# Bases et fonctionnalités avancées



F. Michel

# Plan

# Plan

# Git : logiciel de gestion de versions décentralisé

## Logiciel libre sous licence GNU GPL v.2

- **créé par en 2005 par Linus Torvalds**, l'homme qui a révolutionné l'informatique 2 fois !
- principal contributeur : Junio C Hamano
- C'est le SCM le plus utilisé au monde

# Git : logiciel de gestion de versions décentralisé

## Logiciel libre sous licence GNU GPL v.2

- **créé par en 2005 par Linus Torvalds**, l'homme qui a révolutionné l'informatique 2 fois !
- principal contributeur : Junio C Hamano
- C'est le SCM le plus utilisé au monde

## Caractéristiques

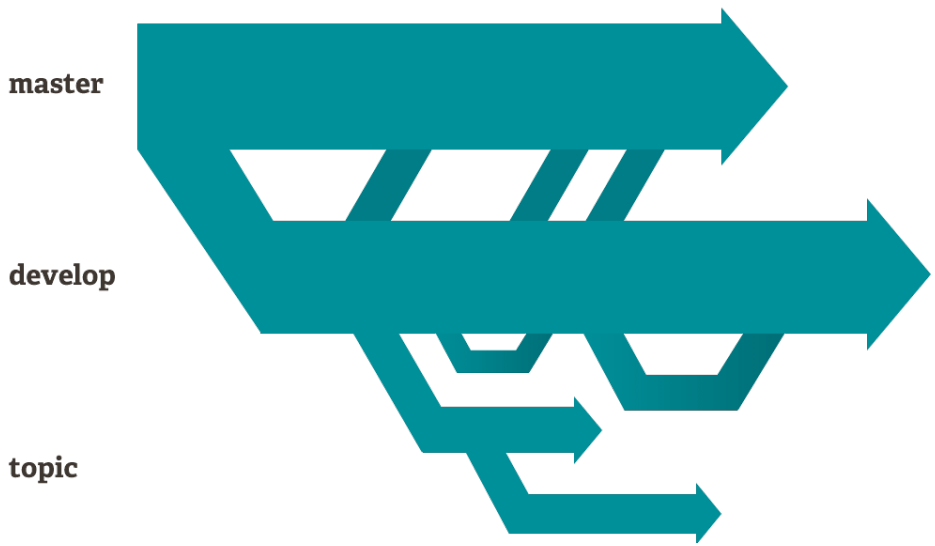
- gère l'évolution du contenu d'une arborescence
- pas de serveur centralisé
- multi plates-formes, simple, léger et efficace
- permet le travail hors ligne
- intégrité cryptographique

# Branching and Merging

Travailler avec de multiples branches est naturel avec Git

## Atouts

- Branches totalement indépendantes : désinhibe la création
- Elles ont chacune un rôle clair : production, test, quotidien. . .
- Simplifie un travail ciblé sur les *features*

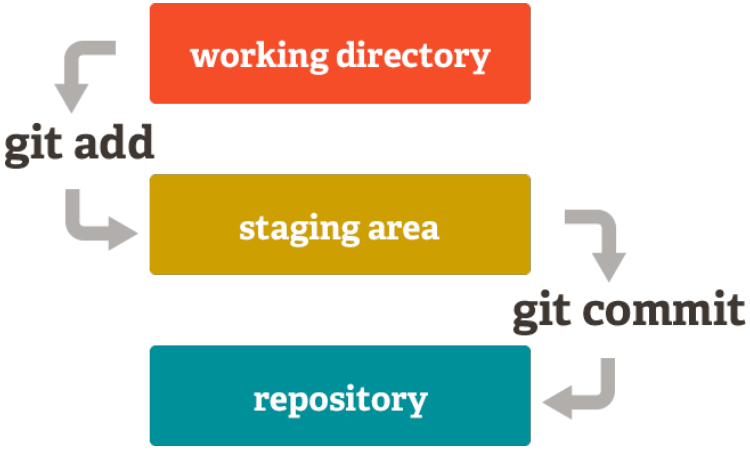


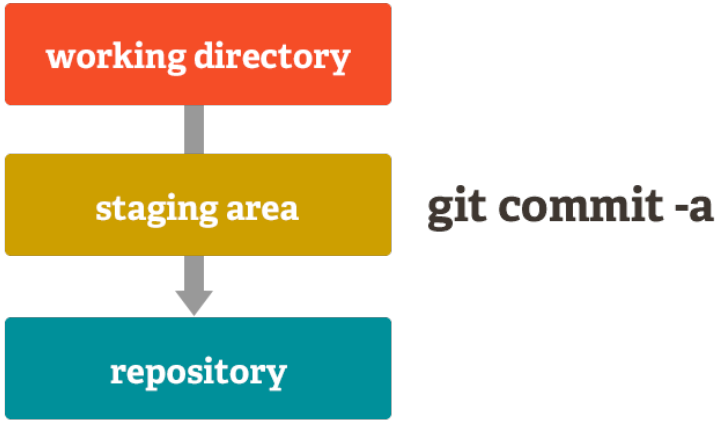
# *Vraiment* distribué

## *Clone*, pas *checkout* !

- $\Rightarrow$ Copie l'entièreté du dépôt : tout le monde a un backup.
- $\Rightarrow$ permet d'implémenter toutes sortes de *workflow* cf. section 8

# *Staging Area* : indexation fine du commit




Voir aussi la commande ▸ git status

# Plan

# Installation

## Linux : paquet `git`

- `$ apt-get install git` *Debian-based*
- `$ pacman -S git` *Arch-based*

## Windows

- https://gitforwindows.org/

## Mac OS

- https://git-scm.com/download/mac

```
$ git --version
```
git version 2.41.0

# Configuration

⇒ `$ git config`

Trois localisations pour les fichiers de config.

## Pour tous les utilisateurs de la machine : `/etc/gitconfig`

- `git config --system` pour modifier

## Utilisateur : `/.gitconfig` ou `~/.config/git/config`

- `git config --global`

## Spécifique au dépôt : `.git/config`

- `git config --local` ou `git config` (option par défaut)

La plus locale écrase les autres.

# Configuration : `clé/valeur`

## Réglage de l'identité

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

# Configuration : `clé/valeur`

## Réglage de l'identité

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

## éditeur de texte préféré

```
$ git config --global core.editor emacs
```

# Configuration : `clé/valeur`

## Réglage de l'identité

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

## éditeur de texte préféré

```
$ git config --global core.editor emacs
```

## lister les propriétés connues

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

# Plan

# Initialiser un dépôt

## 1. Création dans un répertoire existant

git init

- `$ cd /pwd/myAwesomeProject`
- `$ git init`

```
~/Ztmp/myAwesomeProject
❯ git init
astuce: Utilisation de 'master' comme nom de la branche initiale. Le nom de la branche
astuce: par défaut peut changer. Pour configurer le nom de la branche initiale
astuce: pour tous les nouveaux dépôts, et supprimer cet avertissement, lancez :
astuce:
astuce:         git config --global init.defaultBranch <nom>
astuce:
astuce: Les noms les plus utilisés à la place de 'master' sont 'main', 'trunk' et
astuce: 'development'. La branche nouvellement créée peut être rénommée avec :
astuce:
astuce:         git branch -m <nom>
Dépôt Git vide initialisé dans /tmp/zsessionTmp/myAwesomeProject/.git/

myAwesomeProject on  master
❯ l
drwxr-xr-x - fab 16 juil. 11:00 .git
```

# Initialiser un dépôt

## 2. Clonage d'un dépôt distant ▸ git clone

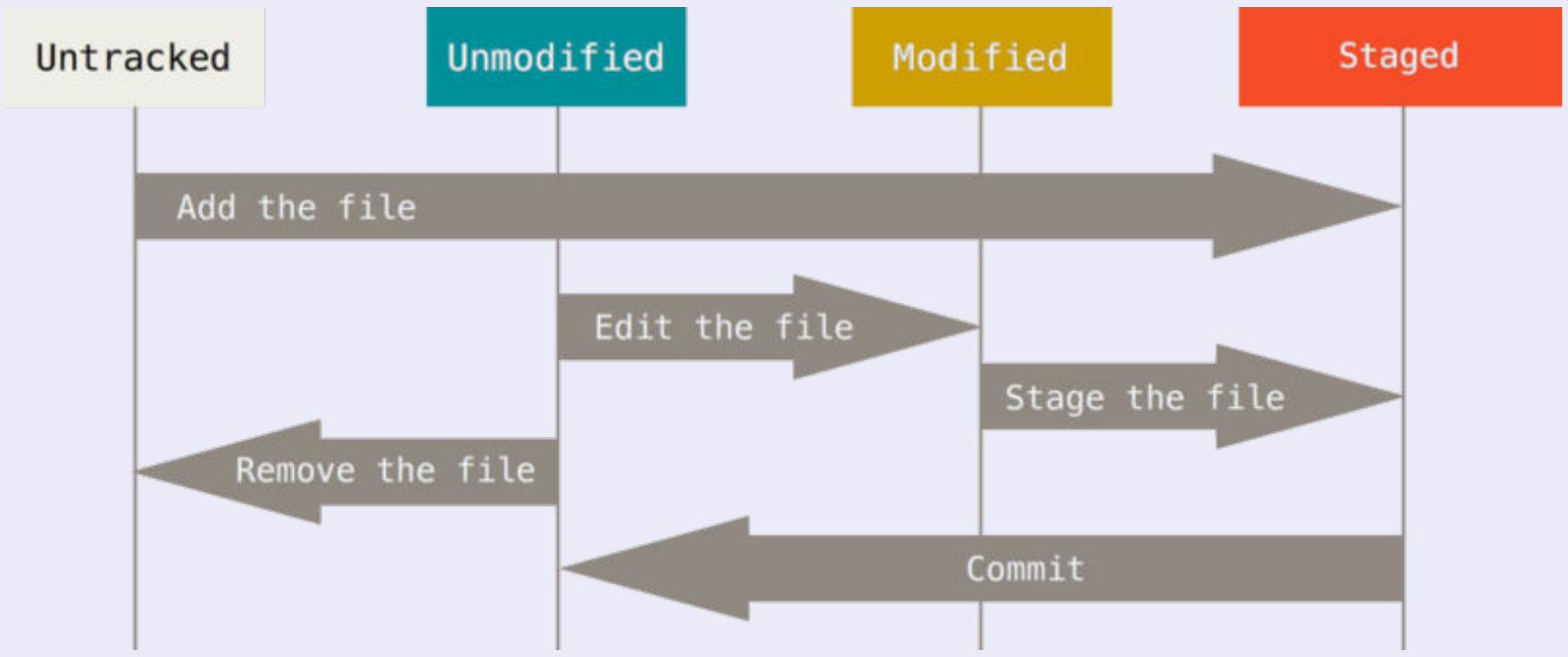
- `$ git clone URL [dir name]`

```
~/Ztmp
❯ git clone https://github.com/fmichel/MaDKit.git MaDKit-5
Clonage dans 'MaDKit-5'...
remote: Enumerating objects: 7610, done.
remote: Counting objects: 100% (75/75), done.
remote: Compressing objects: 100% (65/65), done.
remote: Total 7610 (delta 25), reused 24 (delta 10), pack-reused 7535
Réception d'objets: 100% (7610/7610), 18.02 Mio | 1.98 Mio/s, fait.
Résolution des deltas: 100% (5645/5645), fait.

~/Ztmp took 9s
❯ l
drwxr-xr-x - fab 16 juil. 11:17 MaDKit-5
drwxr-xr-x - fab 16 juil. 11:00 myAwesomeProject

~/Ztmp
❯ cd MaDKit-5

MaDKit-5 on  master is 📦 v5.3.2 via ☕ v17.0.7
❯
```

# Suivre les modifications

**Très important à comprendre**

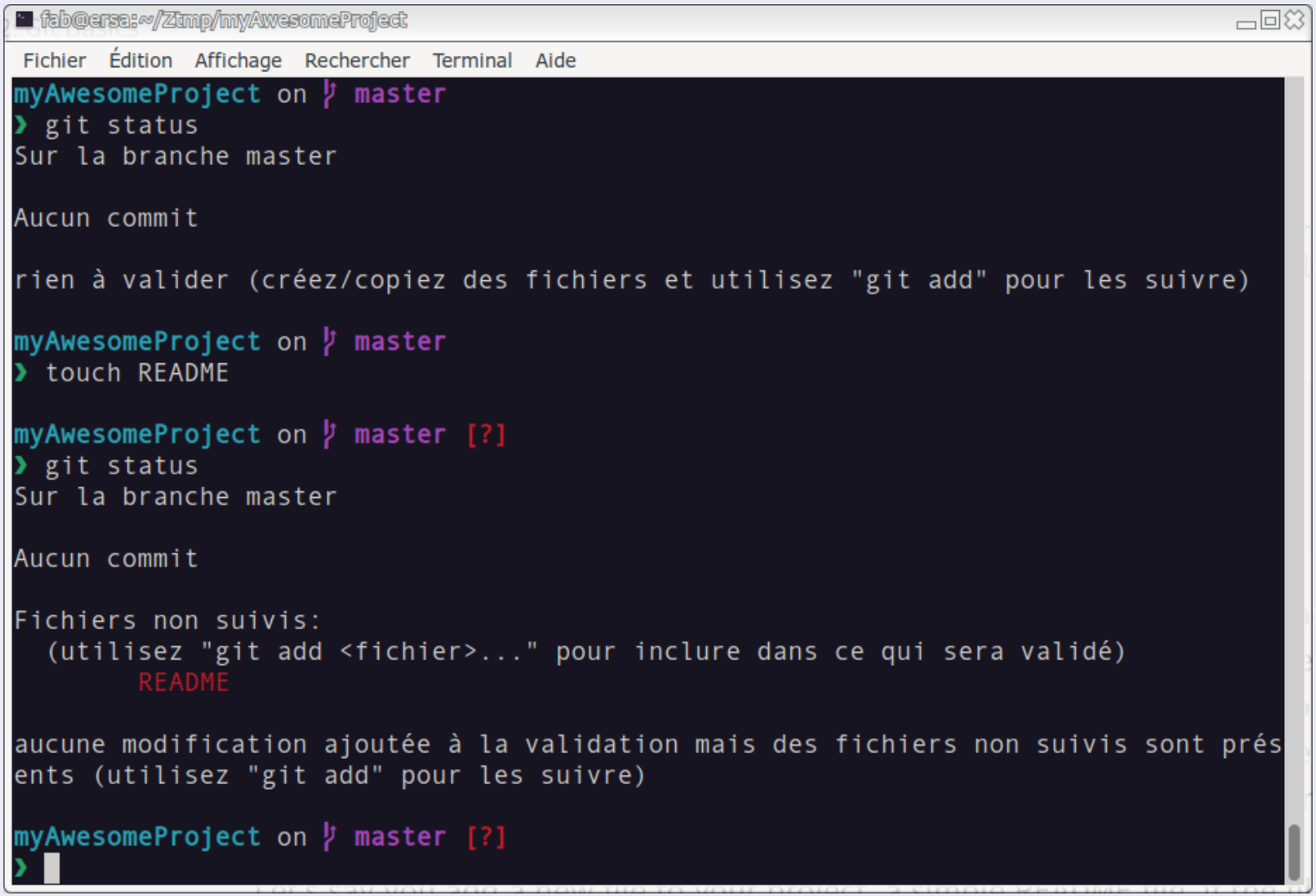
## Les fichiers sont dans l'un des 4 états suivants :



# Suivre les modifications

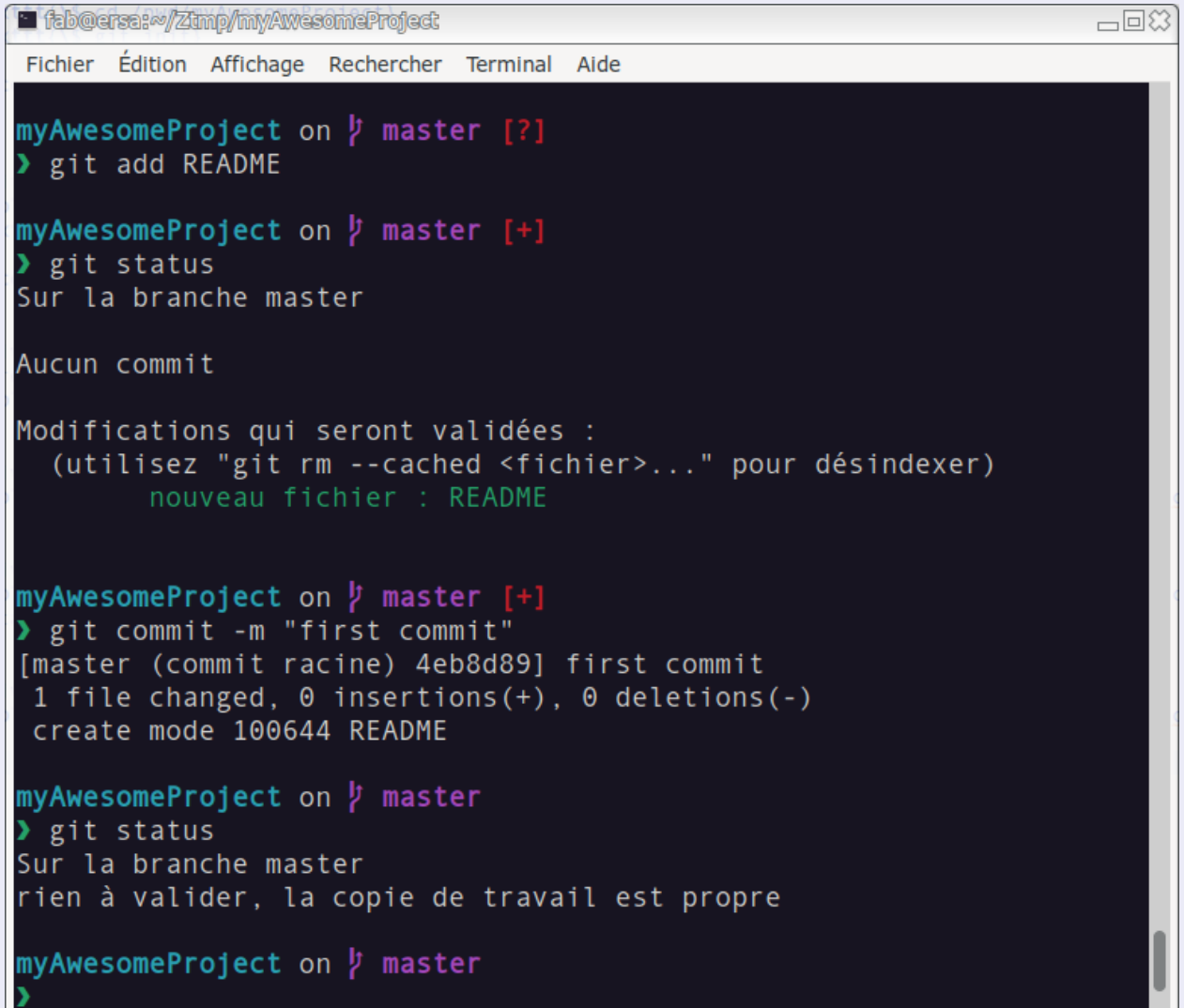
## `git status` : vérifie l'état des fichiers

