

# Formation Git

Cette formation vous est fournie sous licence Creative Commons AttributionNonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0)

**Vous êtes libres de :**

- Copier, distribuer et communiquer le matériel par tous moyens et sous tous formats

**Selon les conditions suivantes :**

– Attribution : Vous devez créditer l'Oeuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'Oeuvre. Vous devez indiquer ces informations par tous les moyens possibles mais vous ne pouvez pas suggérer que l'Offrant vous soutient ou soutient la façon dont vous avez utilisé son Oeuvre.

– Pas d'Utilisation Commerciale: Vous n'êtes pas autoriser à faire un usage commercial de cette Oeuvre, tout ou partie du matériel la composant.

– Pas de modifications: Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant l'Oeuvre originale, vous n'êtes pas autorisé à distribuer ou mettre à disposition l'Oeuvre modifiée.



- **Introduction**
- **Installation et configuration**
- **Git avec un dépôt local**
  - Premiers pas
  - Branches
  - Checkout / Reset / Tag
  - Reflog
  - Merge et rebase
- **Git avec un dépôt distant**
  - Repository distant
  - Branches distantes
- **Commandes diverses**
- **Scénarios classiques**

# Introduction

# Ancêtres

- **GNU RCS** (Revision Control System) et `diff` : 1982
  - un fichier (source, binaire) à la fois
- **SCCS** (Source Code Control System) : 1986-89
- **CVS** (Concurrent Versions System) : 1990
  - client-serveur
  - CLI & GUI
- **SVN** (Apache Subversion) : 2000
  - *commits atomiques*
  - renommage et déplacement sans perte d'historique
  - prise en charge des répertoires et de méta-données
  - numéros de révision uniques sur tout le dépôt
  - *NB: il est possible d'utiliser Git avec un dépôt SVN via Git-SVN)*

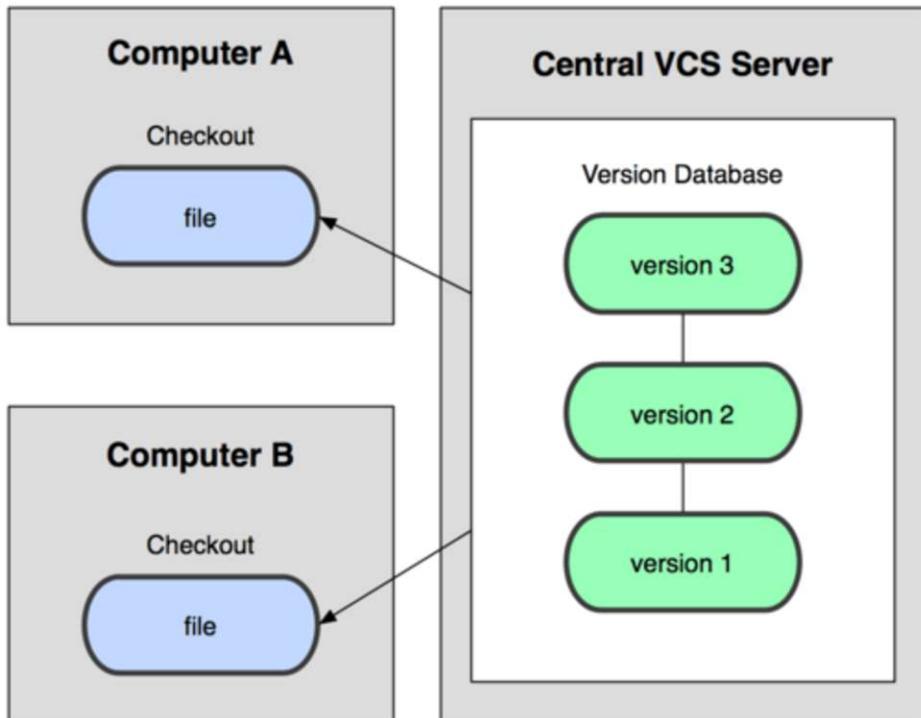
# Historique

- Créé en avril 2005 par Linus Torvalds
- Objectif : gérer le *workflow* d'intégration des *patches* du noyau Linux
- Remplacement de BitKeeper
- En Mai 2013, 36% des professionnels utilisent Git en tant que VCS principal (source : Eclipse Foundation)
- En Avril 2013 Github déclare avoir 3.5 millions d'utilisateurs

# Rappel VCS

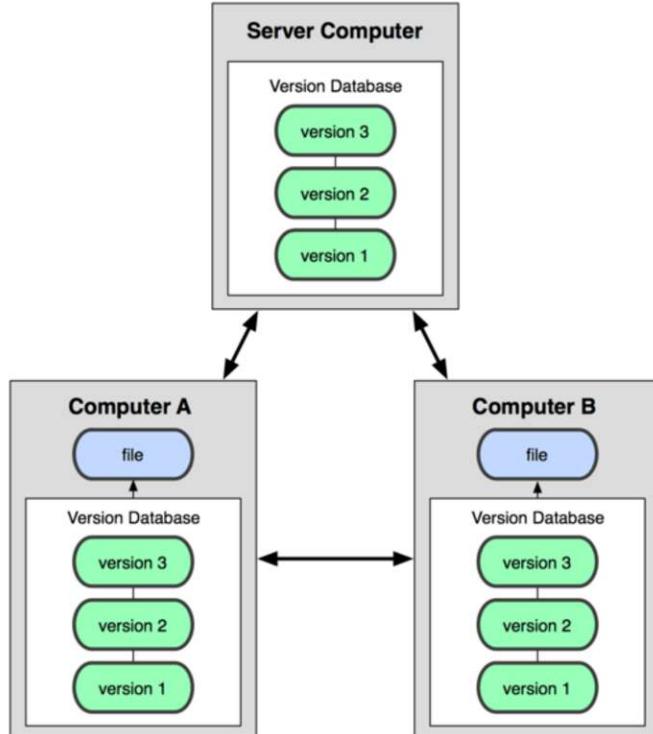
- VCS == Version Control System
- Gestion des versions et historiques de fichiers
- Gestion des branches
- Gestion des *tags*
- Gestion des conflits / *merges*

# VCS centralisé (CVS, SVN...)



- Centralisé == *repository* (dépôt) central
- On “emprunte” et on travaille sur des *working copies* (copies de travail)

# VCS distribué (Git, Mercurial...)



- Décentralisé : Les versions / branches / *tags* sont en local
- On travaille sur son *repository* local et on publie sur les autres *repositories*
- Possibilité d'avoir un *repository* central (mais pas obligé)

# Git a pour objectif :

- D'être **rapide**
- D'avoir une architecture **simple**
- De faciliter le développement parallèle (branches, *merges*...)
- D'être complètement **distribué**
- De gérer des projets de taille importante (Gnome, KDE, XORG, PostgreSQL, Android...)

# Git avec un dépôt local

# Installation et Configuration

## Installation :

- Sous Linux : via le gestionnaire de paquet  
(ex: apt-get install git)
- Sous OSX : via homebrew (brew install git)
- Sous Windows : via msysgit (<http://msysgit.github.com/>)

## Clients graphique :

- De nombreux clients graphiques et outils de *merge* sont disponibles sur chaque OS parmi lesquels :
  - Sous linux : gitg, git gui, p4merge ...
  - Sous OSX : gitx-dev , p4merge ...
  - Sous windows : git extensions, p4merge

# Configuration:

- La configuration globale de Git est située dans `~/.gitconfig`
- La configuration propre à chaque *repository* Git est située dans `<repository>/.git/config`
- A minima, il faut configurer son nom d'utilisateur et son adresse *email* (informations qui apparaîtront dans chaque commit) :
  - `git config --global user.name "John Doe"`
  - `git config --global user.email johndoe@example.com`

# Premiers Pas

## Création d'un dépôt et commits

# Définitions

- *Commit* : ensemble cohérent de modifications
- *Repository* : ensemble des *commits* du projet (et les branches, les *tags* (ou libellés), ...)
- *Working copy* (ou copie de travail) : contient les modifications en cours (c'est le répertoire courant)
- *Staging area* (ou index) : liste des modifications effectuées dans la *working copy* qu'on veut inclure dans le prochain *commit*

# Configuration

- `git config --global user.name "mon nom"` : configuration du nom de l'utilisateur (**inclus dans chaque commit**)
- `git config --global user.email "mon email"` : configuration de l'*email* de l'utilisateur (**inclus dans chaque commit**)
- `git config --global core.autocrlf true` : conversion automatique des caractères de fin de ligne (**Windows**)

# Repository (dépôt)

- C'est l'endroit où Git va stocker tous ses objets : versions, branches, *tags*...
- Situé dans le sous-répertoire `.git` de l'emplacement où on a initialisé le dépôt
- Organisé comme un *filesystem* versionné, contenant l'intégralité des fichiers de chaque version (ou *commit*)

# Commit

Fonctionnellement : **Unité d'oeuvre**

- Doit **compiler**
- Doit **fonctionner**
- Doit **signifier** quelque chose (correction d'anomalie, développement d'une fonctionnalité / fragment de fonctionnalité)

# Commit

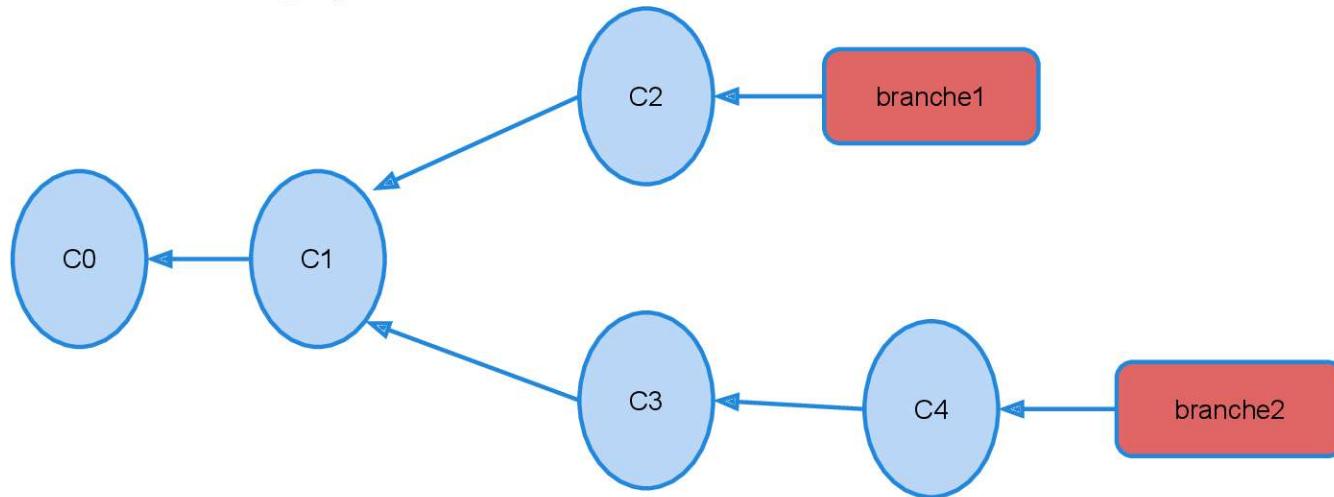
Techniquement : Pointeur

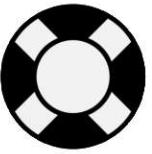
vers un *snapshot* du *filesystem* dans son ensemble

- Connaît son ou ses **parents**
- Possède un **identifiant unique** (hash SHA1) basé sur le contenu et sur le ou les parents



- Le *repository* contient l'ensemble des *commits* organisés sous forme de **graphe acyclique direct** :
  - Depuis un *commit*, on peut accéder à tous ses ancêtres
  - Un *commit* ne peut pas connaître ses descendants
  - On peut accéder à un *commit* via son ID unique
  - Des pointeurs vers les *commits* permettent d'y accéder facilement (branches, tags)





# HELP

- git help <commande>
- git help <concept>

# Création d'un *repository* Git

- git init
- Répertoire .git == dépôt
- Fichier de conf .git/config
- Répertoire racine == *working copy*

# Ajouter un changement dans le *repository*

- Faire des modifications dans la *working copy* (ajout / modification / suppression de fichiers)
- Ajouter les modifications dans la *staging area*
- Commiter == générer un *commit* à partir des changements dans la *staging area* pour l'ajouter au *repository*



## ***Staging area***

C'est la **liste des modifications** effectuées dans la *working copy* et qu'on veut inclure dans le prochain *commit*.

