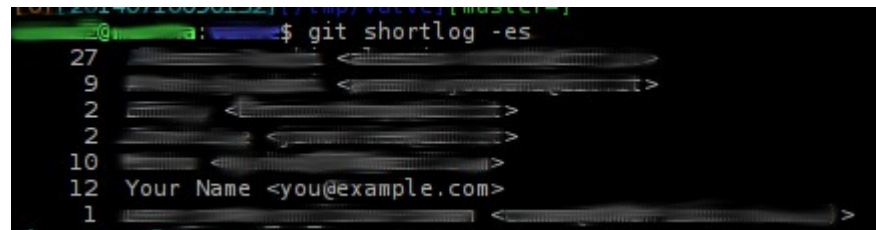
Identity

- The first thing you should do is to set your user name and e-mail address.

```
git config --global user.name "John Doe"
```

```
git config --global user.email "john.doe@example.com"
```

- Please, please, please replace "John Doe" and "john.doe@example.com" with **your real name and email!!**

A terminal window showing the output of the command 'git shortlog -es'. The output lists commit counts for various authors, with the last entry being '12 Your Name <you@example.com>'. The terminal has a dark background with light-colored text.

```
git shortlog -es
27 <redacted>
9 <redacted>
2 <redacted>
2 <redacted>
10 <redacted>
12 Your Name <you@example.com>
1 <redacted>
```


Some Color

- If you use git mostly from the command line, colors are very useful, you can enable them by running:

```
git config --global color.pager true
```

```
git config --global color.ui auto
```

- Status and diff are a lot more readable with colors.

Editor, Diff Tool, and Pager

- `git config --global core.editor "vim"`
- `git config --global merge.tool "vimdiff"`
- `git config --global core.pager "less -FRSX"`

End of Lines

- This avoids issues with Unix-style and Windows-style end of lines

```
git config --global core.autocrlf input (Linux/Mac)
```

```
git config --global core.autocrlf true (Windows)
```

Https Authentication Helper

- Caches https username and passwords for `<timeout> seconds`.

```
git config --global credential.helper \  
"cache --timeout=7200"
```

Auto Setup Rebase

- Automatically configures your branches to perform a rebase instead of a merge when you run "git pull"
- `git config --global branch.autosetuprebase "always"`
- **WARNING:** this is a possibly dangerous operation; do not use it unless you understand the implications. We'll discuss about rebase later.

Other Configurable Options

- More options are listed in `git-config` man page
 - `man git-config`
 - `git config --help`

Check Your Settings

- If you want to list all your settings, you can use:

```
git config -list
```

- Or if you want a specific value:

```
git config user.name
```

Generate an SSH Key

- Most of the remote git servers allow you to use several protocols to connect.
- Using SSH protocol with an SSH key is more secure than using username and password, and often faster.
- To create an SSH key you can just run:

```
ssh-keygen -t rsa
```


Bash Prompt

- Bash prompt can be tweaked to show you in which branch you are by using

```
$(__git_ps1)
```