▸ git status



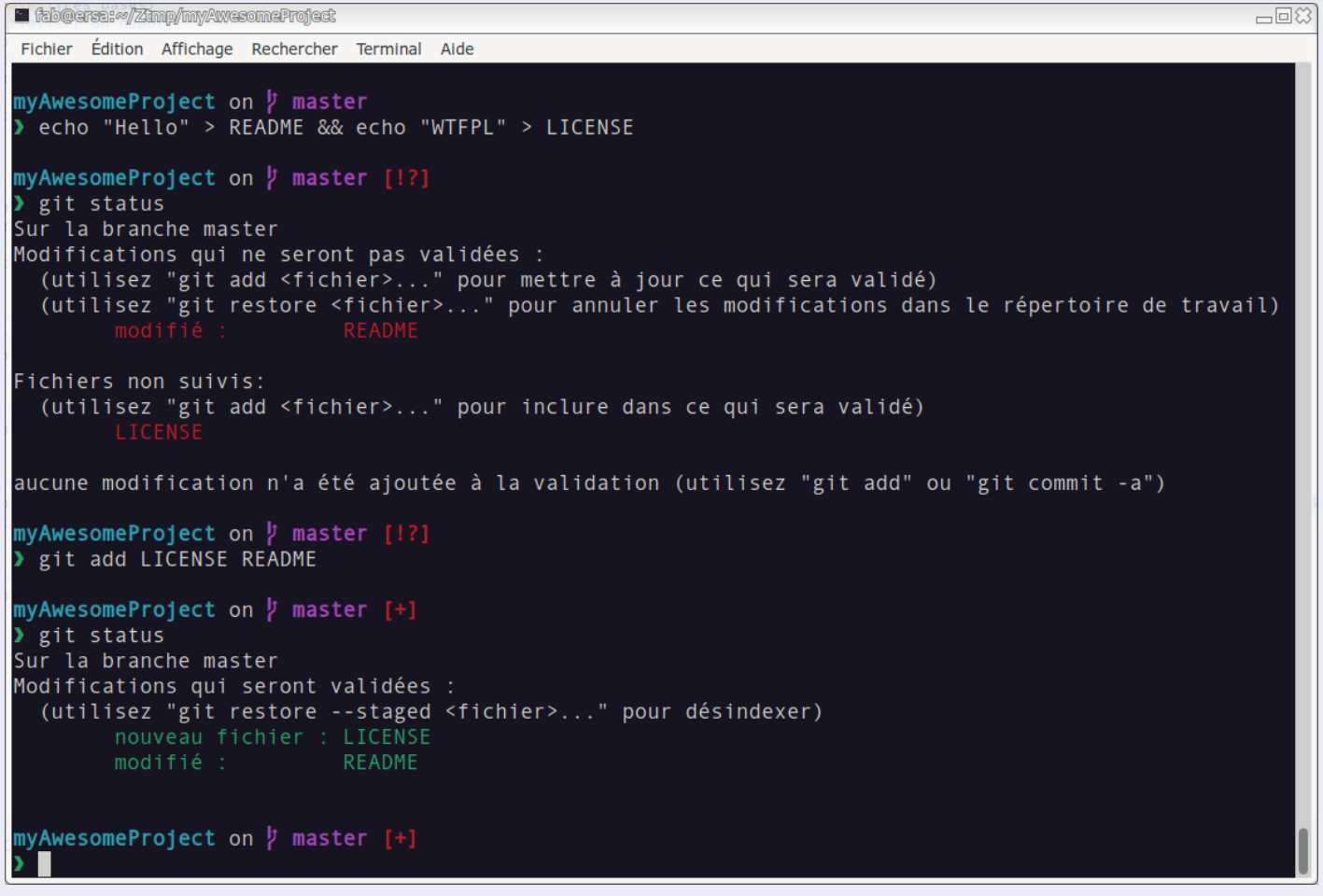
# Suivre les modifications

▸ git add ⇒ *staged* ▸ git commit ⇒ validation *staged*



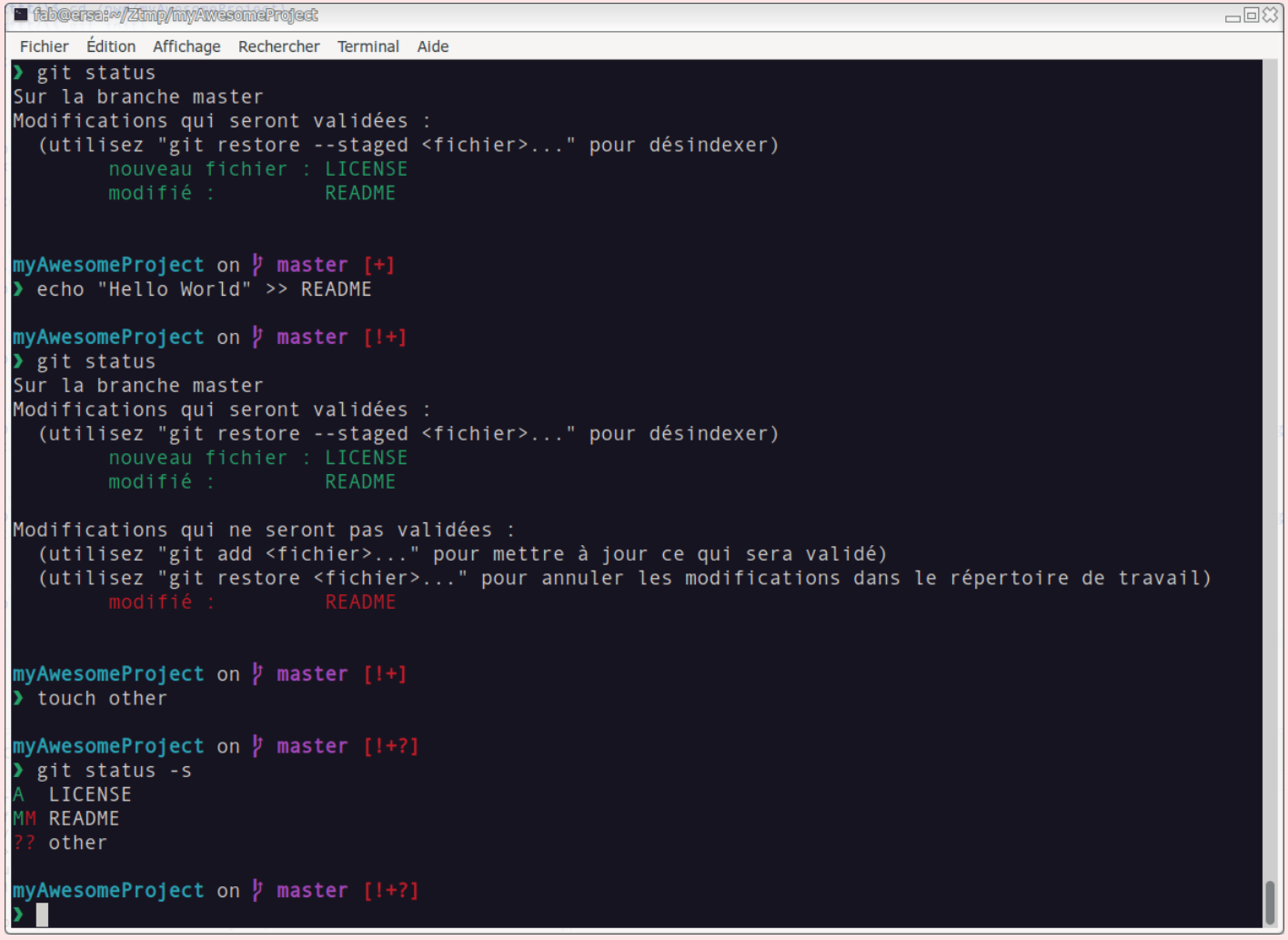
# Suivre les modifications

▸ git add ⇒ *staged* ▸ git commit ⇒ validation *staged*



# Suivre les modifications

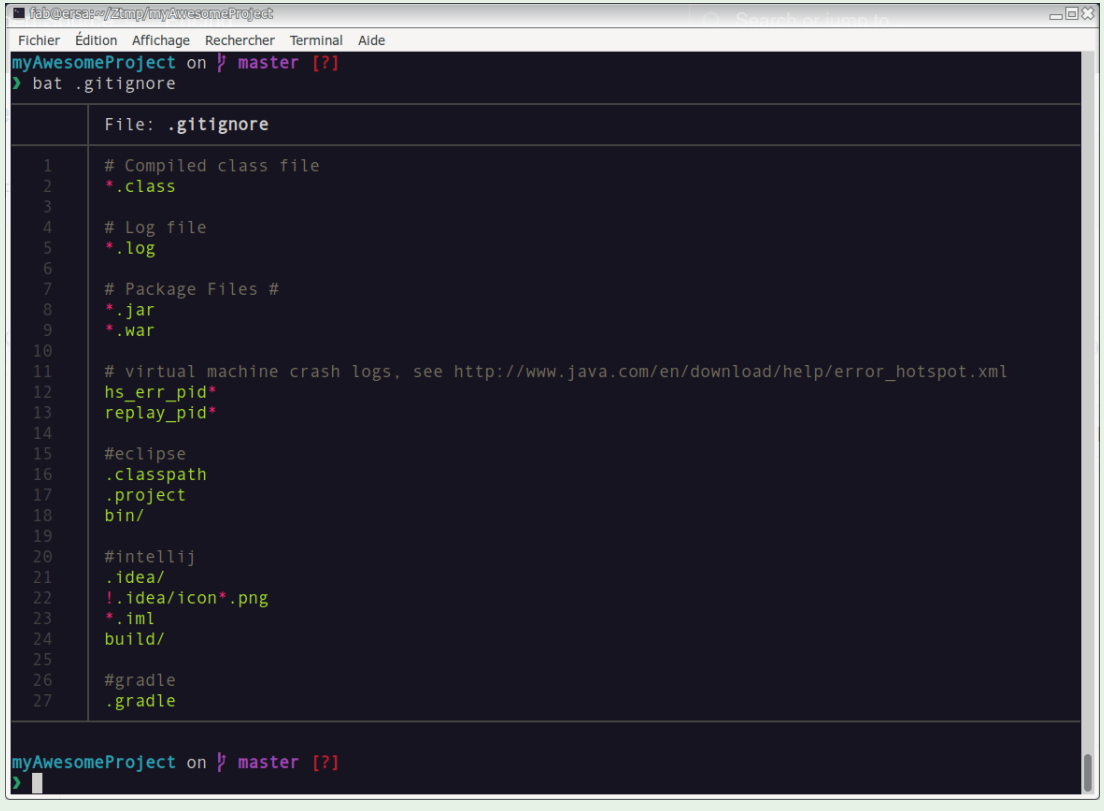
**git add** ⇒ ***staged*** **: le commit prendra cet état**



# Fichiers non suivis : `.gitignore`

C'est l'un des fichiers les plus importants pour Git

▶ gitignore



# Fichiers non suivis : `.gitignore`

## Quels fichiers ignorés ?

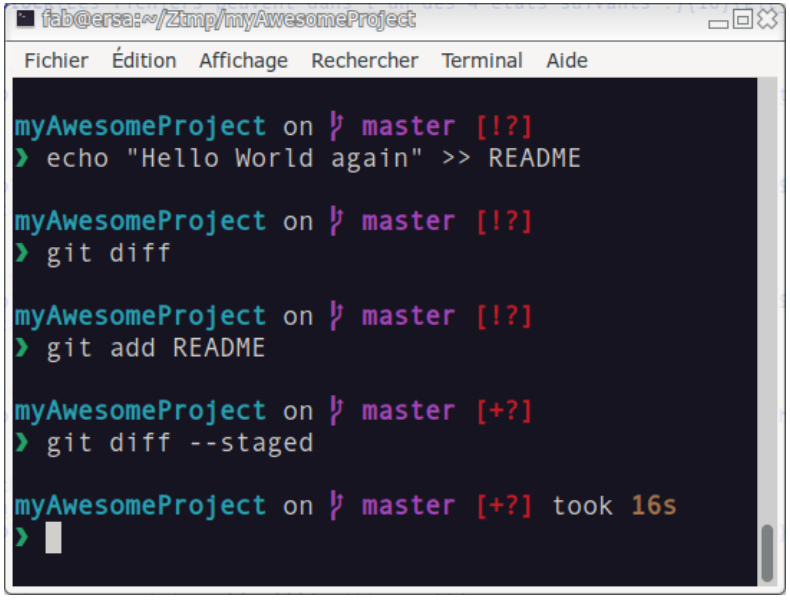
- TOUS les fichiers générés par un *build*
- TOUS les fichiers liés aux IDE
- les fichiers qui ne sont que pour vous

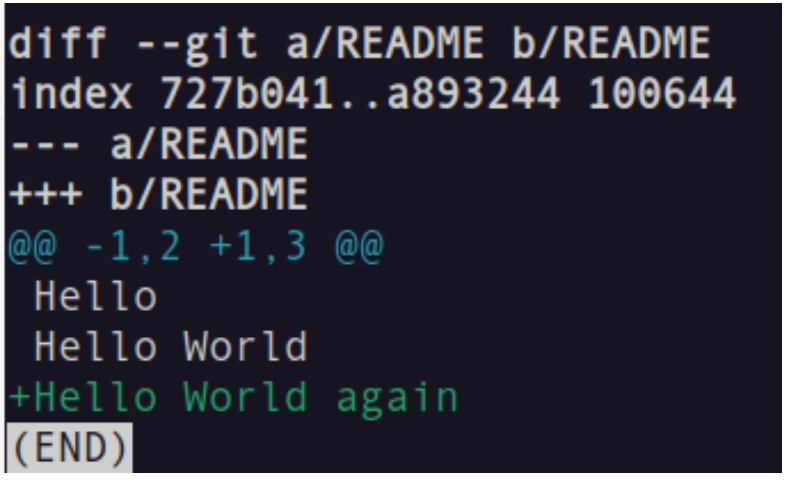
**Créer, remplir et ajouter (add) ce fichier est quasiment la première chose à faire pour tout nouveau dépôt**

# Visualiser les modifications

git diff

`git diff` : compare *working dir* vs. *staged*
`git diff --staged` : compare *staged* vs. *last commit*



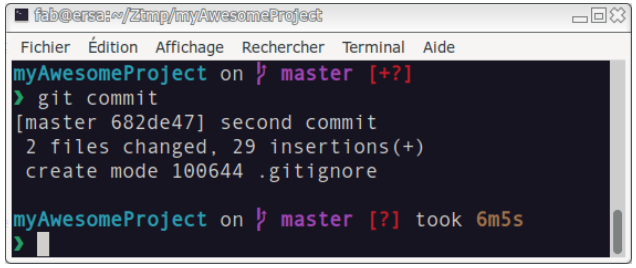



# Valider les modifications

git commit

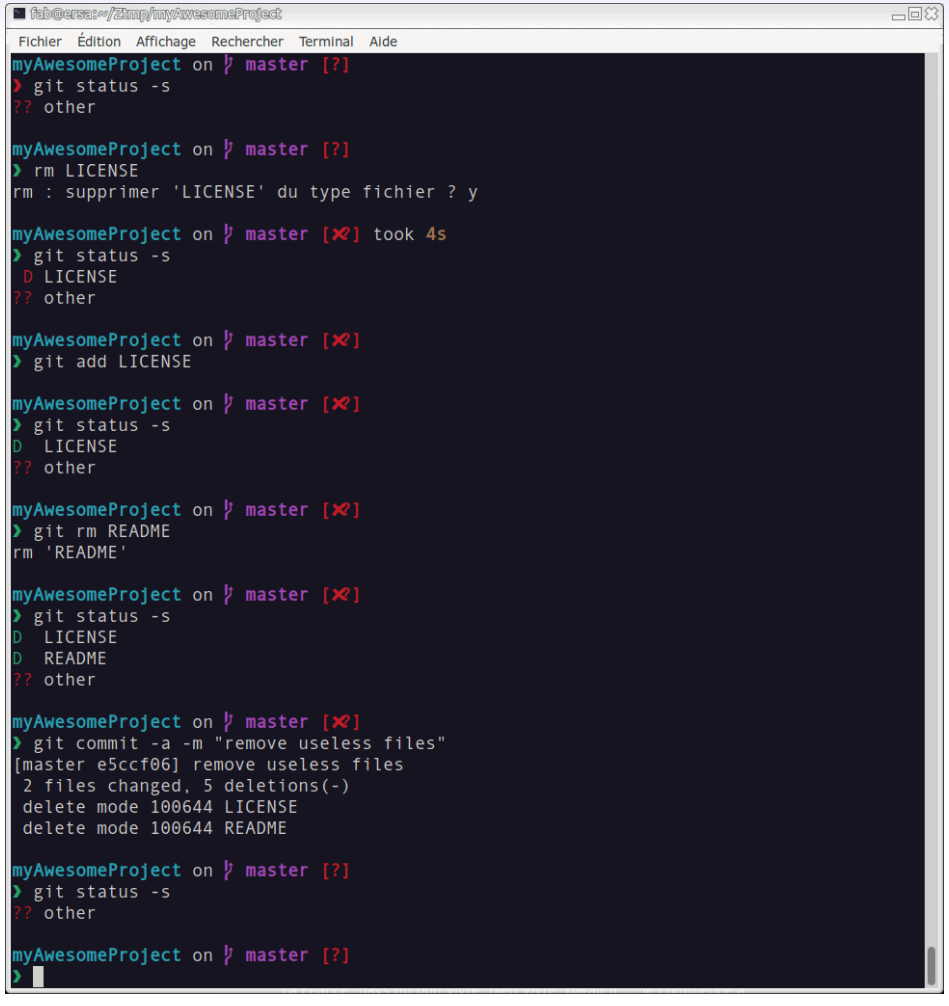
`git commit` : valide les fichiers *staged* et ouvre un éditeur pour définir le *commit* via le fichier `.git/COMMIT_EDITMSG`


équivalent à ⇒ `git commit -m "second commit"`



# Effacer des fichiers

git rm




Variante : `git rm --cached README` arrête le suivi sans effacement (*e.g.* après un oubli dans `.gitignore`)

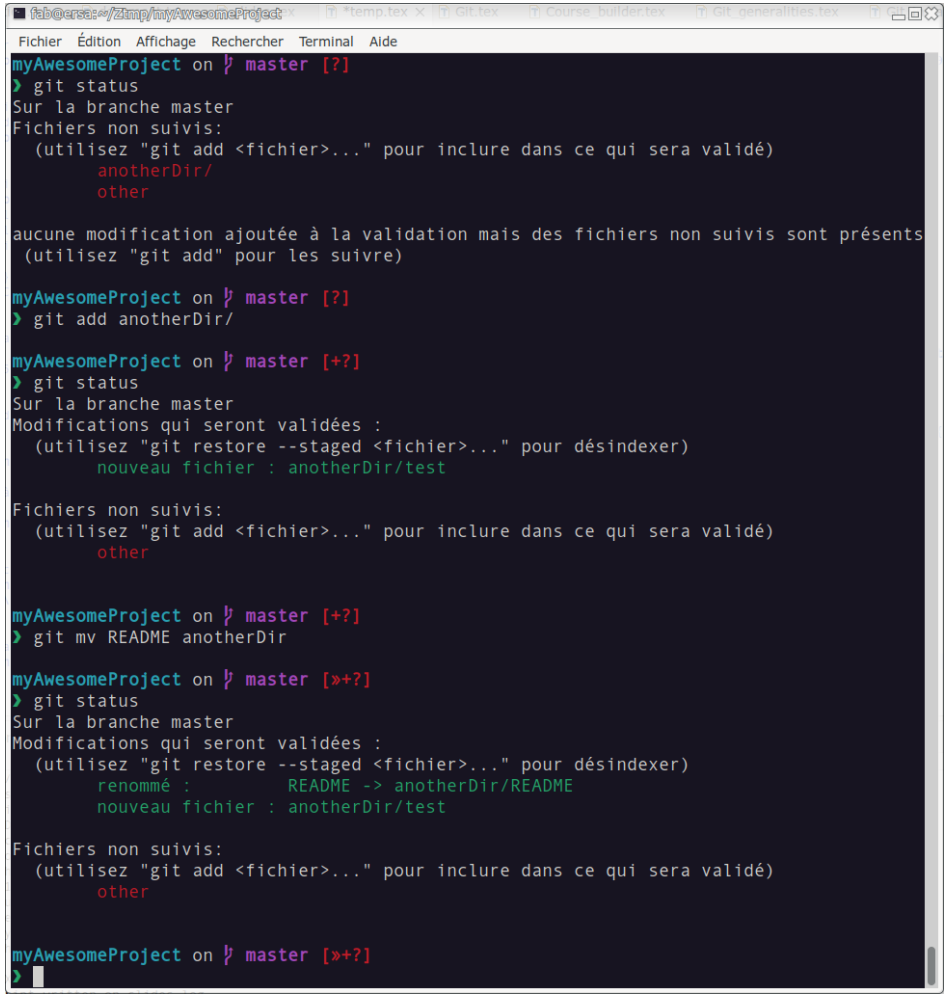
# Déplacer / Renommer des fichiers

git mv

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

    renamed:    README.md -> README
```

However, this is equivalent to running something like this:

