POLITECNICO DI MILANO





Advanced Operating Systems

Control Versioning with GIT

Giuseppe Massari

giuseppe.massari@polimi.it

Outline

Why using version control tools?

Single developer

- Do we need to keep track of the history of changes in a software project?
- How to *effectively* – keep track of / store – such changes?
  - Copying sources into new directories (e.g., MyProject_v20151006_...)?
  - What about disk occupancy?

Multiple developers

- How to merge changes and additions coming from other developers?
  - Manual *copy-and-paste* for each changed file?
- How to keep track of who's the author of a change?
  - Mmm... maybe looking for mails with source code attached?

Why Git?

Distributed approach

- The entire project history database is *local*
  - → Faster browsing among the changes
  - → No traffic generated on the network
  - → Changes can be committed locally before sharing
  - → Higher reliability: *Central server DOWN is not an issue*!

Integrity mechanisms

- Every change is tracked
- SHA-1 checksum

  40-character string composed of hexadecimal characters
  calculated based on the contents of a file or directory structure in Git.

Help

- The command-line usage of Git is based on commands
- To get help about a specific command syntax

```
$ git help [command]
```

- If no command is given an overview of the most common Git commands is returned

```
usage: git [--version] [--help] [-C <path>] [-c name=value]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

The most commonly used git commands are:
   add        Add file contents to the index
   bisect     Find by binary search the change that introduced a bug
   branch     List, create, or delete branches
   checkout   Checkout a branch or paths to the working tree
   clone      Clone a repository into a new directory
   commit     Record changes to the repository
...
```

Configuration

- Setup user name, mail and preferences...

```
$ git config <level> <option> value
```

- Configuration "levels" reference three possible configuration files

`--system`	`/etc/gitconfig`	System wide scope
`--global`	`~/.gitconfig` `~/.config/git`	User scope
	`.git/config`	Local project scope

- First configuration steps

```
$ git config --global user.name "Giuseppe Massari"
$ git config --global user.email giuseppe.massari@polimi.it
```

Basic concepts

- *Repository*

  Concretely, (sub)directory where in Git stores metadata and object database for the project (`.git`)

- *Working directory*

  A checkout of one version of the project

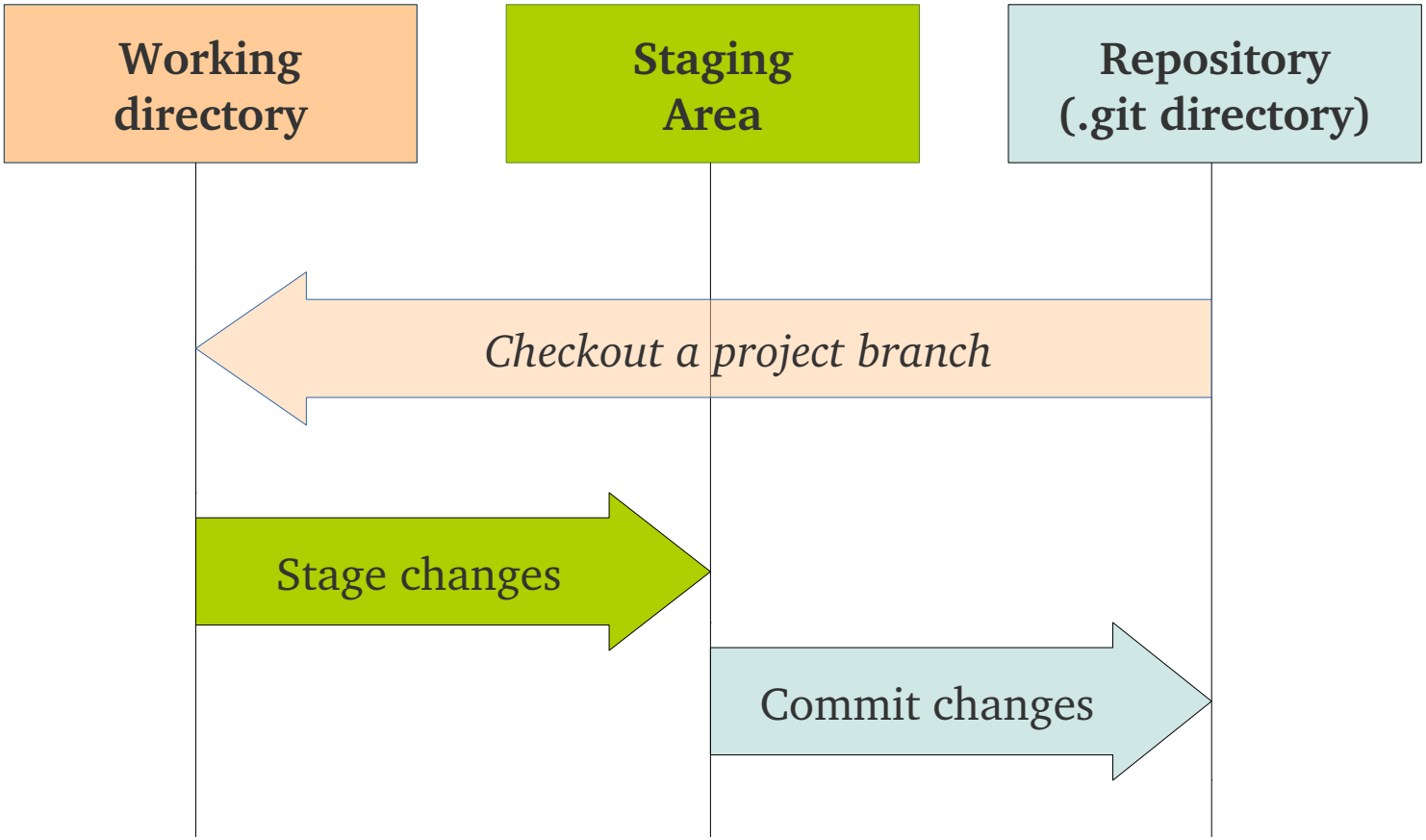
  The tracked files pulled out of the compressed database in the `.git` directory

- *Staging area*

  The set of selected changes, ready to be committed

  In practice, a file ("INDEX") contained in `.git` directory, storing information about the changes to track in the forthcoming next commit

Basic workflow



Your first Git repository

- Enter the new project directory
- Initialize the Git repository

```
$ cd ~/Development/MyProject
$ git init
Initialized empty Git repository in ~/Development/MyProject/.git/
```

This will create the .git directory. From now on, you can track changes in our project committing them in the Git repository

```
~/Development/MyProject/
└── .git
    ├── branches
    ├── config
    ├── description
    ├── HEAD
    ├── hooks
    ├── info
    ├── objects
    └── refs
```

All the “magics” behind Git is here

Local repository management

Your first Git repository

- Create some files and do you first commit

```
$ touch main.cpp
$ git add main.cpp
$ git commit -s
```

- Now we have the very first step of our project *history*

```
$ git log

commit 8c837e5e223ee55bb9c9d1818fca67b8a365e94a
Author: Giuseppe Massari <joe.massanga@gmail.com>
Date:   Fri Oct 16 13:59:53 2015 +0200

    Initial commit

    Signed-off-by: Giuseppe Massari <joe.massanga@gmail.com>
```

Diff

- As we continue introducing changes we can list them for each file
  - If no files are specified all the changes for each file are shown

```
$ git diff [file-name]
```

```
diff --git a/main.cpp b/main.cpp
index e69de29..d9d9396 100644
--- a/main.cpp
+++ b/main.cpp
@@ -0,0 +1,9 @@
+#include <iostream>
+
+int main(int argc, const char *argv[])
+{
+       std::cout << "Hello, GIT World!" << std::endl;
+
+       return 0;
+}
+
```

Local repository management

Status

- After performing some changes we may want to have an overview of the project status
- The command *status* recaps the status of the working directory and the staging area

```
$ git status
```

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

      new file:   main.h

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

      modified:   main.cpp

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

      README
```

Staging area

Local repository management

Commit

- What it is?

  A *tracked* change in the project: new files or changes in already existing files

- When to do it?

  Whenever we consider the changes introduced as a “stable” part of the project development

- What TO commit?

  Whatever can be considered a source file

- What TO DO NOT commit?

  Whatever is generated from source files

- How to commit?

  Add files or changes in files to staging area and then ***commit***

```
$ git add <files>
$ git commit ...
```

How to write a “good” commit?

A “tag/label” to easily identify the project module touched

A brief title to summarize the changes introduced

```
[Model] Power-thermal models support

The purpose of this support is to providing a well-defined interface to
power/thermal resource allocation policy, hiding platform specific details.
A ModelManager allows the registration of platform-specific models,
according to the target platform selected.
Such models are implemented as derived classes of the base class Model.

Signed-off-by: Giuseppe Massari <giuseppe.massari@polimi.it>
```

Author's signature (`git commit -s`)

An exhaustive explanation of what has been done and why

Local repository management

How to write a “good” commit?

- Include only consistent changes!

  A single source file can include changes having different goals

  - Bug fixing
  - Code refactoring
  - New functionality implemented

  Use ***git gui*** to be more productive!

```
$ git gui
```

  - Select lines (also from different files) changed for the same single specific purpose
  - Right-click on changed lines
  - Select “*Stage lines for commit*”

Local repository management

Log

- To recap the *history* of the project development

```
$ git log
```

```
commit 7bb21c0837093be82d391ad9d03d5d8235193704
Author: Giuseppe Massari <joe.massanga@gmail.com>
Date:   Mon Oct 19 18:06:25 2015 +0200

    [Main] Using a People object

    Added an instance of People object (joe), walking and greeting.

    Signed-off-by: Giuseppe Massari <joe.massanga@gmail.com>

commit 5124691db36e150b2839fd923b3ca71186b22fe6
Author: Giuseppe Massari <joe.massanga@gmail.com>
Date:   Mon Oct 19 18:03:08 2015 +0200

    [People] Class first version

    Including member functions Walk() and Greet().