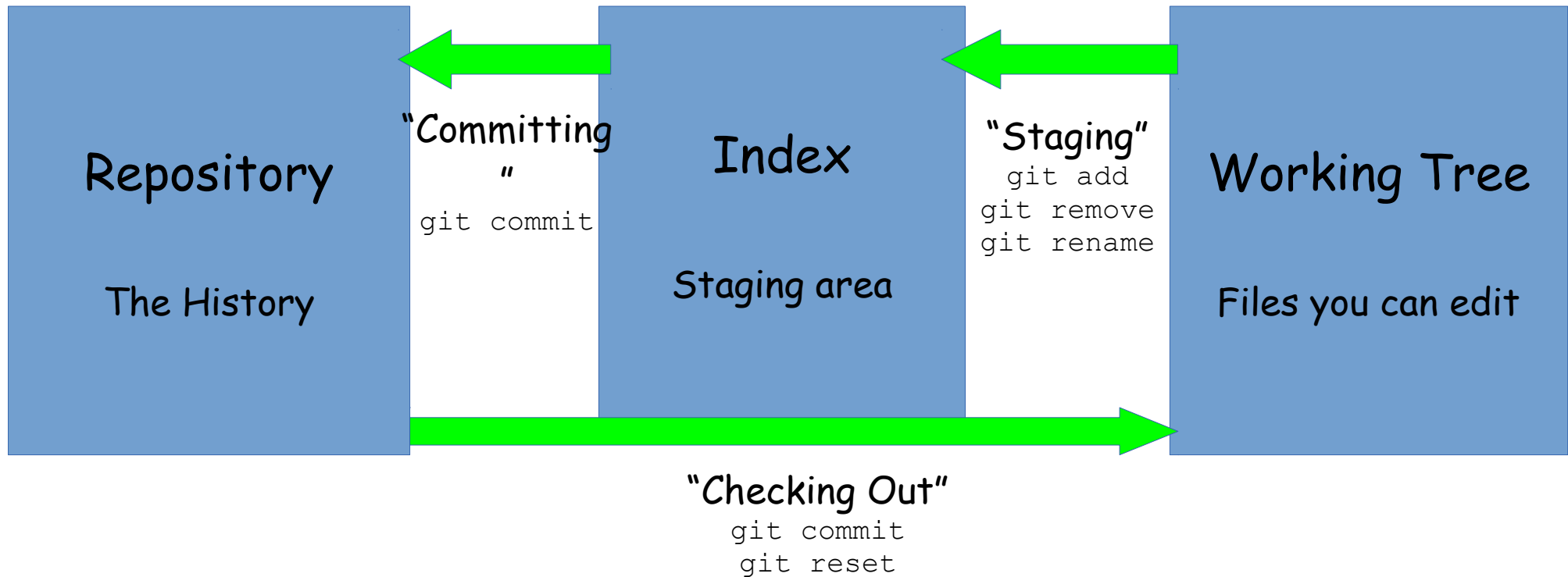
- You can add it to your PS1 variable (see http://wiki.icub.org/wiki/Learning_more_about_git for an example)
- This is very useful when working with more than one branch.

LOCAL WORKFLOW

Creating a git repository

- `git init`

Local Workflow



HEAD is a reference to the commit that you checked out in your working tree

Status

- `git status`
 - Shows staged, unstaged and untracked files
 - Do not be afraid to use it every command
 - Better with colors enabled

Staging

- `git add <file>`
- `git add .`
- `git add -p`
- `git rm <file>`
- `git mv <old> <new>`
- `git reset <file>`
- `git reset`
- `git reset -p`
 - Nothing was committed yet!
 - The "index" is updated
 - Check what changes after each command using "`git status`"

Restoring Files

- **WARNING:** These commands will delete your local changes and there is no way to recover them.
- `git checkout [--] <file>`
 - Discards all uncommitted changes to a file (the "--" is required in some cases, when git might be confused by the name of a branch)
- `git checkout <commit> [--] <file>`
 - Discards all uncommitted changes to a file and restore the file at <ref>
- `git reset --hard`
 - Discard all your uncommitted changes
- `git reset --hard <commit>`
 - Discards all your uncommitted changes and resets the current branch head to <commit>

Committing

- Svn style commits (bad practice)
 - `git commit -a -m "Log message"`
 - `-a` = all modified and trackable files in the working directory
- Staged files only
 - `git commit -m "Log message"`
- Commit on a shared machine, where your name and email address are not configured (e.g. on the iCub computers)
 - `git commit --author="John Doe <john.doe@example.com>"`

Amending Commits

- Amend latest commit
 - `git commit --amend`
 - **WARNING:** Rewrites the history! Do not do it if you already pushed the commit
- Discards all your uncommitted changes and resets the current branch to the previous commit
 - `git reset --hard HEAD~`
 - **WARNING:** Deletes all your local changes!
- `git reset HEAD~`
 - Resets the current branch to the previous commit, but does not touch your working tree, your files are safe.

