Git & GitHub

pt1

Andrea Fornaia, Ph.D.

Department of Mathematics and Computer Science

University of Catania

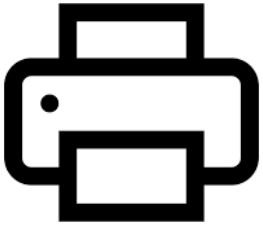
Viale A.Doria, 6 - 95125 Catania Italy

fornaia@dmi.unict.it

http://www.cs.unict.it/~fornaia/

Track changes in your Code

Sharing your Code



Version Control System

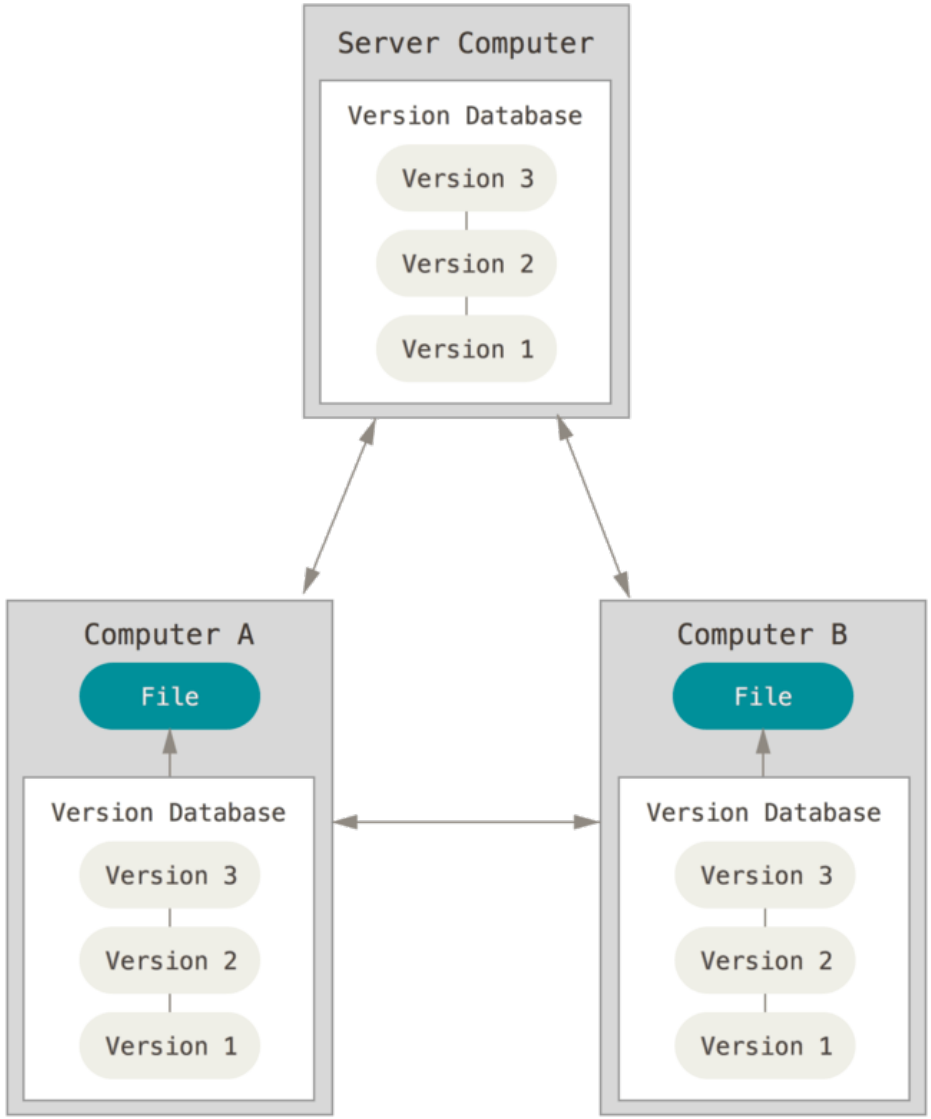
- Tracks file changes over time (**history**)
  - **revert** files (or entire project) to a previous state
  - **compare** changes over time
  - **document** change history
  - see **who** last modified a file and **when**
- Allows developers to **share the code** and **work simultaneously** on it
  - branch oriented
  - avoid overwriting each other's changes

Also Known As

- VCS: **Version Control System**
- SCM: **Source Code Manager**
- **Code Base**
- Or (if you are old as me) **Repository!**

- Its origins are tied with the **Linux Kernel** maintenance:
  - Changes stored as patches and archived files (1991)
  - Start using a proprietary DVCS called BitKeeper (2002)
  - **Linus Torvalds** and Linux dev community create their own DVCS: Git

Git is a Distributed VCS



Git Goals

- Strong support for non-linear development
  - supports thousands of parallel **branches**
  - **branching made easy!**
- Fully distributed
- We can use git just locally or sharing one (or more) remote
- Storing **entire history locally** most operations can be done offline
  - no latency
  - free to try new solutions and make experiments
- Efficient for large projects (e.g. Linux kernel)
  - speed
  - data size

Contents

- Basics
- Brancing & Marging
- Conflicts
- Branching Workflow
- Go Remote
- GitHub

Basics

Install Git

```
$ git --version
git version 2.9.2
```

- Mac
  - via Xcode.
  - Installer:
    - https://sourceforge.net/projects/git-osx-installer/files/
  - Homebrew:
    - $ sudo brew install git
  - MacPort:
    - $ sudo port install git
- Windows
  - Installer:
    - https://git-for-windows.github.io
- Linux
  - Package manager:
    - $ sudo apt-get install git