```
$ mv README.md README
$ git rm README.md
$ git add README
```



# Visualiser l'historique

git log

## `git log` : ordre inversé


Il existe de très nombreuses variantes :

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

The most interesting option is `format`, which allows you to specify your own log output format. This is especially useful when you're generating output for machine parsing – because you specify the format explicitly, you know it won't change with updates to Git:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

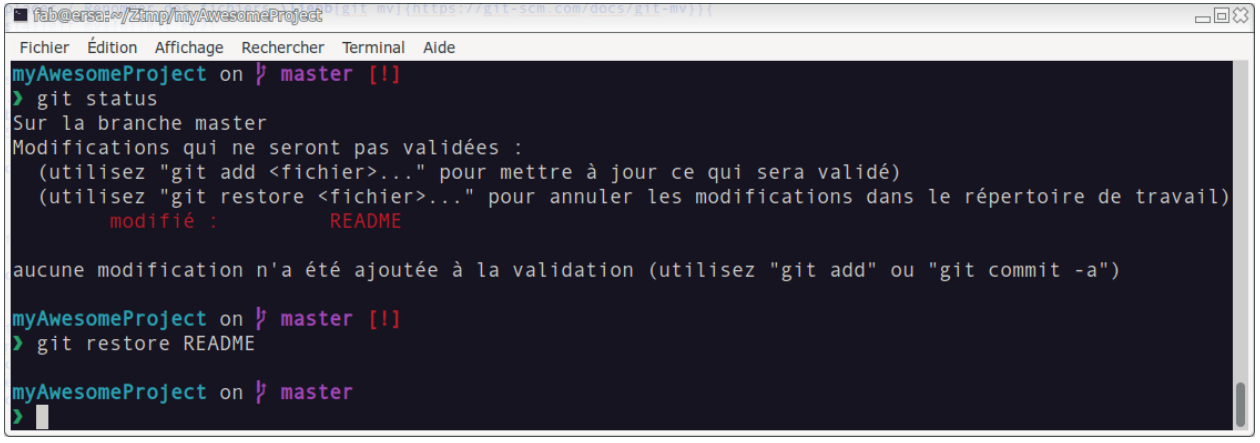
# Refaire et annuler

redéfinir le dernier commit ⇒ `$ git commit --amend`

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

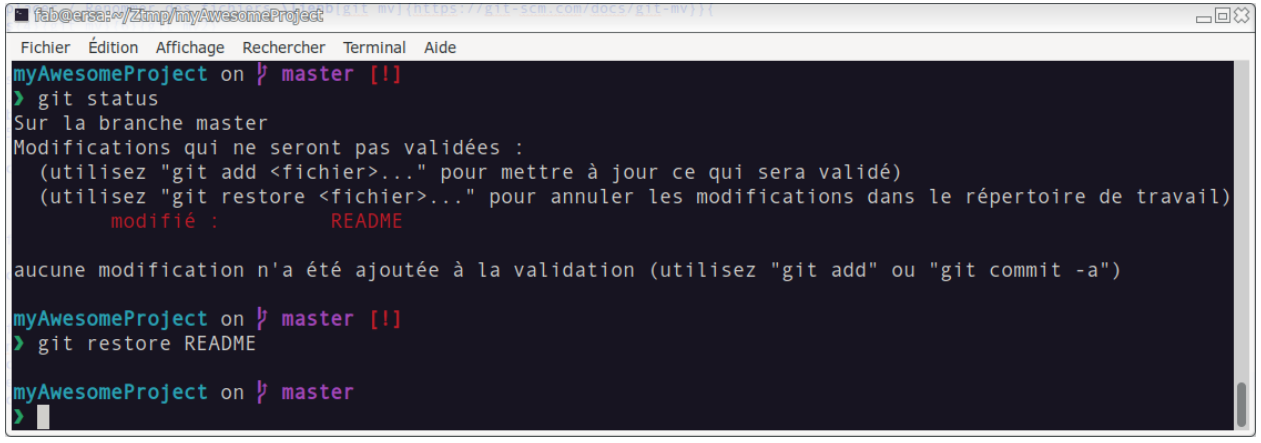
# Refaire et annuler

Annuler des modifications ⇒ `$ git restore file`

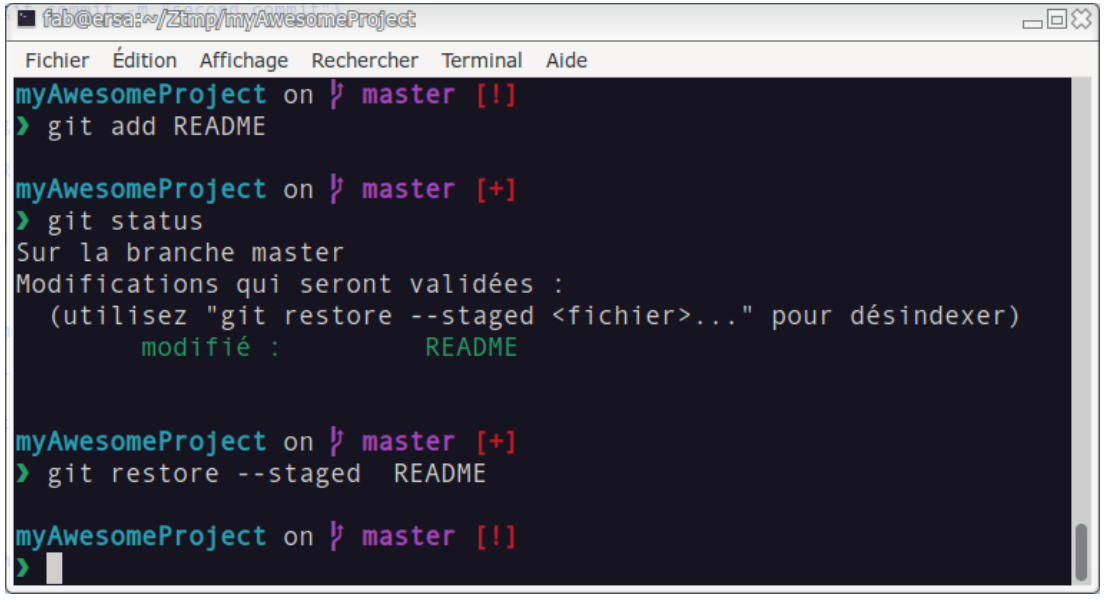


# Refaire et annuler

Annuler des modifications ⇒ `$ git restore file`



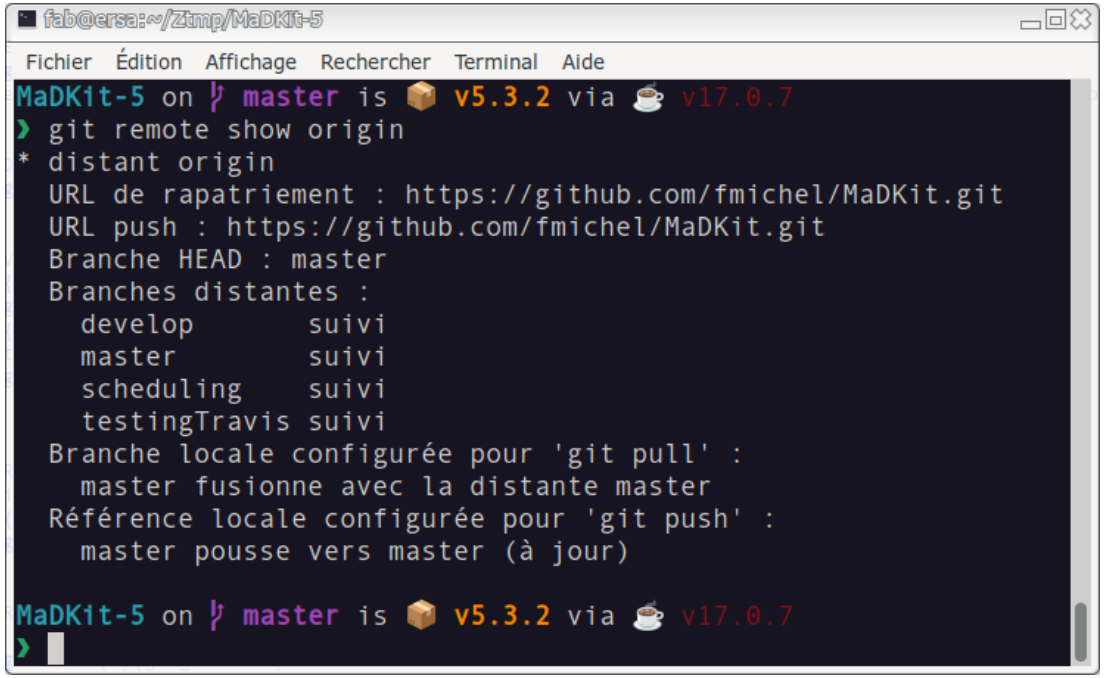

*Unstaging* ⇒ `$ git restore --staged file` ▸ git restore



# Inspecter un dépôt distant

- git remote

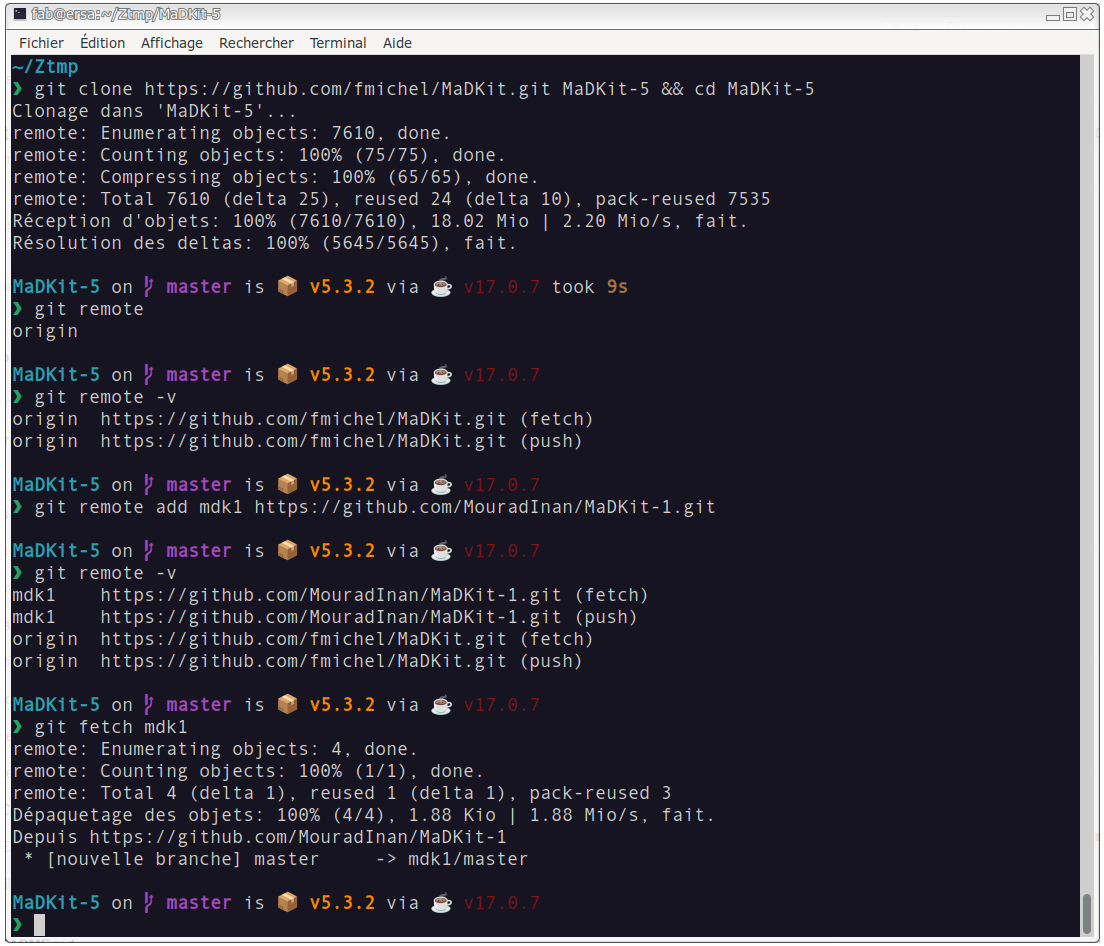
```
$ git remote show [remote-name]
```



# Ajouter des dépôts distants

▸ git remote Lister / ajouter des dépôts distants. Par exemple

⇒ `$ git remote add [remote-name] [repo-URL]`




Il est possible de les renommer, effacer, . . .

# Récupérer des dépôts distants

▸ git fetch ⇒ `$ git fetch [remote-name]`

```
$ git fetch [remote-name]
```

The command goes out to that remote project and pulls down all the data from that remote project that you don't have yet. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time.

Un `git pull` est un `git fetch` enchaîné avec `git merge`

Un `git pull` direct est le cas d'utilisation le plus courant

# Tagging

git tag

`git tag` permet de marquer un commit considéré comme majeur, et d'y ajouter une annotation

Obtenir le listing des tags :

```
$ git tag
v0.1
v1.3
```

```
$ git tag -l 'v1.8.5*'
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

# Tagging

git tag

Créer un tag annoté :

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

# *Tagging* git tag

## Il est possible de

- créer un tag sans info (commit courant) : `$ git tag 1.5`
- créer un tag sur un commit plus ancien :
  `$ git tag -a v1.2 9fceb02`

# Tagging

git tag

## Il est possible de

- créer un tag sans info (commit courant) : `$ git tag 1.5`
- créer un tag sur un commit plus ancien :
  `$ git tag -a v1.2 9fceb02`

## Par défaut les tags sont uniquement locaux

- partager un tag : `$ git push origin v1.5`
- partager tous les tags : `$ git push origin --tags`

# Git Aliases

On peut créer des alias pour certaines commandes :

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

⇒ `$ git ci` pour `$ git commit`

Le mieux est encore de créer des alias dans son fichier de config (e.g. `.bashrc`) !

# Plan

# Git Branching

## Atouts

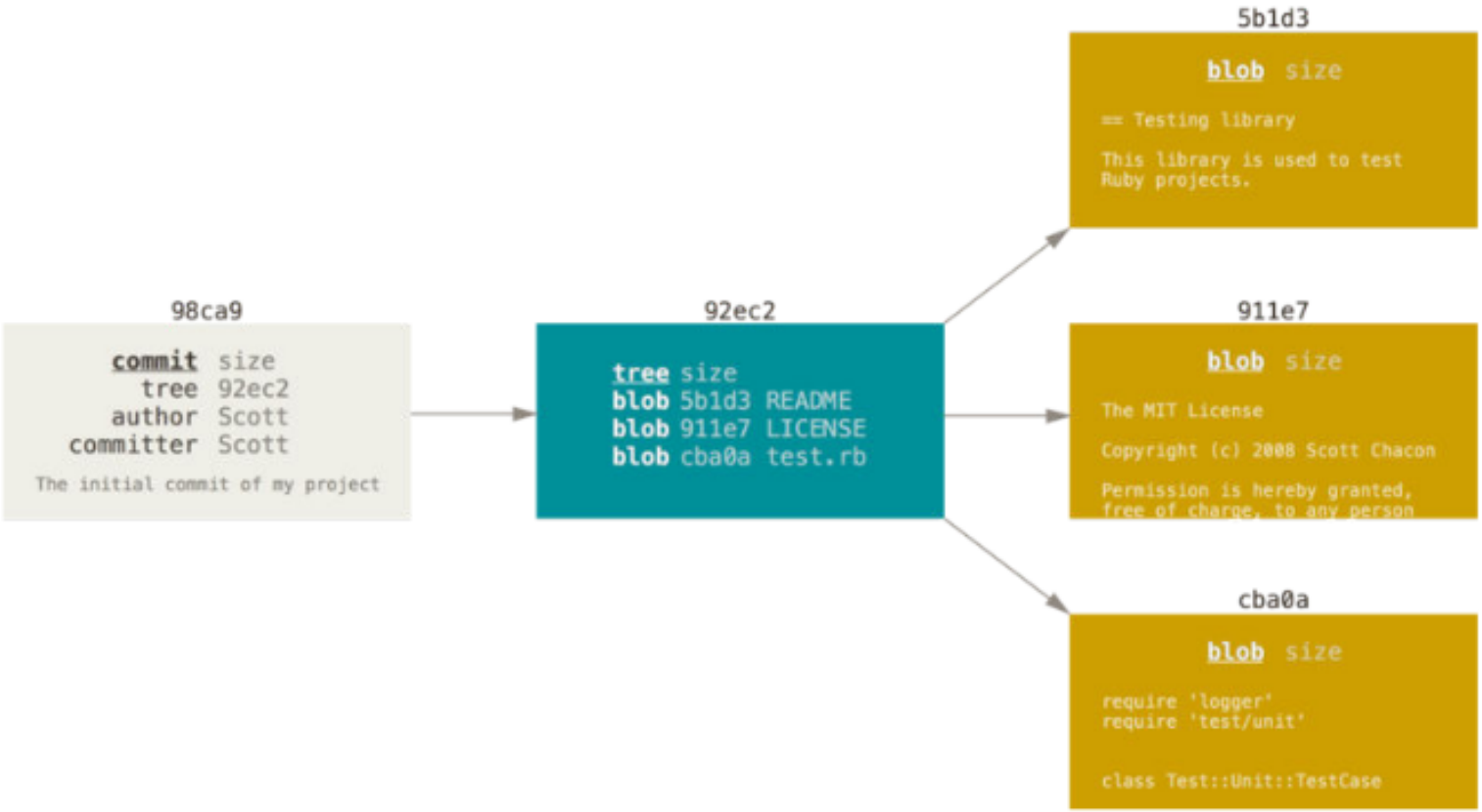
git branch

- C'est la *killer feature* de Git
- Créer une branche est une opération très légère et instantanée
- Passer d'une branche à une autre aussi !
- $\Rightarrow$ créer et fusionner des branches est naturel

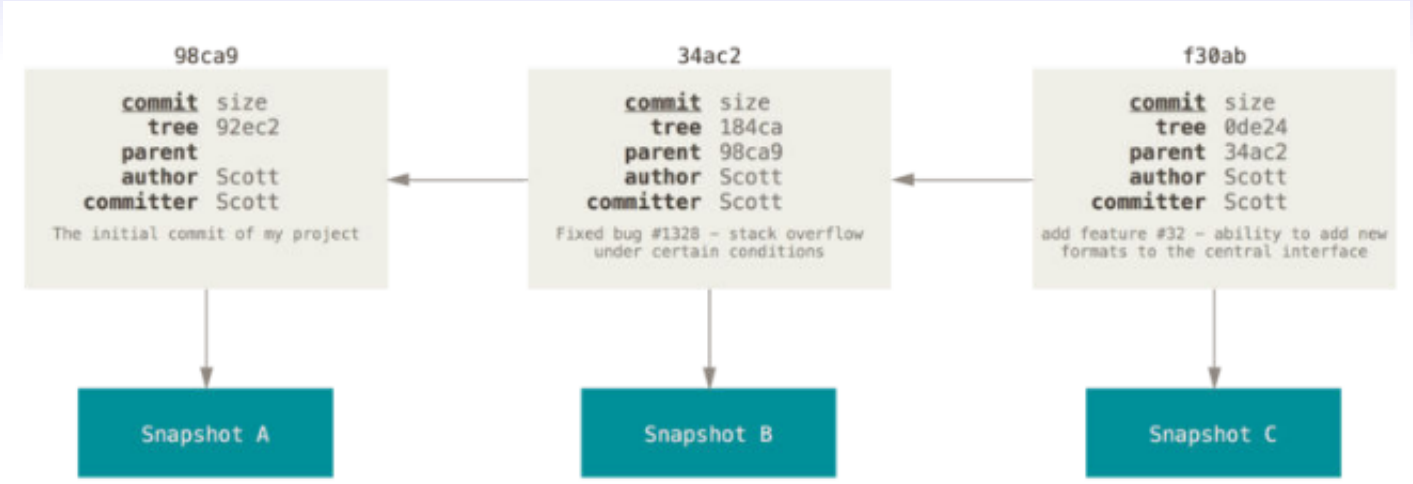
# Fonctionnement interne

```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

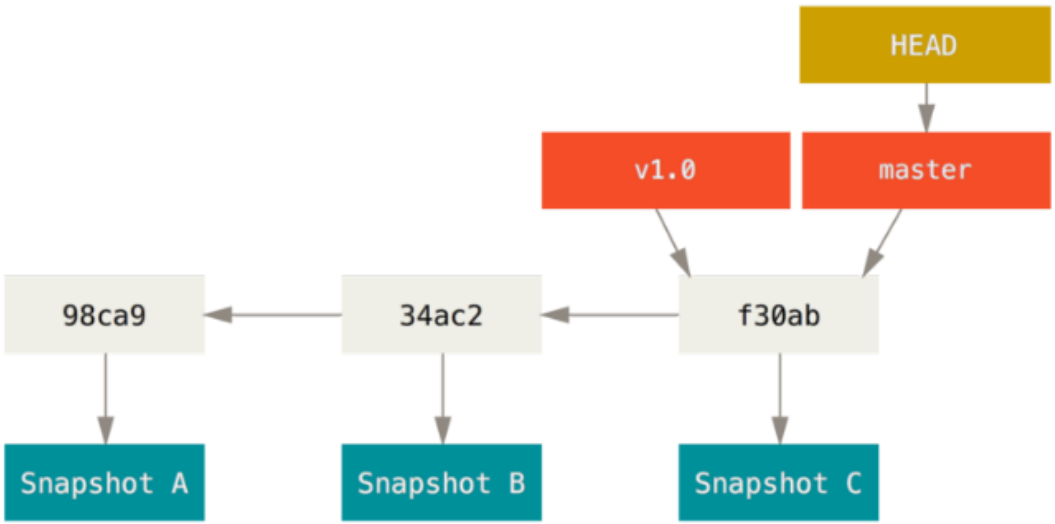
*commit, tree and blob* :



# Fonctionnement interne




Une branche est juste un pointeur sur un commit :

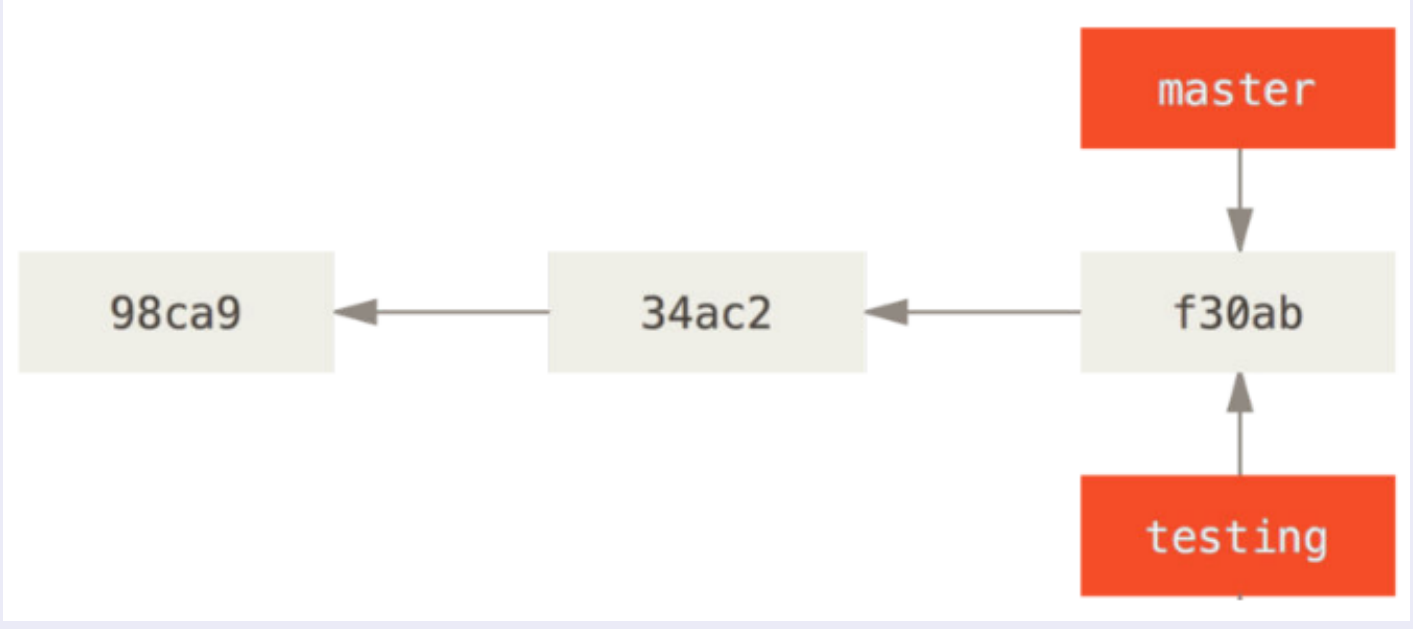


# Créer une branche

git branch

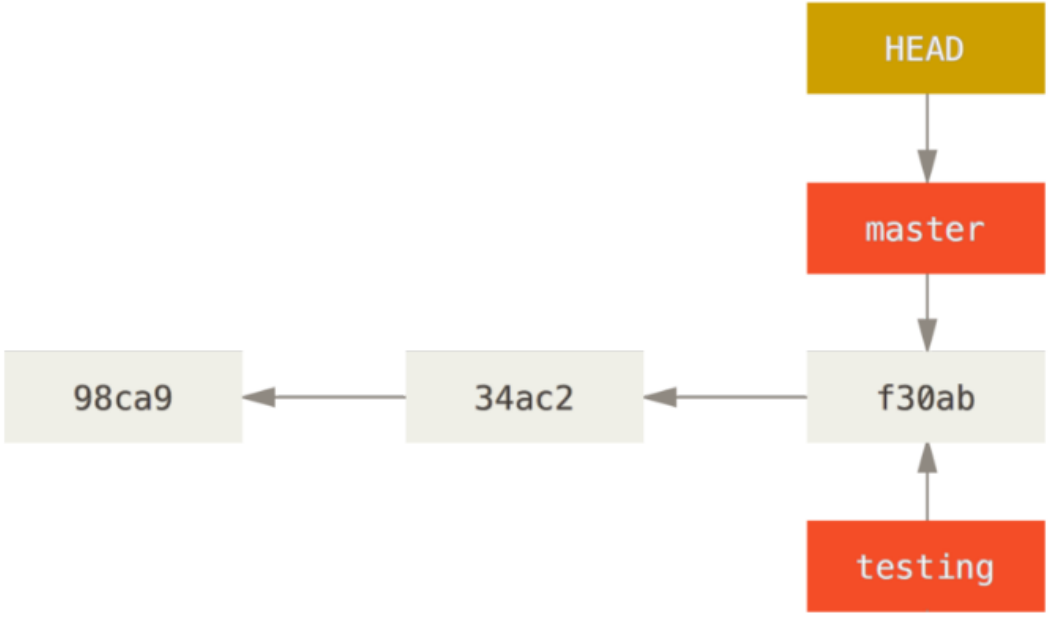
Créer une branche revient à créer un nouveau pointeur :

```
$ git branch testing
```



# Le pointeur `HEAD`

L'état du répertoire correspond au snapshot pointé par `HEAD` :



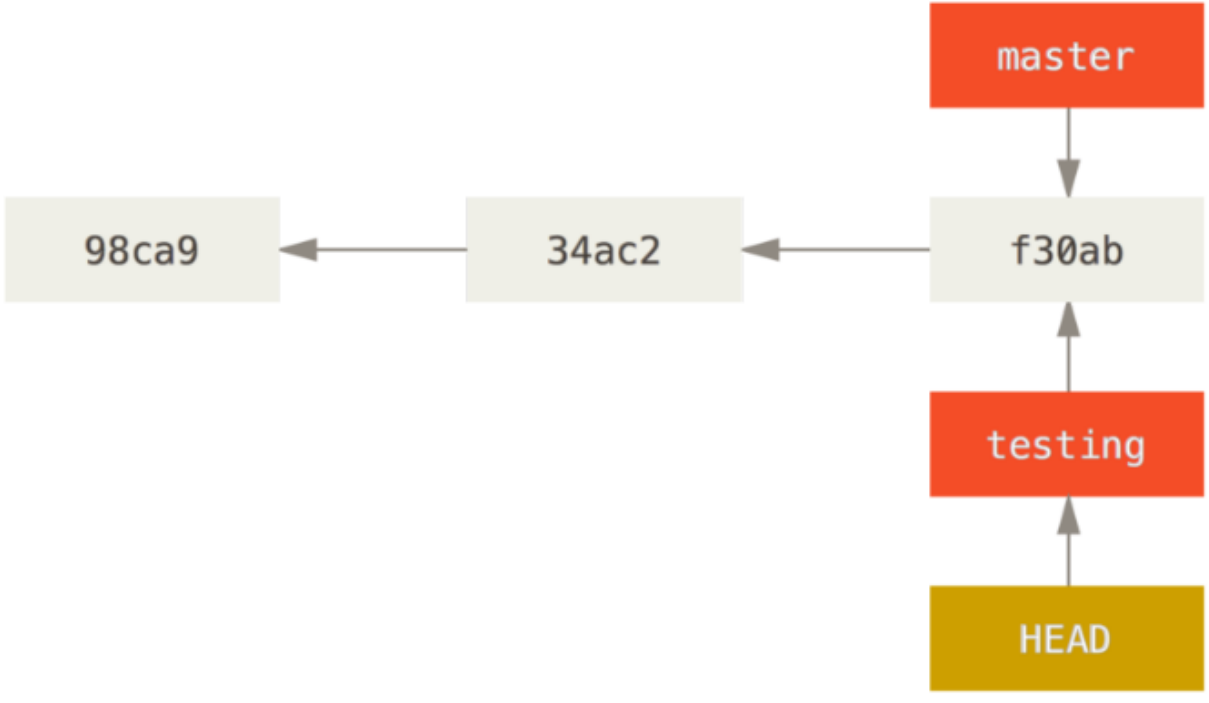

On peut voir quels pointeurs sont sur un commit :