Diff

- `git diff`
 - Shows diff between index and working dir
- `git diff --staged`
 - Shows diff between HEAD and index
- `git diff <object>`
 - Shows diff between object and working dir
- `git diff <object> <object>`
 - Shows diff between two objects

Show

- `git show`
 - "Shows" HEAD (displays the differences between HEAD and the previous commit)
- `git show <object>`
 - "Shows" a specific commit
- `git show --stat <object>`
 - "Shows" a commit, but just the stats, not the full patch

Log

- `git log`
 - Shows the log, starting from `HEAD`
- `git log <object>`
 - Shows the log, starting from `<object>`
- `git log -- <file or directory>`
 - Limit the log to the changes to a file or directory
- `git log <object1>..<object2>`
 - Limit the log to the changes since `<object1>` and until `<object2>`
- `git log --since="2 weeks ago" --until="yesterday"`
- `git log --oneline --graph --decorate --all`

Blame

- `git blame <file>`
 - Shows the file with annotations on each line about the commit that modified that line, and its author

Tag

- `git tag`
 - Shows the tags in the repository
- `git tag -a -m "Message" tag-name`
 - Creates a new annotated tag (in your local repository)

Branch

- `git branch (-l)`
 - Shows local branches only
- `git branch -r`
 - Shows remote branches only
- `git branch -a`
 - Shows all branches (local and remote)

Create local branches

- `git branch new-branch <commit>`
 - Create a branch "new-branch" on HEAD or on specified commit. Does not do a checkout of that branch
- `git checkout -b new-branch <commit>`
 - Create a branch "new-branch" on HEAD or on specified commit and do a checkout of that branch

Switching branch

- `git checkout new-branch`
 - If you have some changes to files that would be overwritten when switching branch, the operation fails, and does not try to do anything

Merge

- `git merge <branch> ...`
 - Create a merge commit with multiple parents.
 - If it results in a conflict, user intervention is required
- `git mergetool`
 - Shows the configured tool to resolve conflicts
- Or you can solve them manually and then fix the index using `git add`
- After fixing the conflict, commit using “`git commit`” (no parameters)

Rebase

- **WARNING:** Rewrites the history!
- `git rebase <commit>`
 - Takes all your commits and applies them onto `<commit>`
- `git rebase -i <commit>`
 - Let you choose what to do with each commit (you can move, modify, delete, squash commits)

Stash

- You have some code that you don't want to commit yet, but you want to switch branch to work on something else
 - `git stash`
 - “Stashes” your changes
 - `git stash pop`
 - Applies your stashed change to your current directory

Bisect

- Very powerful tool to find the commit that caused a bug
- Let you interactively test a commit, and marking it as `good` or `bad`, "bisecting" until the first bad commit is found.
- See `git bisect --help`

Revert, Cherry-Pick

- `git revert <commit>`
 - Applies the reversed patch from <commit> to the repository and creates a new commit
- `git cherry-pick <commit>`
 - Applies a patch from <commit> to the repository and creates a new commit
- `git cherry-pick -x <commit>`
 - Same, but add a line to the commit with the hash of the original commit (useful when cherry-picking from another published branch)

Reflog

- Reflog is a mechanism to record when the tip of branches are updated.
 - `git reflog [ref]`
- Where have I been?
 - `git reflog`
 - `git reflog HEAD`
- Where has my branch <branch> been?
 - `git reflog <branch>`
- Can be very useful to find a previous commit after an error or an unwanted operation
 - `git reset --hard HEAD{2}`
 - reset current branch to where `HEAD` used to be two moves ago
 - **WARNING**: deletes all local changes
- **WARNING**: Reflog expires

Grep

- `git grep`
 - Searches for a string in tracked files
 - Faster than system `grep` (that usually requires a `find`)
 - Searches only in tracked files
 - Does not search in the `.git` directory
 - Does not search in new files

REMOTE WORKFLOW

Working with remotes

- All the commands mentioned until now can be used locally, you don't need network or a server
- A remote can be
 - Another local repository
 - `file:///home/user/repo.git`
 - A remote repository
 - `https://github.com/robotology/yarp.git`
 - `ssh://git@github.com:robotology/yarp.git`
 - `git://github.com/robotology/yarp.git`
- Remotes can be read/write or read only

Clone

- `git clone <remote>`
 - Makes a local copy of a remote repository and makes a checkout of the remote repository HEAD
 - The remote is named "origin"
 - Creates a local branch "master" that corresponds to the remote branch "origin/master"
- `git remote add <name> <url>`
 - Add a new remote, does not download anything
- `git remote mv <name> <new-name>`
 - Rename a remote
- `git remote rm <name>`
 - Delete a remote