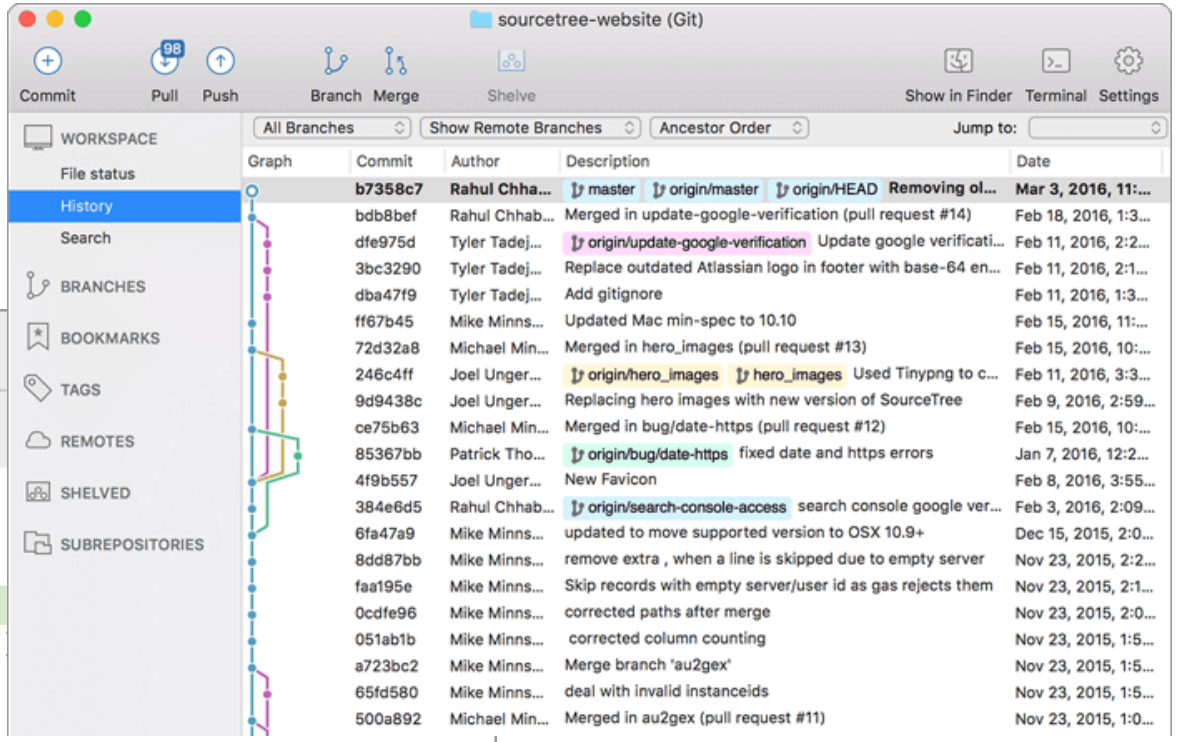
First time Git setup

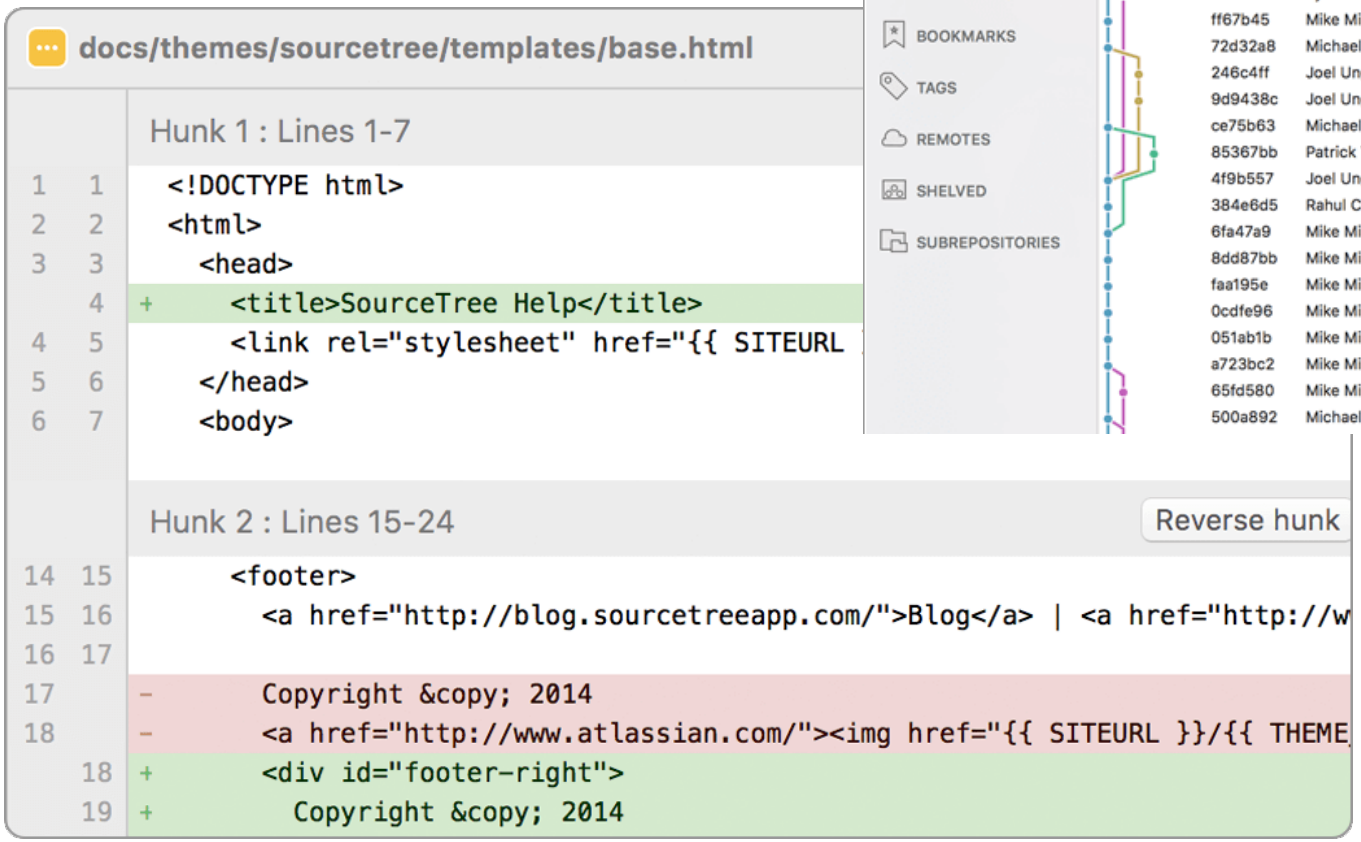
```
$ git config --global user.name ”Andrea Fornaia"
$ git config --global user.email fornaia@example.com
$ git config --global core.editor vim

$ git config --list
user.name=Andrea Fornaia
user.email=fornaia@example.com
core.editor=vim
...

$ git config user.name
Andrea Fornaia
```