    Signed-off-by: Giuseppe Massari <joe.massanga@gmail.com>
```

Show

- To show commit details (description + changes)

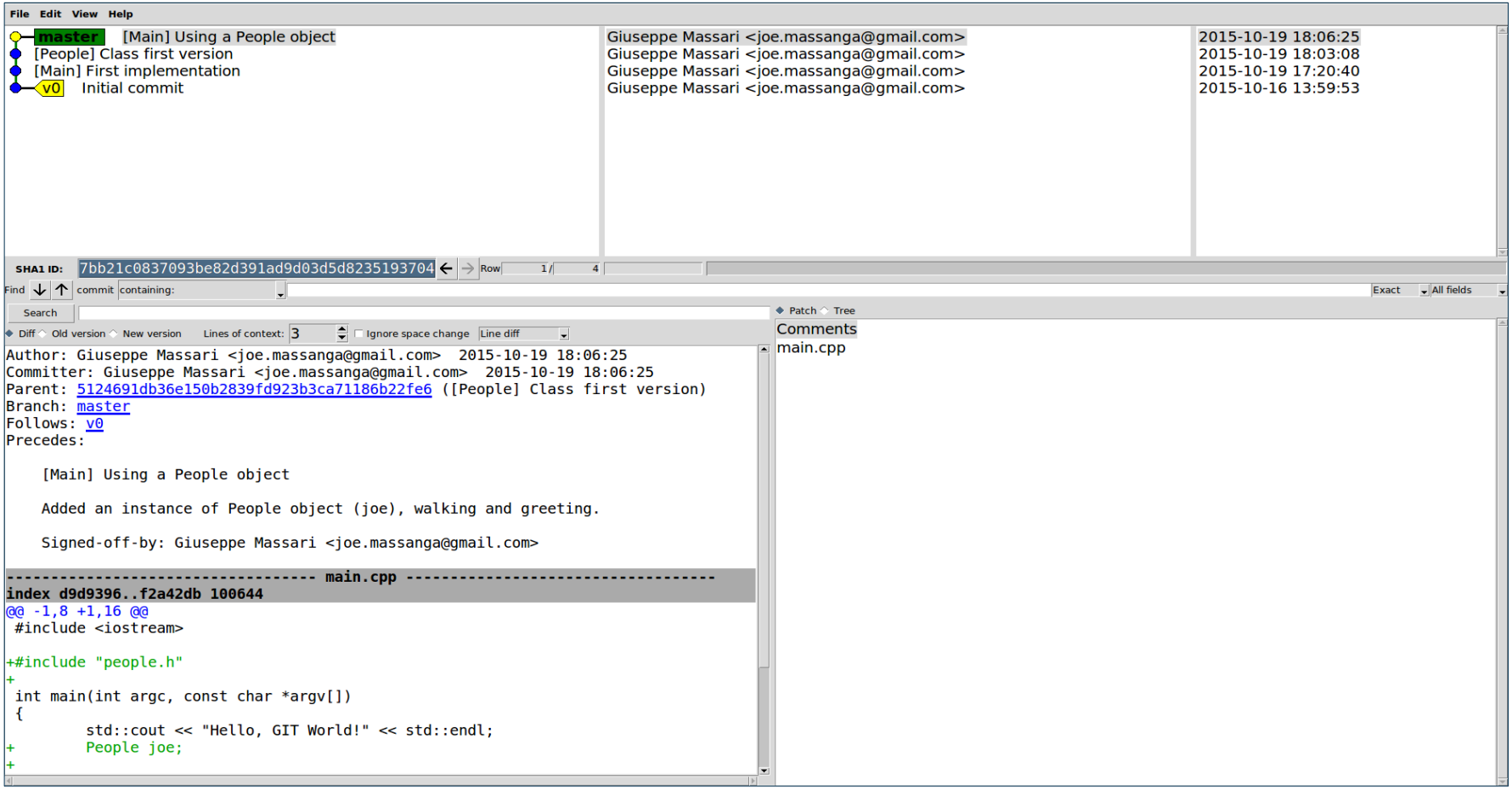
```
$ git show [SHA1_number]
```

```
giuseppe@plutone:~/Development/Test/GitTutorial$ git show 5552cef
commit 5552cefd12be253fd10c50cd03b64525eb0217f5
Author: Giuseppe Massari <joe.massanga@gmail.com>
Date:   Mon Oct 19 17:20:40 2015 +0200

    [Main] First implementation

    The main function now prints a message.

    Signed-off-by: Giuseppe Massari <joe.massanga@gmail.com>

diff --git a/main.cpp b/main.cpp
index e69de29..d9d9396 100644
--- a/main.cpp
+++ b/main.cpp
@@ -0,0 +1,9 @@
+#include <iostream>
+
+int main(int argc, const char *argv[])
+{
+       std::cout << "Hello, GIT World!" << std::endl;
+
+       return 0;
+}
+
```

Local repository management

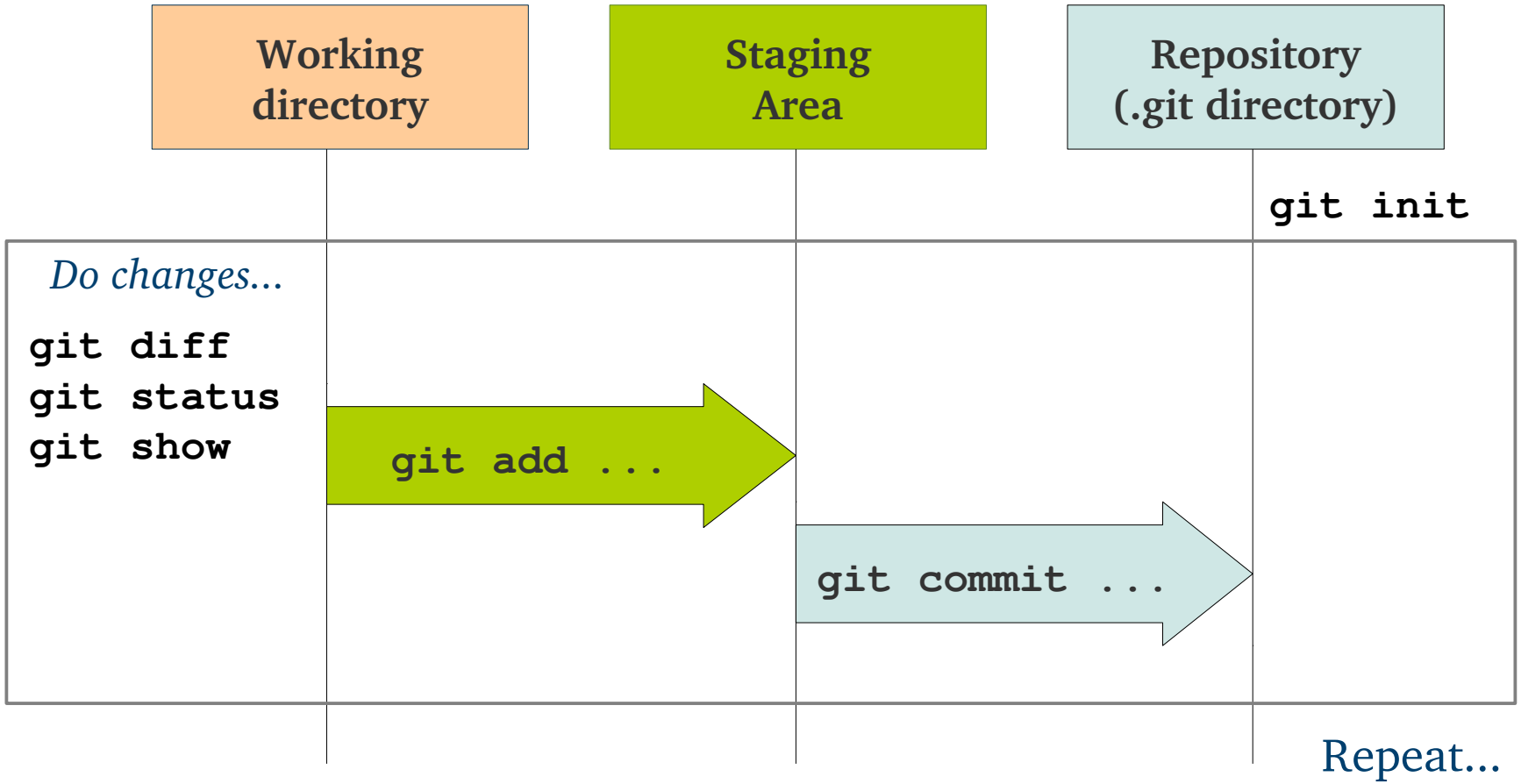
Graphical front-ends

```
$ gitk --all
```



Local repository management

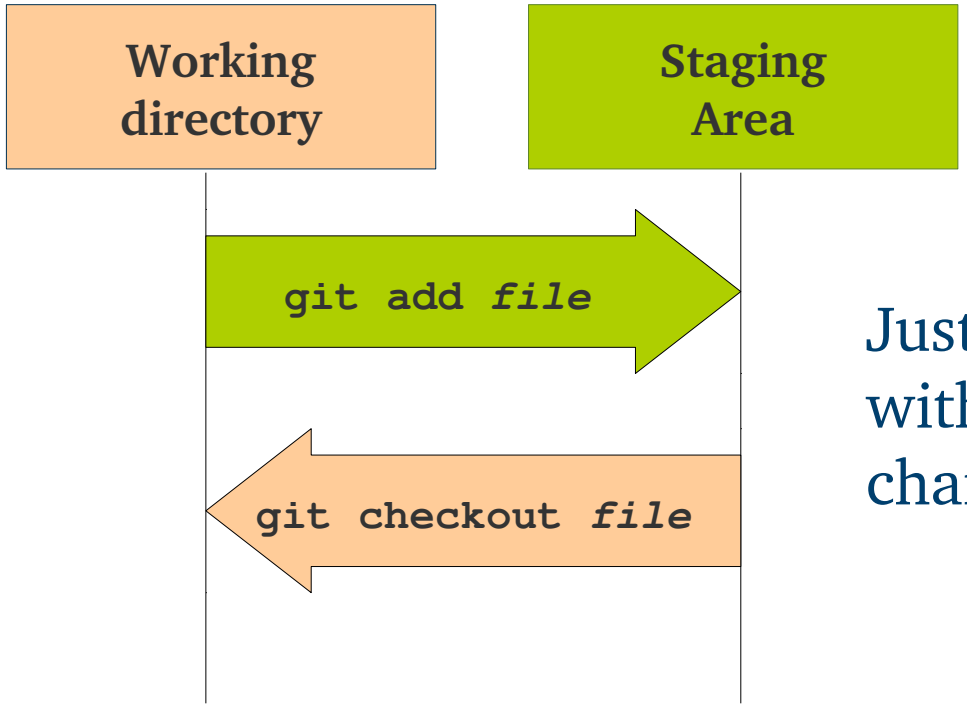
Basic work-flow in commands



Undo commands

- How to “unstage” changes already added to the *staging area*?

```
$ git checkout filename
```



Just a step back, without cleaning the changes in the sources

Undo commands

- How to clean out unstaged changes?

  Clean out all the (unstaged/uncommited) changes in the tracked files

```
$ git reset --hard
```

- How to “correct” a commit?