Fetch, Pull

- Someone made some commits on a remote repository and you want to import these changes locally
- `git fetch [remote]`
 - Retrieves all the new objects (commits, branches, tags) in the `[remote]` remote and saves them in your local `.git` folder, but does not apply anything to your working directory
- `git fetch --all --prune`
 - Retrieves all the new objects from all your configured remotes
- `git pull <remote> <branch>`
 - Retrieves all the new commits in the `<branch>` branch from the `<remote>` remote and merge them into your current branch (you get a new local merge commit)
 - Might result in a conflict
- `git pull --rebase <remote> <branch>`
 - Same as previous command, but does not create an extra merge commit. Instead it rebases your local changes on top of the remote branch.

Push

- You did some great work, and some commits locally, and now you are ready to make your changes available to all the others. Assuming that nobody else sent commits on the repository since your last pull

- `git push <remote> <branch>`

will push all the missing commits in your `<branch>` that are not in the `<remote>/<branch>` branch to the `<remote>` remote. The remote will update its local `<branch>` to your latest commit.

- Commits must be fast-forward only (i.e. you append new commits to the branch, you don't modify any of the past commits), otherwise some commits could be lost.
- You can force a non fast-forward push. There are some reasons why you would do that, for example in your own private repository, or because someone messed up with the history

- `git push --force <remote> <branch>`

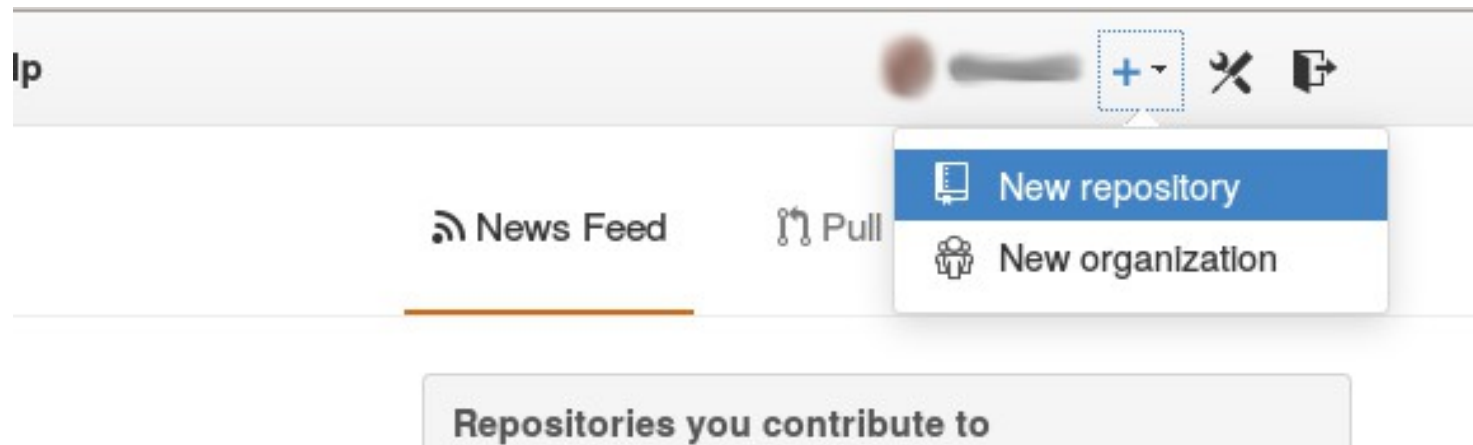
Now forget about it, and never do it on a shared repository.