Atlassian

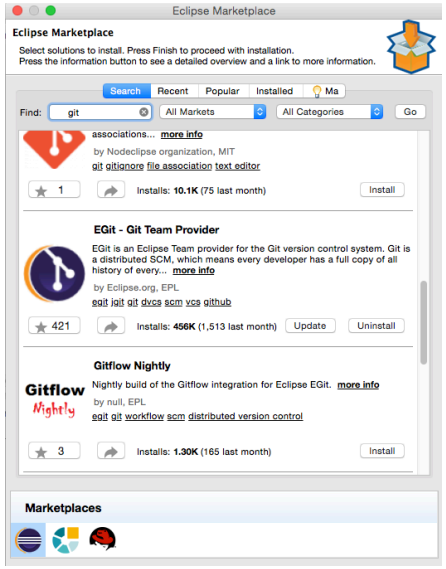
SourceTree

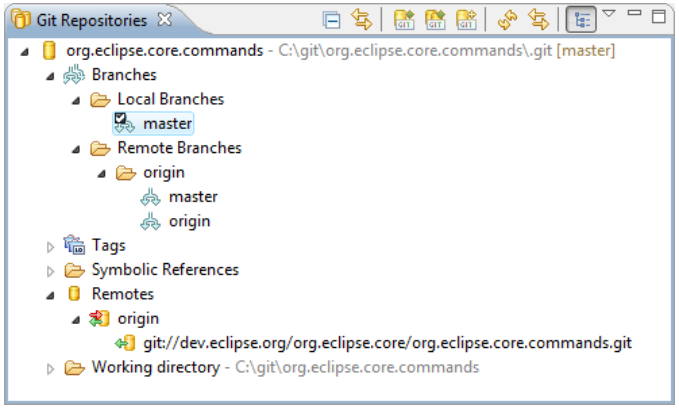
Mac & Windows

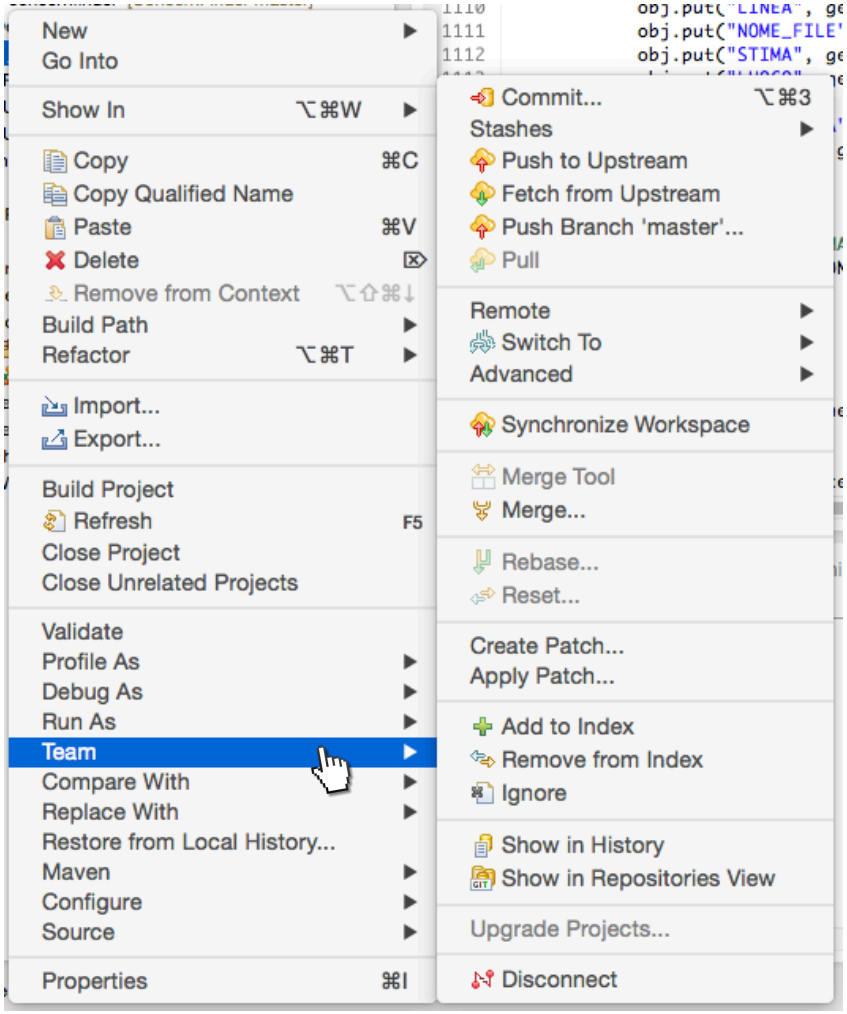




EGit





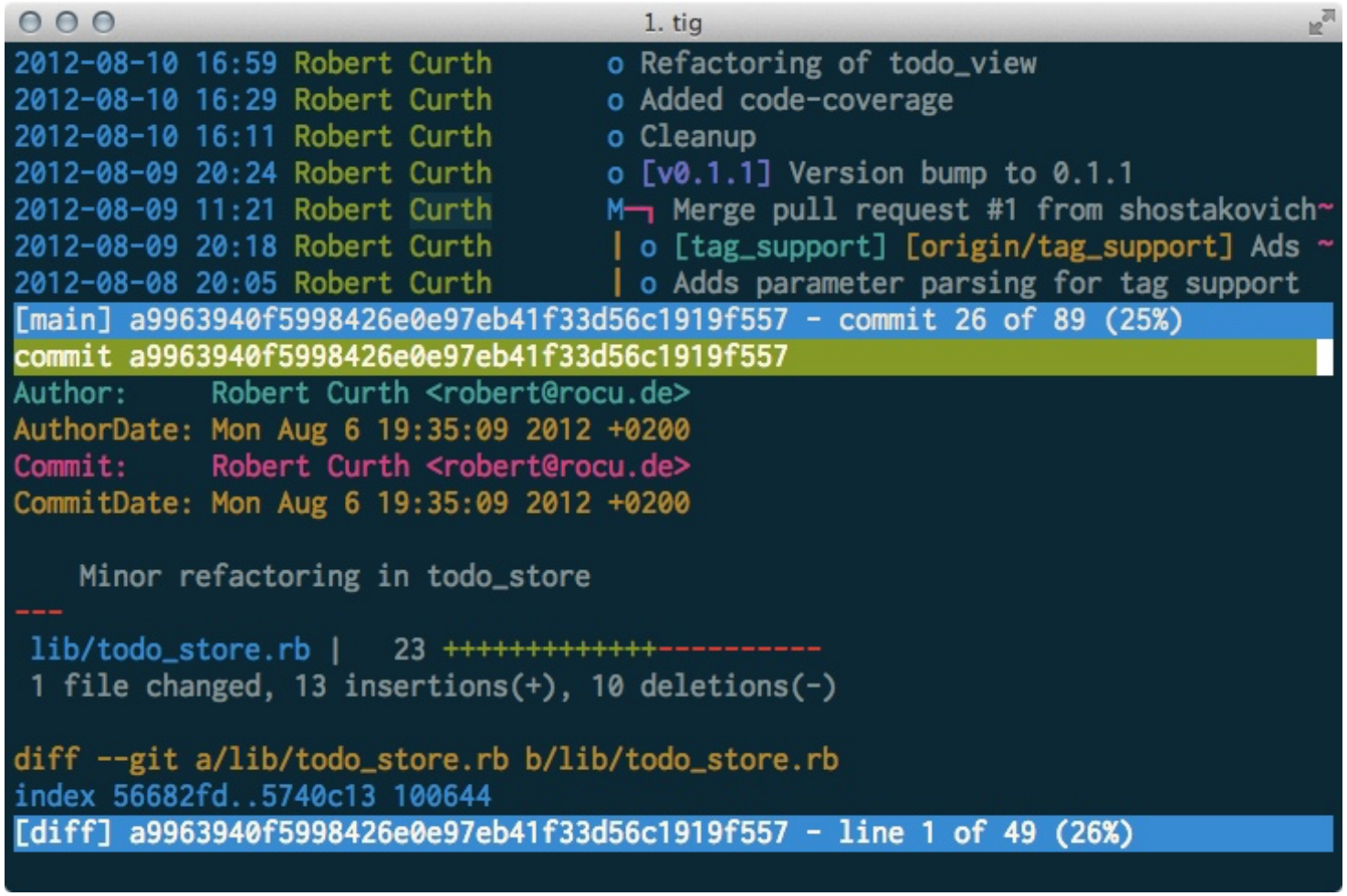


[http://www.eclipse.org/egit/]

Other solutions

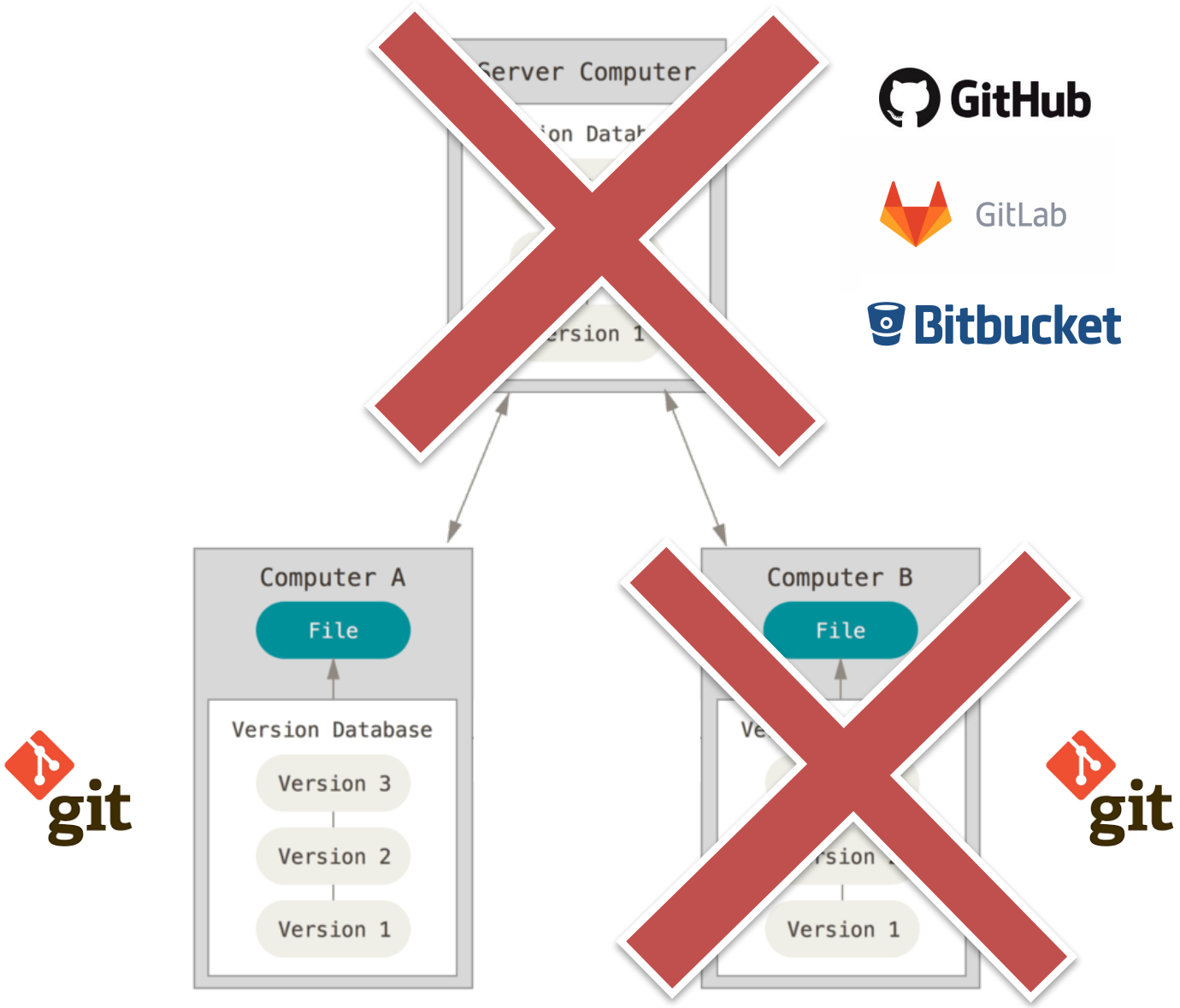
Tig

Tig



[https://rocu.de/tig-a-nice-git-repository-browser]

Typical Git Scenario



Initialising a **New** Local Repository

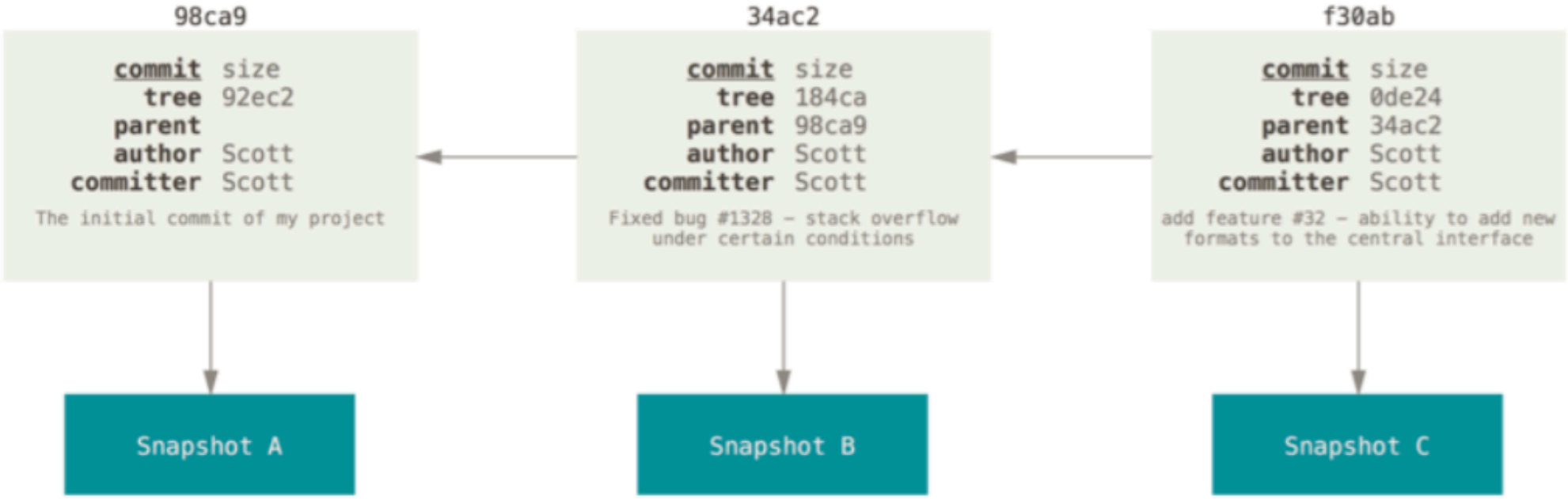
- Setup a repository to track an existing project (some *.txt files): move to project dir and type:

```
$ git init
```

- This creates a new subdirectory named **.git**
- Contains all of your necessary repository files (Git repository skeleton)
- **Nothing in your project is tracked yet!**

```
$ git add *.txt
$ git add README
$ git commit -m 'first commit'
```