```
$ git commit --amend
```

  “Open” the current commit and allows us to add/remove files/lines or modify the commit message

  It overwrites the commit, creating a new one (with a different SHA-1)

  Not recommended if the commit has been already pushed on the remote repository

```
$ git revert SHA-1_number
```

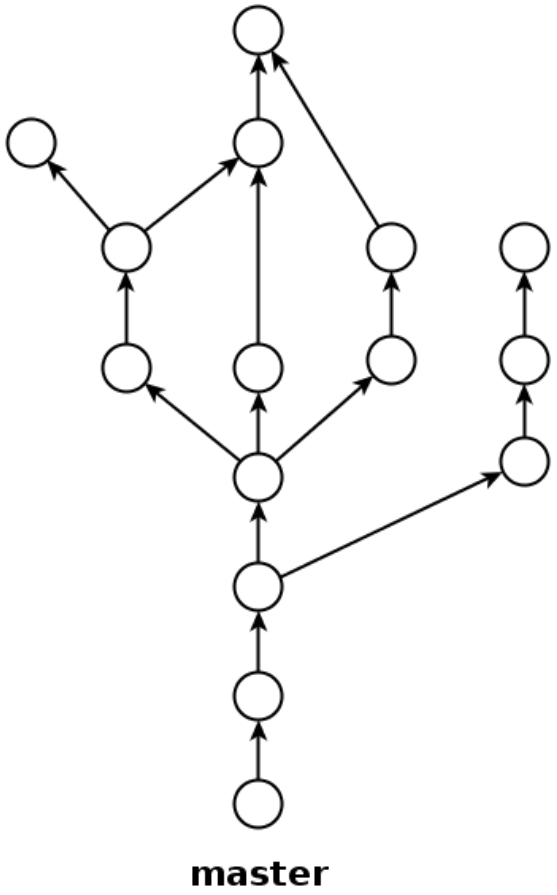
  Revert the changes of the specified commit

  - *Added* lines become *removed* lines and viceversa...

Local repository management

Branches

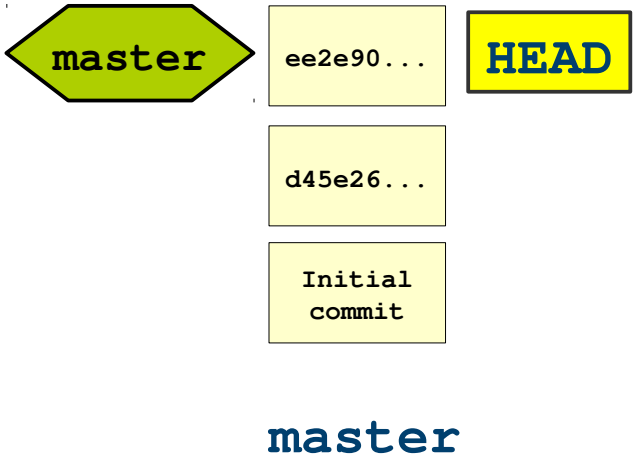
- In real projects, the development history is far from progressing on a straight line
- Split developers to work in parallel, each focusing on specific ***features***
- Software versions targeting different ***versions*** following different paths
- The concept of *branch* is all about this
- By default the Git repository initialization creates the *master* branch



Branches

- Creating (and switching to) a new branch

```
$ git checkout -b dev
```



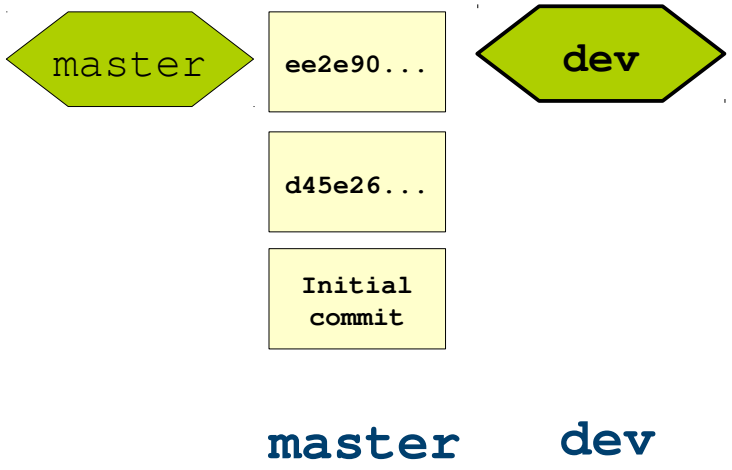
HEAD is basically a reference to the topmost commit on the current branch

Branches

- Creating (and switching to) a new branch

```
$ git checkout -b dev
```

Command **checkout** action is different in case the argument is a branch, instead of a file name

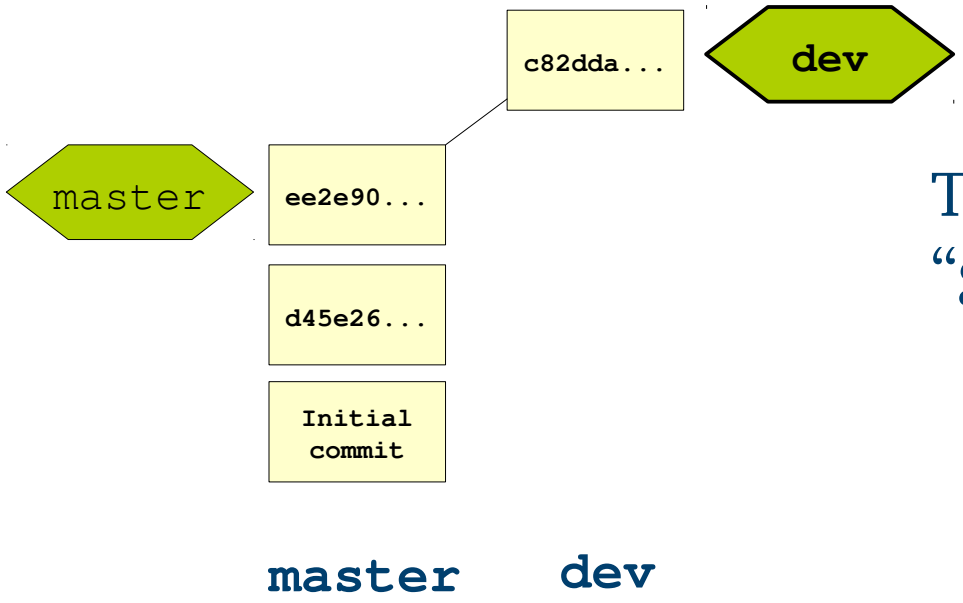


The new *dev* branch points to the same topmost *master* commit

Branches

- Committing on top of a new branch

```
$ git commit ...
```



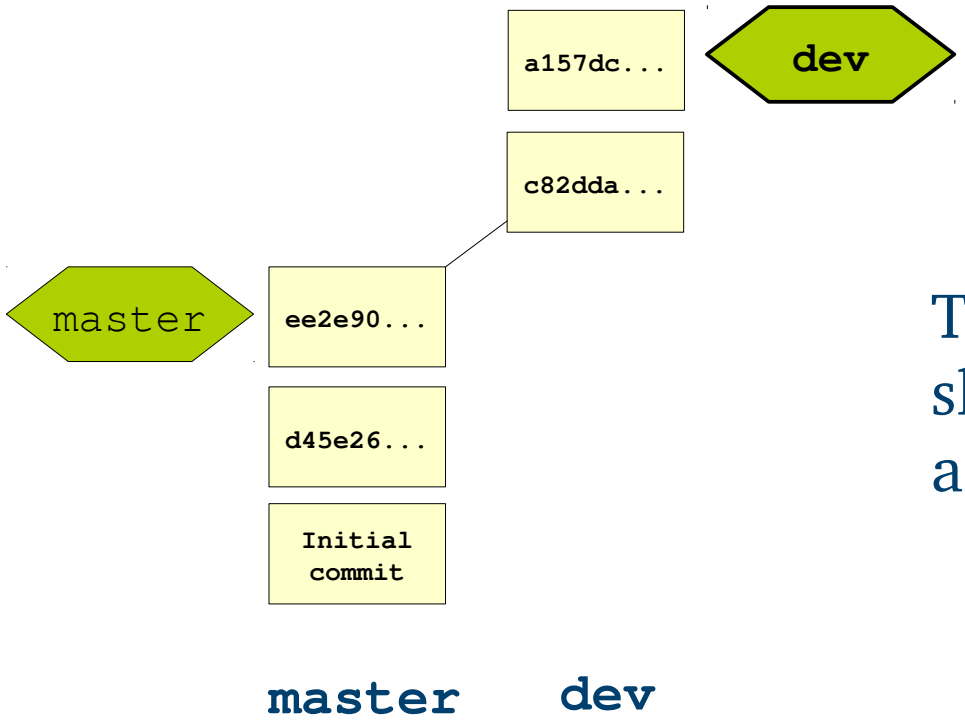
The *dev* branch starts "growing"

Local repository management

Branches

- Committing on top of a new branch

```
$ git commit ...
```



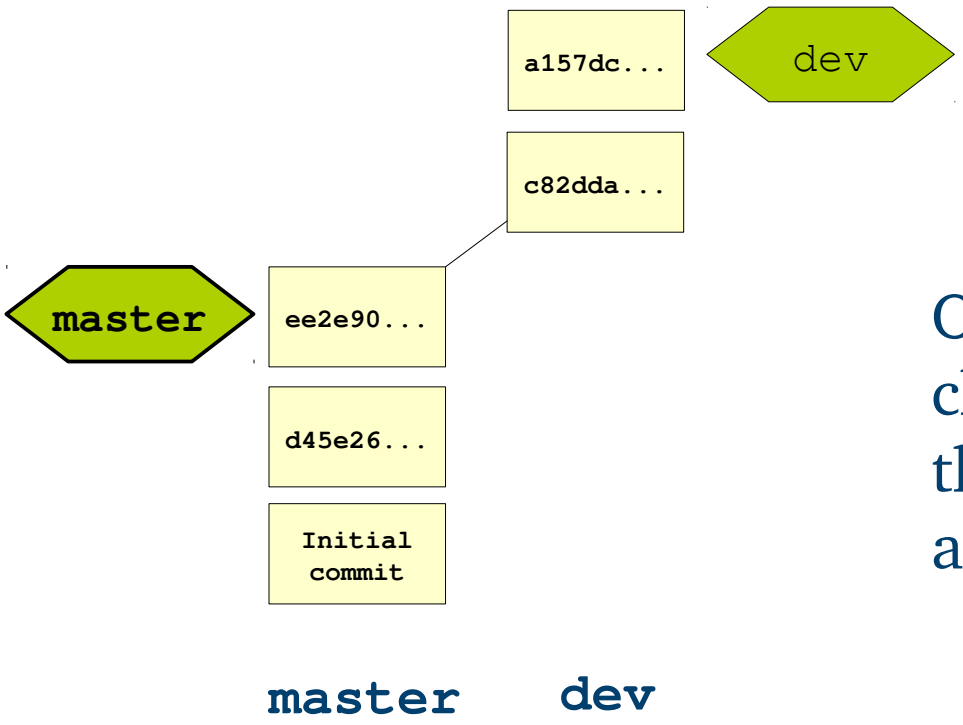
The project history is slowly starting to show a “tree” shape

Local repository management

Branches

- Switching back to *master* branch