```
$ git log --oneline --decorate
f30ab (HEAD, master, testing) add feature #32 - ability to add new
34ac2 fixed bug #1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

# Changement de branche

git checkout

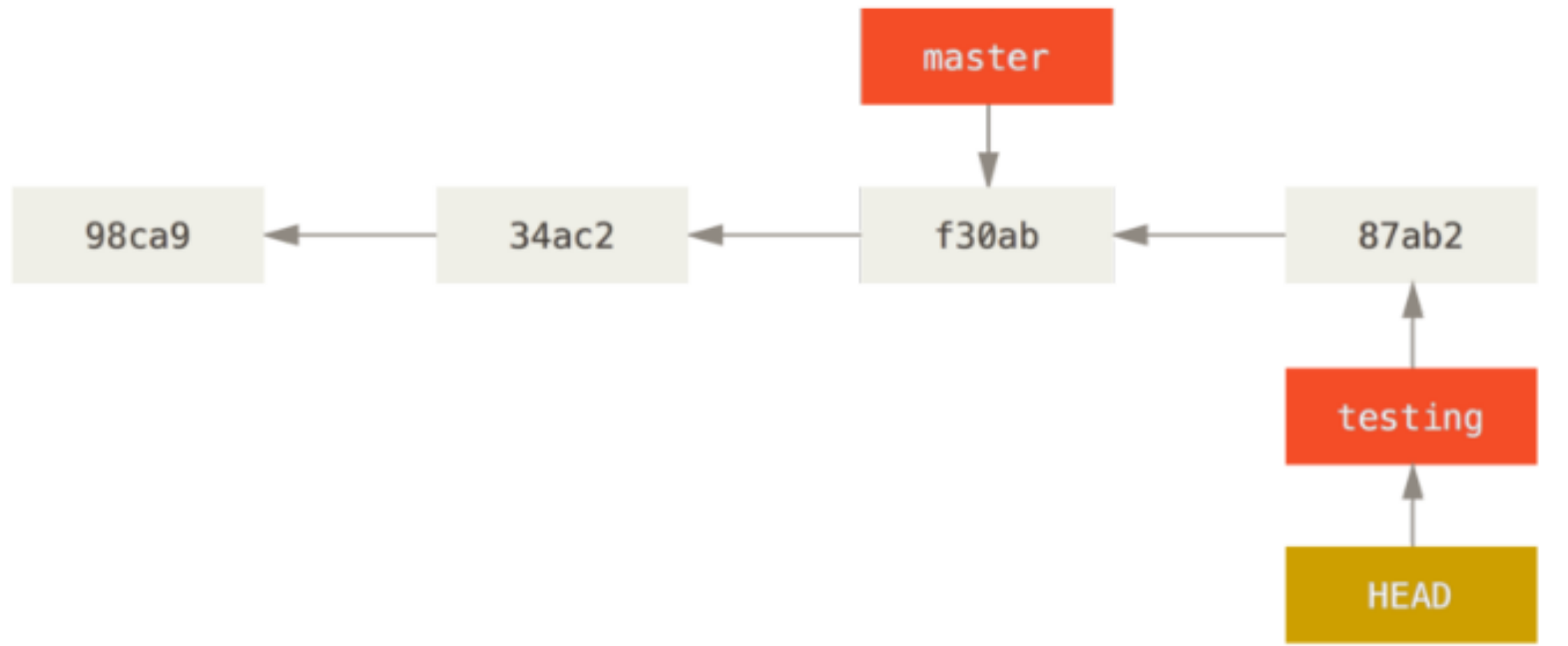
`$ git checkout testing` ⇒ HEAD pointe sur `testing`



# Working on testing

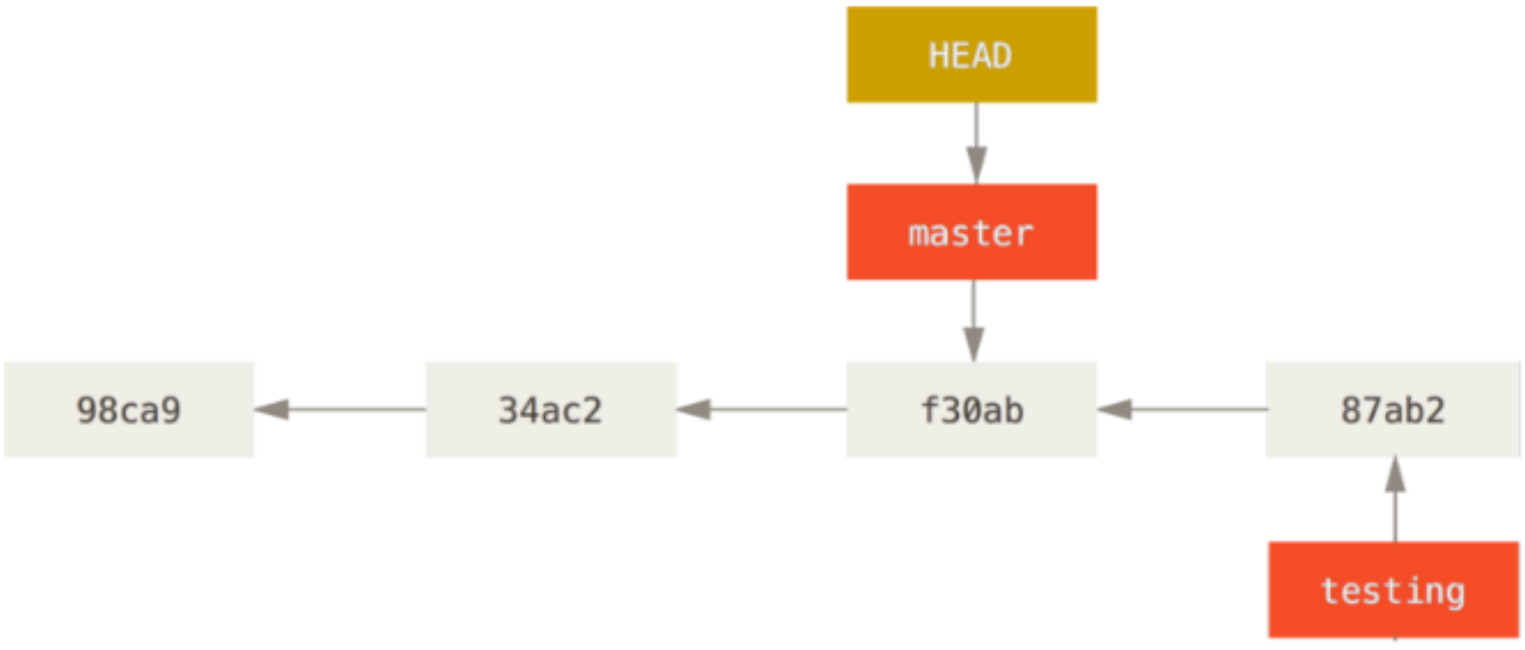
Un commit déplace `HEAD` et le pointeur de la branche de travail

```
$ vim test.rb
$ git commit -a -m 'made a change'
```



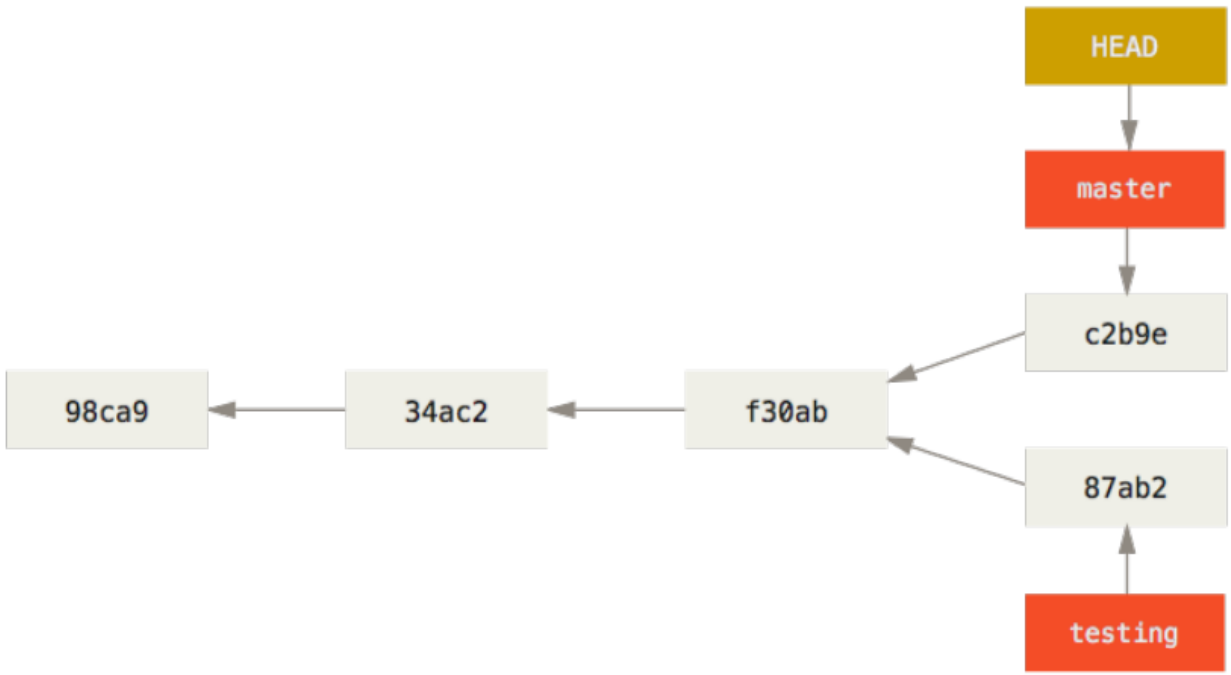
# Retour sur *master*

```
$ git checkout master
```



# *Working on master* ⇒ historique divergent

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```




```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

# Plan

# Branching and Merging 1

Un scénario usuel :

1. Do work on a web site.
2. Create a branch for a new story you're working on.
3. Do some work in that branch.

At this stage, you'll receive a call that another issue is critical and you need a hotfix. You'll do the following:

1. Switch to your production branch.
2. Create a branch to add the hotfix.
3. After it's tested, merge the hotfix branch, and push to production.
4. Switch back to your original story and continue working.

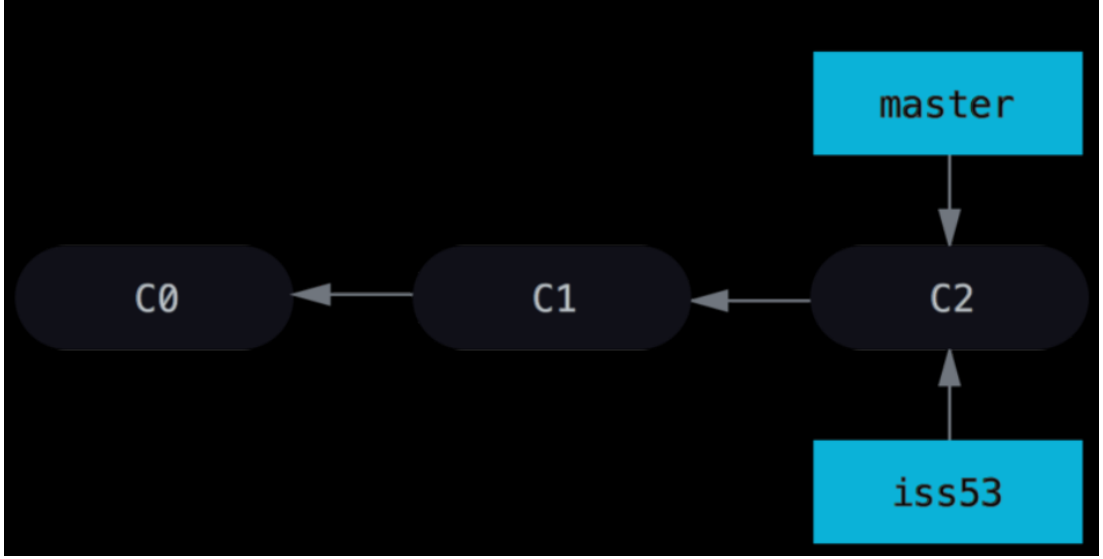
# Branching and Merging 2

2. *Create a branch for a new story you're working on*

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

This is shorthand for:

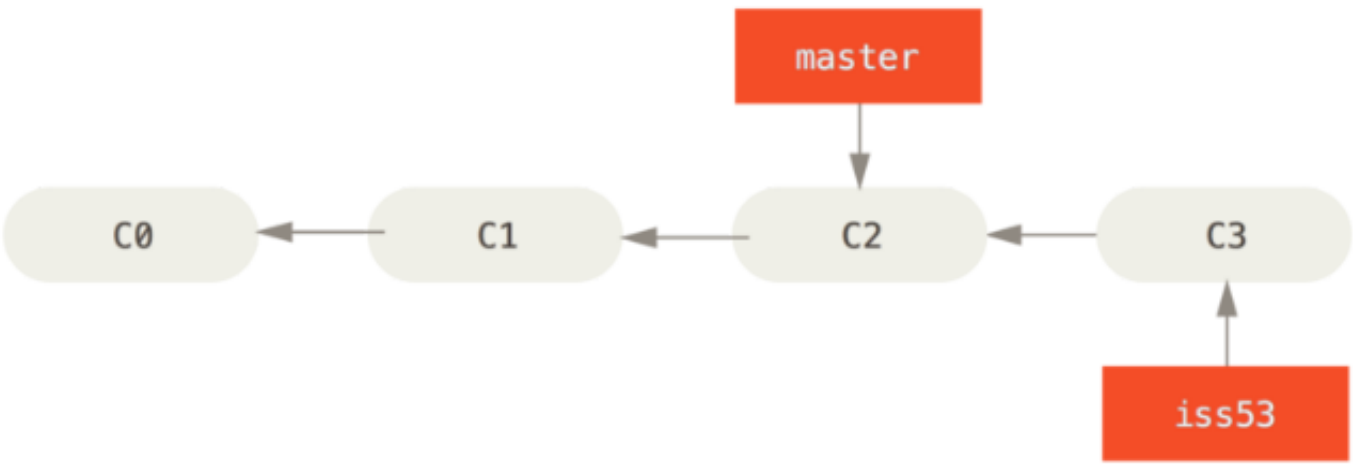
```
$ git branch iss53
$ git checkout iss53
```



# Branching and Merging 3

*3. Do some work in that branch*

```
$ vim index.html
$ git commit -a -m 'added a footer [issue 53]'
```



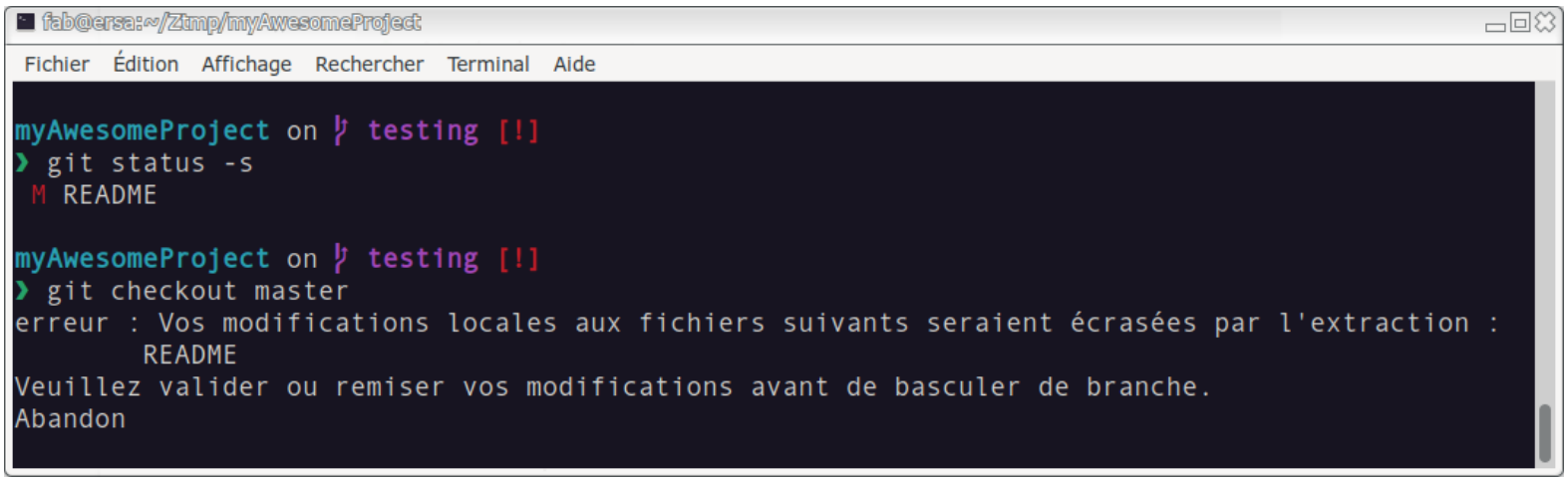
# Branching and Merging 4

*The call: a fix is needed immediately!*

*1. Switch to your production branch*

```
$ git checkout master
$ Switched to branch 'master'
```

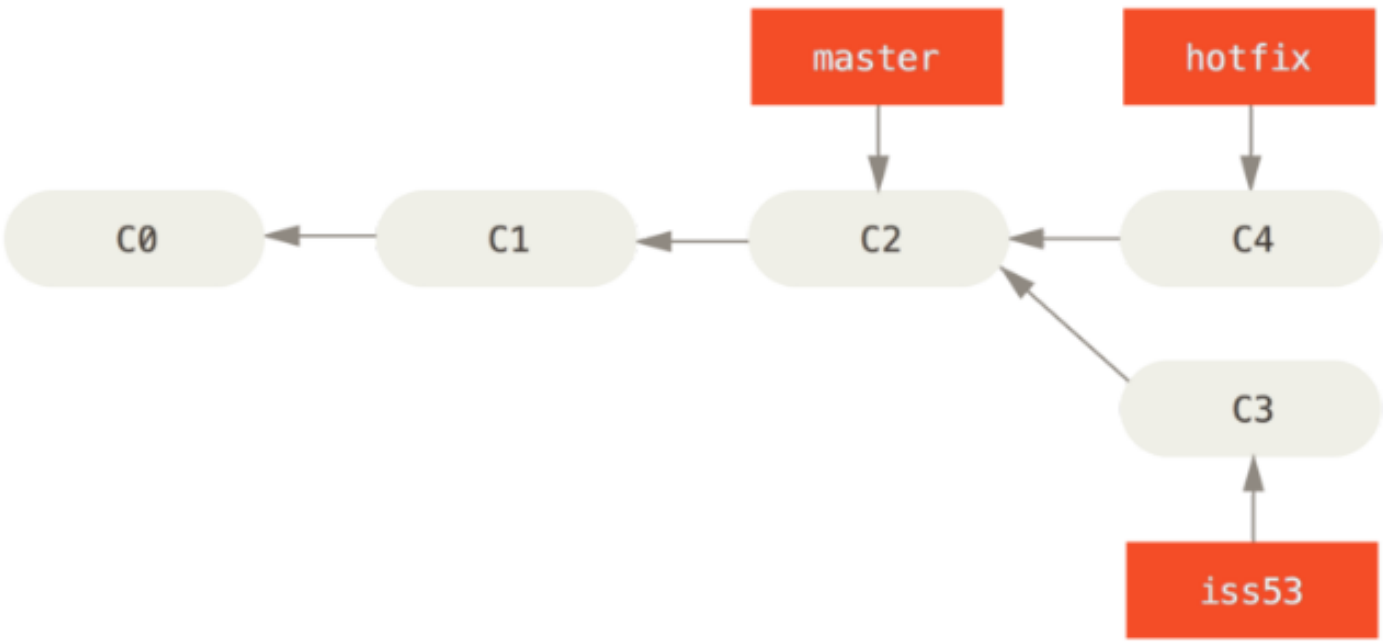
**Pour que le `checkout` marche, il faut que la branche de départ soit *clean* ou bien *stashed*…**



# Branching and Merging 5

## 2. Create a branch to add the hotfix

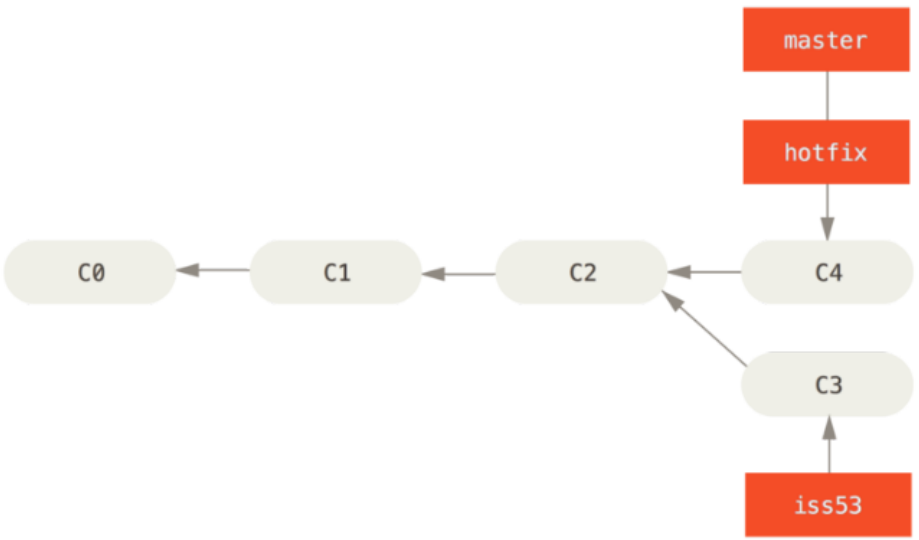
```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed broken email'
```



# Branching and Merging 6

*3. After it's tested, merge the hotfix branch, and push to production* ▸ git merge

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```




Un *fast-forward* du pointeur *master* est effectué

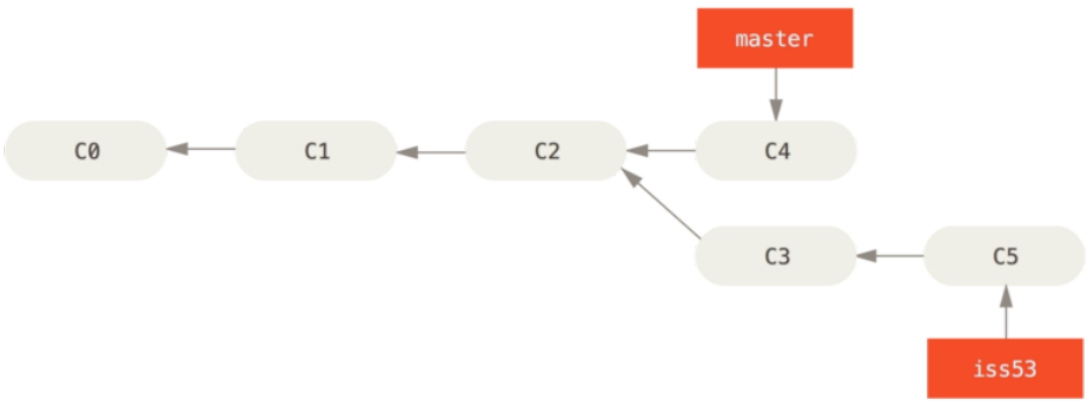
# Branching and Merging 7

## 4. Switch back to your original story and continue working

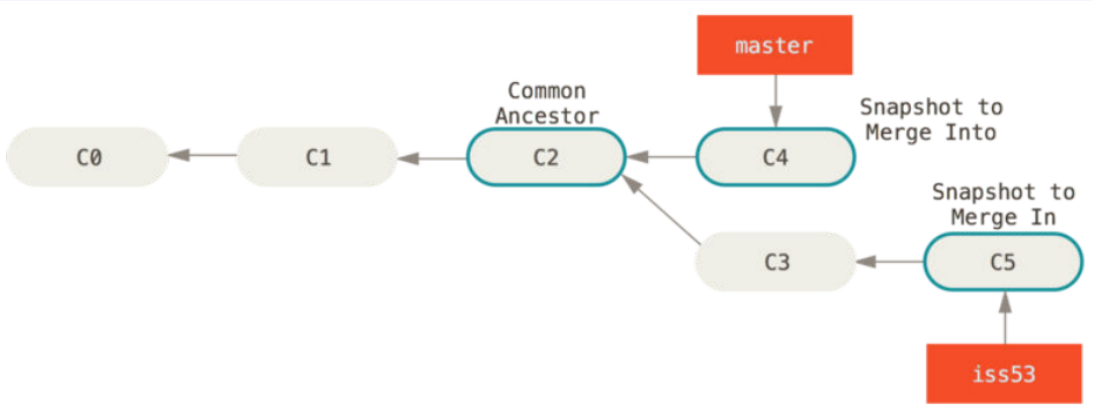
```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Now you can switch back to your work-in-progress branch on issue #53 and continue working on it.