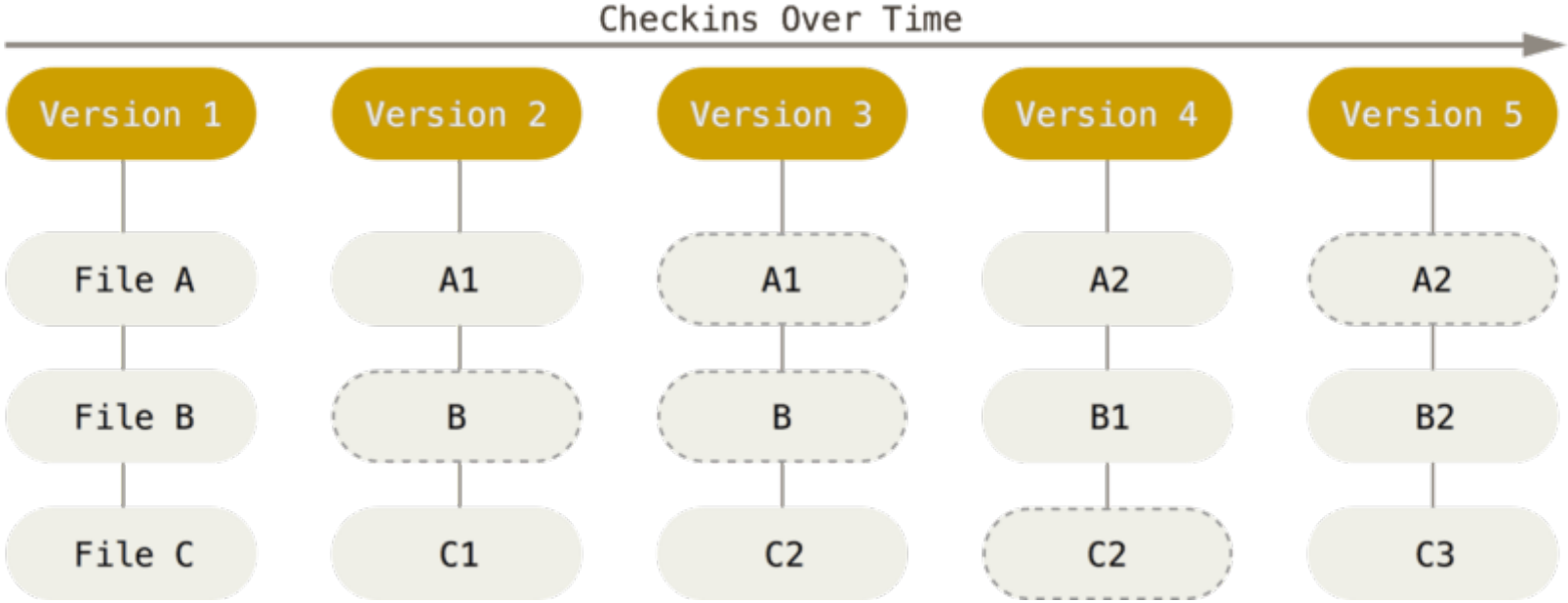
Commit



- ***Snapshot (tree):*** a structure of nested files and directories (complete state of the project)
- ***Author:*** name, email address, and date/time (or “timestamp”) of who made the commit
- ***Committer:*** who added this commit to the repository (may differ from author)
- ***Message:*** text used to comment on the changes made by this commit
- ***List of parent commits (0, 1 or 2)*****:** immediately preceding states of the project content

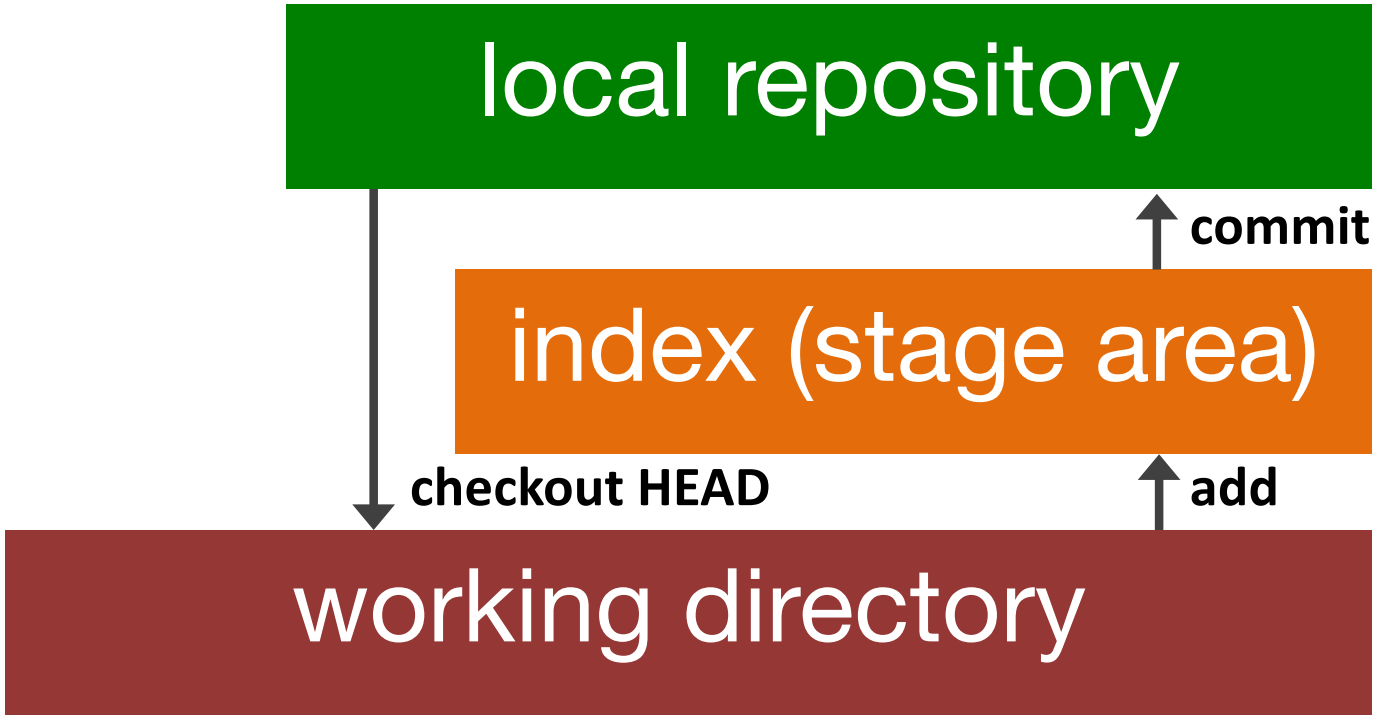
Snapshots, Not Differences

- Git thinks of its data more like a set of **snapshots** of a **miniature filesystem** that a sequence of diffs
- At every commit, Git takes a **picture** of what all files look like at that moment (**snapshot**)
- To be efficient, if files have **not changed**, Git doesn't store the file again, just a **link** to the **previous identical file** it has already stored.
- Makes branching operations fast and easy.



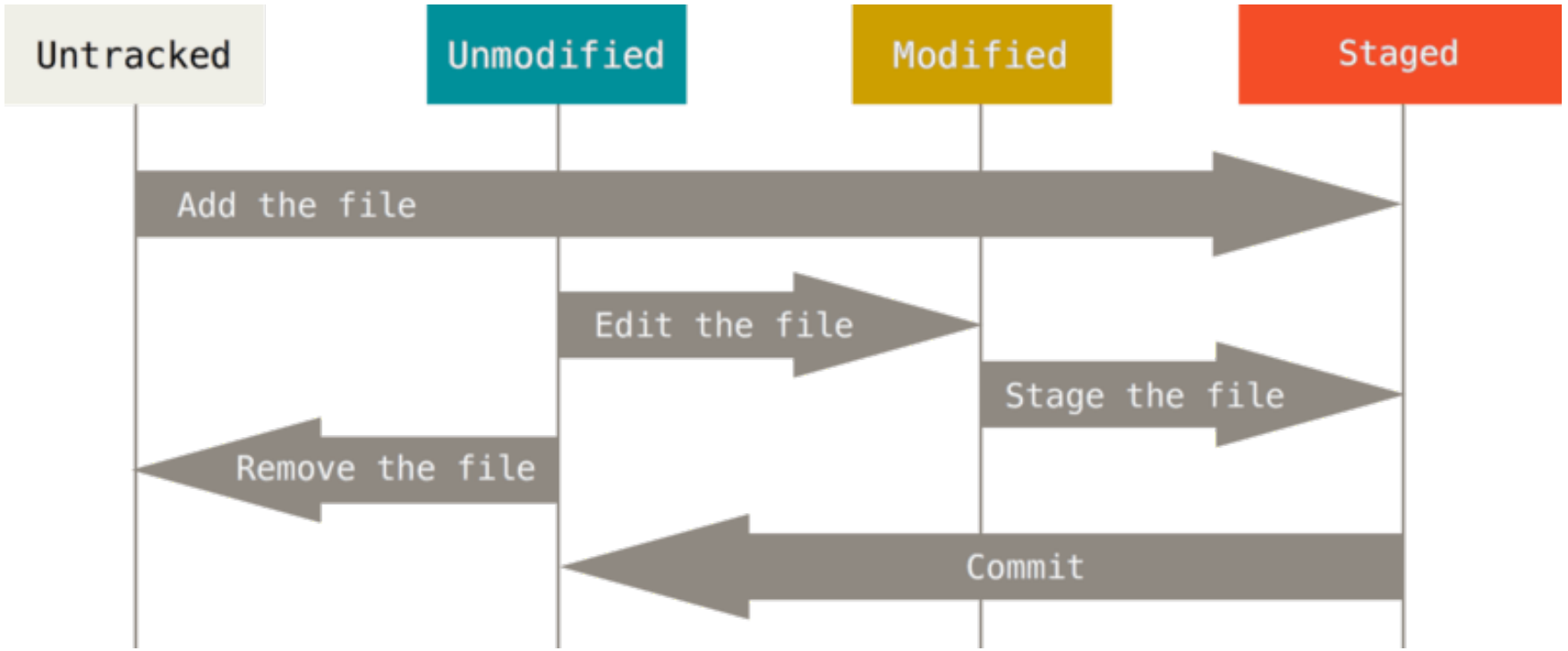
Basic Git Workflow

- Git has three main states that files can reside in:
  - **Modified:** file changed but not committed yet
  - **Staged:** file marked to be included into the next commit
  - **Committed:** file stored in your **local** database