```
$ git checkout master
```



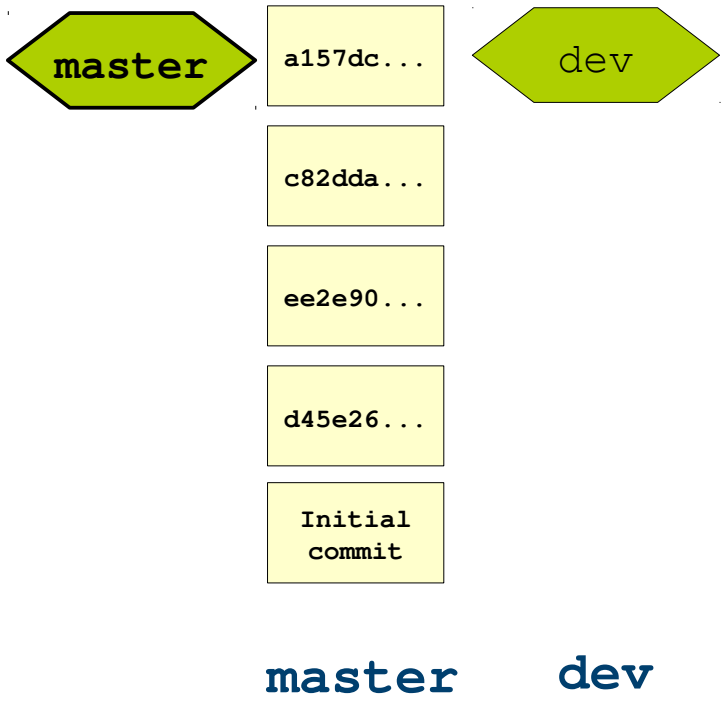
Once back to *master* the changes introduced in the last *dev* commits apparently “disappear”

Local repository management

Branches

- Merging *master* and *dev* branches

```
$ git merge dev
```



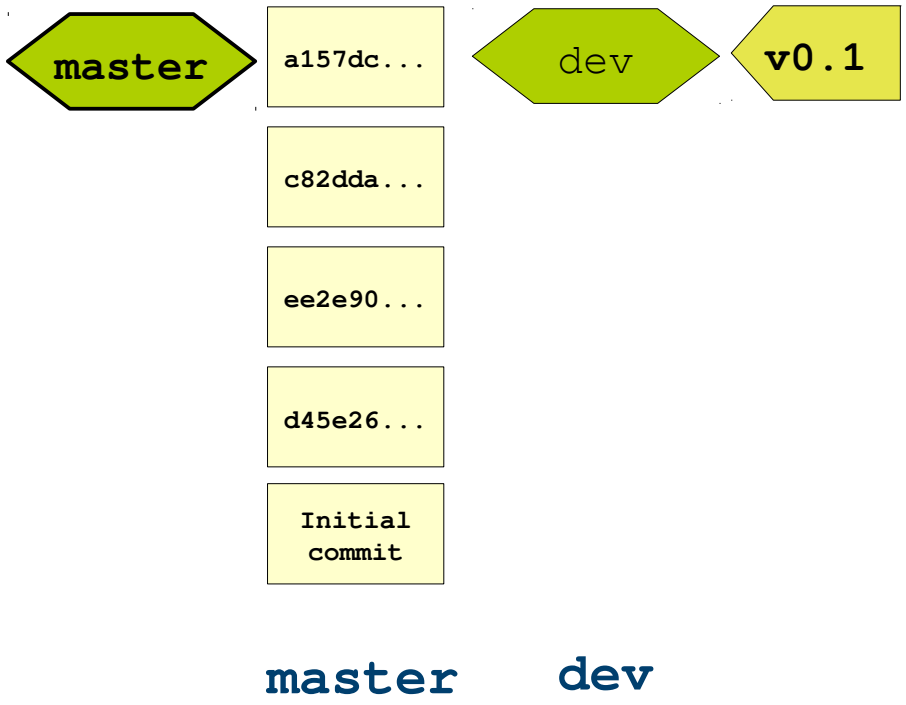
Now, if we switch between *master* and *dev* we would not see changes in the project

Local repository management

Branches

- Labels on commits

```
$ git tag -a v0.1 a157dc
```



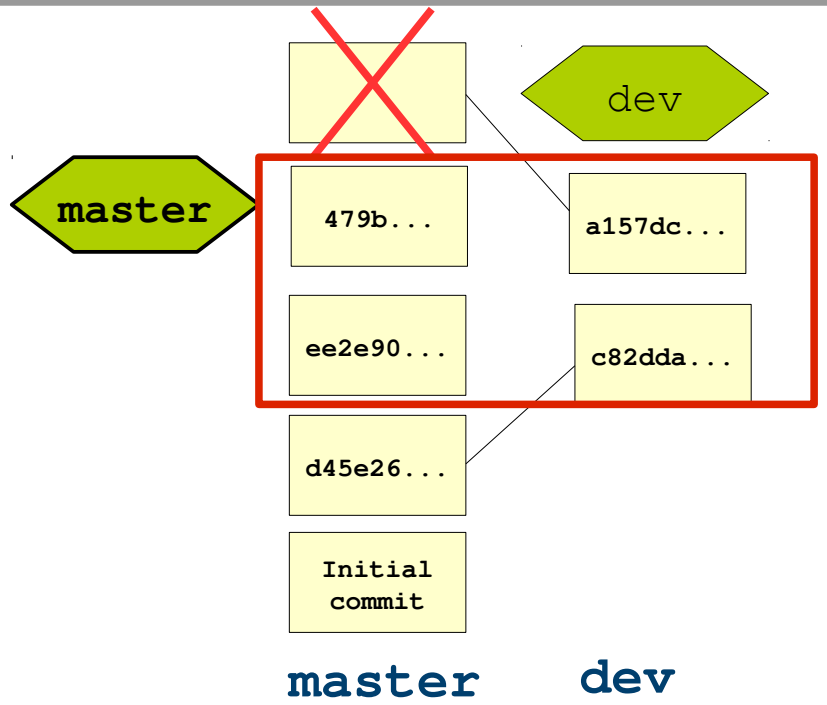
The tag can be thought as a label to mark our releases and improve the git tree readability

Conflicts

- In some cases, merging branches may lead to conflicts

```
$ git merge dev
```

```
CONFLICT (content): Merge conflict in main.cpp
Automatic merge failed; fix conflicts and then commit the result.
```



Master and dev branches include commits that touched the same lines of the same file

Local repository management

Conflicts

- The status command reports the conflict in this way

```
$ git status dev
```

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      main.cpp
```

- To solve the conflict the conflict we need to open the files involved and check the content
- Git automatically add “special lines” into the file to identify the conflict and help us

Conflicts

- main.cpp is the file containing the conflict

```
 7 <<<<<<< HEAD
 8     std::cout << "Hello, walking man!" << std::endl;
 9 =======
10 >>>>>>> dev
```

Lines 7-9: HEAD status, changes in commit on top of current branch (*master*)

Lines 9-10: changes coming from the merging branch (*dev*)

- Resolution

Make a choice about the lines to keep or deleted, remove

Delete special lines “<<<<...====...>>>>”

Add the files to the staging area (INDEX)

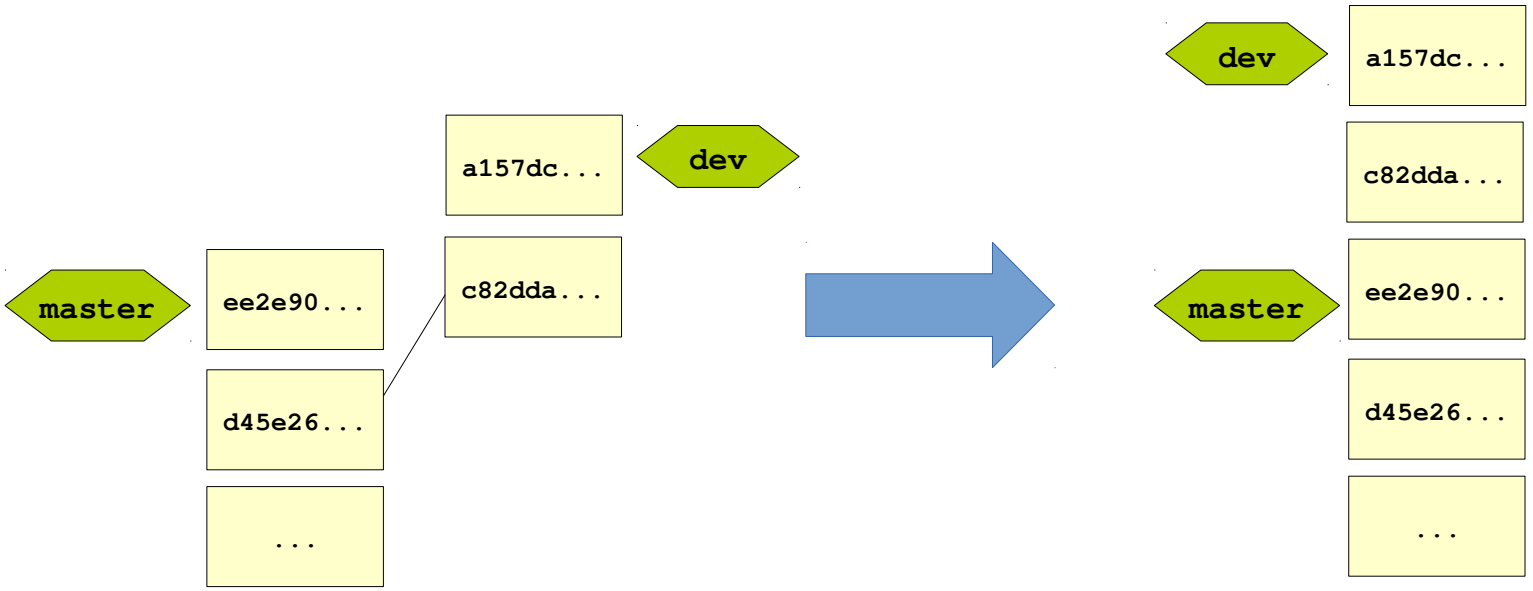
Commit

Local repository management

Rebasing

- Move a branch on top of another
  - May lead to conflicts
  - Do it as long as your are moving a local branch

```
$ git rebase master dev
```