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

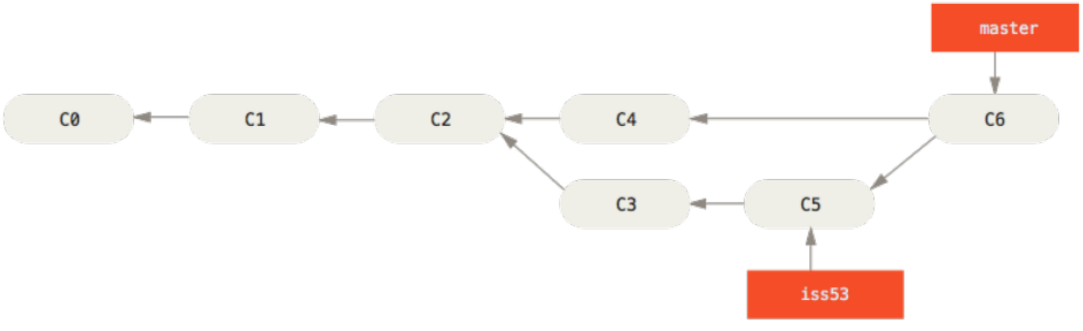


# Branching and Merging 7



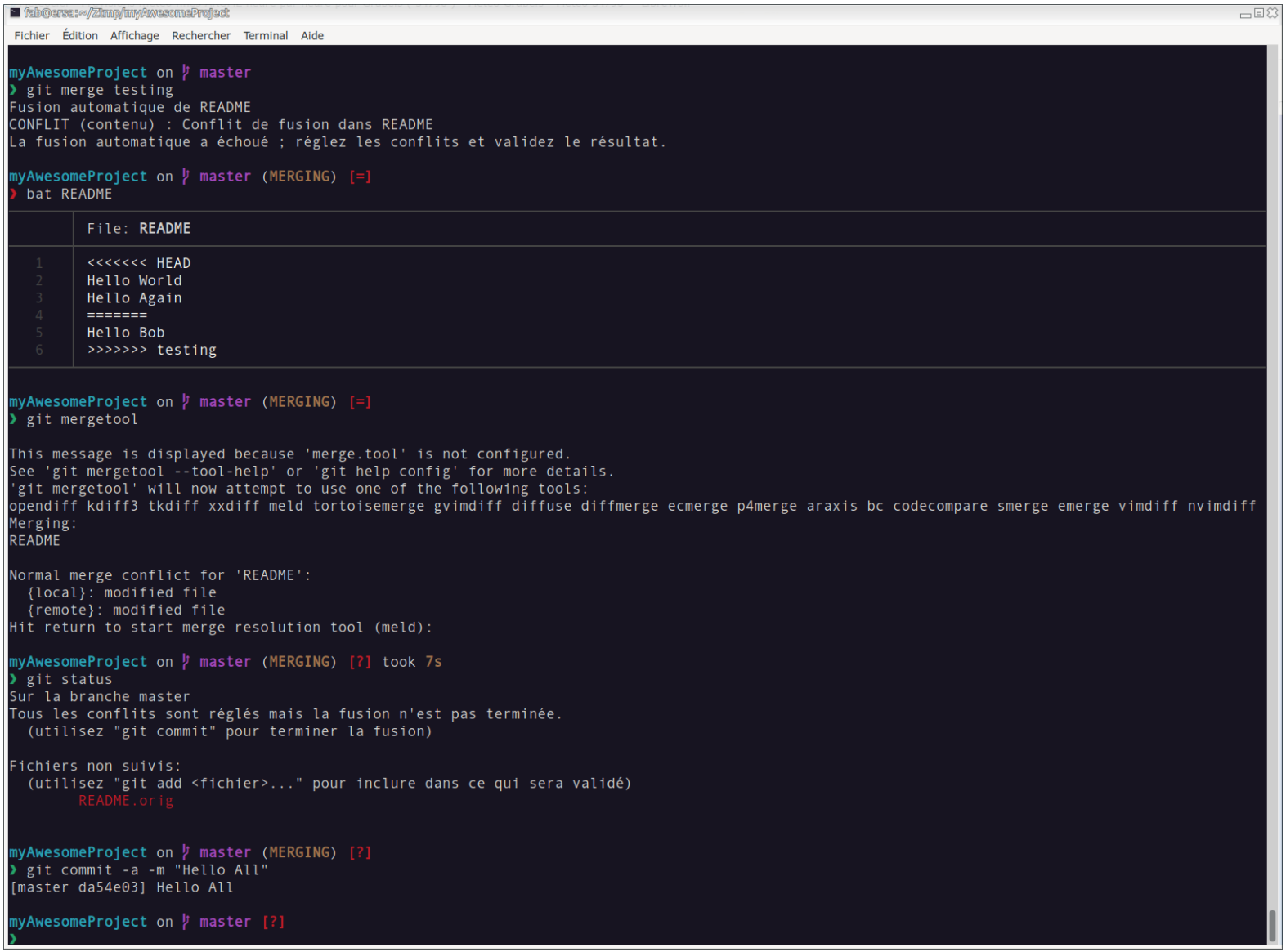

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

stratégie récursive ⇒ création d'un commit, résultat du merge :



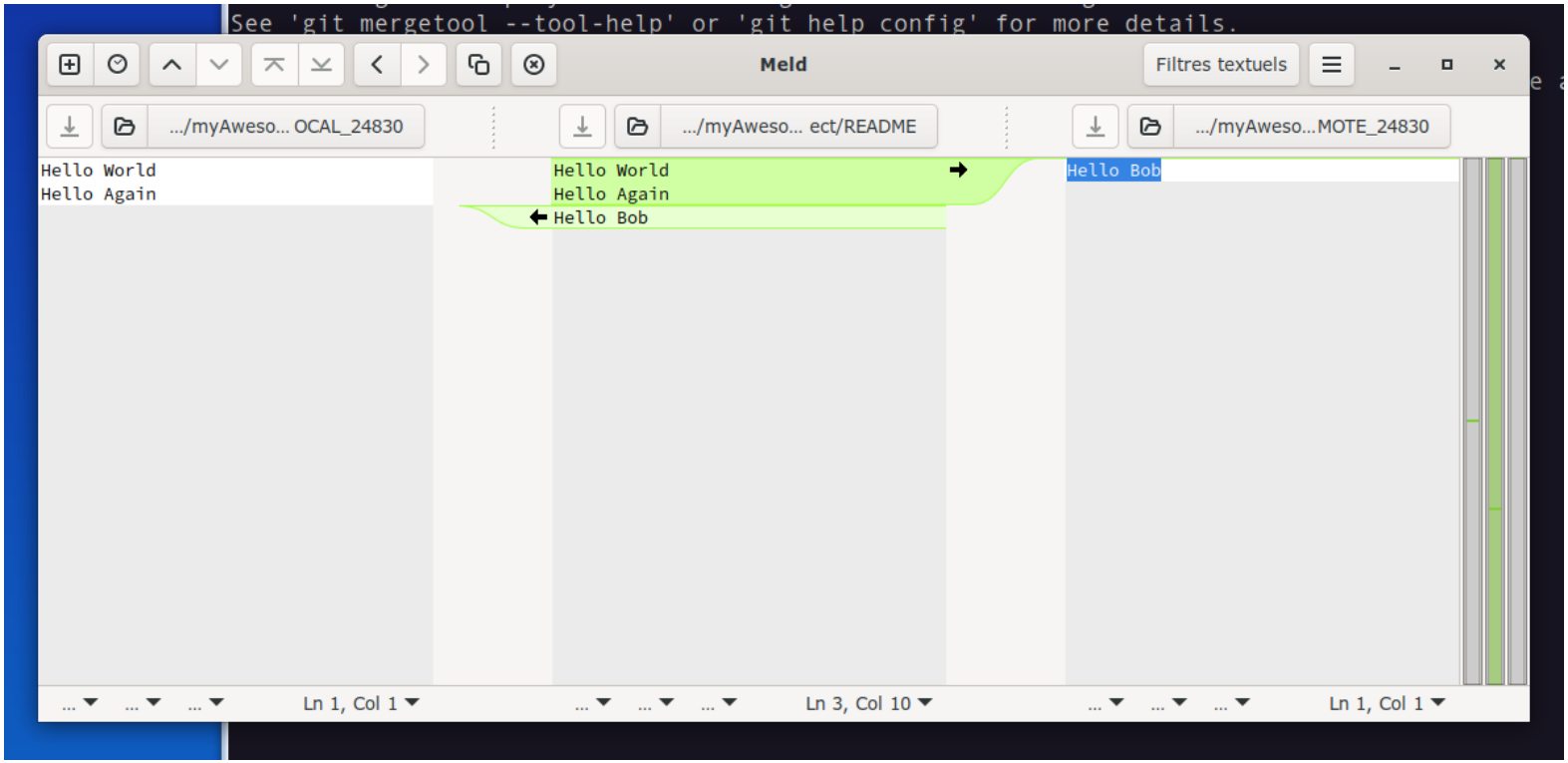
# Merge conflictuel

- git mergetool



# git config --global merge.tool meld

- git mergetool




```
$ git config --global mergetool.keepBackup false
```

⇒ supprime les fichiers créés pour merger, *e.g.* `REAME.origin`

# Gestion des branches

git branch

## Listing

```
$ git branch
  iss53
* master
  testing
```

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

# Gestion des branches

git branch

## Information sur les merges :

```
$ git branch --merged
  iss53
* master
```

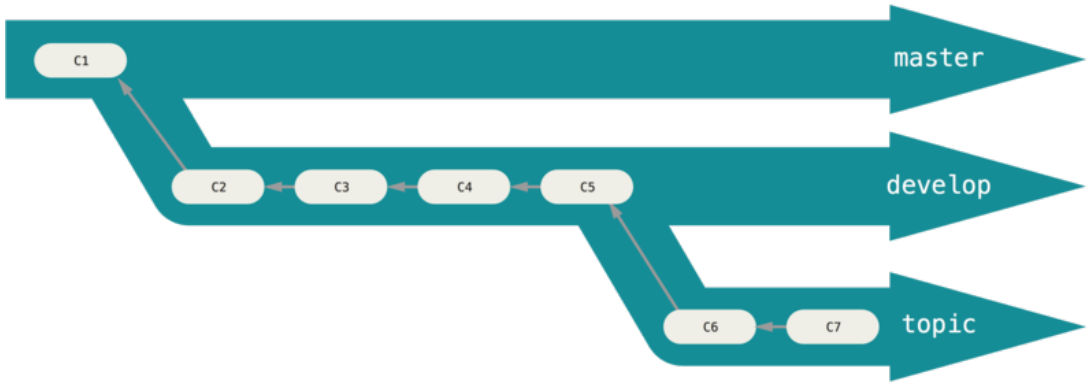
## On ne peut effacer directement une branche non mergée :

```
$ git branch --no-merged
  testing
```