Checking the status of your file

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```



git add & git commit

```
$ echo "file content" > file.txt          (untrcked)
$ git add file.txt                        (new file: staged for commit)
$ echo "added new content" >> file.txt    (changes not staged for commit)
$ git add file.txt                        (changes staged for commit)


$ git commit -m “second commit”           (it’s now unmodified)
```

Log

```
$ git log
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Author: Andrea Fornaia <mrossi@email.com>
Date:   Thu May 07 15:08:43 2020 -0800

    second commit

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Andrea Fornaia <mrossi@email.com>
Date:   Thu May 07 14:58:32 2020 -0800

    first commit
```

Basic Git Workflow

1. Modify files in your working tree.

2. **Selectively stage** just those changes you want to be part of your next commit, which adds only those changes to the 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.

A few points on commits

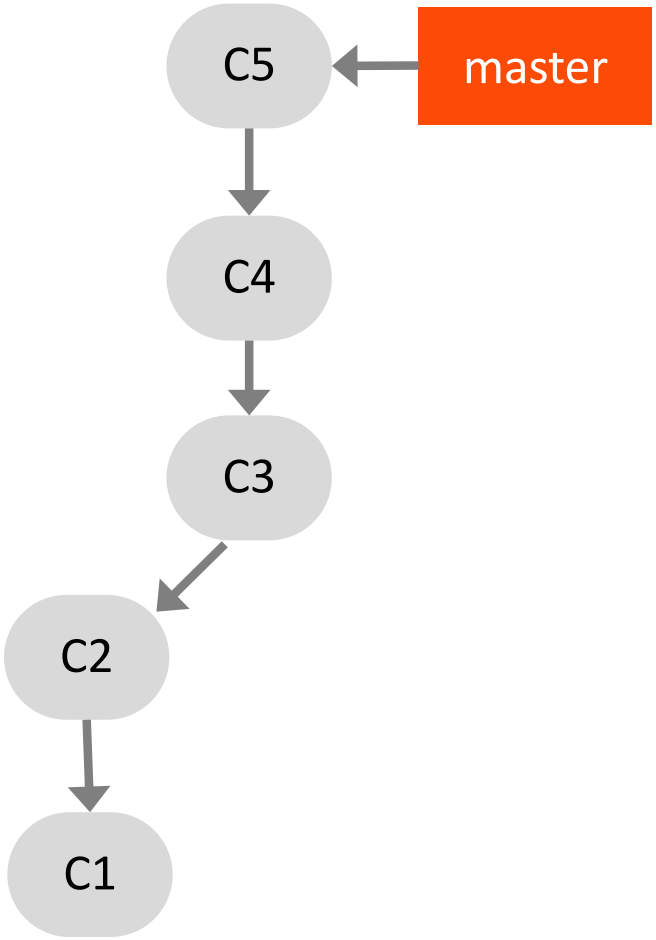
- Use **frequent, small, cohesive commits**
  - No need to wait, they are local at first
- Don't get out of sync with your collaborators
- Commit the sources, **not the derived files**
- Use a **.gitignore** file to indicate files to be ignored

```
$ cat .gitignore
*.class
*.log
build/
doc/*.pdf
doc/**/*.txt
```

Good comments

Branching & Merging

Branch – Linear History



Every repository starts with one commit (root) that has no predecessors (C1)

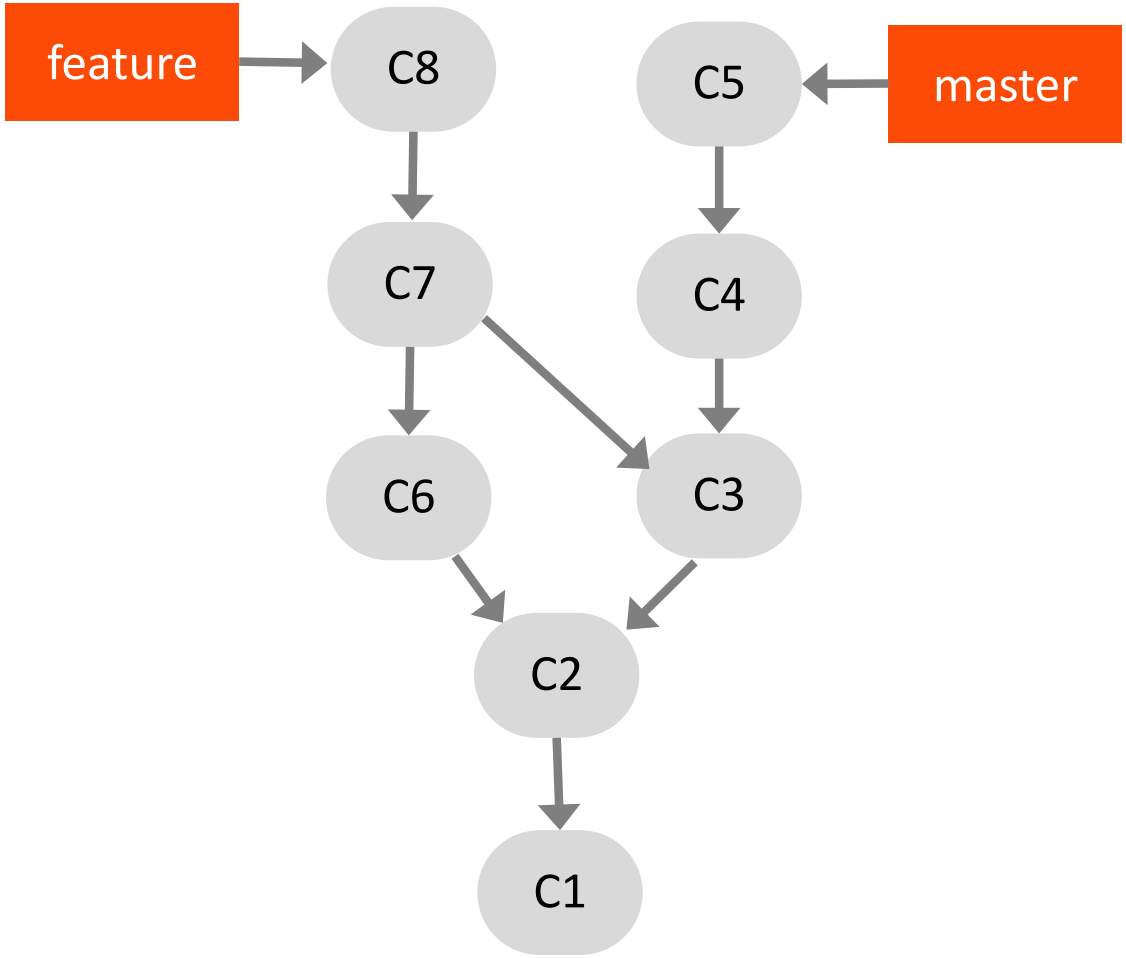
Every repository starts with a branch called “master”

The **branch** label points to the last commit on that branch, called “branch tip” (e.g. master -> C5)

A branch is the **collection** of all commits reachable from the **tip** down **to** the **root** (e.g. master = {C1 .. C5})

A **new commit** is added to the branch as a node pointing to the branch tip (**on top of branch**), then moving the branch label to the new tip

Branch – Non Linear History



Git supports and encourages non-linear development. Commits can be represented as a **DAG**