Stashing

- Temporarily hide current (uncommited) changes on a “stack”

  Useful in case of switching on a branch already including changes on files we currently working on...

```
$ git stash save “Changes about feature1 implementation”
```

- I can stack multiple stashed changes

```
$ git stash list
```

- To discard stashed changes:

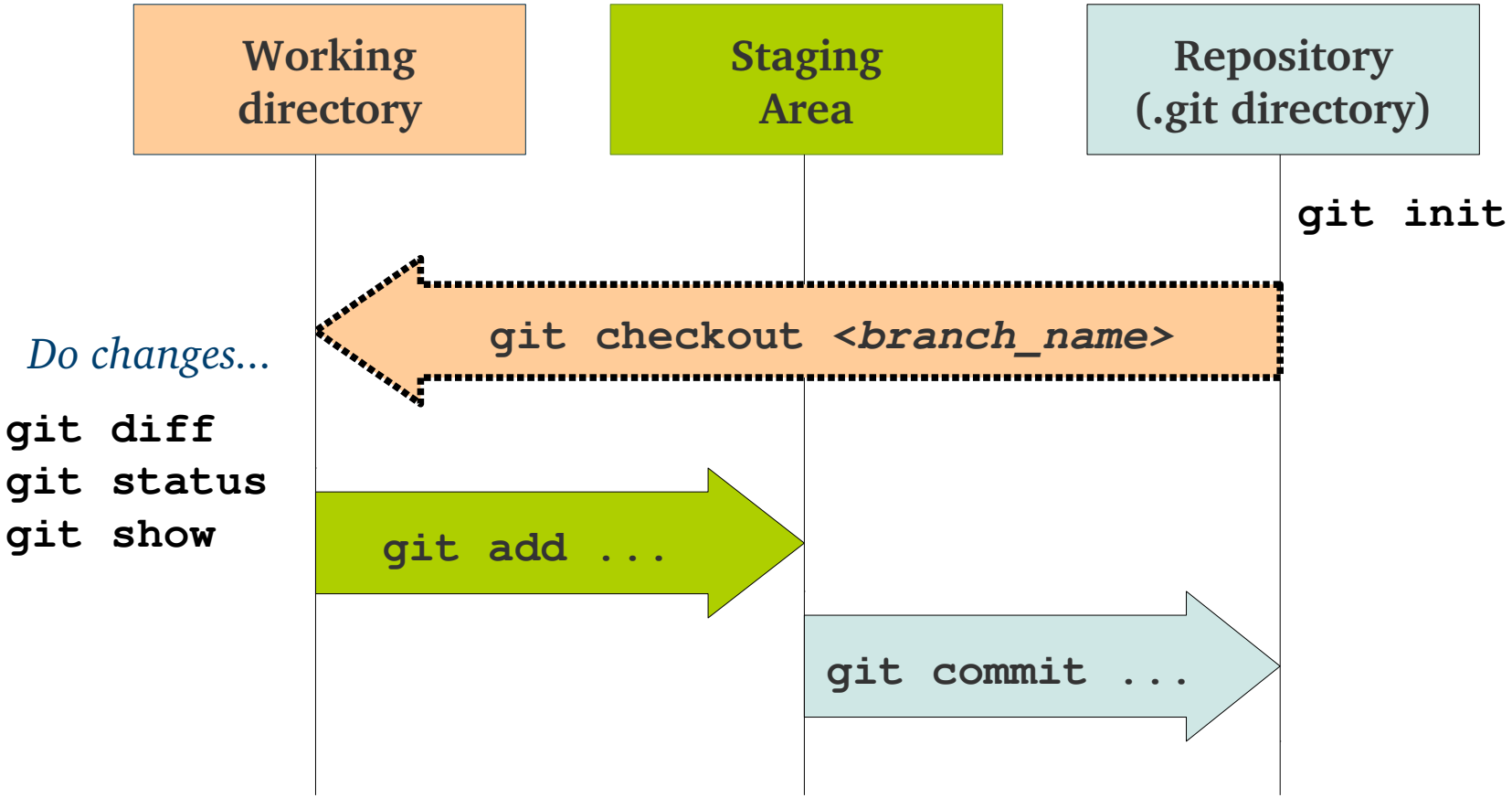
```
$ git stash drop stash@{N}
```

- To resume the stashed changes:

```
$ git stash pop [stash@{N}]
```

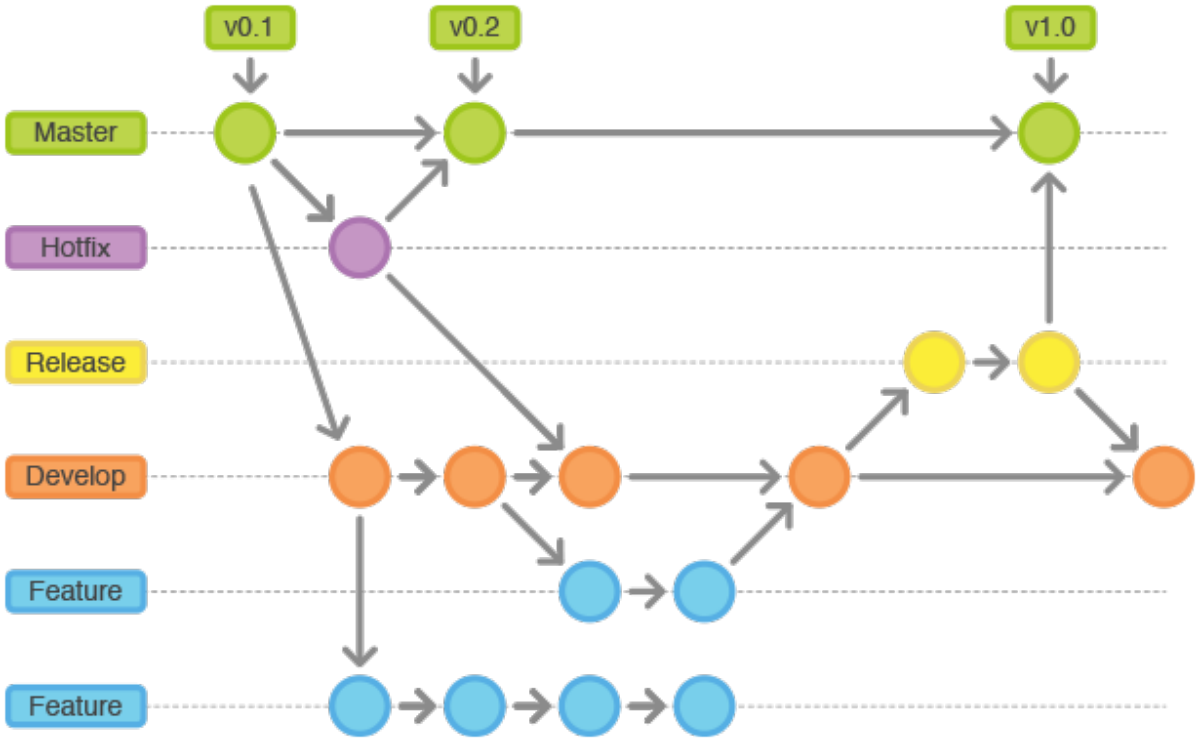
Local repository management

Basic work-flow in commands



Work-flow example

- To manage the software release cycles, we can design a work-flow, by splitting bug fixing, feature development and release-candidate versions in dedicated branches