On construit cette liste explicitement.

- git status : affiche le statut de la *working copy* et de la *staging area*
- git add : ajoute un fichier à la *staging area*
- git rm --cached : *unstage* un nouveau fichier
- git checkout -- : retire un fichier de la *staging area*

# Commit

- git commit -m "mon commentaire de commit"  
→ génère un *commit* avec les modifications contenues dans la *staging area*
- git commit -a -m "mon commentaire de commit"  
→ ajoute tous les fichiers modifiés (pas les ajouts / suppressions) à la *staging area* et committe
- git commit --amend  
→ corrige le *commit* précédent

# Historique des versions

- `git log [-n] [-p] [--oneline]`: historique
  - affiche les ID des *commits*, les messages, les modifications
  - `-n` : limite à n *commits*
  - `-p` : affiche le diff avec le *commit* précédent
  - `--oneline` : affiche uniquement le début de l'ID du *commit* et le commentaire sur une seule ligne pour chaque *commit*
- `git show [--stat]` : branche, tag, commit-id ...
  - montre le contenu d'un objet
- `git diff`:
  - `git diff id_commit` : diff entre *working copy* et *commit*
  - `git diff id_commit1 id_commit2` : diff entre deux *commits*

# Ancêtres et références

- `id_commit^` : parent du *commit*
- `id_commit^^` : grand-père du *commit*...
- `id_commit~n` : n-ième ancêtre du *commit*
- `id_commit^2` : deuxième parent du *commit* (*merge*)
- `id_commit1..id_commit2` :  
variations entre le *commit 1* et le *commit 2*  
(ex. `git log id_commit1..id_commit2` : tous les  
commits accessibles depuis *commit2* sans ceux  
accessibles depuis *commit1*)

- Créer un nouveau *repository* Git
- Ajouter un fichier et le commiter
- Modifier le fichier et le commiter
- Observer l'historique (on doit voir les deux *commits*)

# Branches

# Introduction

- Déviation par rapport à la route principale
- Permet le développement de différentes versions en parallèle
  - Version en cours de développement
  - Version en production (correction de bugs)
  - Version en recette
  - ...
- On parle de “**merge**” lorsque tout ou partie des modifications d'une branche sont rapatriées dans une autre
- On parle de “**feature branch**” pour une branche dédiée au développement d'une fonctionnalité (ex : gestion des contrats...)

# Introduction

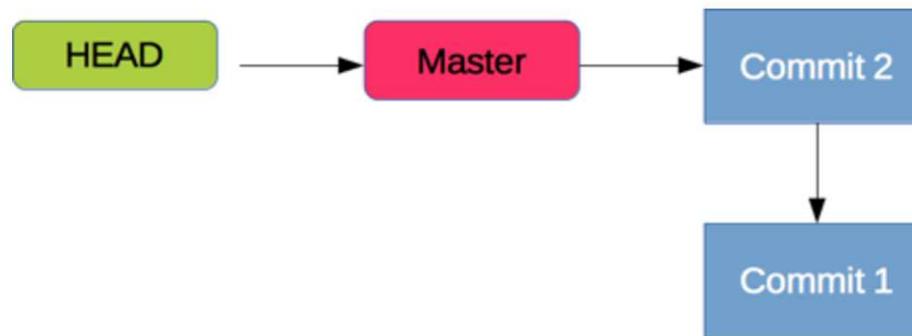
- **branch** == pointeur sur le dernier *commit* (sommet) de la branche
  - les branches sont des **références**
- **master** == branche principale (*trunk*)
- **HEAD** == pointeur sur la position actuelle de la **working copy**

# Création

- git branch <mabranche> (**création**) + git checkout <mabranche> (**se positionner dessus**)
- Ou git checkout -b <mabranche> (**création + se positionner dessus**)
- git branch → liste des branches (locales)

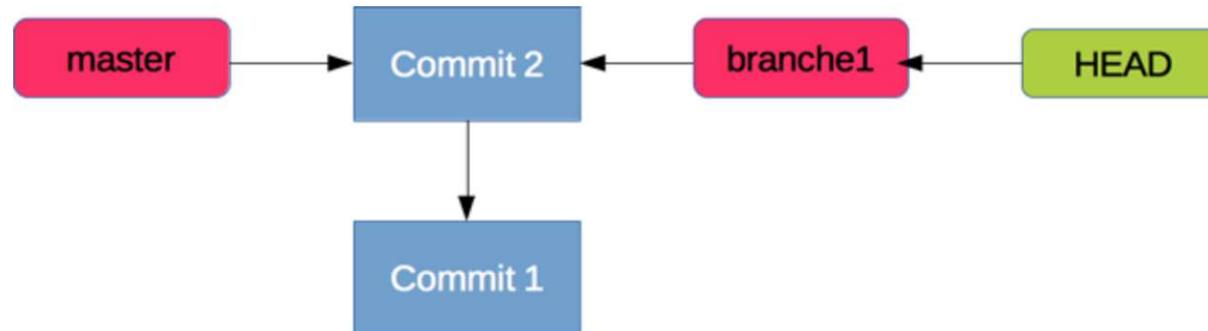
# Création

- Situation initiale



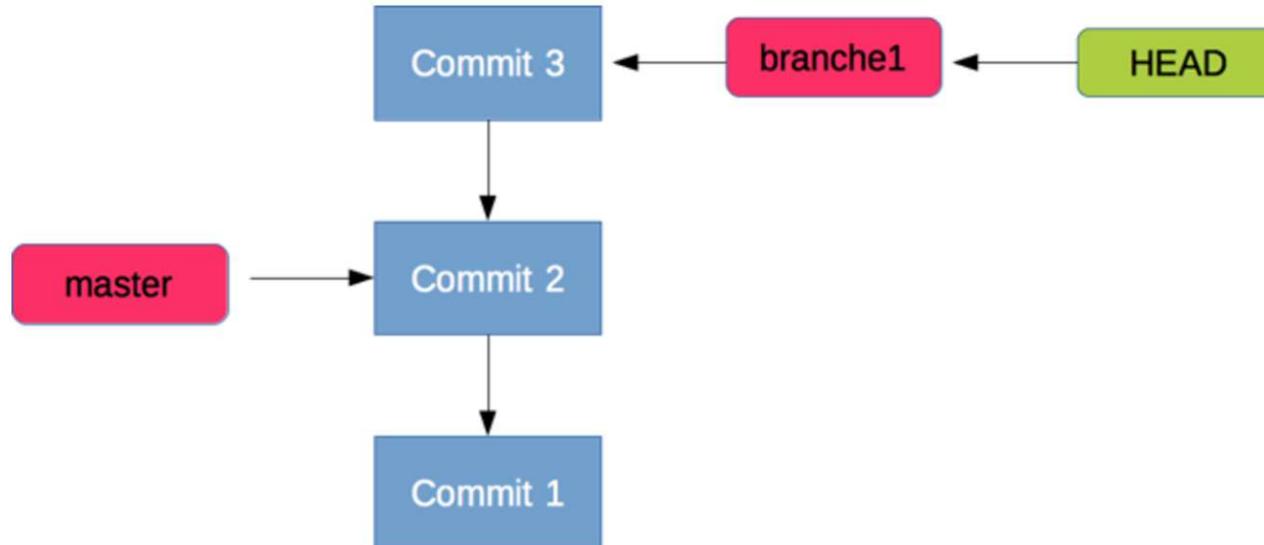
# Création

- Après git checkout -b branche1 on obtient :



# Création

- Après un troisième commit (`git commit -a -m "commit 3"`) on obtient :



# Suppression

- git branch -d mabranche (**erreur si pas mergé**)
- git branch -D mabranche (**forcé**)
- Supprime la référence, pas les *commits* (on peut toujours récupérer via reflog en cas d'erreur)

# Ancêtres et Références

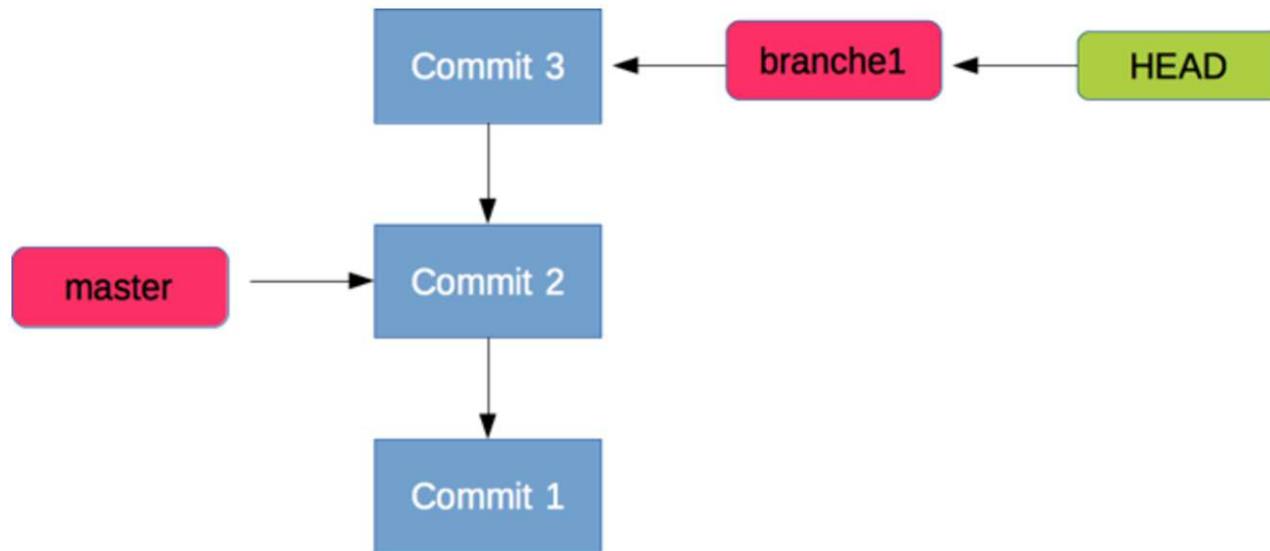
- Les branches sont des références vers le *commit* du sommet de la branche,  
on peut donc utiliser les notations ^ ou ~ sur la branche
  - branche1^^ : le grand-père du commit au sommet de branche 1
  - on peut aussi le faire sur un *tag*