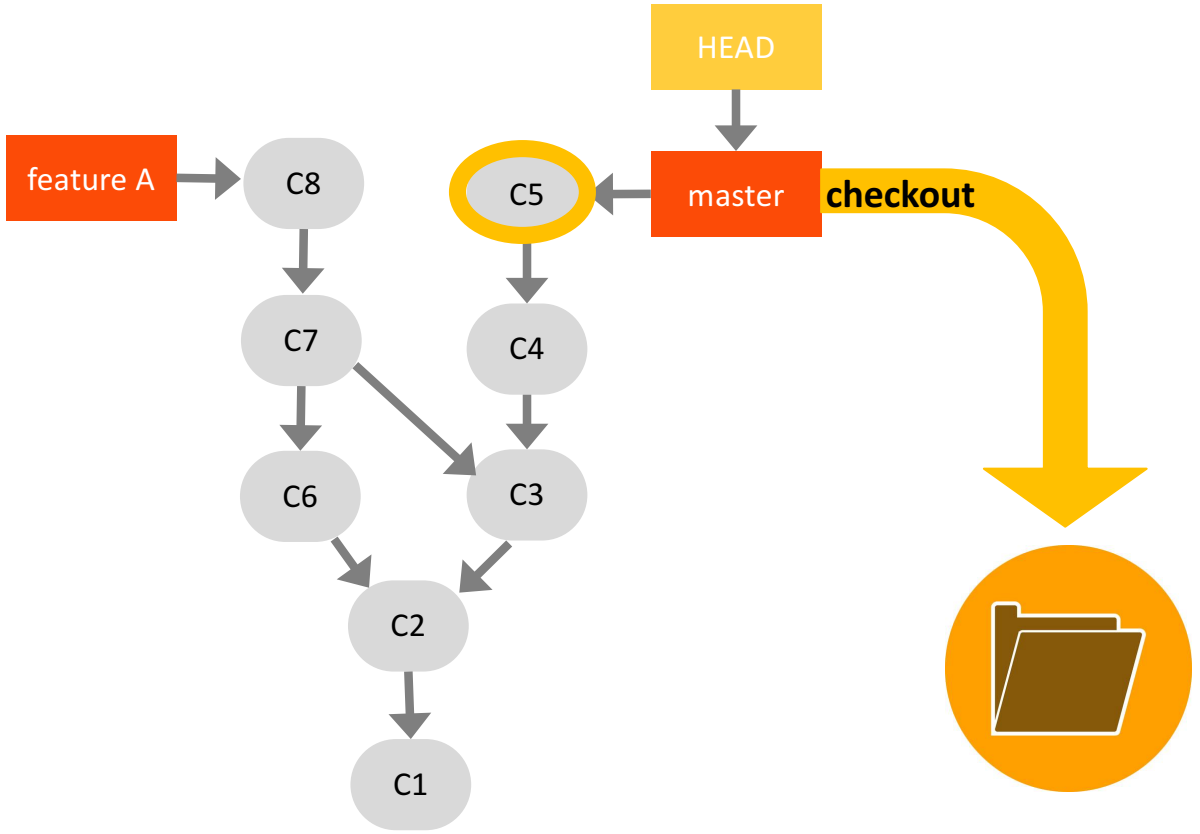
A commit may have:
- no predecessor (first commit, **root**)
- one predecessor (**normal** commit)
- two predecessors (**merge** commit)

We may have several branches; in this example:
- master {C1; C2; C3; C4; C5}
- feature {C1; C2; C3; C6; C7; C8}

Note that branches **can overlap**

References are **not bidirectional**: by deleting a branch you may loose access to commits (garbage collector)

Only one active branch at a time



Working Directory (active commit)

We can **switch** from one branch to another without loosing our work

HEAD is a special label that points to the **active branch**

Working directory in your file system will contain the snapshot (files and dirs) of the **active commit**

Git checkout

```
$ git checkout <branch>
```

- If branch doesn't exist both on local and remote repositories, will rise an **error**. To create a **new branch** use **–b** option

```
$ git checkout –b <new-branch>
# same as:
$ git branch <new-branch> && git checkout <new-branch>
```

- If branch doesn't exist into local repository, but exists into exactly one remote repository, will create a **tracking branch**

Git graph

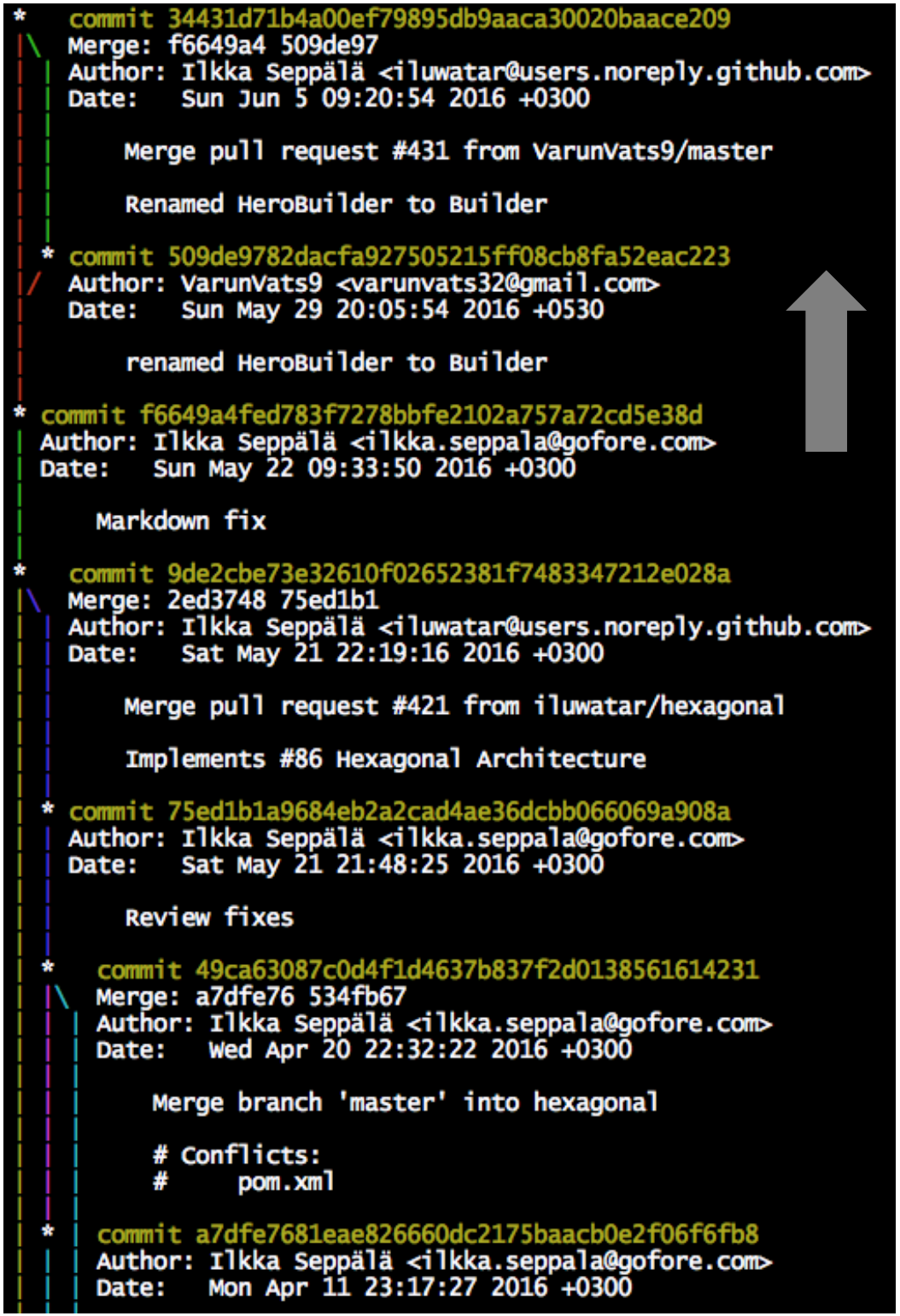
```
$ git log
  --graph
  --decorate
  --oneline
  --all
```

Possiamo creare un alias

```
$ git config --global
   alias.graph "log
       --graph
       --decorate
       --oneline"
```

Creando un nuovo comando

```
$ git graph
$ git graph --all
```



Git merge

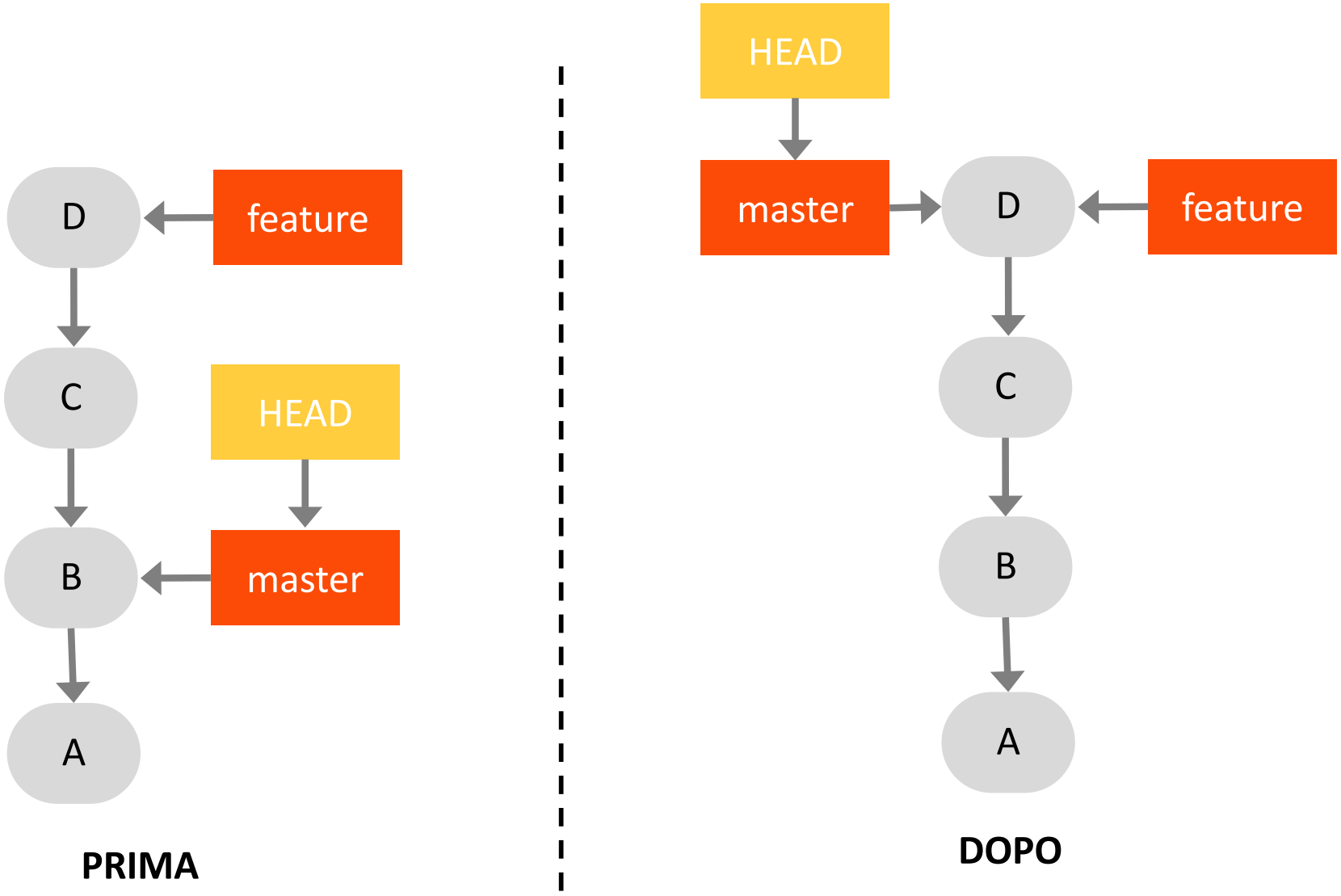
- Merge a source branch **into** the active branch

```
$ git merge <branch>
```