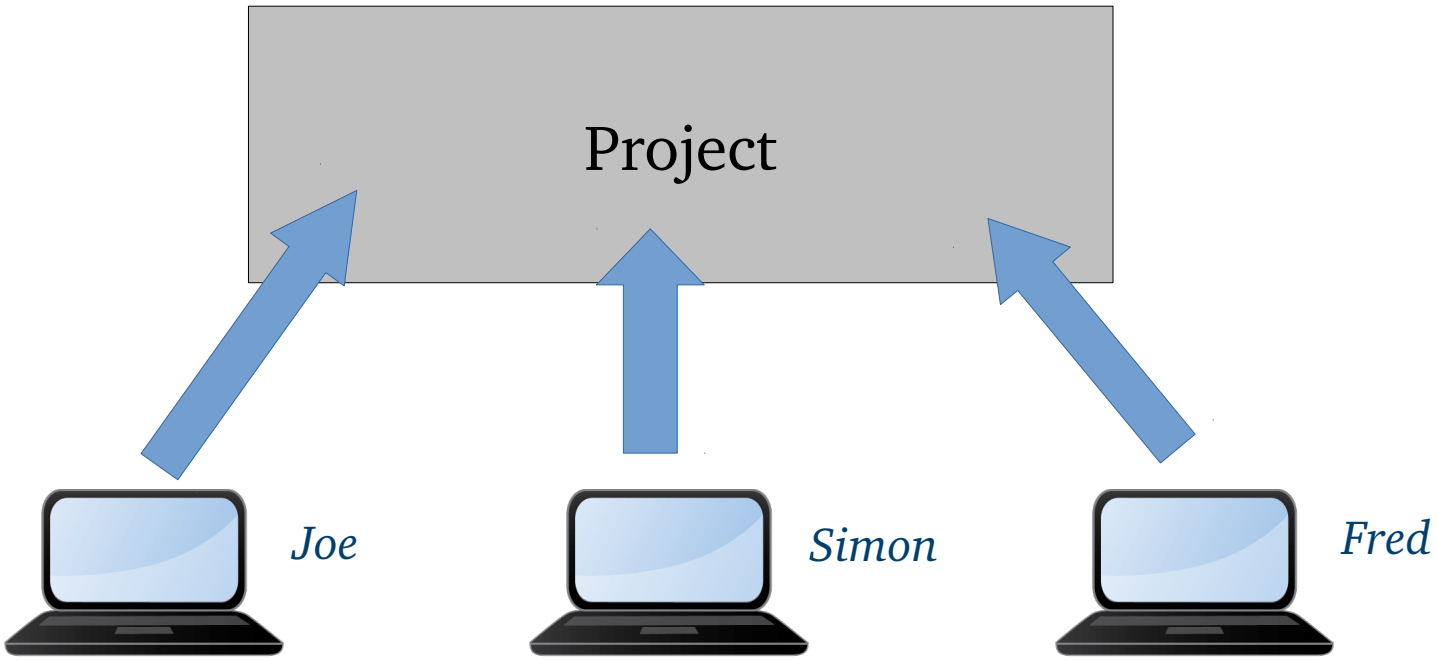


Outline

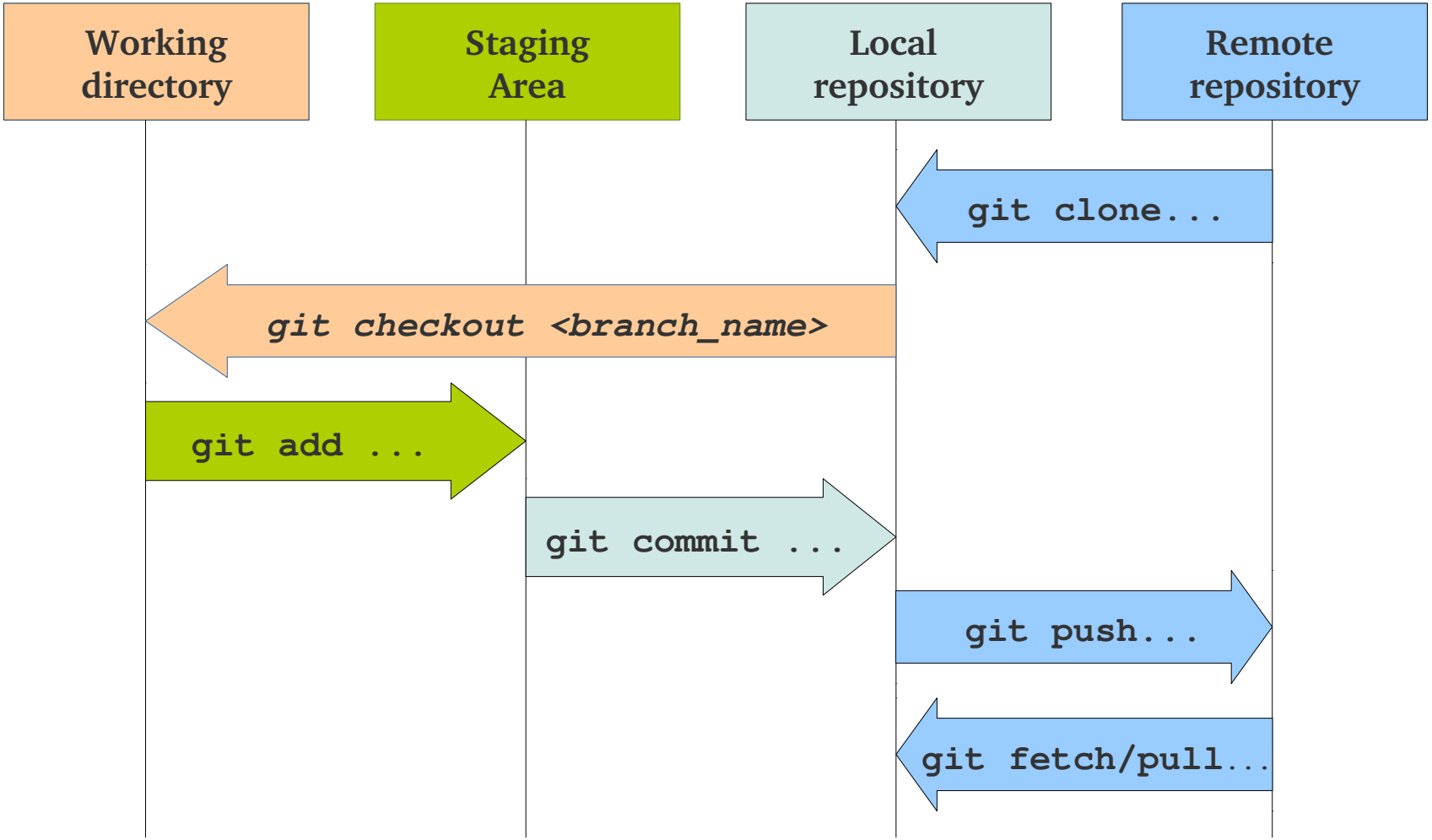
Remote repository management

Cooperative development

- Several developers working on the same project



Remote repository work-flow



Remote repository management

Basic commands

- To clone a project (remote repository)

```
$ git clone repository_path
```

- To download all the changes from the remote repository

  New commits, new branches, new tags...

```
$ git fetch repository_name
```

- To download changes and merge it in the current local branch

  Fetch + merge

```
$ git pull repository_name [branch_name]
```

- To upload local changes to remote repository

```
$ git push repository_name [branch_name]
```

Cooperative development

Cooperative development

- Joe starts a project tracking changes on a local repository


```
[Joe]$ git init
[Joe]$ git add …
[Joe]$ git commit …
```

Cooperative development

- Joe adds a remote repository to the project

Git server

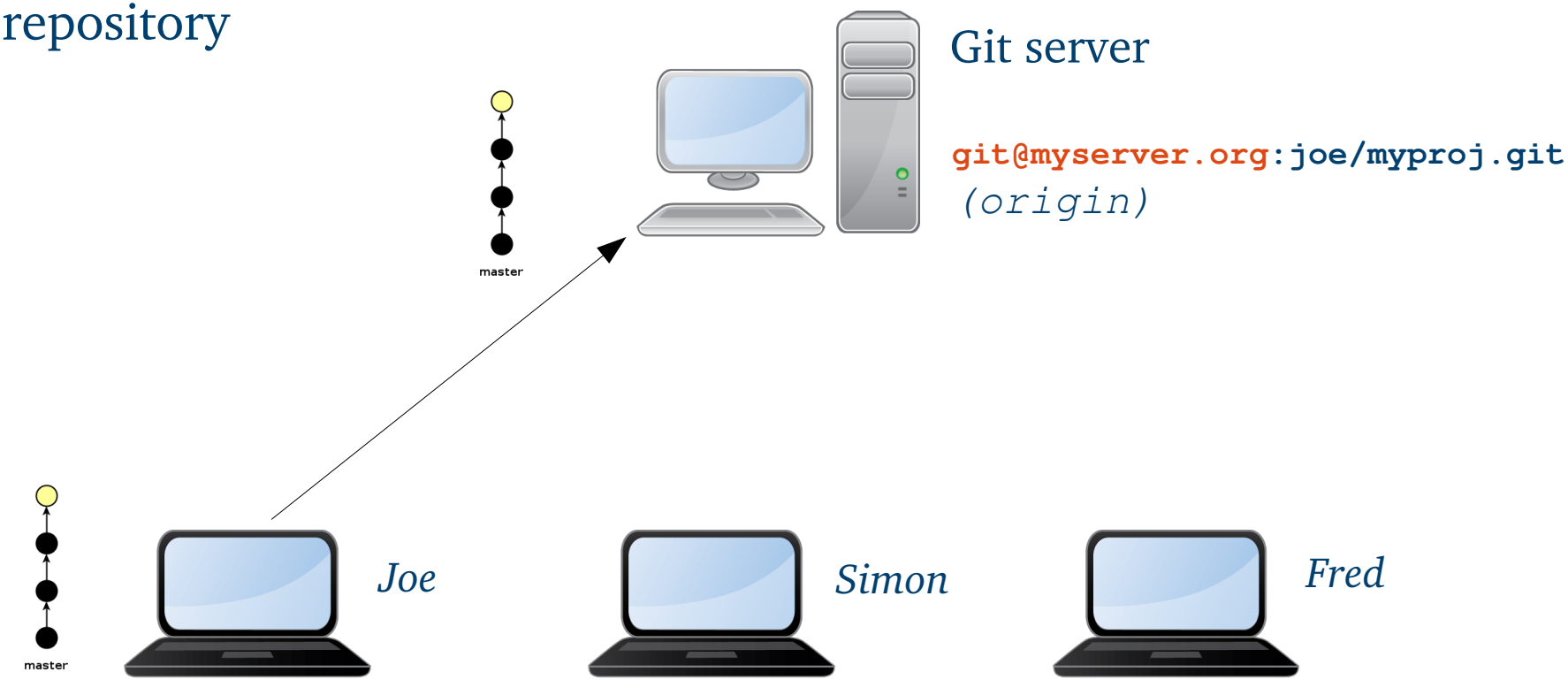
`git@myserver.org:joe/myproj.git`
(origin)


```
[Joe]$ git remote add origin git@myserver.org:joe/myproj.git
```

Remote repository management

Cooperative development

- Joe shares its work by uploading its current git tree to the remote repository

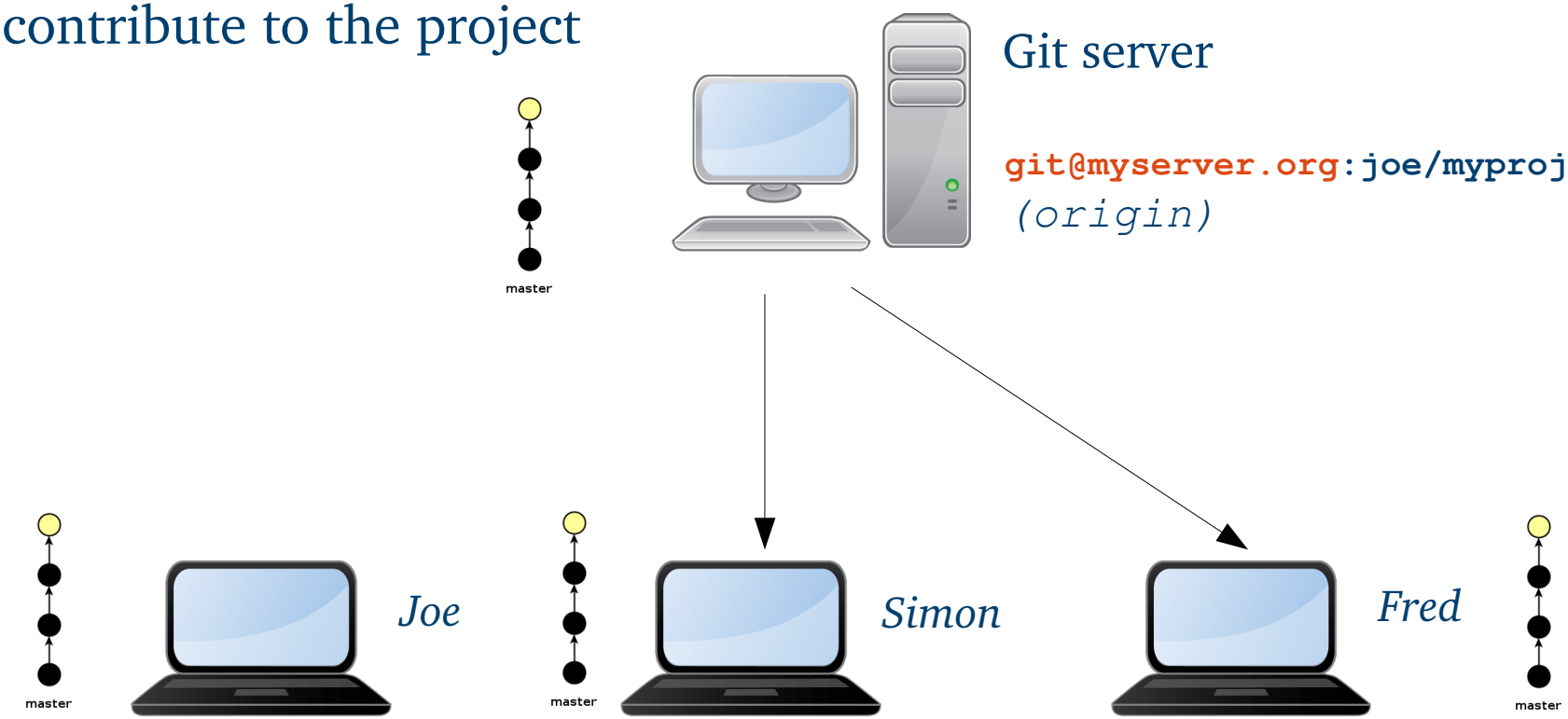



```
[Joe]$ git push -u --all origin
```