Tips

- Discard all the local changes and restore a branch to the status on the remote
 - `git fetch origin`
`git reset --hard origin/<branch>`

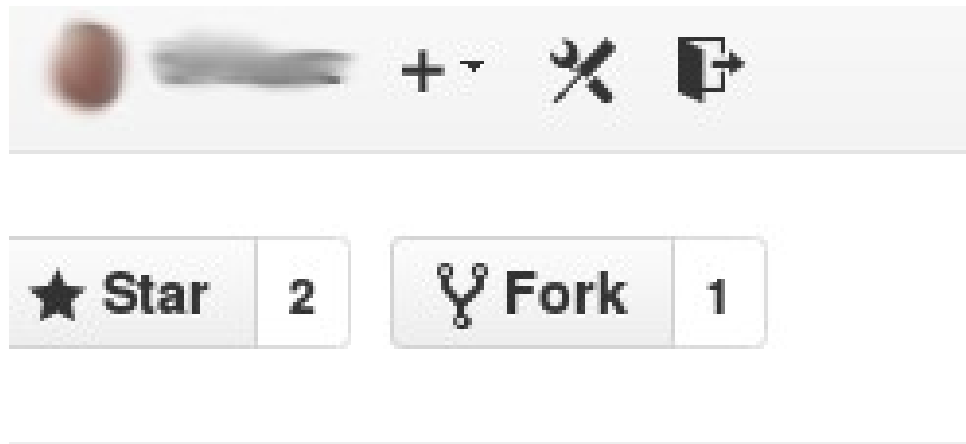
WORKING WITH GITHUB

Creating a git Repository on GitHub



- Get an account if you don't have one
- Log in
- Click on "New repository"
- Follow the instructions

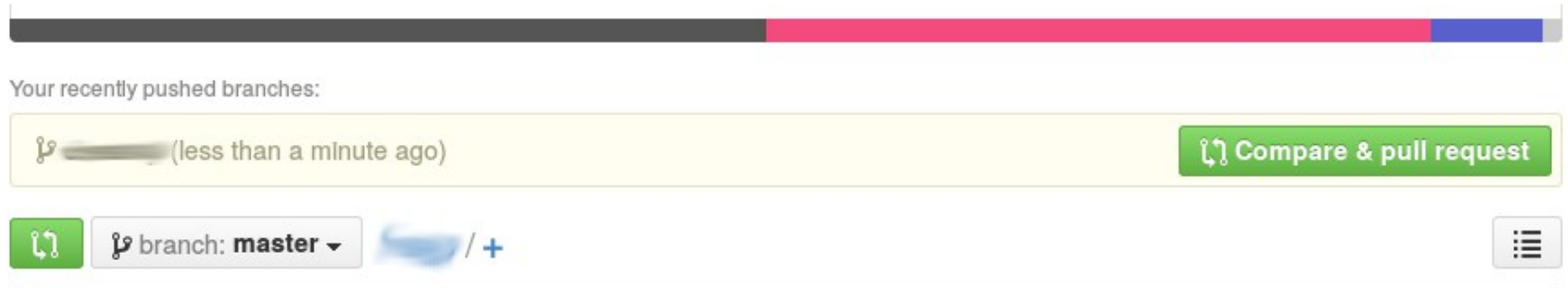
Forking a Repository on GitHub



- Open the page of the repository you want to fork
- Click the fork button
- Add the new remote to your local clone

```
git remote add <remote name> git@github.com:<username>/<repository name>.git
```

Creating a Pull Request on GitHub



- Push your changes to a branch in your fork
 - `git push <remote name> <branch>`
- Open the main page of your fork
- Click the "Compare & pull request" button
- Write a comment and click "Create pull request"

GOOD PRACTICES

Good practices 1:

What am I pushing?

- Always ask yourself this question before pushing on a shared repository
- Use `git log (<remote>/<branch>..<branch>)` and gui tools (`git gui`, `gitk`, `gitg`, `qgit`, `git cola`) to ensure you are not pushing useless stuff
- Get rid of useless commits and merges using `git rebase`
- Review your own commits before pushing (`git show` and `git log --patch`), follow guidelines (for YARP avoid tabs and trailing white spaces)
- Read the manuals, search the internet, if you are still not sure ask someone before pushing