- Merge the **file modifications** made in the target branch into to the active branch
- It can be seen as a **set operation**:
  - After the merge, the active branch must contain **all the commits** of the source branch

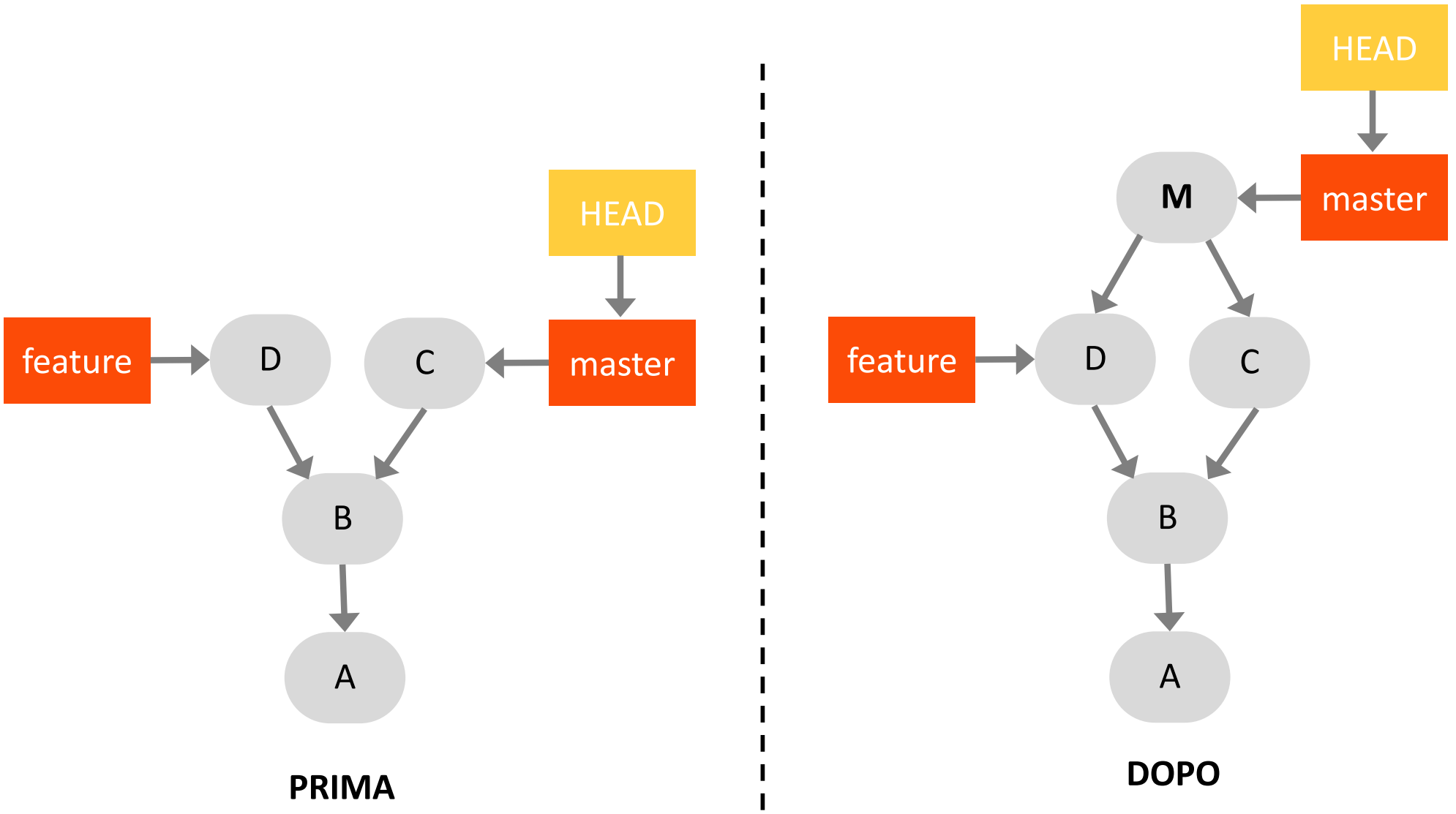
Fast Forward Merge

master = {A,B} feature = {A,B,C,D} ==> master = {A,B,C,D}



Commit Merge

master = {A,B,C} feature = {A,B,D} ==> master = {A,B,C,D,**M**}



Conflicts

Auto-merging differences on separate lines

<master> <feature> <master>

```
$ git checkout master
$ git merge feature
Auto-merging long.txt
Merge made by the 'recursive' strategy.
 long.txt | 2 +-
 1 file changed, 1 insertion(+), 1
deletion(-)
```

recursive strategy

Auto-merging rocks… but?

- Occasionally, this process doesn't go smoothly…
- Did you change the **same part** of **the same file**?
  - Git won't be able to merge them cleanly!
  - Git will ask you what to do to solve the **conflict**


PUSH REJECTED, REBASE OR MERGE
GIT PUSH --FORCE
memegenerator.net

Conflict example

master

file3.txt <master>

```
added file3 for bug fix
```

file1.txt <master>

```
original file1 content
bug fixed directly on master!
```

feature

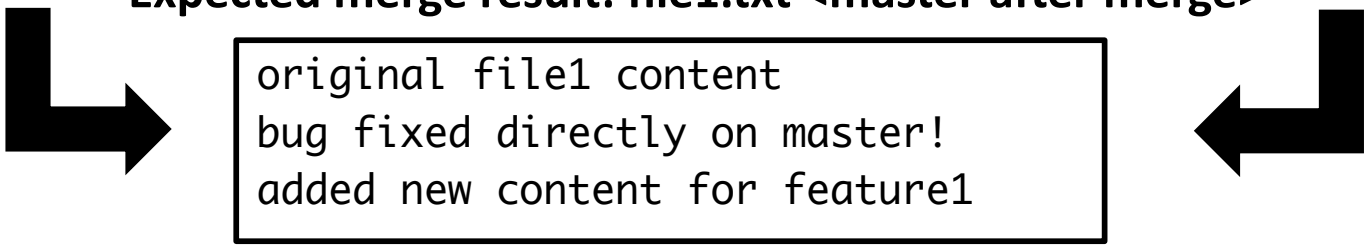
file2.txt <feature>

```
file2 content for feature
```

file1.txt <feature>

```
original file1 content
added new content for feature
```