# Checkout

- La commande `checkout` permet de déplacer HEAD sur une autre référence : (branche, tag, commit...)
- `git checkout <ref>` :  
*checkoute une référence*
- `git checkout -b <branch>` :  
crée une branche et la *checkoute*

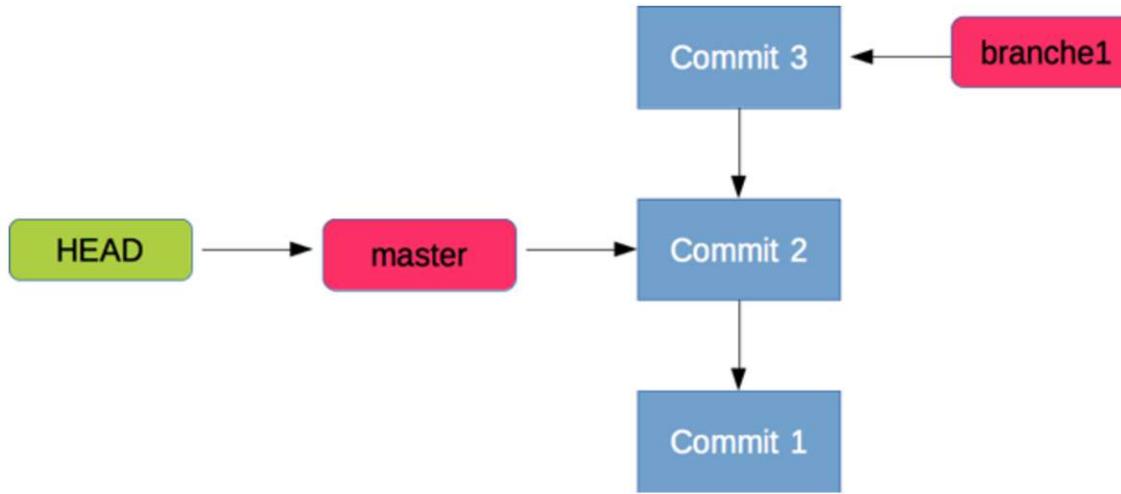
# Exemple

- Situation initiale : HEAD sur branche1



# Exemple

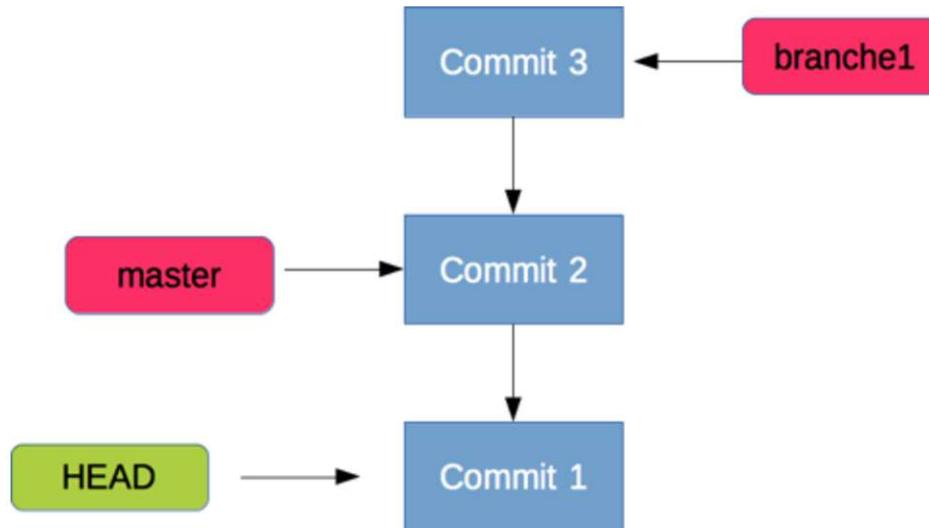
- On peut repasser sur **master** avec `git checkout master`



- On a juste pointé **HEAD** vers **master** plutôt que **branche1**
- **Checkout déplace HEAD (et met à jour la *working copy*)**

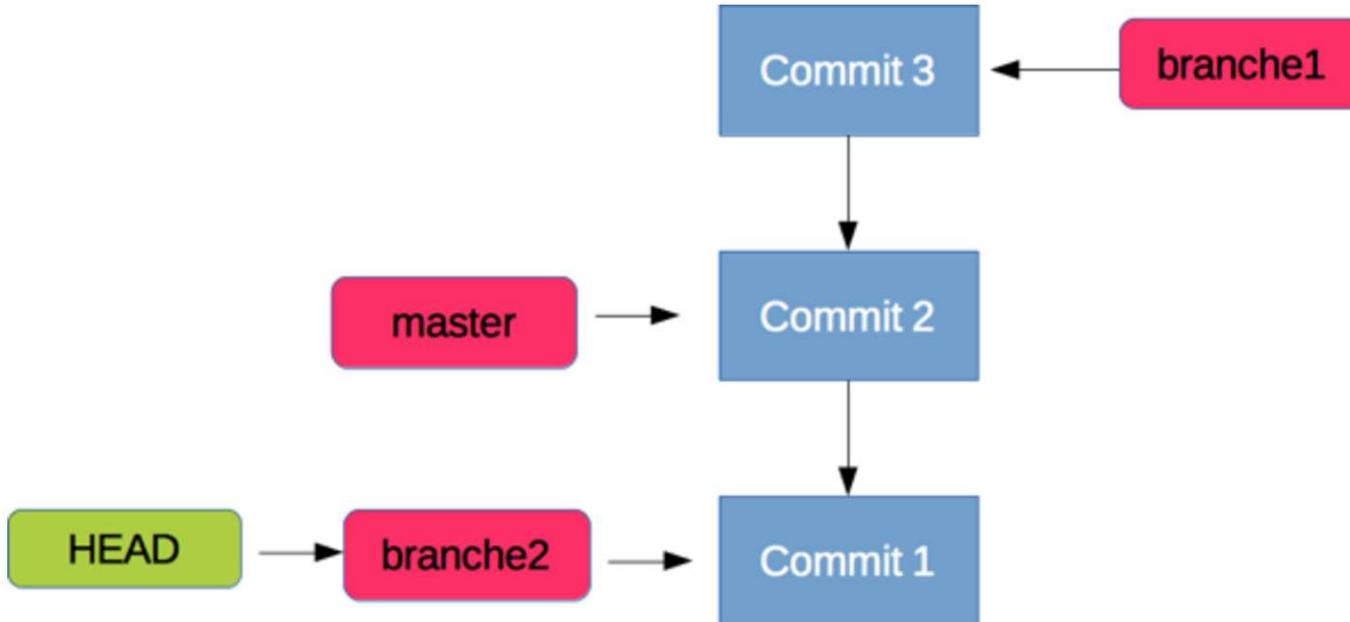
# Detached HEAD

- On peut aussi faire un checkout sur un *commit* (ou un *tag*) :
  - git checkout <id\_du\_commit>
  - On parle de “detached HEAD” car la **HEAD** n'est pas sur une branche



# Création de branche à posteriori

- Avec une *detached HEAD*, on peut créer une branche “après coup” sur le commit 1 (git branch branche2)



- Les branches sont des références vers le *commit* du sommet de la branche.  
On peut donc utiliser les notations ^ ou ~ pour un *checkout* :
  - `checkout branch1^` : on *checkoute* le grand-père du *commit* au sommet de branche 1 (*detached head*)
- Impossible de faire un *checkout* si on a des fichiers non commités modifiés, il faut faire un *commit* ou un *reset* (ou un *stash* comme on le verra plus tard)
- Les nouveaux fichiers restent dans la *working copy* (ils ne sont pas perdus suite au *checkout*).

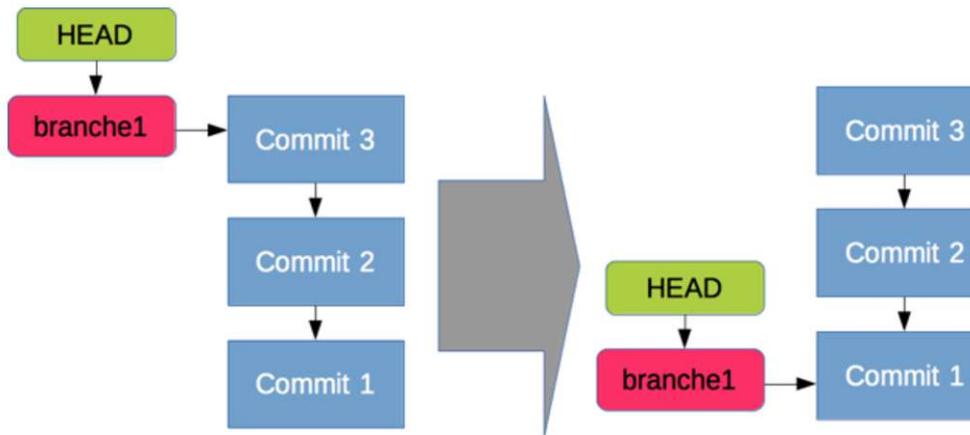
# Reset

- Permet de déplacer le **sommet d'une branche** sur un *commit* particulier, en resettant éventuellement l'index et la *working copy*
- 2 utilisations principales :
  - annuler les modifications en cours sur la *working copy*
  - faire “reculer” une branche  
→ annuler un ou plusieurs derniers *commits*

- `git reset [mode] [commit]` : *resette* la branche courante
  - commit :
    - id du commit sur lequel on veut positionner le sommet de la branche
    - si vide, on laisse la branche où elle est (utile pour *resetter* l'index ou la *working copy*)
  - mode :
    - `--soft` : ne touche ni à l'index, ni à la *working copy*  
(alias “je travaillais sur la mauvaise branche”)
    - `--hard` : *resette* l'index et la *working copy*  
(alias “je mets tout à la poubelle”)
    - `--mixed` : *resette* l'index mais pas la *working copy*  
(alias “finalement je ne vais pas commiter tout ça”)  
→ **c'est le mode par défaut**
  - Le mode par défaut (*mixed*) n'entraîne pas de perte de données, on retire juste les changements de l'index

- Pour revenir sur une *working copy* propre (c'est-à-dire supprimer tous les changements non commis) :
  - `git reset --hard`

- Le *reset* permet de déplacer le sommet d'une branche
- Ex : git reset --hard HEAD<sup>^^</sup>



- Si on passe --hard, on se retrouve sur commit1 et la *working copy* est propre
- Si on ne passe pas --hard, on se retrouve aussi sur commit 1 et la *working copy* contient les modifications de commit 3 et commit 2 (non committées non indexées)

# Tag

- Littéralement “étiquette” → permet de marquer / retrouver une version précise du code source
- `git tag -a nom_du_tag -m "message"` : crée un tag
- `git tag -l` : liste les tags
- C'est une référence vers un commit
- On peut faire un checkout sur un tag (comme une branche ou un commit) → detached HEAD
- Les tags sont des références vers un commit on peut donc utiliser les notations ^ ou ~ pour un checkout :
  - → `checkout mon_tag^` : on checkout le grand-père du commit du tag (detached head)