This shows your other branch. Because it contains work that isn't merged in yet, trying to delete it with `git branch -d` will fail:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'
```

# Quelles branches créer et quand ?

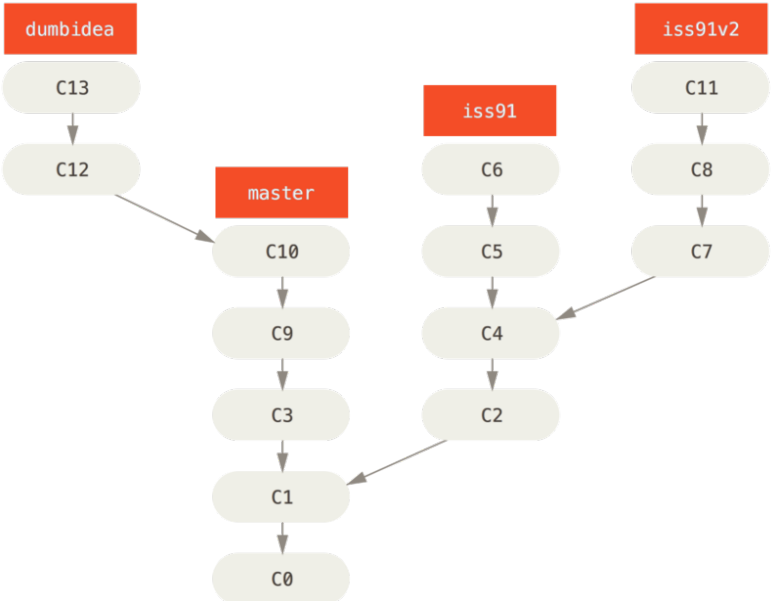


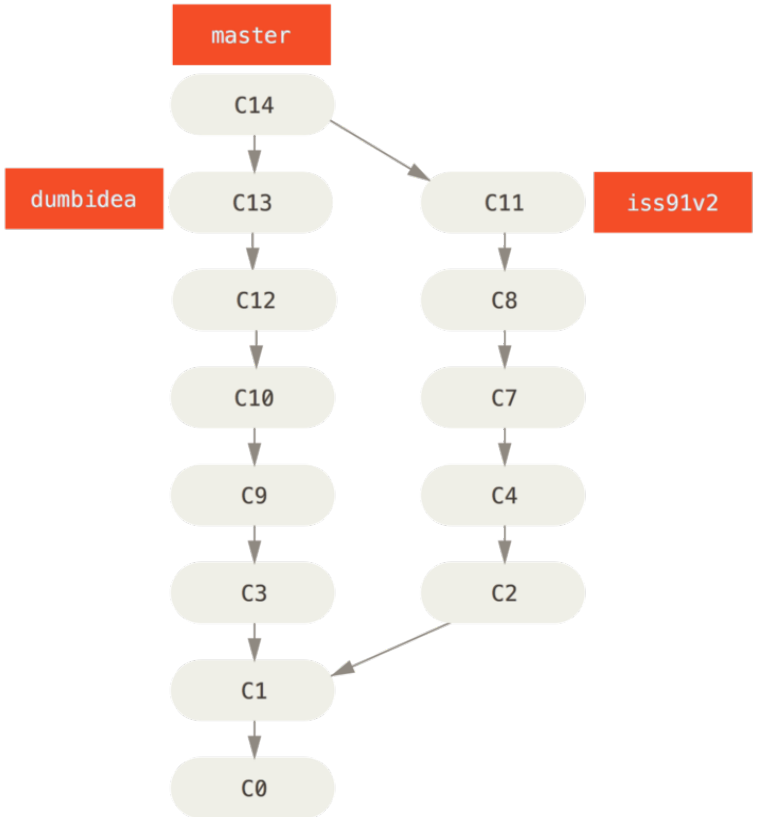

## Approche usuelle

- master ⇒ code en production
- develop ⇒ branche de test avant mise en prod
- topic ⇒ durée de vie courte : à tester avant merge

# En pratique

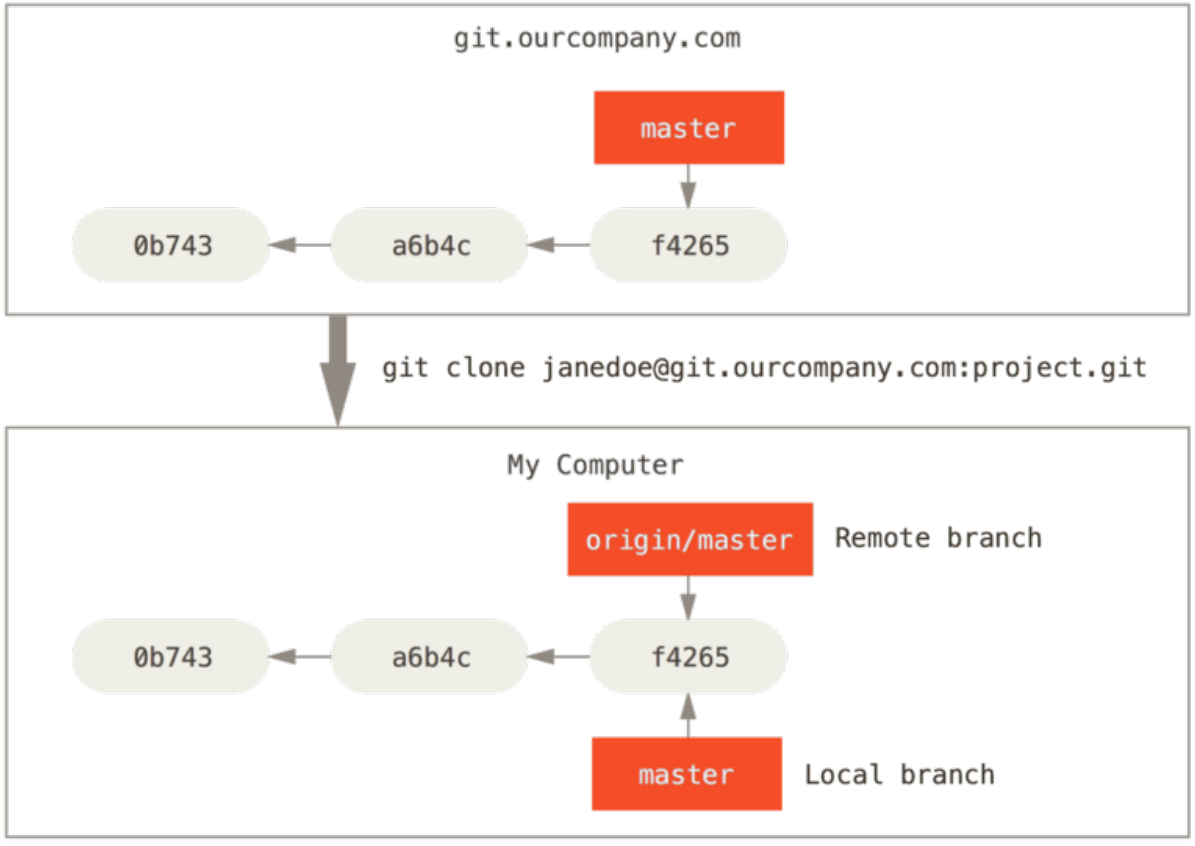
Avant et après merge de dumbidea et iss91v2 (*fast-forward* en premier) :



# Plan

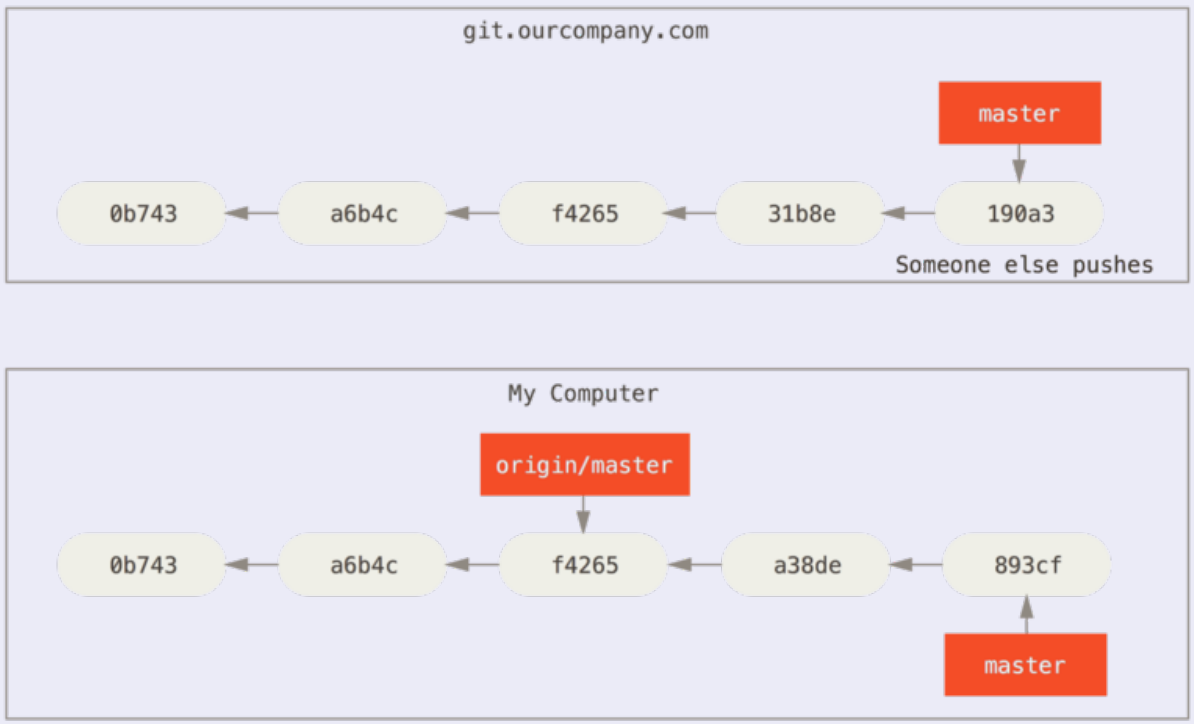
# Branches distantes



# Branches distantes

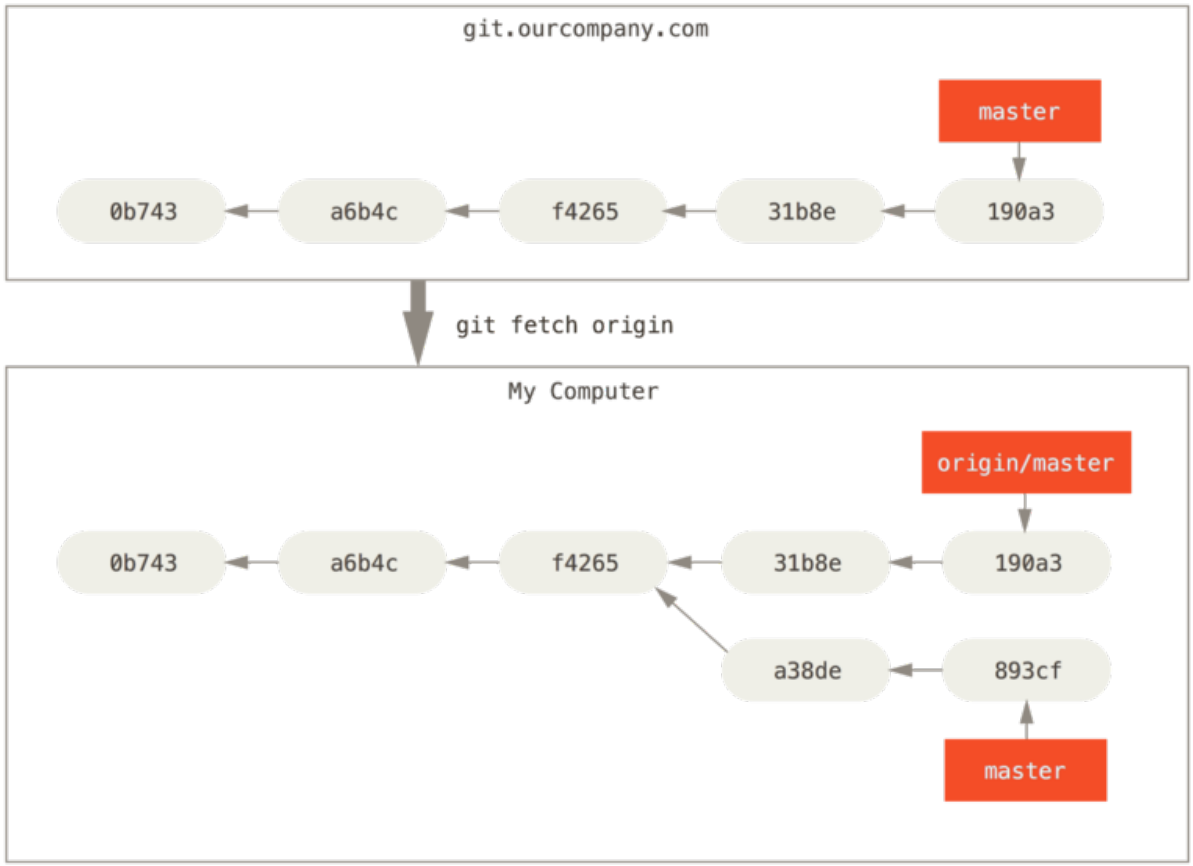
Les branches distantes n'ont absolument rien de spécial, à part qu'elles peuvent évoluer sans travail local !

## Exemple de divergence

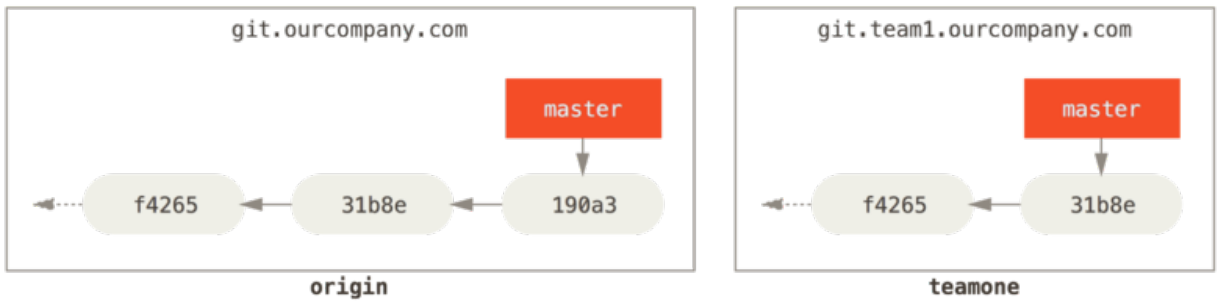


# Branches distantes

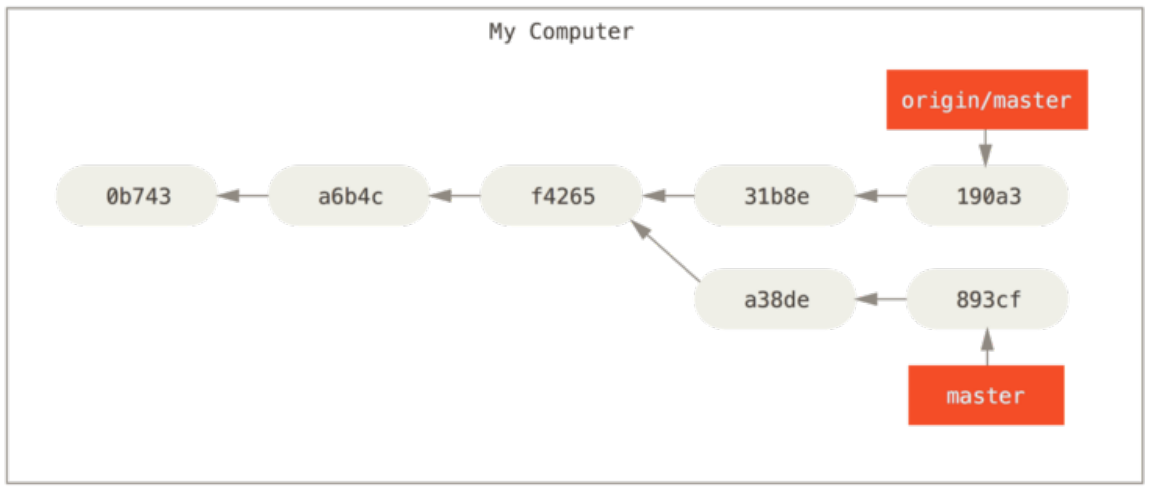
Récupération des modifications distantes ▸ git fetch



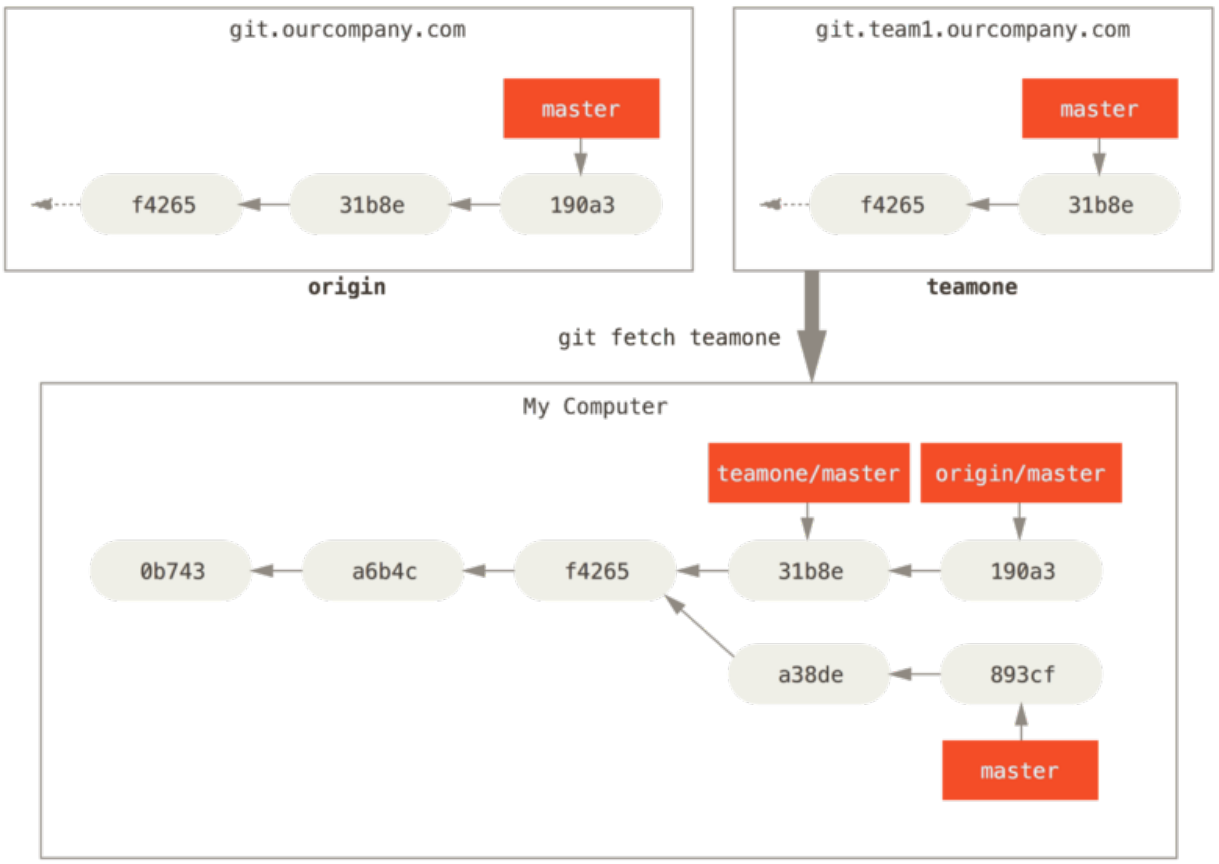
# Ajouter un remote additionnel




```
git remote add teamone git://git.team1.ourcompany.com
```



# Ajouter un remote additionnel



# Pushing

▸ git push

Partager une branche sur le remote :

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

# *tracking a branch*

▸ git checkout

Vos collaborateurs :

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

*tracking the branch*

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

# *tracking a branch* ▸ git checkout

Vos collaborateurs :

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

*tracking the branch*

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

*En une seule commande*

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

# Plan

# Rebasing

▸ git rebase

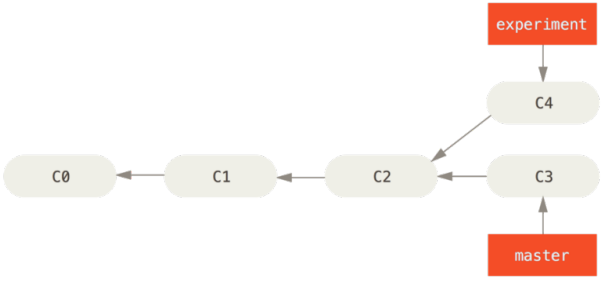
`git rebase` est un autre moyen d'intégrer des changements

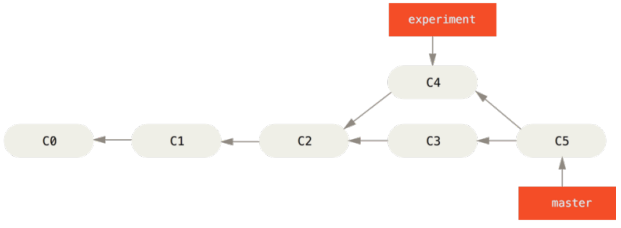
## Intérêt vs. `git merge`

- Les deux sont équivalents au niveau du contenu du commit final
- mais `git rebase` permet un **historique plus propre et une intégration simplifiée**
- ⇒ `git rebase` produit un historique linéaire
- ⇒ `git rebase` permet le fast-forward

# Rebasing

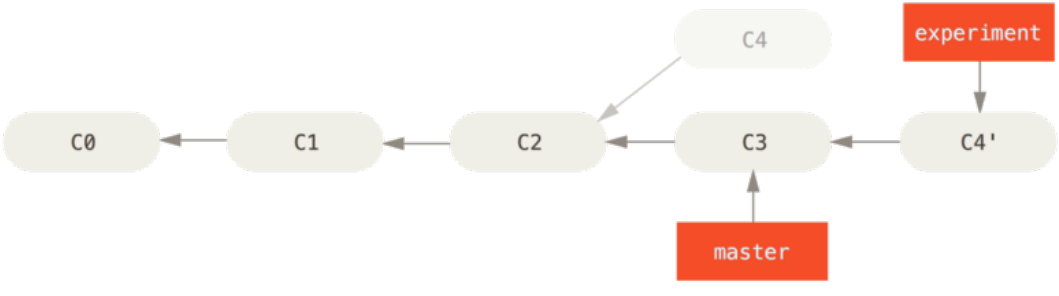
merge :



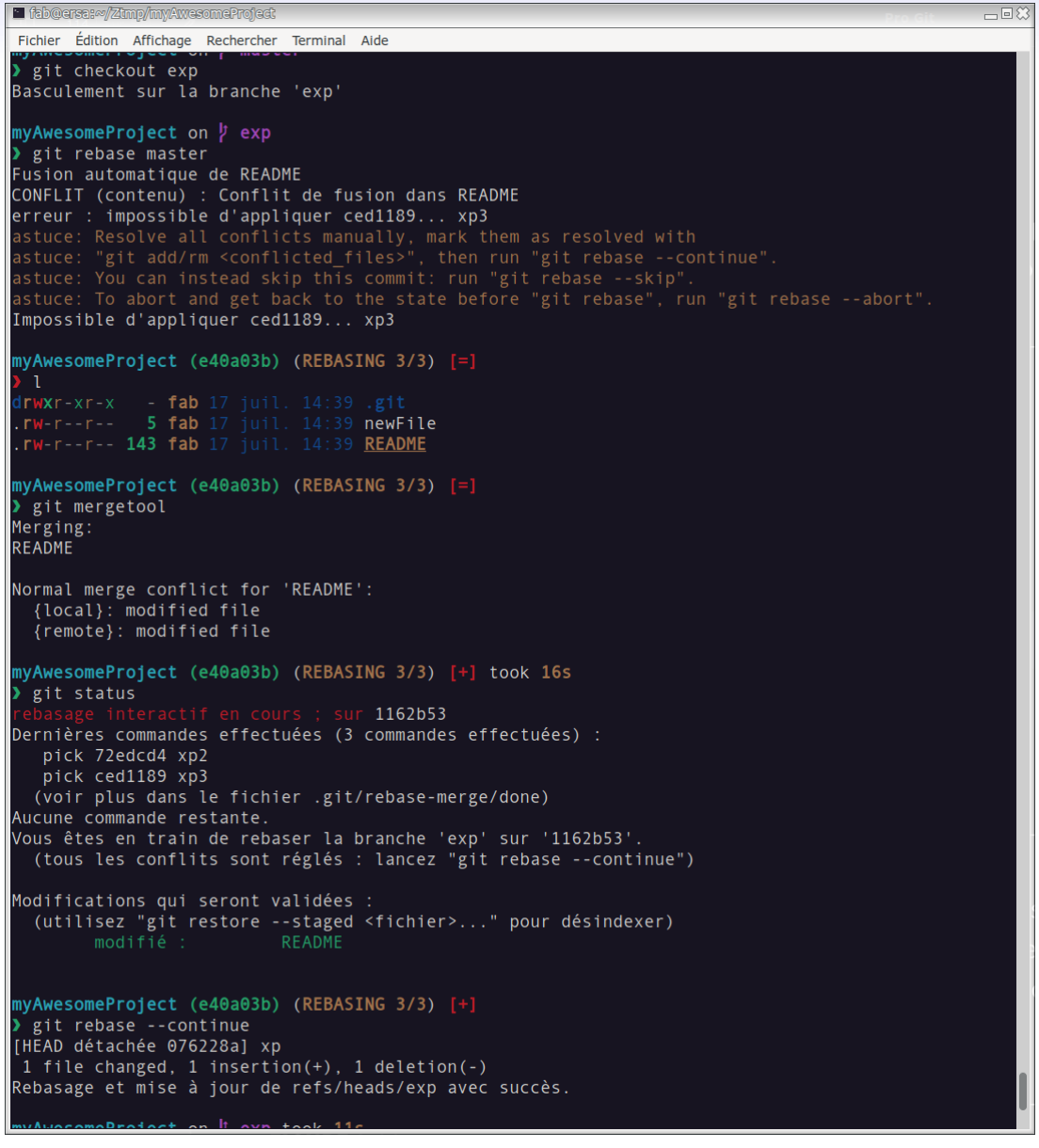




vs. `git rebase`

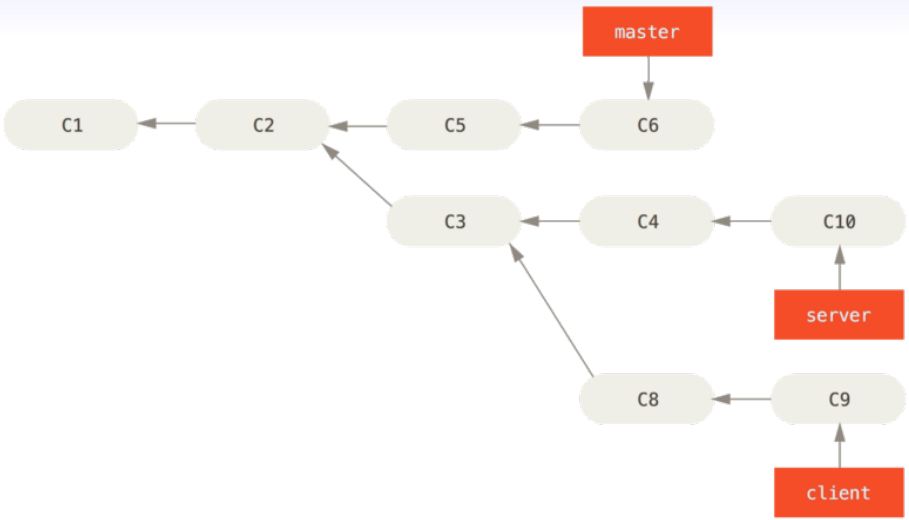
```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```



# Rebase avec conflit



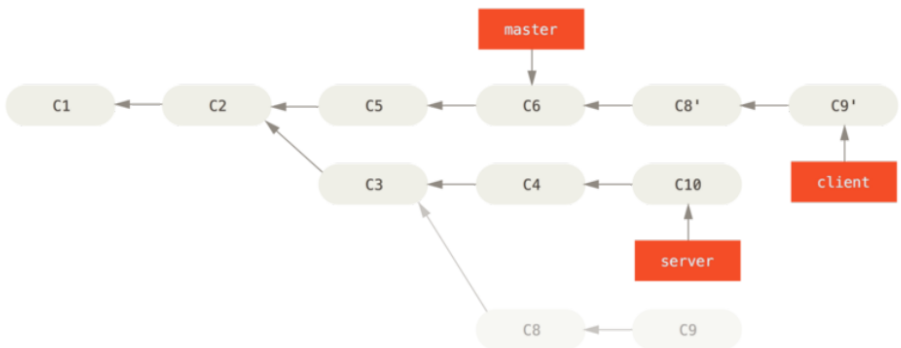
# Rebase plus complexe




```
$ git rebase --onto master server client
```

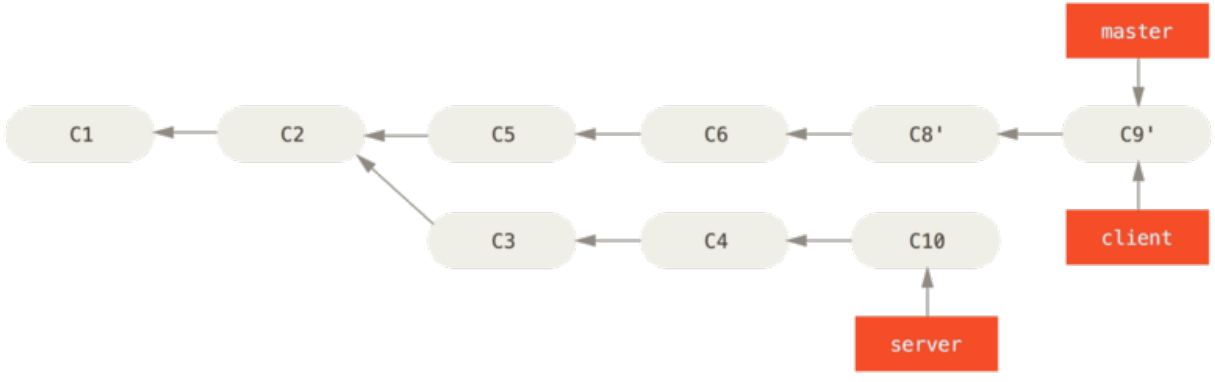
This basically says, “Check out the client branch, figure out the patches from the common ancestor of the `client` and `server` branches, and then replay them onto `master`.” It’s a bit complex, but the result is pretty cool.