Expected merge result: file1.txt <master after merge>

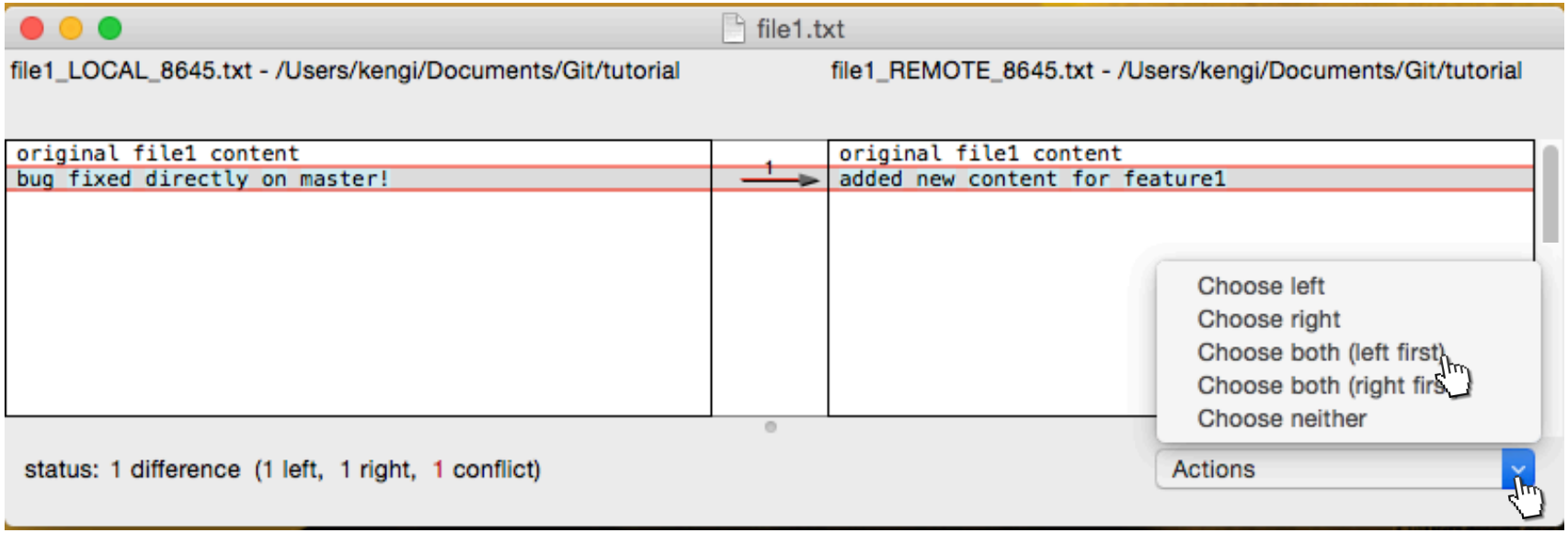


Try to merge **master ← feature**

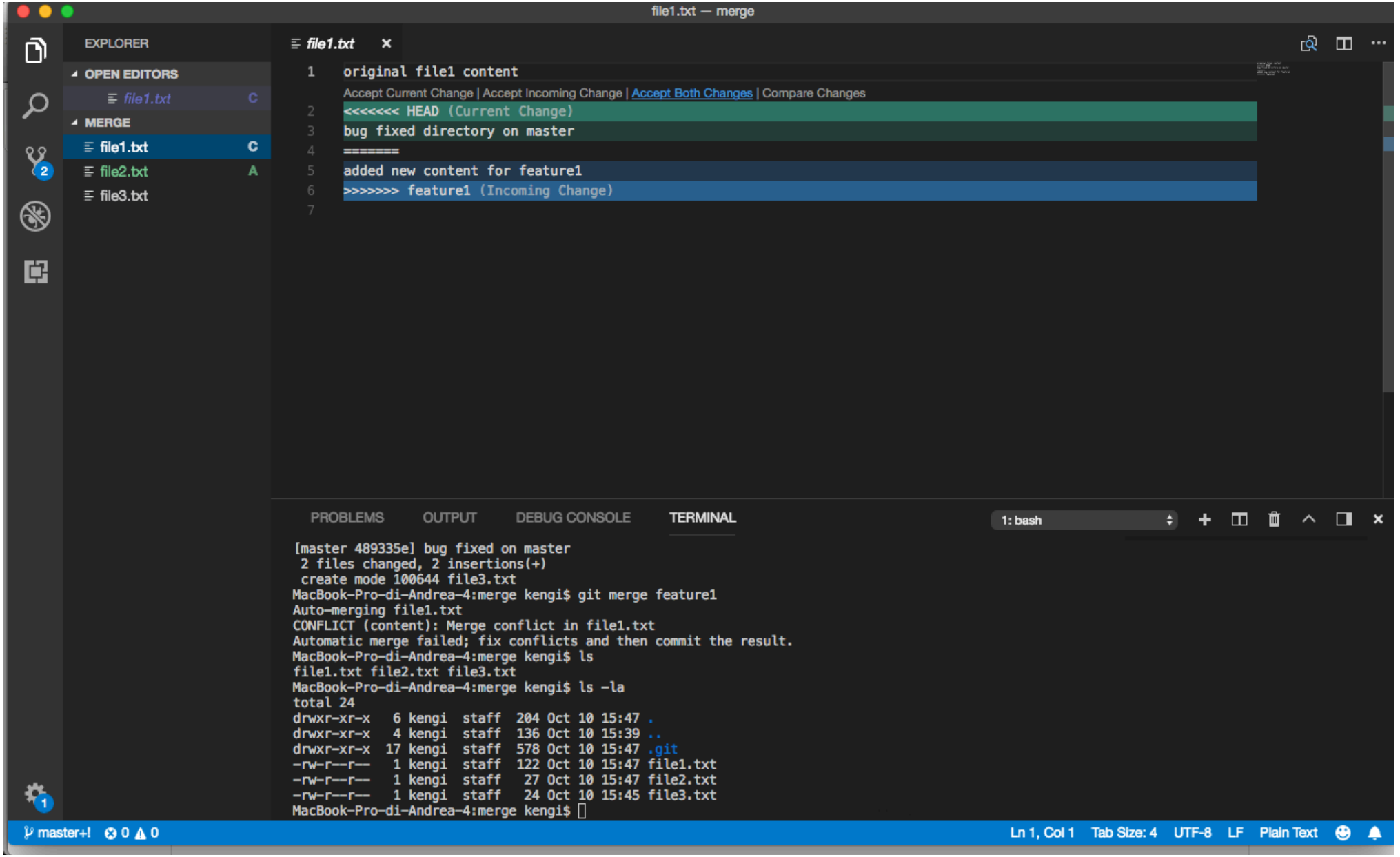
```
$ git merge feature
Auto-merging file1.txt
CONFLICT (content): Merge conflict in file1.txt
Automatic merge failed; fix conflicts and then commit the result.

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

Changes to be committed:

    new file:   file2.txt

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

    both modified:   file1.txt
```

Solve conflicts on file1.txt

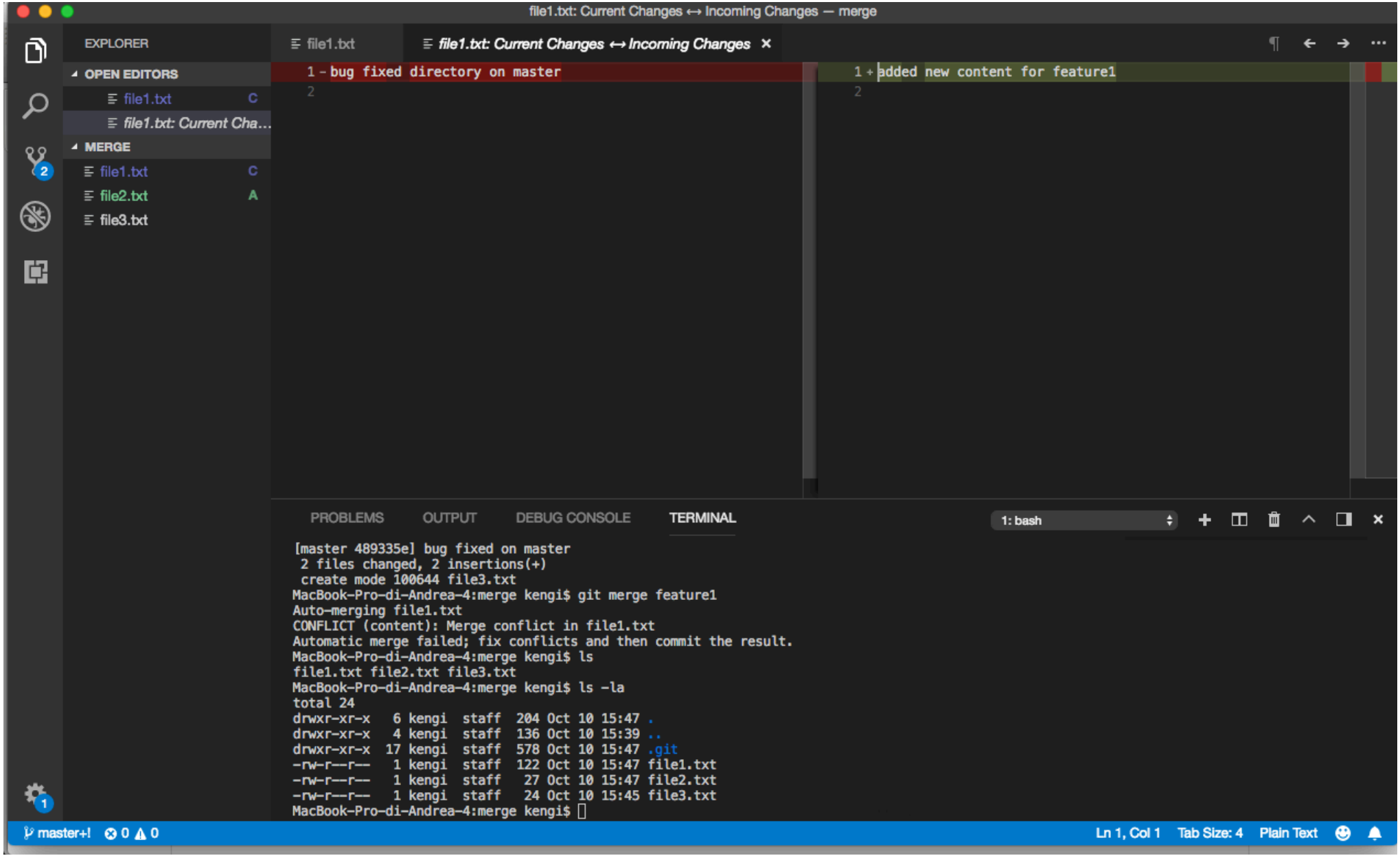
```
$ vim file1.txt
original file1 content
<<<<<<< HEAD
bug fixed directly on master!
=======
added new content for feature1
>>>>>>> feature1

$ git mergetool
```



Merge with Visual Studio Code



Merge with Visual Studio Code



Commit. New merge node created

```
$ git commit
Merge branch 'feature1'
# Conflicts:
#   file1.txt
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#   .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#   modified:   file1.txt
#   new file:   file2.txt
```