- Créer un nouveau *repository* Git
- Ajouter un fichier et le commiter
- Ajouter un deuxième fichier et le commiter
- Vérifier l'historique (on doit avoir 2 commits)
- Faire des modifications sur le deuxième fichier et le commiter
- Annuler les modifications du dernier commit
- Vérifier l'historique (on doit avoir 2 commits)
- Créer une branche à partir du 1er commit
- Faire un commit sur la branche
- Vérifier l'historique de la branche (on doit avoir 2 commits)

- Lister les branches (on doit avoir 1 branche)
- Tagger la version
- Revenir au sommet de la branche *master*
- Lister les tags (on doit avoir un *tag*)
- Supprimer la branche
- Lister les branches (on doit avoir une seule branche :  
*master*)

# Reflog

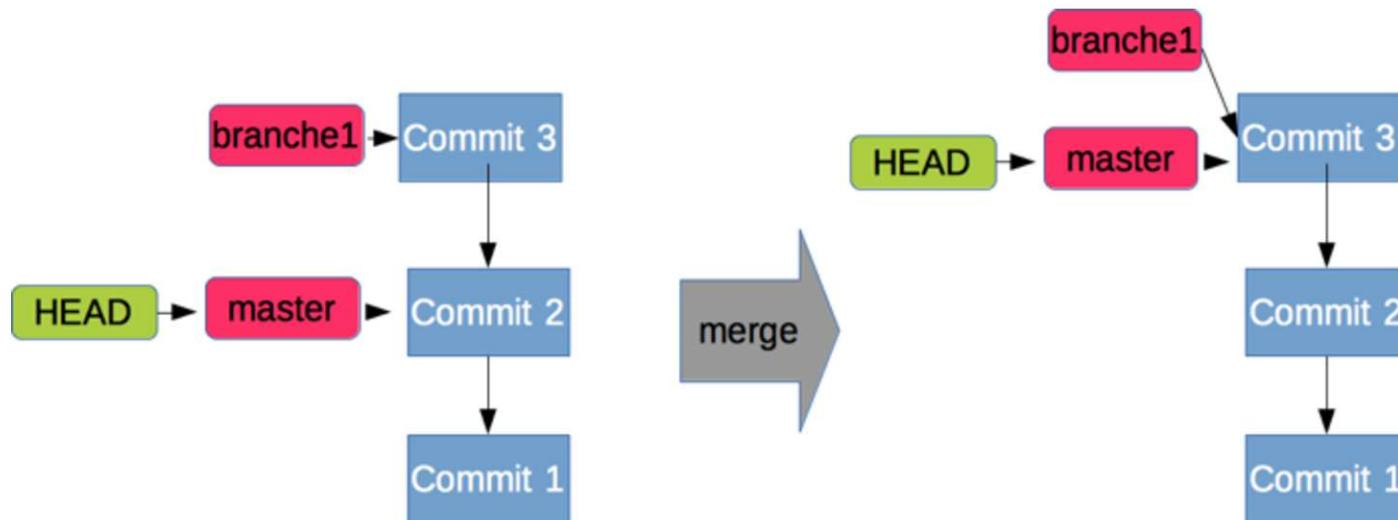
- Reflog → Reference Log
- Commit inaccessible (reset malencontreux / pas de branche / id oublié ?)
- 30 jours avant suppression
- git reflog
- git reset --hard HEAD@{n} → **repositionne la branche sur la ligne n du reflog**

# Merge

- Fusionner 2 branches / Réconcilier 2 historiques
- Rapatrier les modifications d'une branche dans une autre
- ATTENTION: par défaut le *merge* concerne tous les *commits* depuis le dernier *merge* / création de la branche
- Depuis la branche de destination : git merge nom\_branche\_a\_merger
- On peut aussi spécifier un ID de *commit* ou un *tag*, plutôt qu'une branche
- 2 cas : *fast forward* et *non fast forward*

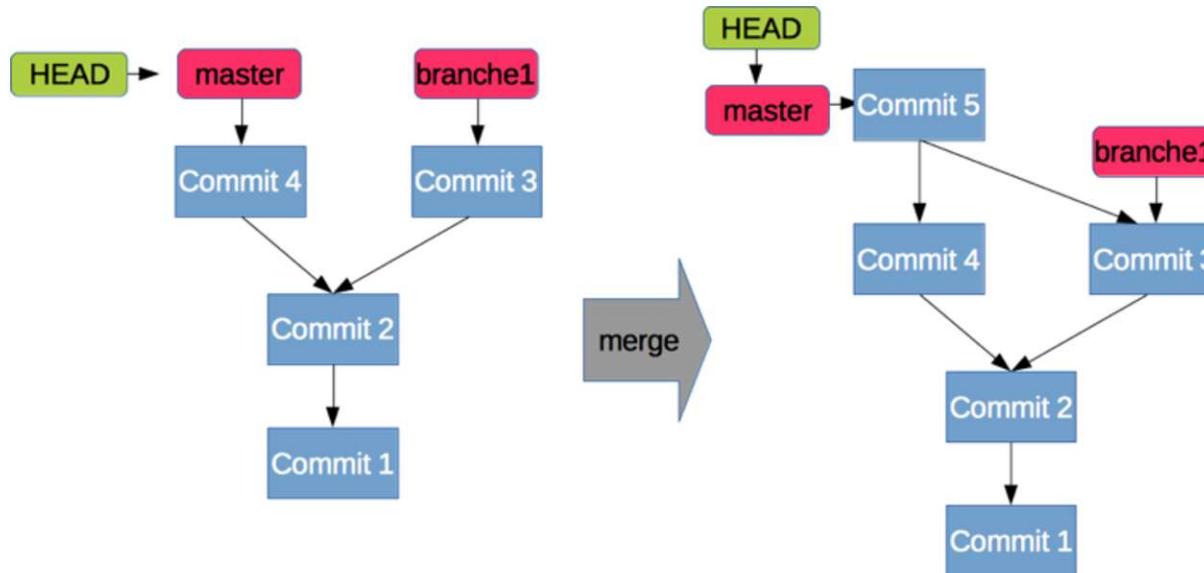
# Fast-forward

- Cas simple / automatique
- Quand il n'y a pas d'ambiguïté sur l'historique



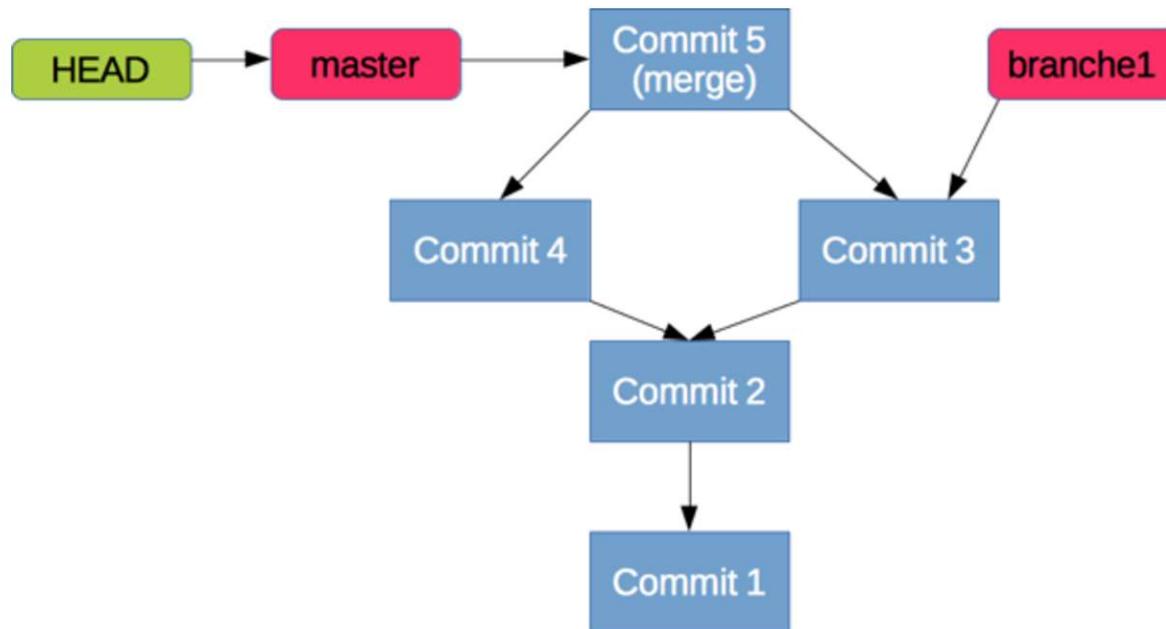
# Non fast-forward

- Quand il y a ambiguïté sur l'historique
- Création d'un *commit de merge*



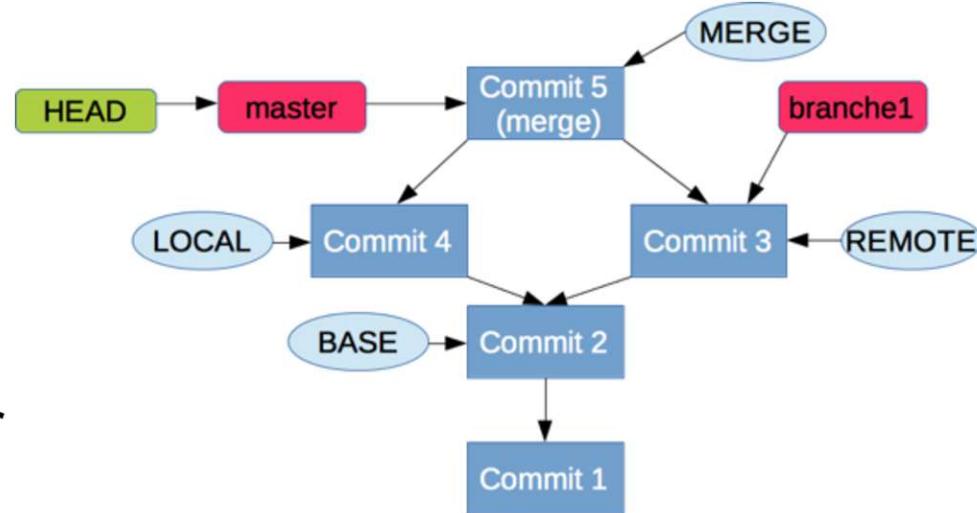
# Conflit

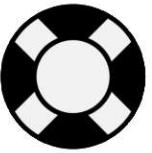
- On souhaite merger la branche branche1 sur master pour obtenir :



# Conflit

- Commit 4 et commit 3 modifient la même ligne du fichier
- Git ne sait pas quoi choisir  
→ conflit  
→ suspension **avant** le commit de merge
- git mergetool / Résolution du conflit / git commit
- Ou git merge --abort ou git reset --merge ou git reset --hard HEAD pour annuler
- NB : branche1 ne bougera pas