Remote repository management

Cooperative development

- Simon and Fred clone the remote git repository, so that they can contribute to the project




```
[Simon]$ git clone git@myserver.org:joe/myproj.git
[Fred]$ git clone git@myserver.org:joe/myproj.git
```

Remote repository management

Cooperative development

- Simon fix bugs, committing changes on a new branch (*hotfix*)

Git server

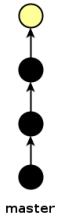
`git@myserver.org:joe/myproj.git`
(origin)


```
[Simon]$ git checkout -b hotfix
[Simon]$ git commit ...
```

Remote repository management

Cooperative development

- Fred implements a new feature, committing changes on a new branch (*feature1*)



Git server

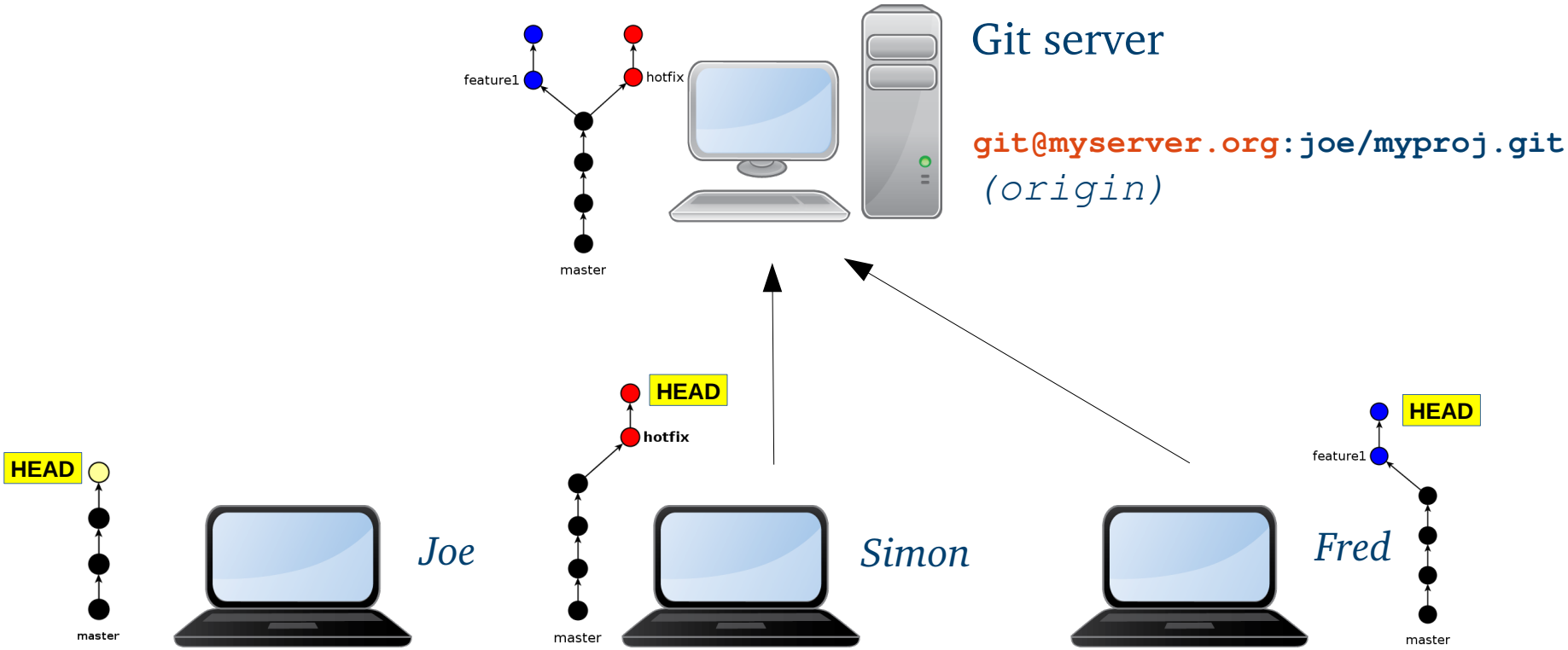
`git@myserver.org:joe/myproj.git`
(origin)


```
[Fred]$ git checkout -b feature1
[Fred]$ git commit ...
```

Remote repository management

Cooperative development

- Fred and Simon upload their changes (branches)

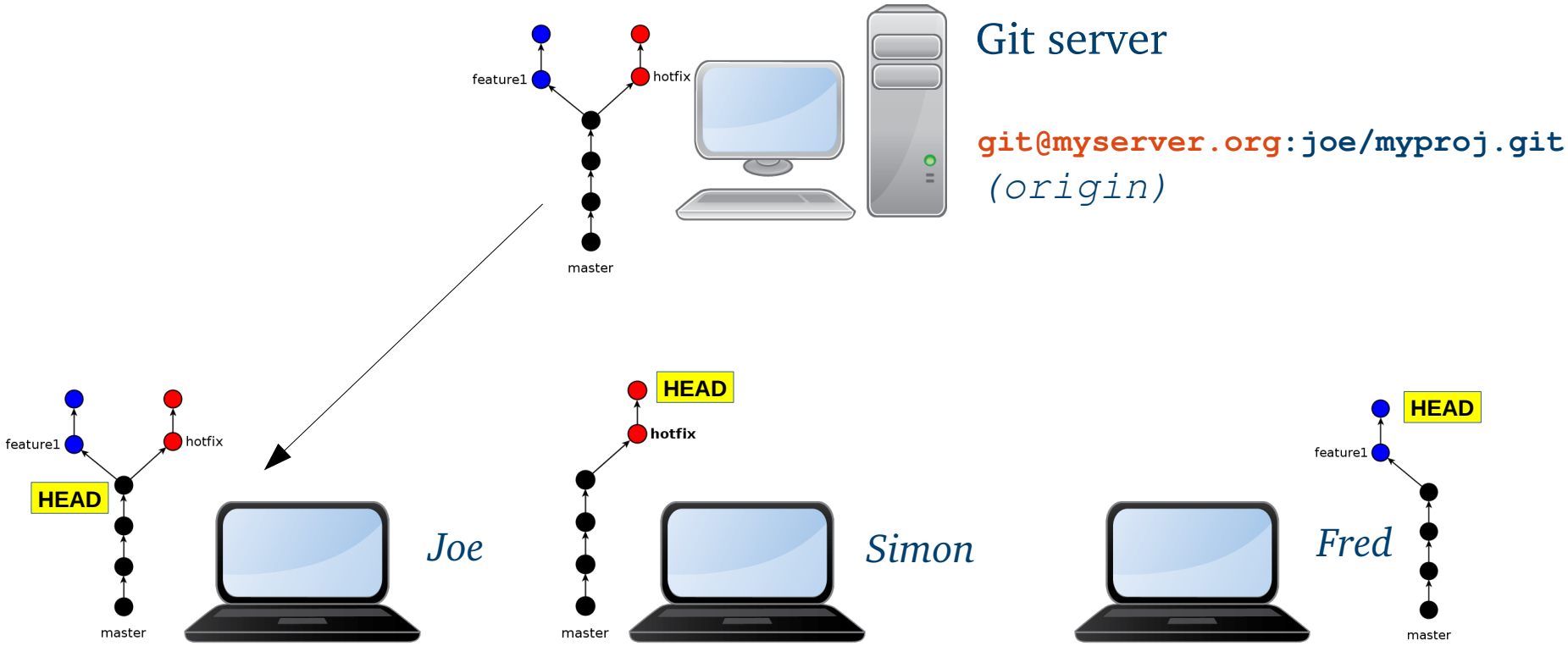



```
[Fred]$ git push origin feature1
[Simon]$ git push origin hotfix
```