Good Practice 1: Example

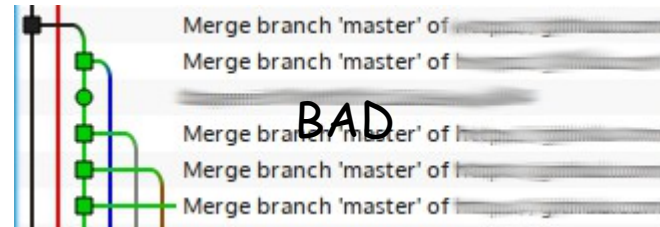
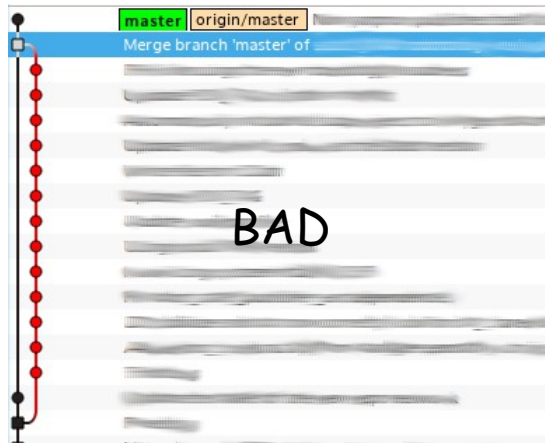
●BAD

- `git commit -a -m "Fix bug"`
- `git pull`
- `git push origin master`

●GOOD

- `git add <file>` (even better stage the changes one by one using `git add -p <file>`)
- `git diff --cached` (always check what you are pushing)
- `git commit` (enter a proper commit log)
- `git pull --rebase origin master` (rebase your changes over the remote branch instead of creating an useless merge commit)
- `git log --oneline --graph --decorate origin/master..master` (always check which commits you are pushing)
- `git log -p origin/master..master` (always check the content of the commits that you are pushing)
- `git push origin master`

Good Practice 1: Example



- "Merge branch 'master' of ..." usually means that you performed a "git pull" without rebasing your changes. One commit like this is bad enough, several developers doing this make a huge mess
- Use "git pull --rebase".
- master history should always be on the left, this kind of commits makes the history dirty and hard to understand.
- If you cannot rebase, (for example for a shared branch that is developed for a long time) then use "git pull", and either rebase the whole branch when it is ready to be merged, or merge back your changes into master before pushing to master, and not the opposite

Good practices 2:

Write good log messages

- Git log is very powerful, but with poor log messages, it's useless
- Do not commit with empty or useless log messages
 - Brief explanation of the commit
 - Leave an empty line
 - Detailed information about the commit
 - Use `"git commit --amend"` to edit the commit message of the last commit or use `"git rebase -i"` to edit the commit message of a commit that is not the last (do not edit published commits)

Good Practice 2: Example

- BAD

- `git commit -a -m "Fix bug"`

- GOOD

- `git commit` (enter a proper commit log)

Good practices 3:

Work in branches

- Create a branch for each bug you start fixing, or for each new feature you start to work on
- Switching branch is very easy and fast
- Merge into master only when you are ready, and you are sure that you will not break the build
- `master` should be always in a stable and releasable state.
- Feature branches are usually merged with "`git merge --no-ff <branch>`". The `--no-ff` options forces git to create a merge commit even if the history could be linear.

Good practices 4:

Let someone else review your code

- Nobody writes bug-free code and knows all the best practices for writing code. Having someone else review your code helps reducing them.
- YARP and iCub use GitHub as main repository. GitHub has "pull requests". If you are writing a non-trivial feature, push you commits in a branch on your private fork and open a "pull request" to ask to someone else to review your code
- Other projects use other tools (gerrit, reviewboard) or a hierarchical access to repositories.

Good practices 5:

Never rewrite published history

- There are very few reasons why published history should be rewritten. Some projects enforce fast forward commits only. Unfortunately GitHub allows to do it only if you pay for your account.
- Never `push --force` on a public repository.
- Never ever `push --force` on a public repository.
- If your push fails, and you are tempted to `push --force`, you are doing it wrong. `pull` or `rebase`, and try again.

Questions?

Let the Fun Begin

- Install and configure git
- Clone the repository at
`https://github.com/drdanz/git-training`
- Make a local commit
- Push your commit
- Pull someone else's commits
- Fork the repository on GitHub
- Make a "feature branch", make some commits and merge them into master
- Make another "feature branch", make some commits, push it to your fork and create a merge request