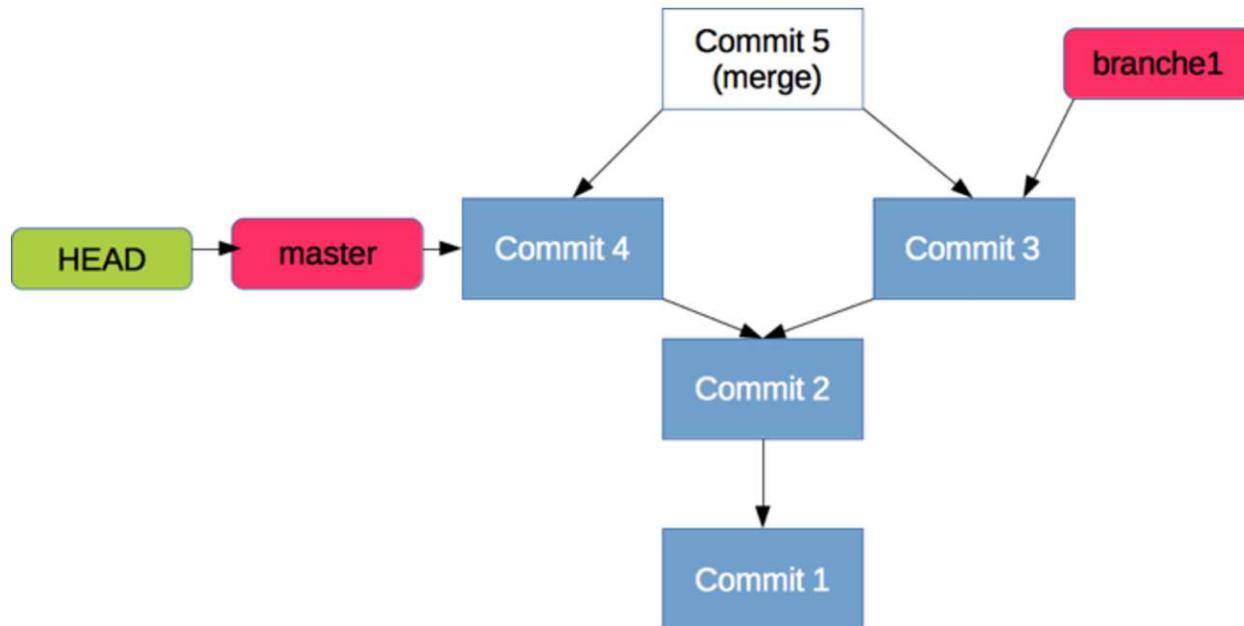




- Si on veut éviter le *fast forward* (*merge d'une feature branch*) on utilise le flag `-no-ff`
- Ex : `git merge branch1 --no-ff`

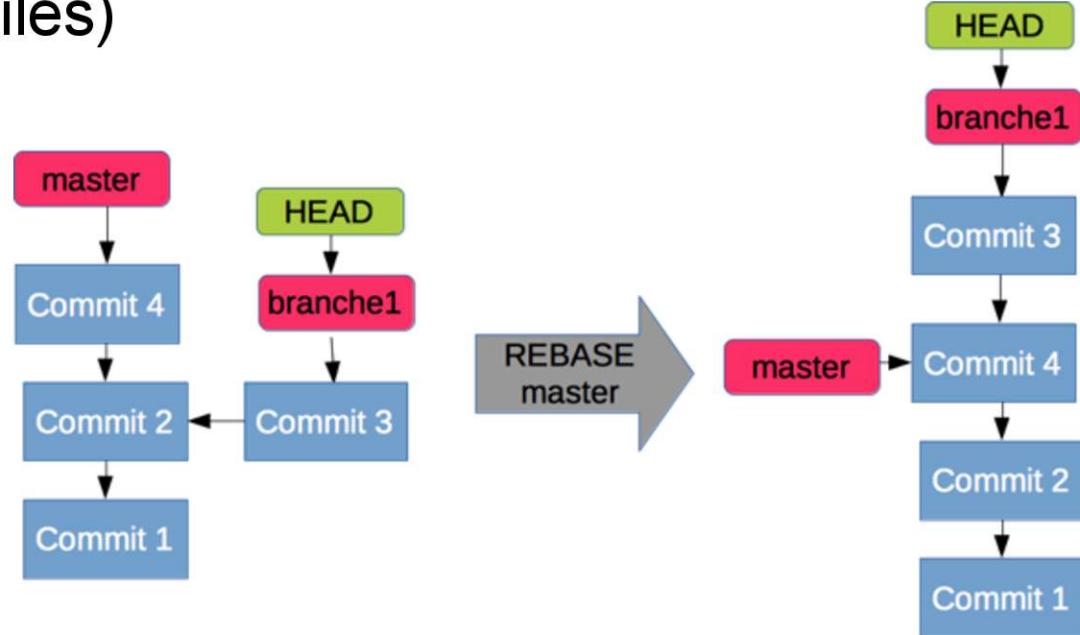
# Annulation (après merge)

- git reset --hard HEAD^

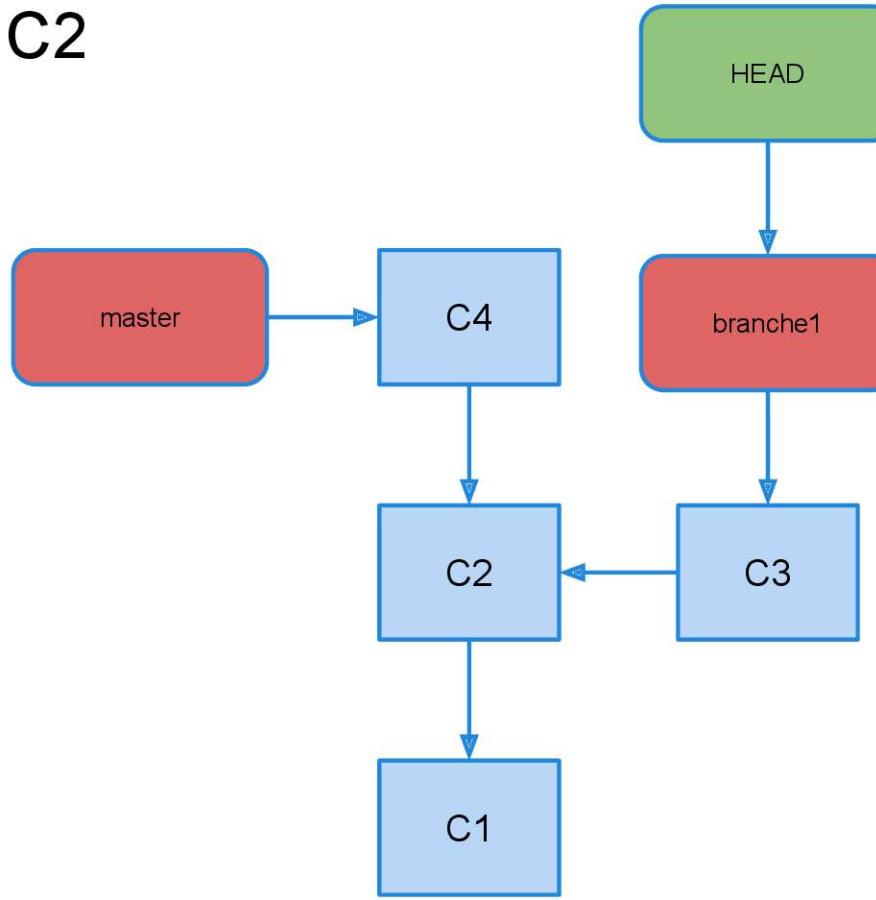


# Rebase

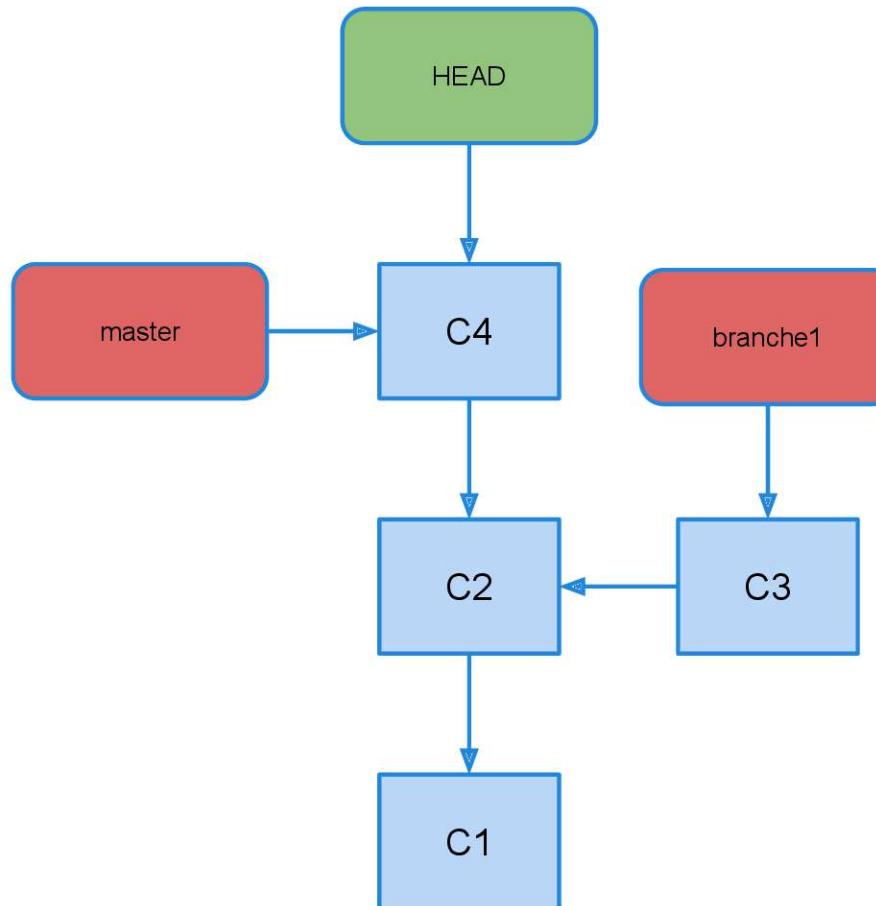
- Modifie / réécrit l'historique
- Modifie / actualise le point de départ de la branche
- Remet nos commits au dessus de la branche contre laquelle on rebase
- Linéarise (évite de polluer l'historique avec des commits de merge inutiles)