---

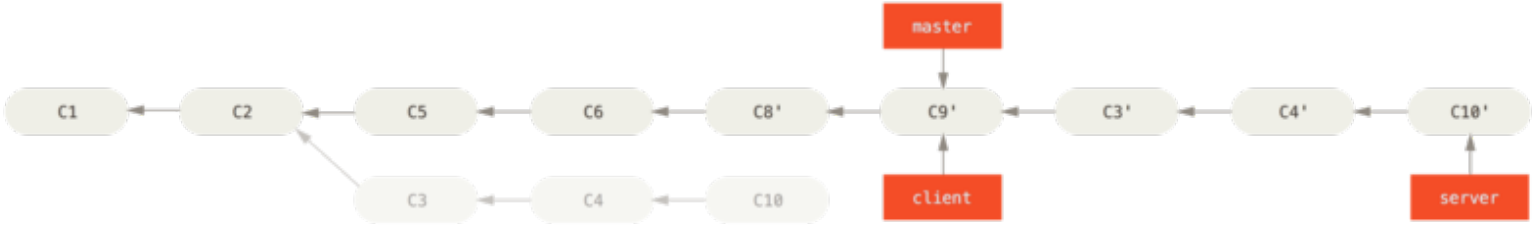


# Rebase plus complexe

```
$ git checkout master
$ git merge client
```




```
$ git rebase master server
```



# Rebase plus complexe

*Fast-forward* et nettoyage :

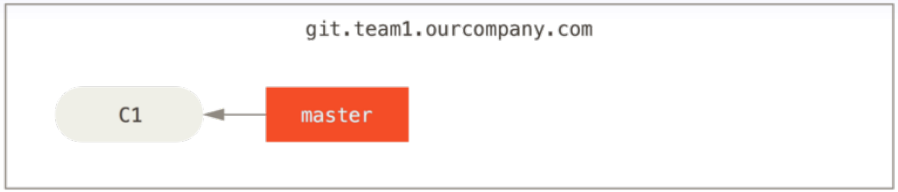
```
$ git checkout master
$ git merge server
$ git branch -d client
$ git branch -d server
```

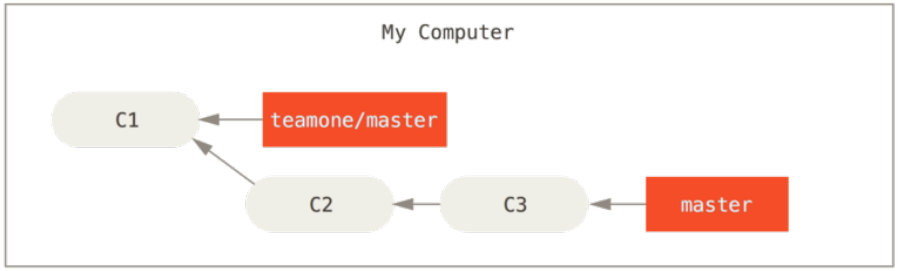

Avec merge, on garde l'historique de la divergence, même après suppression de la branche

# !! ATTENTION !!

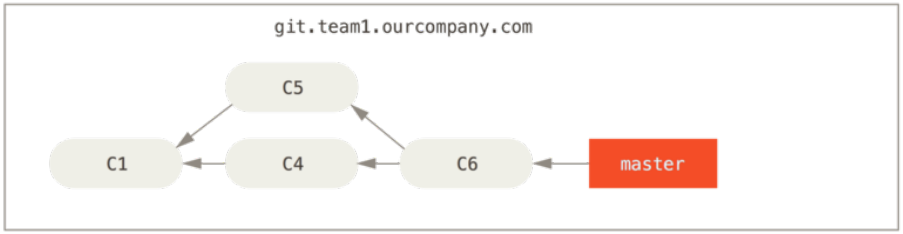
**NE JAMAIS REBASER LORSQUE DES COMMITS PUSHÉS SUR UN REMOTE SONT IMPLIQUÉS !!**

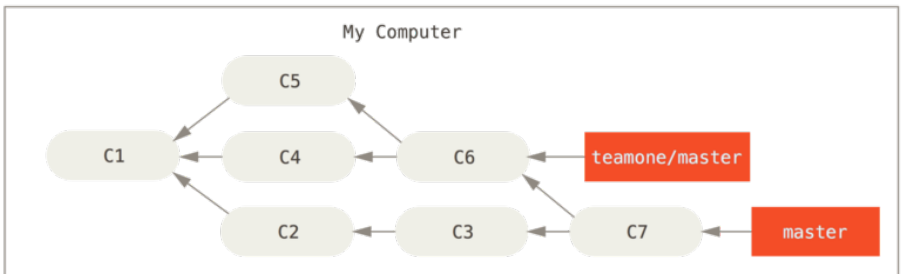
# Exemple de problème



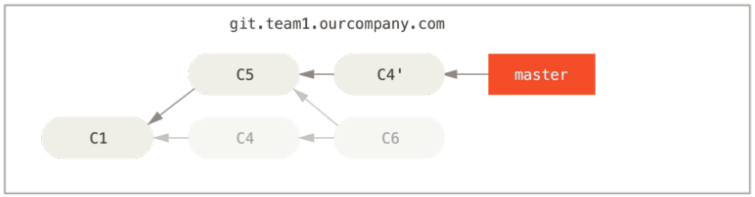

Travail après clonage :

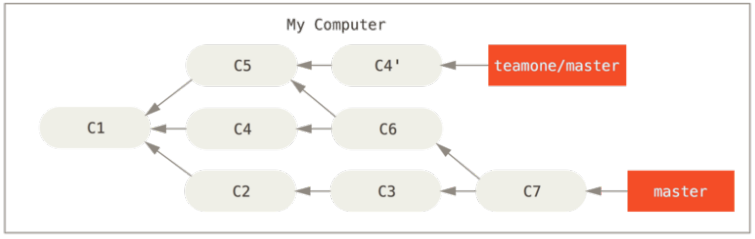


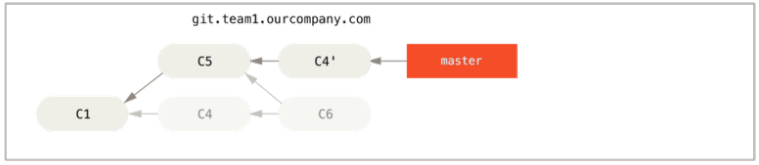
# Exemple de problème

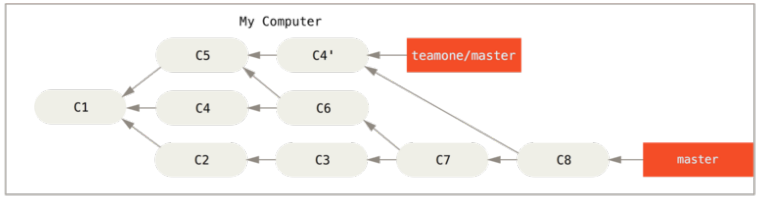
Quelqu'un push après un rebase local. . . Après un `fetch`, c'est la cata chez vous !



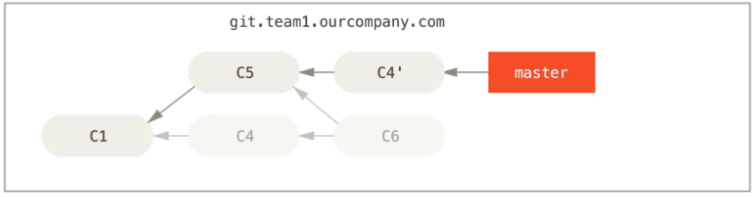

Obliger de merger à nouveau. . .

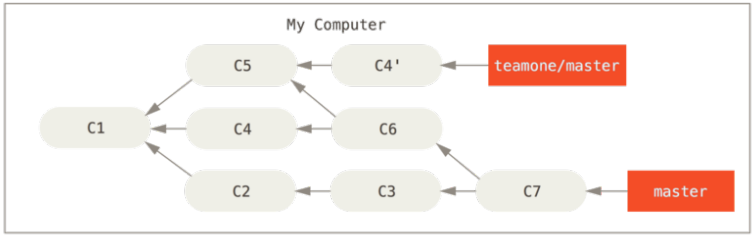


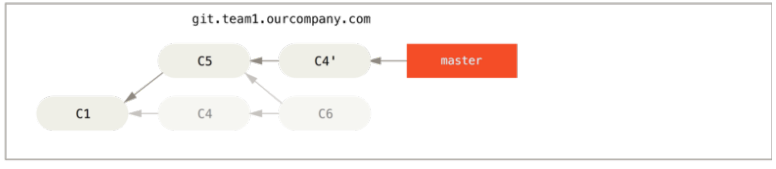
# Exemple de problème

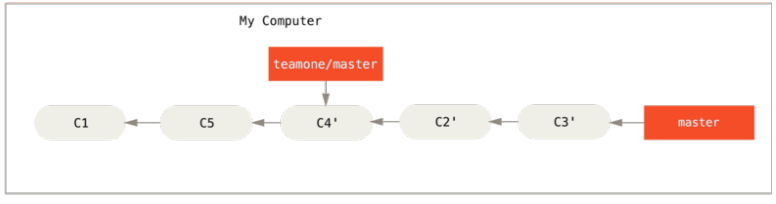
Cependant, on peut faire mieux.




```
$ git rebase teamone/master
```
ou
```
$ git pull --rebase
```

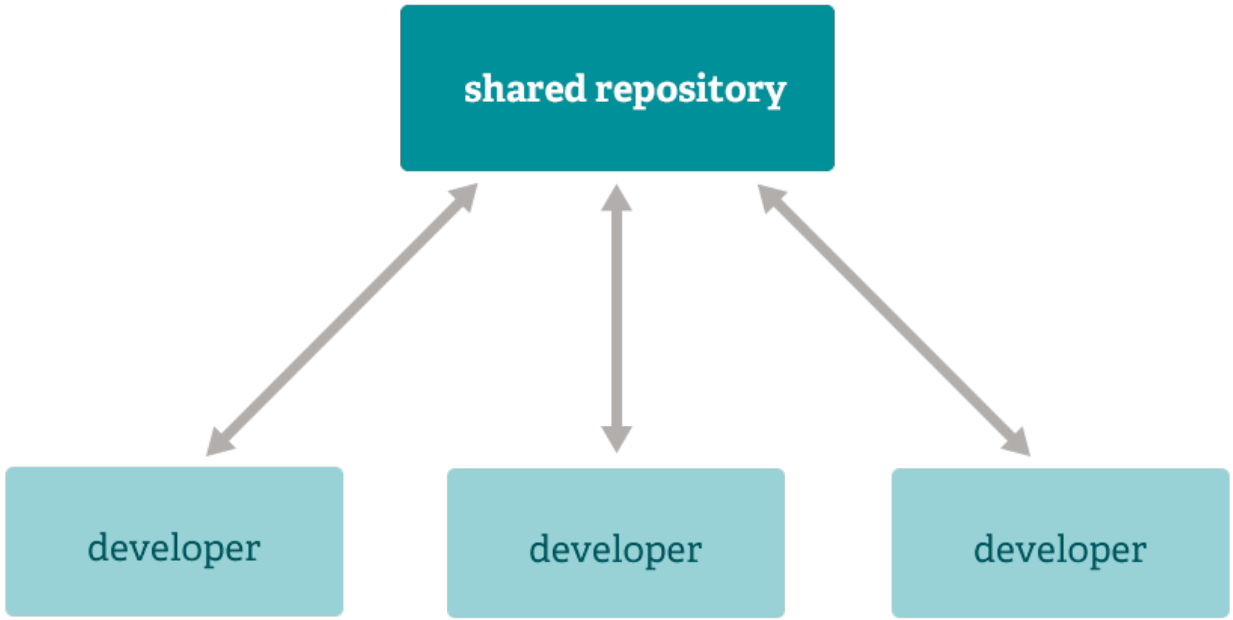


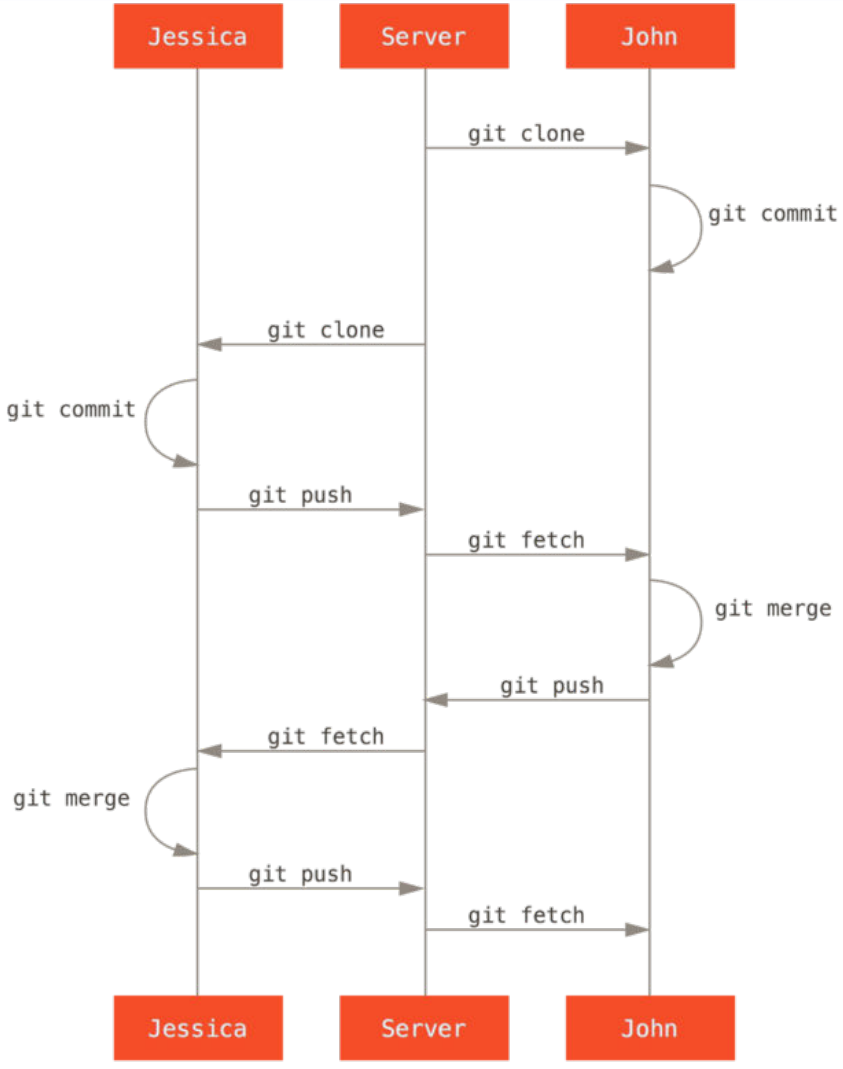
# Plan

# Centralized Workflow

Suffisant pour une petite équipe

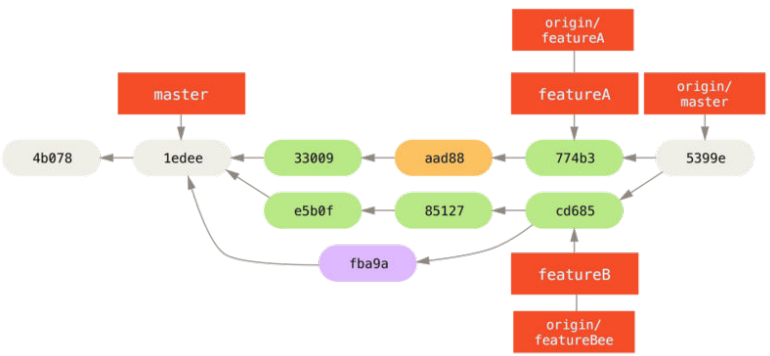


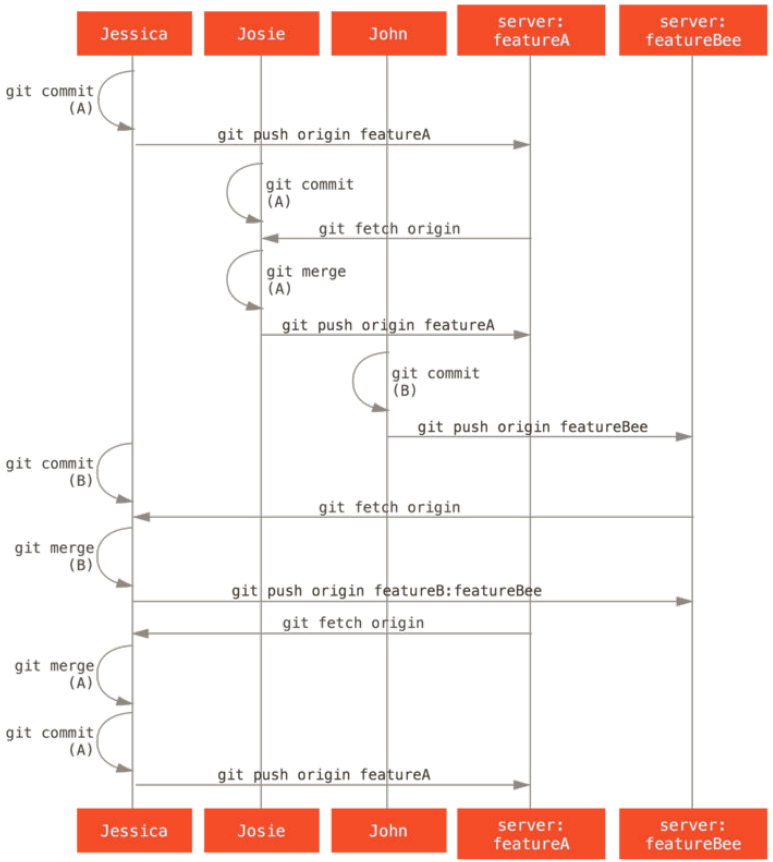
# Centralized Workflow



# Pour une petite équipe

Codage sur différentes branches sans se gêner

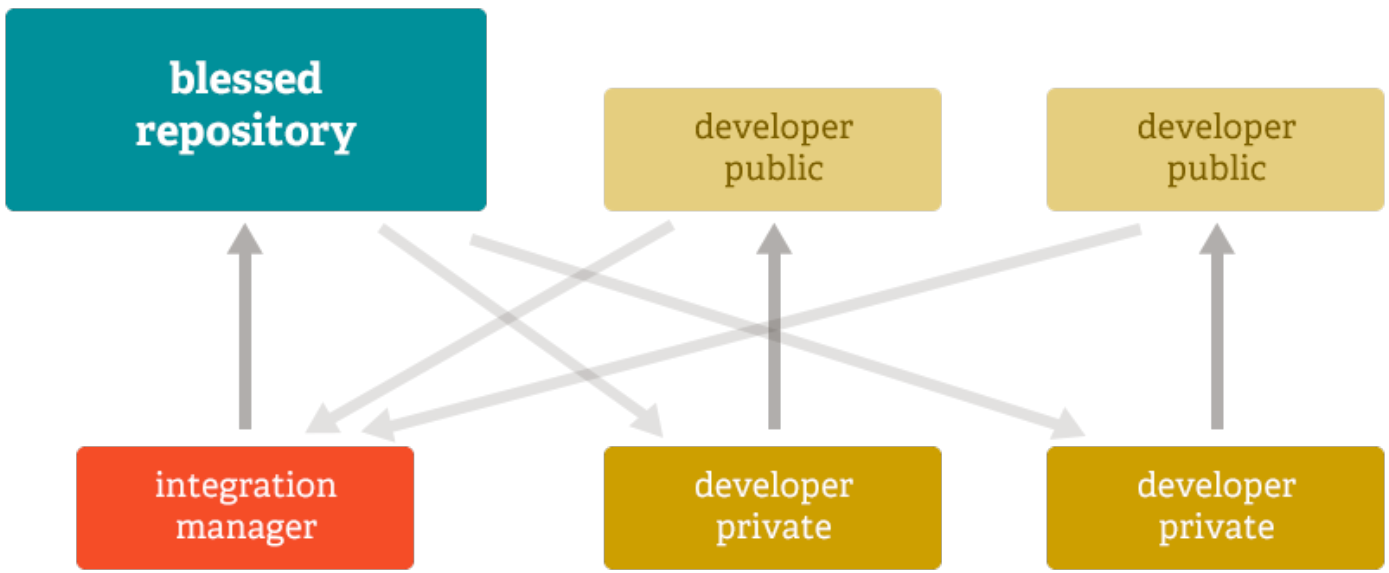


# Integration Manager Workflow

## Workflow usuel sur GitHub, GitLab,...

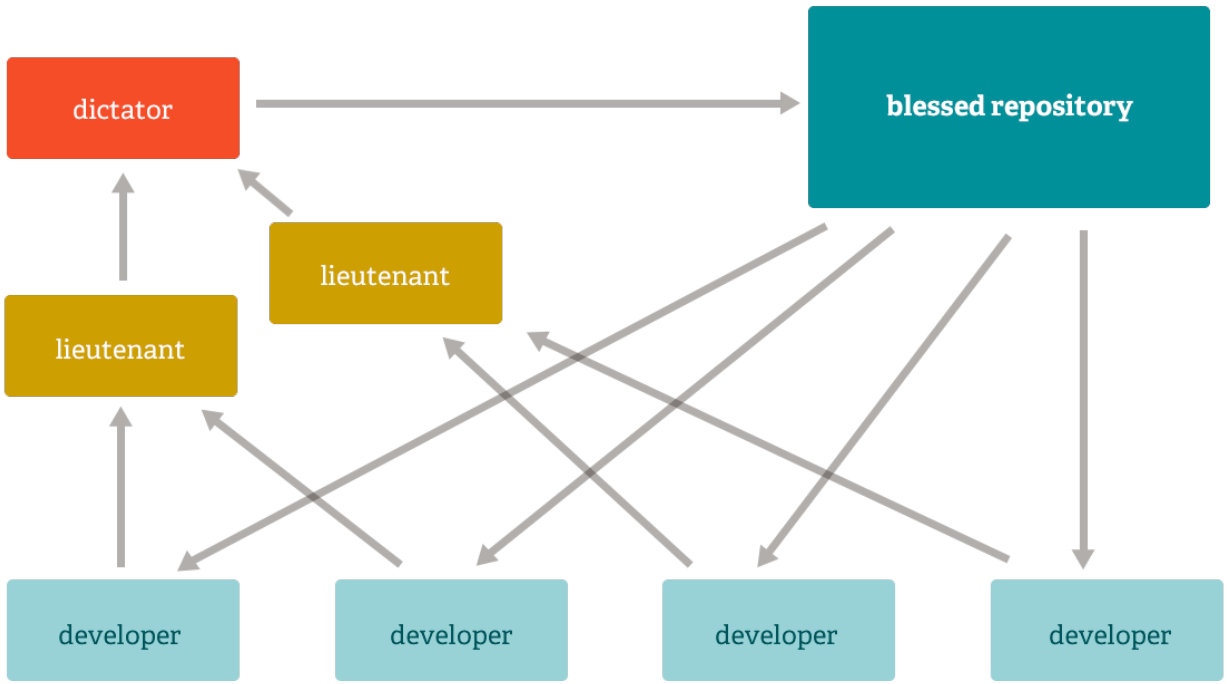
1. The project maintainer pushes to their public repository.
2. A contributor clones that repository and makes changes.
3. The contributor pushes to their own public copy.
4. The contributor sends the maintainer an e-mail asking them to pull changes.
5. The maintainer adds the contributor's repo as a remote and merges locally.
6. The maintainer pushes merged changes to the main repository.



# Dictator and Lieutenants Workflow

## Workflow pour les très gros projet, *e.g.* Linux

1. The project maintainer pushes to their public repository.
2. Regular developers work on their topic branch and rebase their work on top of master. The master branch is that of the dictator.
3. Lieutenants merge the developers' topic branches into their master branch.
4. The dictator merges the lieutenants' master branches into the dictator's master branch.
5. The dictator pushes their master to the reference repository so the other developers can rebase on it.