Remote repository management

Cooperative development

- Joe updates its local repository

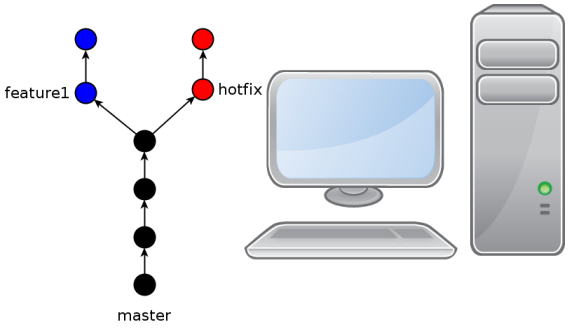



```
[Joe]$ git fetch origin
```

Remote repository management

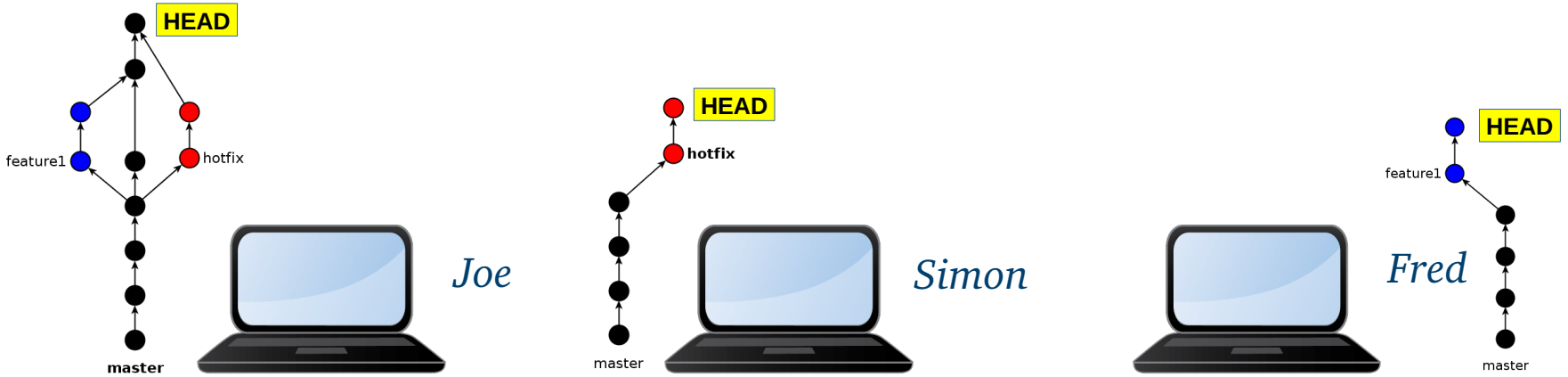
Cooperative development

- Joe merge the contributions in the project “main line” (i.e., *master*)



Git server

`git@myserver.org:joe/myproj.git`
(origin)

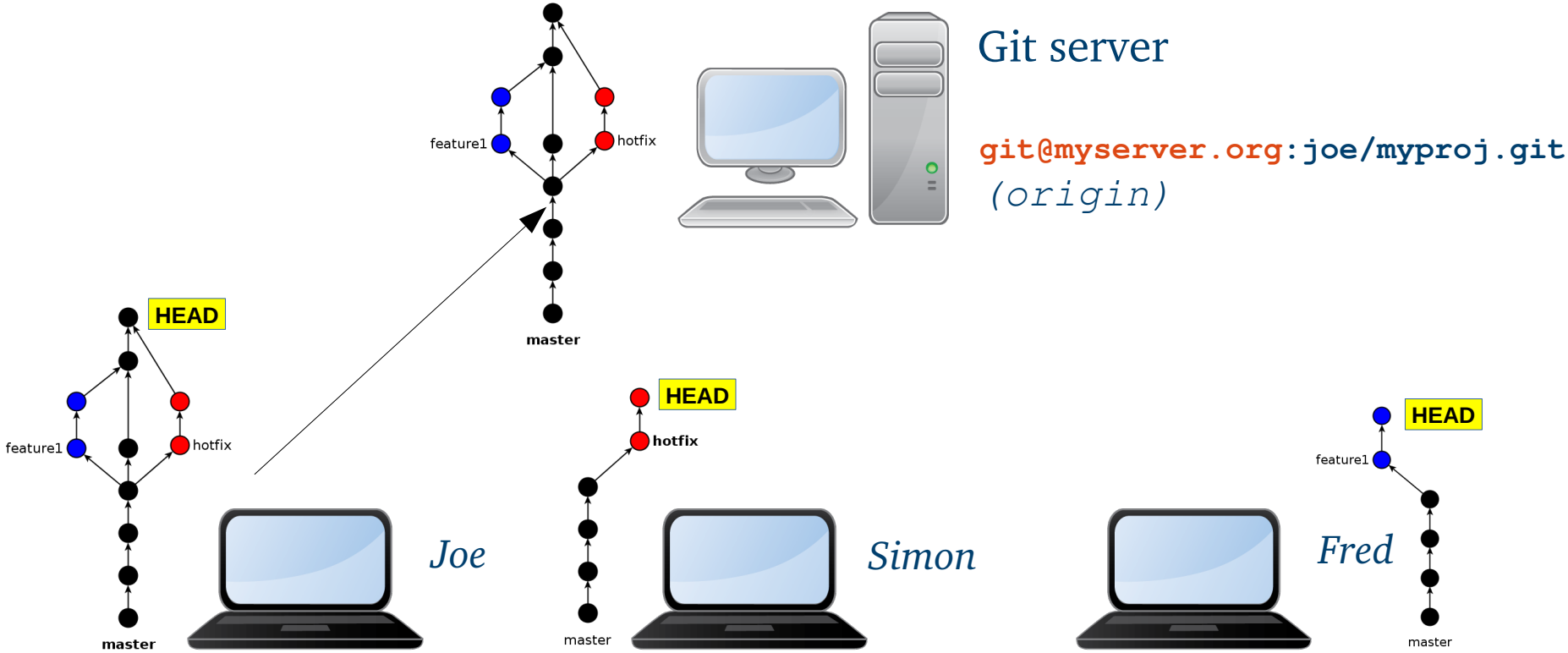



```
[Joe]$ git merge origin/feature1
[Joe]$ git merge origin/hotfix
```

Remote repository management

Cooperative development

- Joe uploads the (updated) *master* branch to the remote repository

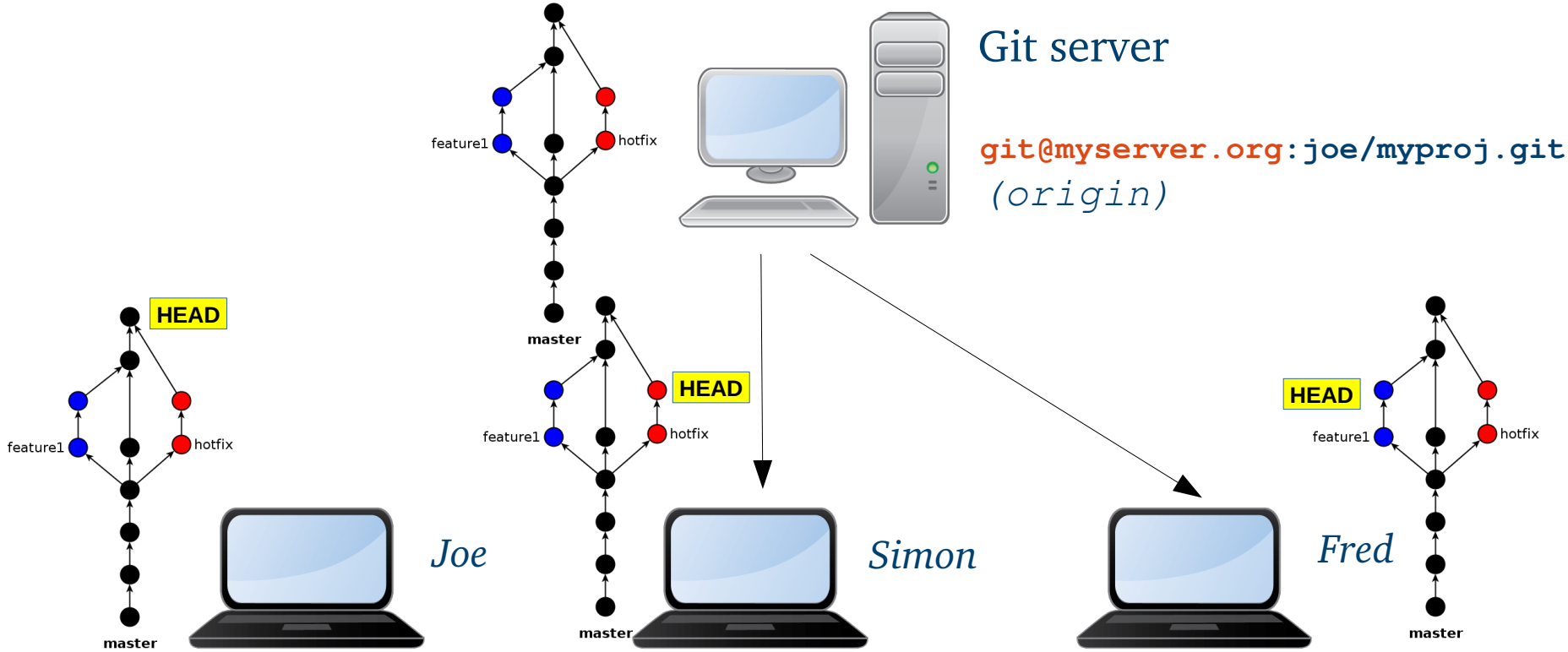



```
[Joe]$ git push origin master
```

Remote repository management

Cooperative development

- Simon and Fred synchronize their local repository with the remote

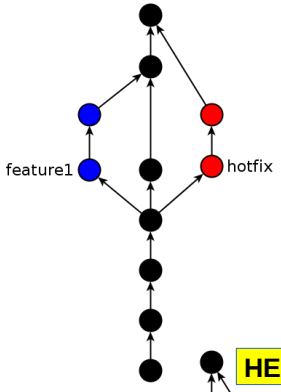



```
[Fred]$ git fetch origin
[Simon]$ git fetch origin
```

Remote repository management

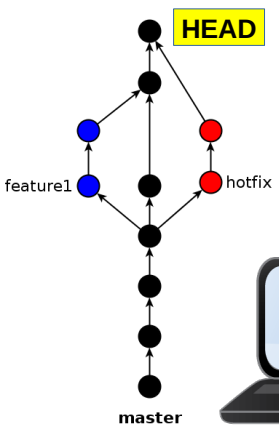
Cooperative development

- Simon and Fred get the updated master

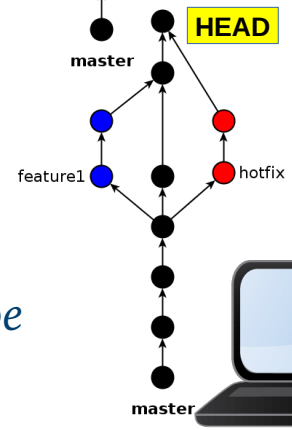


Git server

`git@myserver.org:joe/myproj.git`
(origin)

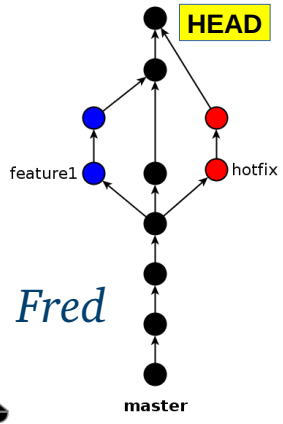


Joe



Simon

Fred




```
[Fred/Simon]$ git checkout master
[Fred/Simon]$ git pull origin master
```

Questions?