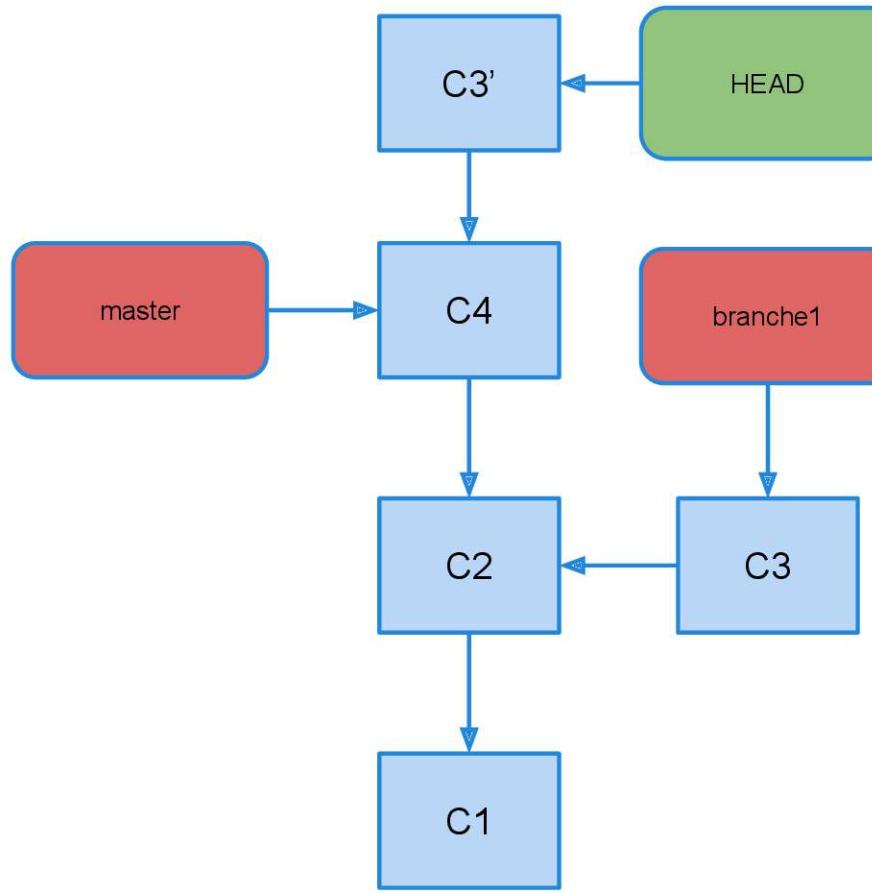
- Situation de départ : 3 commits sur master (C1,C2 et C4) , 3 commits sur branche1 (C1, C2 et C3) , création de branche 1 à partir de C2



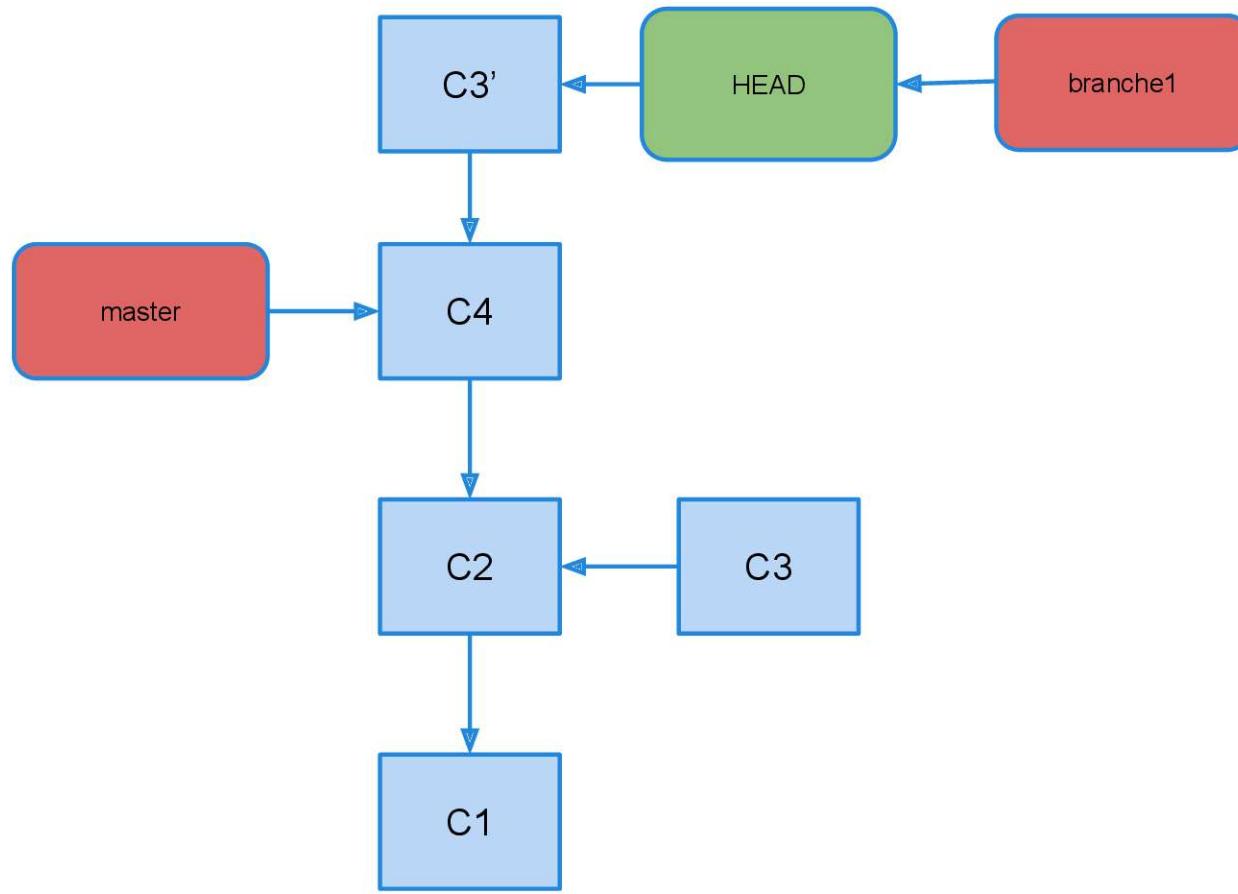
- Depuis branche 1 on fait un git rebase master
- HEAD est déplacé sur C4



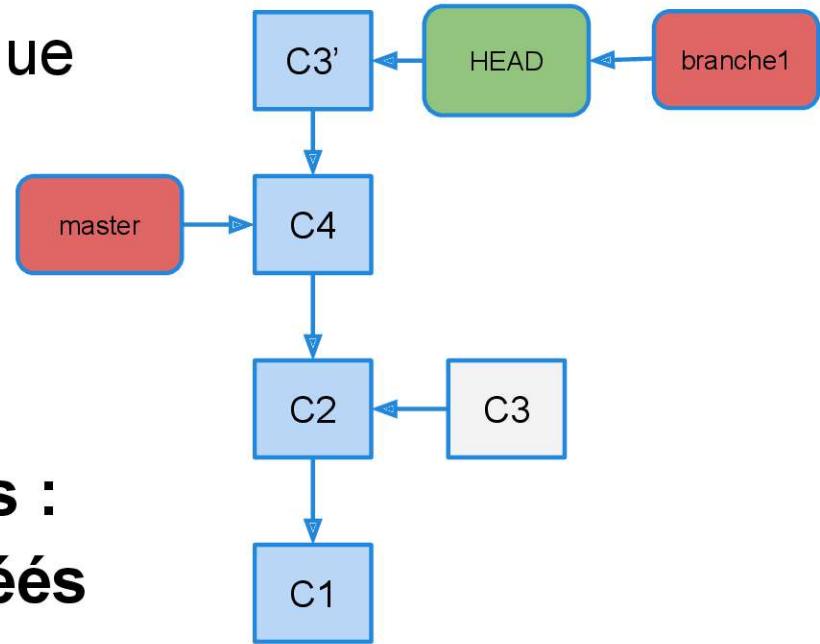
- Git fait un diff entre C3 et C2 et l'applique à C4 pour “recréer” un nouveau C3 ( C3' ) dont le père est C4



- Git reset branche 1 sur la HEAD , le rebase est terminé



- Rebase modifie / réécrit l'historique
- Les commits de branche1 deviennent des descendants de ceux de master (la hiérarchie devient linéaire)
- **On ne modifie pas les commits : de nouveaux commits sont créés à partir de ceux qu'on rebase (on peut toujours les récupérer via id ou reflog)**
- Si on merge branche1 dans master on aura un fast forward
- **Le commit C3 n'est plus accessible que par son id, dans 30 jours il sera effacé**

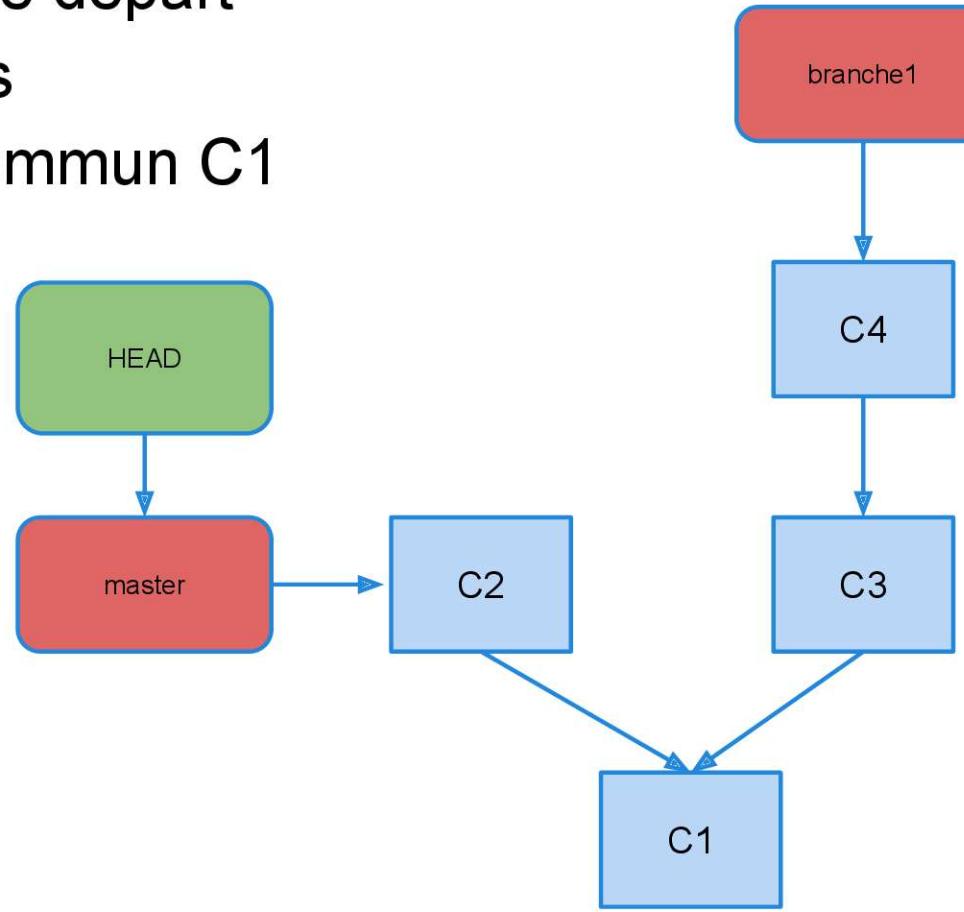


# Merge VS Rebase

- Rebase : pour la mise à jour des branches avant merge linéaire (commits indépendants) ex : corrections d'anomalies → on ne veut pas de commit de merge
- Merge sans rebase : pour la réintégration des feature branches (on veut garder l'historique des commits indépendants sans polluer l'historique de la branche principale)