# Contribuer à un projet externe : *fork*

```
$ git clone (url)
$ cd project
$ git checkout -b featureA
# (work)
$ git commit
# (work)
$ git commit
```

url : fork créé avec le gestionnaire distant (GitHub, GitLab, ...)

```
$ git remote add myfork (url)
```

```
$ git push -u myfork featureA
```

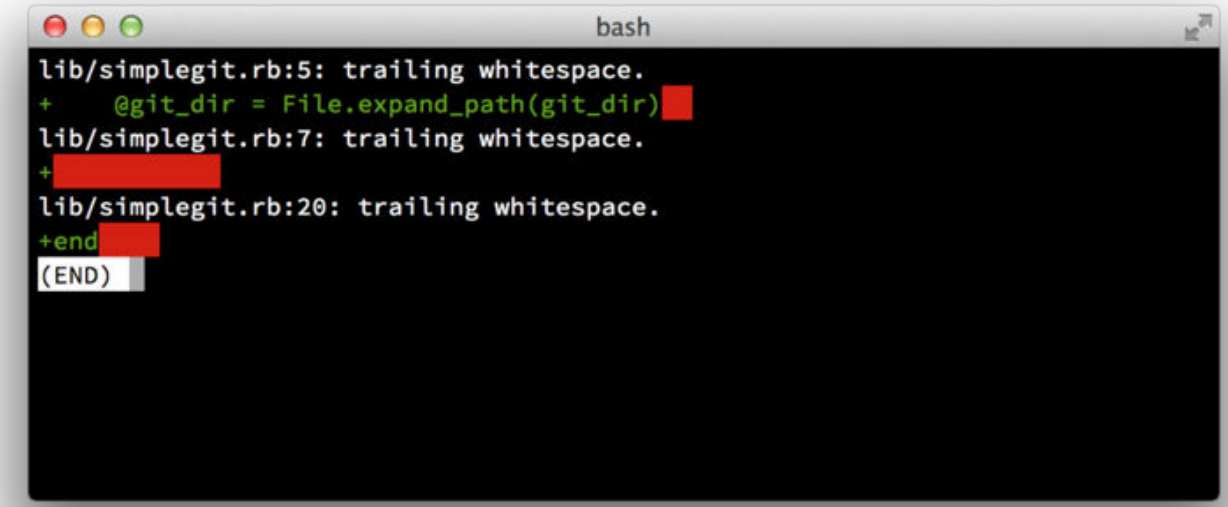
```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d708(
  John Smith (1):
        added a new function

are available in the git repository at:

  git://githost/simplegit.git featureA

Jessica Smith (2):
      add limit to log function
      change log output to 30 from 25

 lib/simplegit.rb |   10 +++++++++-
 1 files changed, 9 insertions(+), 1 deletions(-)
```

En pratique on crée une *pull/merge request* depuis le site de gestion de dépôt

# Contribuer à un projet externe : *fork*

Astuce : vérifier s'il n'y a pas d'espaces inutiles :

```
$ git diff --check
```



# Plan

# Something's wrong!

```
* 0f8727e (HEAD -> master) fix issue 834
* 505d343 fix issue 434
* b56187a fix issue 431
| * 7bae2af (feature_A) Forgot testing... Now it really works
| * 2e6ccc8 At last it works!
| * c713155 fix a typo
| * a9d0d0d End of the day: Backup! :)
| * b08c8a9 Have to do a hot fix on master... And I do not know how to stash
| * b4c2620 looks nicer
| * 78371cf not working anymore... need sleeping
| * c5230ff it works a bit
| * 324c4f9 go eat something
| * 97f7733 write some nice code
|/
* 6fc2753 first commit
```

# Soigner votre historique

**Please!**

- Un commit est un snapshot stable dans la mesure du possible : il faut tester d'abord
- L'historique doit être lisible : les commentaires sont importants !
- Le moins de commits, meilleur l'historique
- Un commit est peu coûteux, mais beaucoup. . . : réfléchissez à deux fois

**Git n'est pas un logiciel de backup**

# $ git commit --amend

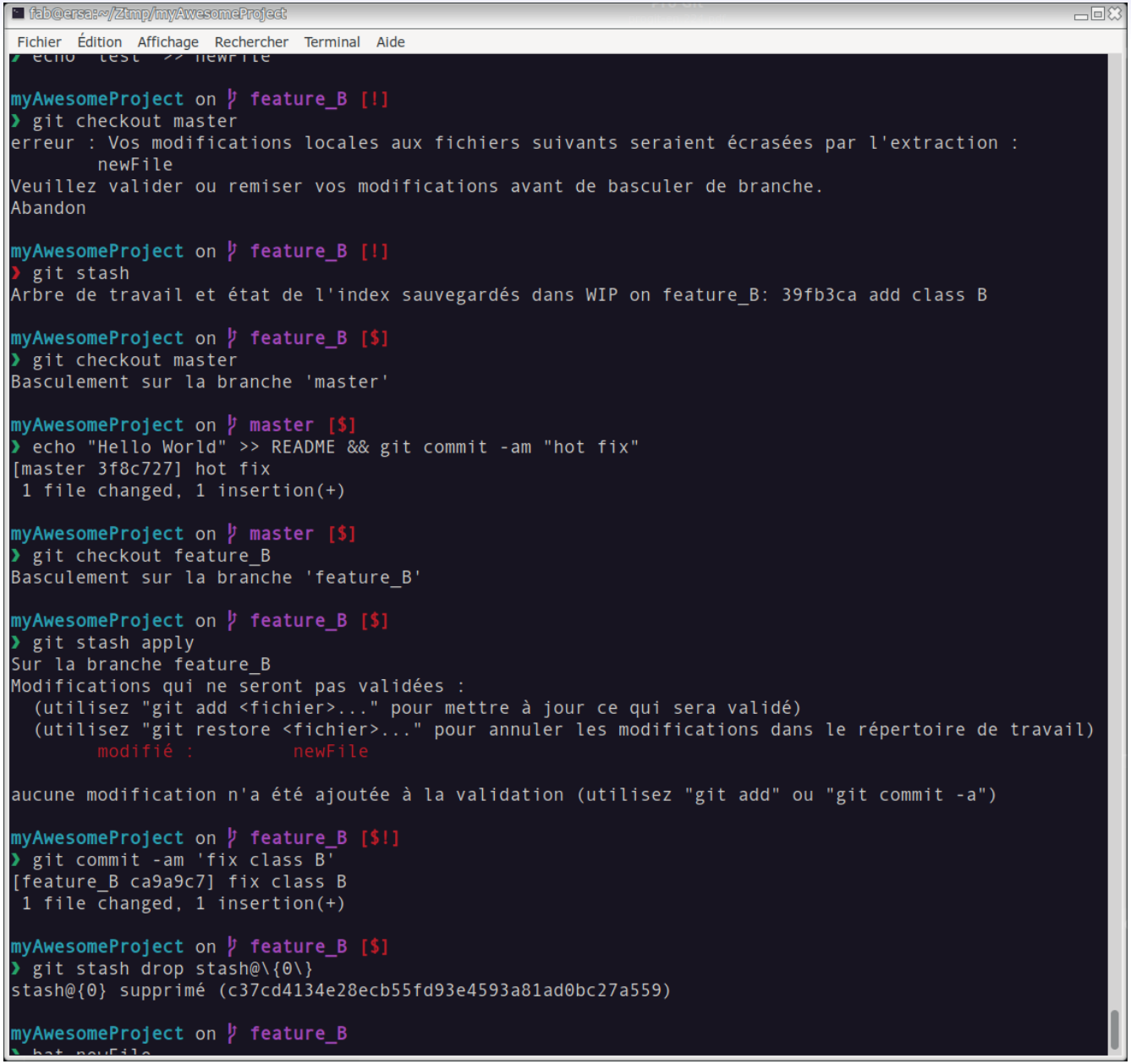
C'est votre ami : il écrase le dernier commit

## *Use cases*

- Vous avez raté votre message de commit
- Vous avez oublié des fichiers
- Vous n'avez pas vérifier les caractères vides
- Vous n'avez pas testé...
- Vous aviez ajouté des fichiers non désirés
- ...

**À ne pas utiliser sur un commit pushé !**

# Changer de branche sans commiter : git stash



# Récrire l'histoire avec ▸ git rebase -i

Do you remember?

```
* 9300a78 (master) fix issue em-666
* 4b72e2e fix issue 434
* 06c4135 fix issue 834
| * 4e0c042 (HEAD -> featureA) Ooops
| * 821f0cd Forgot testing... Now it really works
| * 6e8187a At last it works!
| * 3e4c381 fix a typo
| * f5b93f1 End of the day: Backup! :)
| * 8b57b90 looks nicer
| * dae337d not working anymore... need sleeping
| * 9d0a436 it works a bit
| * a419926 go eat something
| * f8d5132 write some nice code
|/
* a2eb407 first commit
```

# Récrire l'histoire avec ▸ git rebase -i

`$ git rebase -i HEAD~10` ouvre l'éditeur :

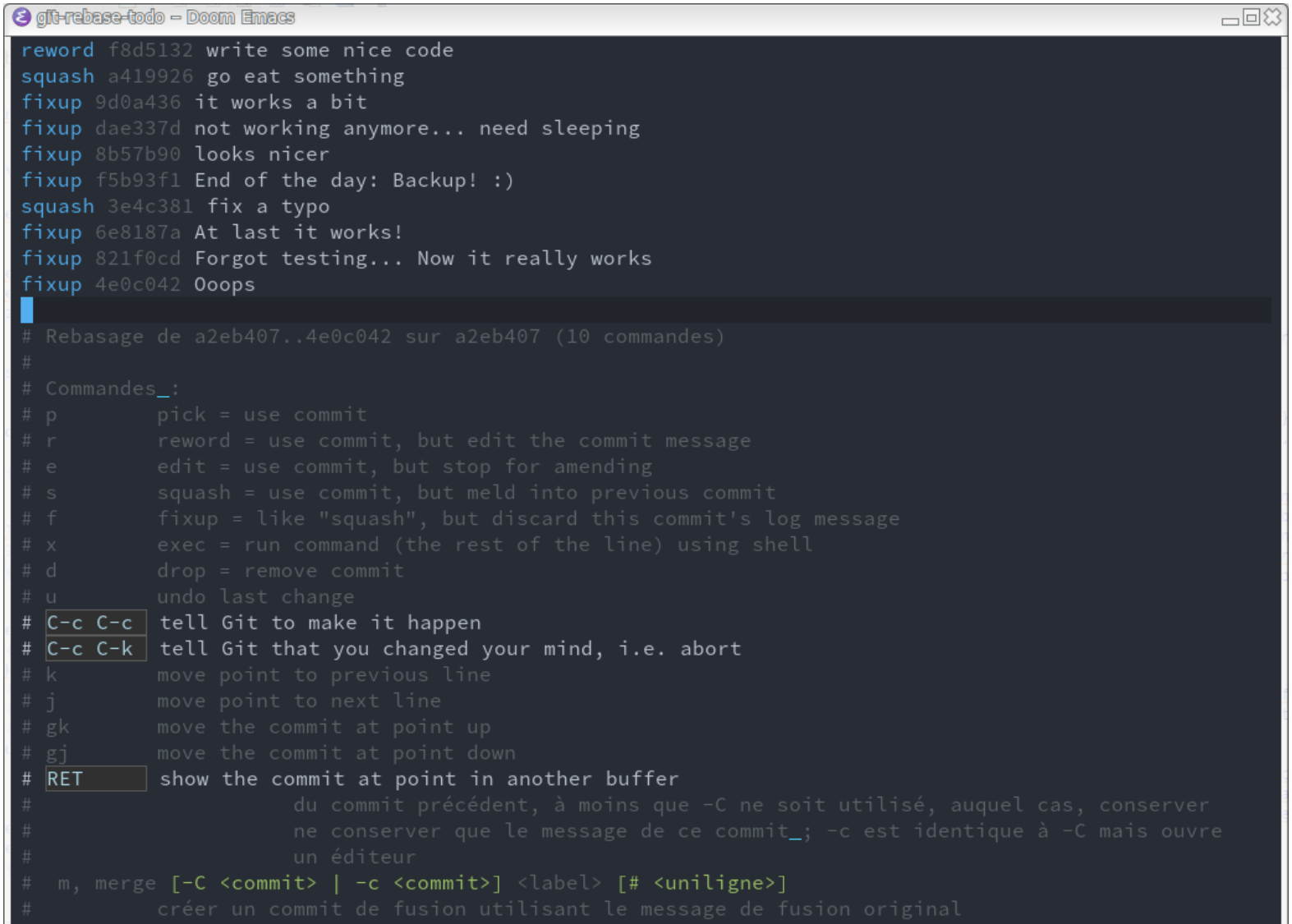
git-rebase-todo – Doom Emacs

```
pick f8d5132 write some nice code
pick a419926 go eat something
pick 9d0a436 it works a bit
pick dae337d not working anymore... need sleeping
pick 8b57b90 looks nicer
pick f5b93f1 End of the day: Backup! :)
pick 3e4c381 fix a typo
pick 6e8187a At last it works!
pick 821f0cd Forgot testing... Now it really works
pick 4e0c042 Ooops

# Rebasage de a2eb407..4e0c042 sur a2eb407 (10 commandes)
#
# Commandes :
# p         pick = use commit
# r         reword = use commit, but edit the commit message
# e         edit = use commit, but stop for amending
# s         squash = use commit, but meld into previous commit
# f         fixup = like "squash", but discard this commit's log message
# x         exec = run command (the rest of the line) using shell
# d         drop = remove commit
# u         undo last change
# C-c C-c   tell Git to make it happen
# C-c C-k   tell Git that you changed your mind, i.e. abort
# k         move point to previous line
# j         move point to next line
# gk        move the commit at point up
# gj        move the commit at point down
# RET       show the commit at point in another buffer
#                     du commit précédent, à moins que -C ne soit utilisé, auquel cas, conserver
#                     ne conserver que le message de ce commit ; -c est identique à -C mais ouvre
#                     un éditeur
#  m, merge [-C <commit> | -c <commit>] <label> [# <uniligne>]
#          créer un commit de fusion utilisant le message de fusion original
#          (ou l'uniligne, si aucun commit de fusion n'a été spécifié).
#          Utilisez -c <commit> pour reformuler le message de validation.
#                       à cette position dans les nouveaux commits. La <réf> est
#                       mise à jour à la fin du rebasage
#
# Vous pouvez réordonner ces lignes ; elles sont exécutées de haut en bas.
#
# Si vous éliminez une ligne ici, LE COMMIT CORRESPONDANT SERA PERDU.
#
# Cependant, si vous effacez tout, le rebasage sera annulé.
```

2.0k myAwesomeProject/.git/rebase-merge/git-rebase-todo 1:0 All Git Rebase (+1)

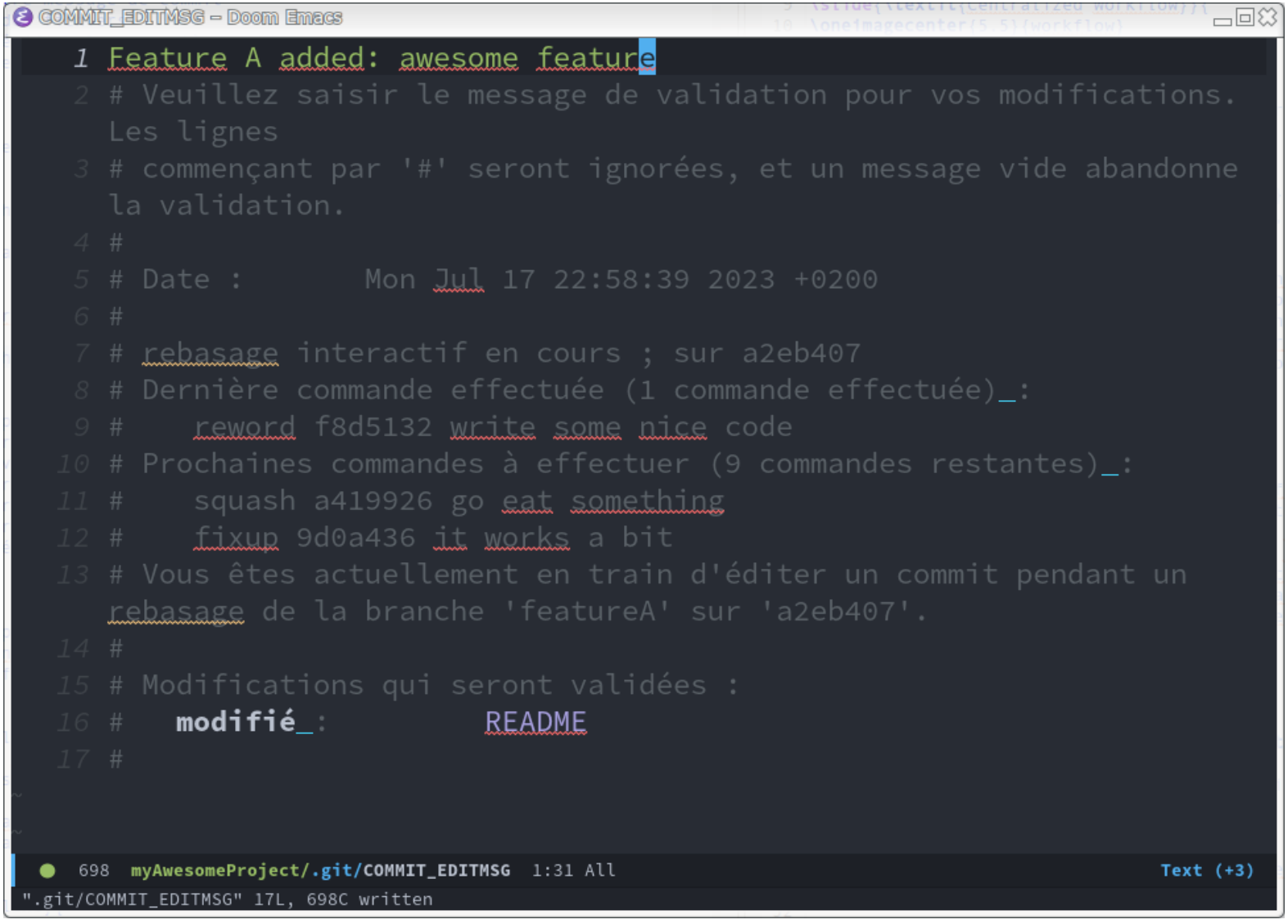
# Récrire l'histoire avec ▸ git rebase -i

`pick, reword, edit, squash, fixup, exec, drop, undo :`



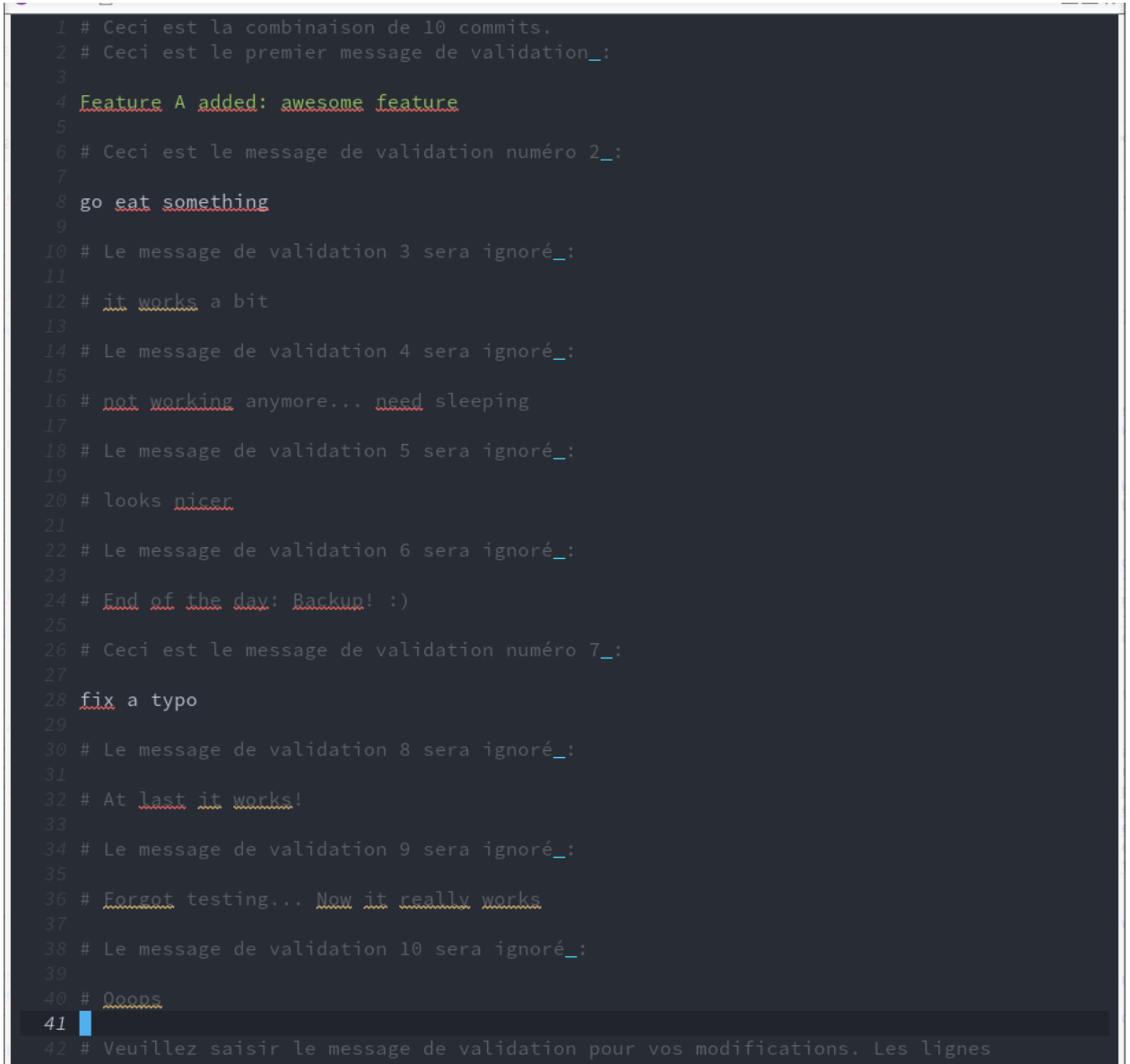
# Récrire l'histoire avec ▸ git rebase -i

Édition du message du commit f8d5132 :

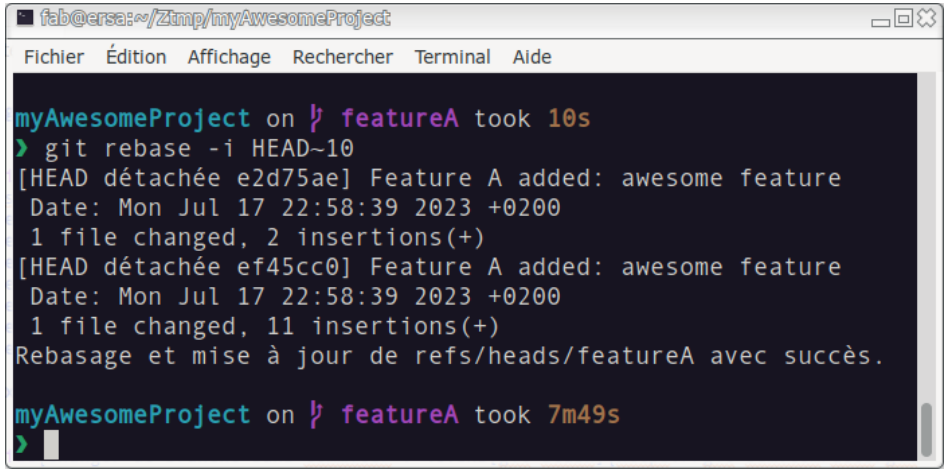


# Récrire l'histoire avec ▸ git rebase -i

Édition du message pour le commit résultat :



# Récrire l'histoire avec ▸ git rebase -i




⇒