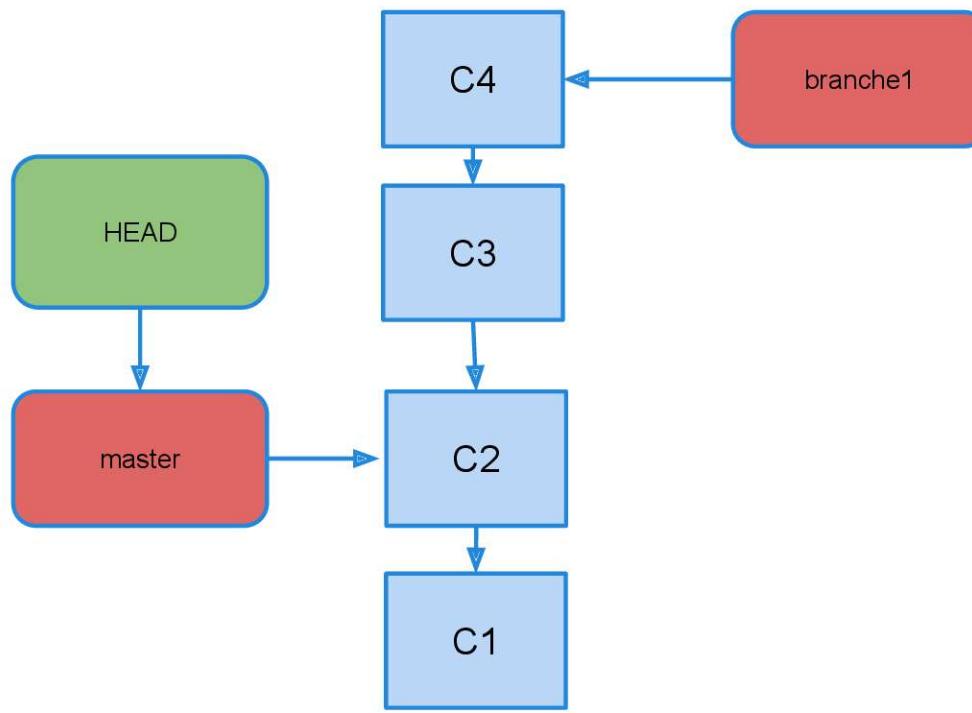
# Merge avec Rebase (1/3)

- Situation de départ
- 2 branches
- Ancêtre commun C1



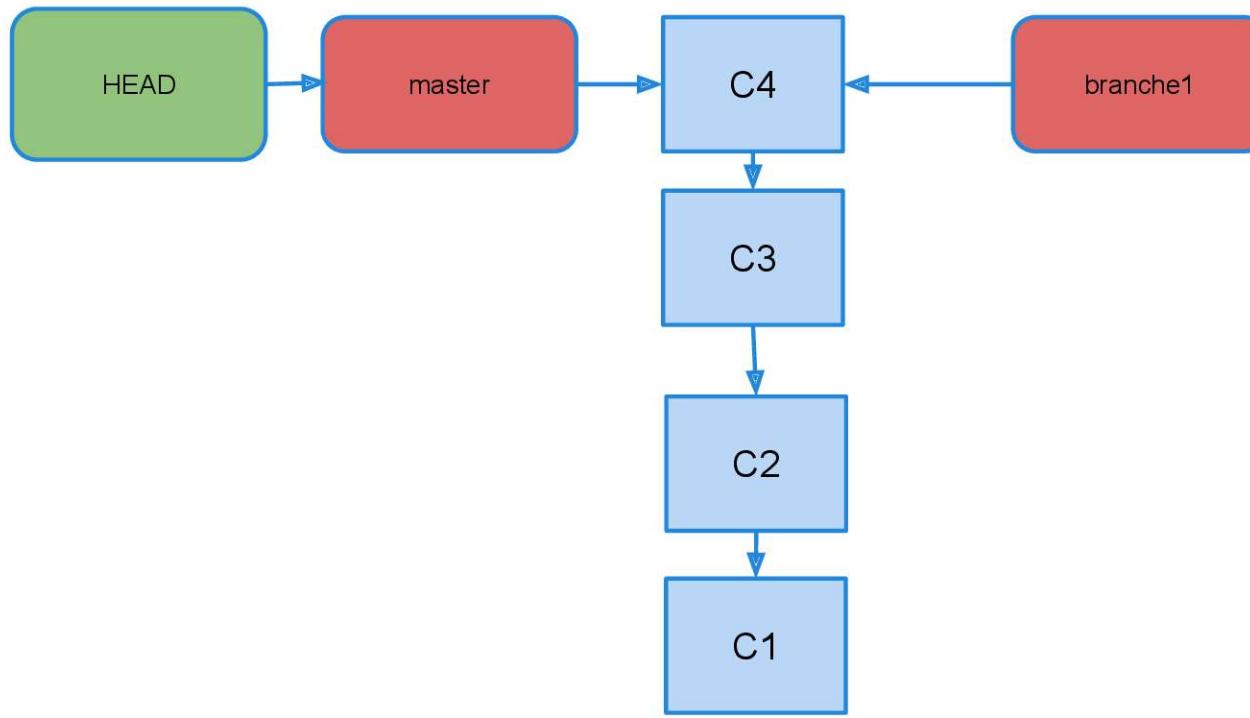
# Merge avec Rebase (2/3)

- Rebase de branche1 sur master



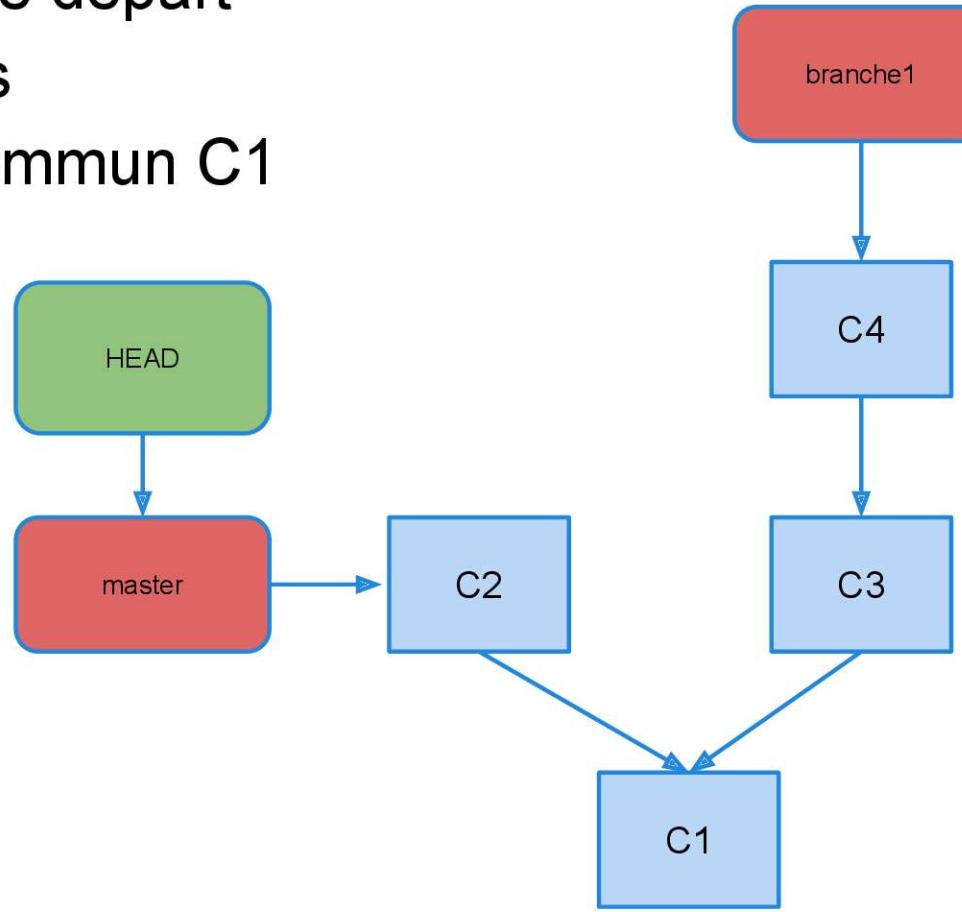
# Merge avec Rebase (3/3)

- Merge de branche 1 dans master
- Fast forward



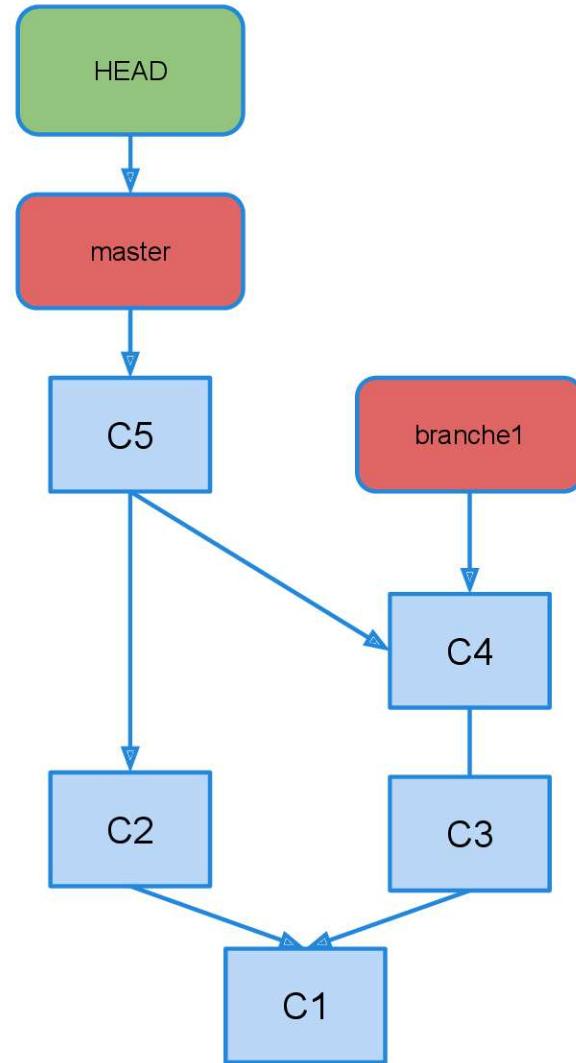
# Merge sans Rebase (1/2)

- Situation de départ
- 2 branches
- Ancêtre commun C1



# Merge sans Rebase (2/2)

- Merge de branche 1 dans master
- Non fast forward
- Création d'un commit de merge (C5)



# TP Rebasing

- Créer un nouveau *repository* Git
- Ajouter un fichier et le commiter (C1), le modifier et le commiter (C2)
- Créer une branche B1 à partir de C1
- Faire une modification du fichier et commiter C3
- Merger B1 dans *master* de manière à avoir un historique linéaire

# TP Merge

- Créer un nouveau *repository git*
- Ajouter un fichier et le commiter (C1)
- Créer une *feature branch* B1 à partir de C1
- Faire une modification du fichier et commiter (C2)
- Merger B1 dans *master* de manière à avoir un commit de *merge* dans *master*

# TP Conflit

- Créer un nouveau *repository* Git
- Ajouter un fichier et le commiter (C1)
- Modifier la première ligne du fichier et commiter (C2)
- Créer une feature branch B1 à partir de C1
- Faire une modification de la première ligne du fichier et commiter (C3)
- Merger B1 dans *master* en résolvant les conflits

# Git avec un dépôt distant

# Repository distant

# **Utilisations d'un repository distant :**

- Pour partager son travail via un repository central (ex svn / cvs ...)
- Repository read only qu'on peut fork (ex : github)
- Pour déployer son code (ex: heroku)
- Dans Git chaque repository peut être “cloné” (copié)
  - Le repository cloné devient de fait le repository distant du clone

# Clone

- Clone complet du repository distant
  - branches, tags → tout est cloné
  - le repository distant peut être exposé via ssh, http, file ...
- `git clone url_du_repository`

# Remote

- C'est la définition d'un repository distant
- Nom + url du repository
- `git remote add url_du_repo` : ajoute une remote
- Créeé par défaut avec clone
- Remote par défaut == origin

# Bare repository

- Repository n'ayant pas vocation à être utilisé pour le développement :
  - Pas de working copy
  - Utilisé notamment pour avoir un repository central
- `git init --bare` : initialise un nouveau bare repository
- `git clone --bare` : clone un repository en tant que bare repository

# Branches distantes

# Remote branch

- Lien vers la branche correspondante du dépôt distant
- Miroir de la branche distante
- Crées par défaut avec clone
- Manipulée via la branche locale correspondante ex master  
→ remotes/origin/master
- git branch -a : liste toutes les branches locales et **remotes**

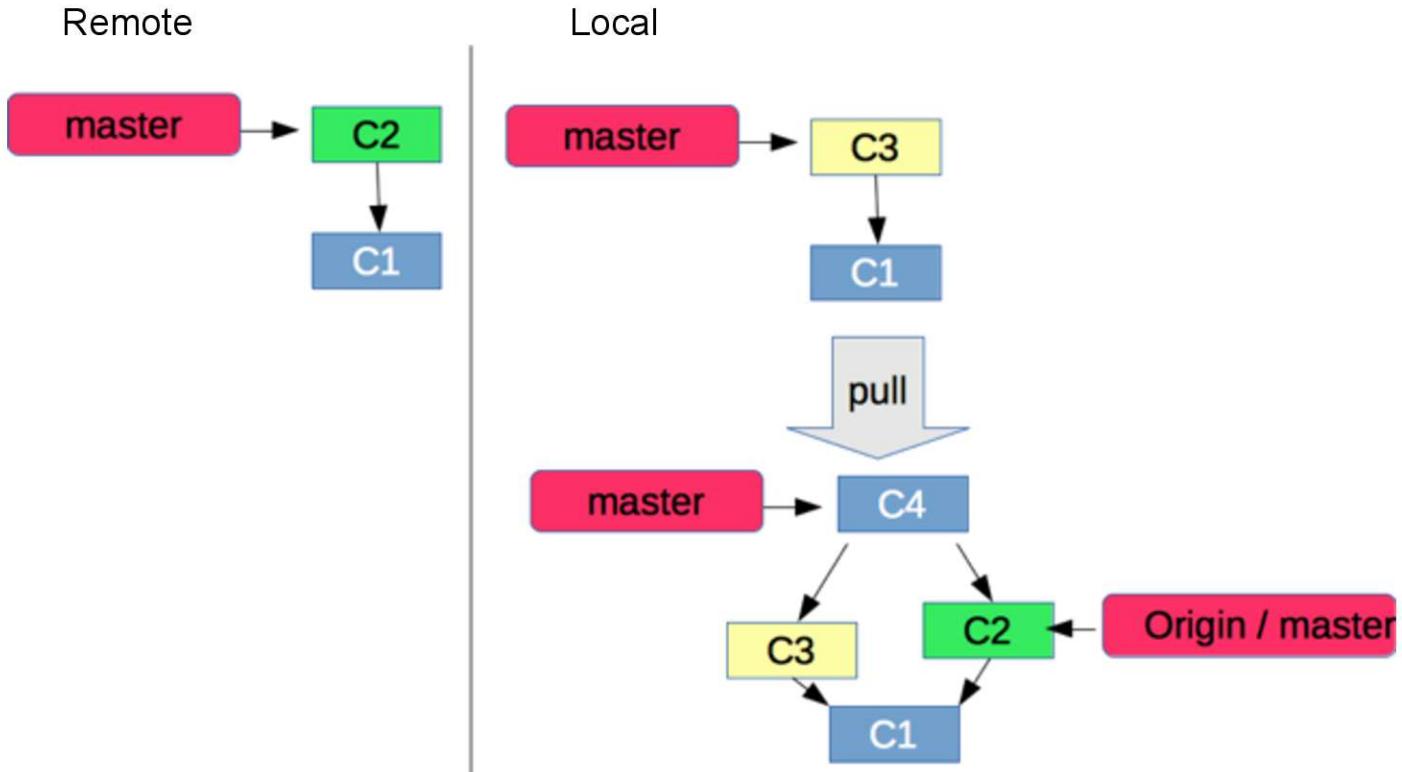
# Fetch

- `git fetch [<remote>]`
- Met à jour les informations d'une remote
  - récupère les commits accessibles par les branches distantes référencées
  - met à jour les références des branches distantes
  - ne touche pas aux références des branches locales

# Pull

- Équivalent de fetch + merge remote/branch
- Update la branche locale à partir de la branche remote
- A éviter peut générer un commit de merge → pas très esthétique
- Se comporte comme un merge d'une branche locale dans une autre

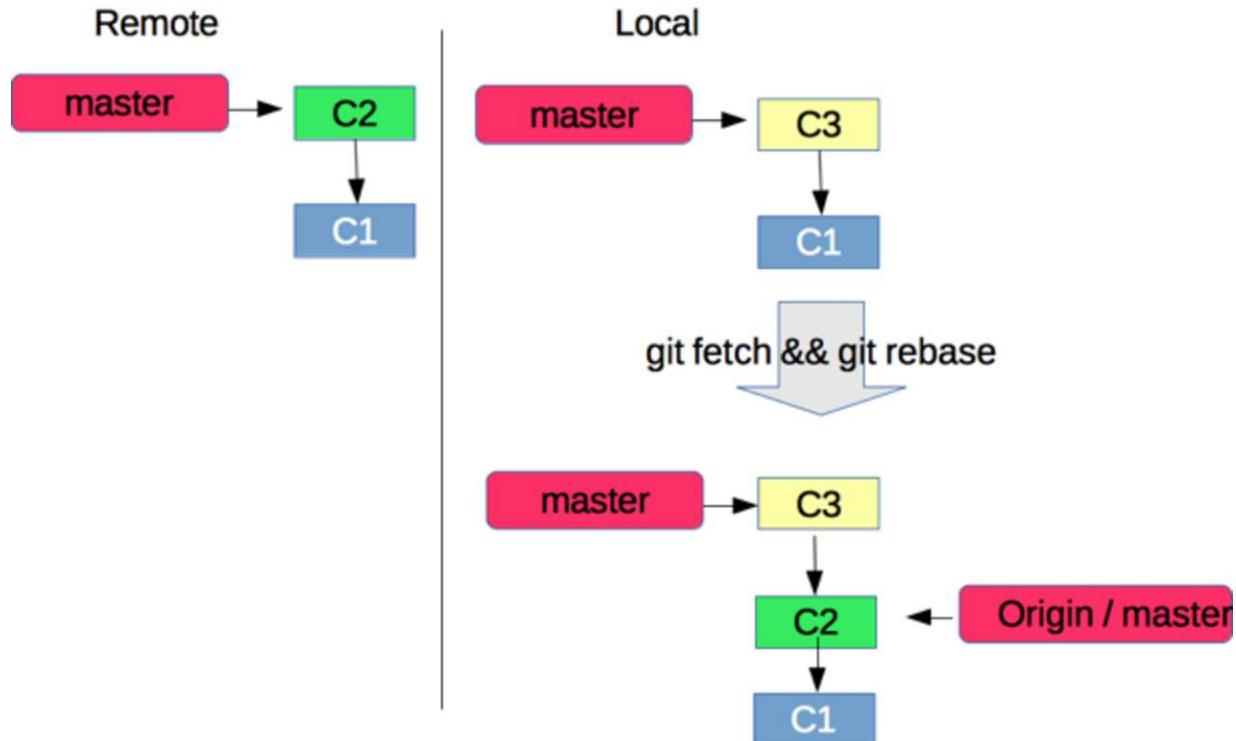
# Pull



## Fetch + rebase

- Permet de récupérer les modifications de la remote et de placer les nôtres “au dessus”
- Plus “propre” que pull → pas de commit de merge
- Se comporte comme un rebase d’une branche locale sur une autre
- Équivalent à `pull --rebase` (**configurable par défaut**)

# Fetch + rebase



# Push

- Publie les commits locaux sur le repository distant
- git status → donne le nombre de commit d'avance / de retard sur la remote
- Refuse de pusher si retard → faire un fetch + rebase -p et recommencer

# Push

- Par défaut publie tous les commits de la branche courante non présents sur la remote
- On peut publier jusqu'à un commit via :  
`git push nom_remote id_commit:nom_branche_remote`

# Push

git push -f : force le push même en cas d'historique divergent : notre historique “remplace” celui du repository distant

- Utile pour corriger une erreur de push avant que les autres users n'aient récupéré les changements
- Attention nécessite des interventions de la part des autres utilisateurs s'ils ont updaté leur repository avant le push -f (ils risquent de merger l'ancien et le nouvel historique)
- On préfère généralement faire un revert

# Créer une branche remote

- Créer une branche locale et se placer dessus :

```
git checkout -b mabranche
```

- Publier la branche :

```
git push -u nom_remote nom_branche
```

- Le -u permet de dire que l'on track la remote (pas besoin de spécifier la remote)

# Emprunter une branche remote

- Updater les références de la remote : `git fetch [nom_remote]` → récupère la branche remote
- `git branch -a` → liste toutes les branches
- Créer la branche locale correspondante :  
`git checkout --track nom_remote/nom_branche_remote`

## **Supprimer une branche distante**

- `git push nom_remote :nom_branche`

# **Créer un tag remote**

- **Créer le tag en local :**

```
git tag -a nom_tag -m "message"
```

- **Publier le tag :**

```
git push nom_remote nom_tag
```

- Créer un nouveau *repository* Git (R1)
- Ajouter un fichier et le commiter (C1)
- Cloner le *repository* (protocole *file*) (R2)
- Lister toutes les branches locales et distantes (on doit avoir une branche locale, une branche *remote* et une *remote head*)
- Sur R1 modifier le fichier et commiter (C2)
- Sur R2 récupérer le commit C2 (vérifier avec git log)
- Sur R2 créer une nouvelle branche (B1), faire une modification du fichier, commiter (C3)
- Publier B1 sur sur R1 (vérifier avec git branch -a sur R1)
- Créer une branche B2 sur R1

- Récupérer B2 sur R2 (vérifier avec `git branch -a` sur R2)
- Tagger B2 sur R2 (T1)
- Publier T1 sur R1
- Vérifier que le Tag T1 est sur R1 (`git tag -l`)
- Sur R1 B1 modifier la première ligne du fichier et commiter (C4)
- Sur R2 B1 modifier la première ligne du fichier et commiter (C5)
- Publier C5 sur R1 B1 (conflit)
- Résoudre le conflit
- Vérifier la présence d'un commit de *merge* sur R1 B1

# Commandes diverses

# **Revert**

- git revert id\_du\_commit
- → génère un antécommit == annulation des modifications du commit

# **Blame**

- Indique l'auteur de chaque ligne d'un fichier
- `git blame <file>`

# Stash

- Cachette / planque
- Sauvegarder sa working copy sans commiter (ex : pour un changement de branche rapide )
- git stash : Déplace le contenu de la working copy et de l' index dans une stash
- git stash list : list des stash
- git stash pop [stash@{n} ] : pop la dernière stash (ou la n-ième)

# Bisect

- Permet de chercher la version d'introduction d'un bug dans une branche :
  - On fournit une bonne version et une mauvaise
  - Git empreinte une succession de versions et nous demande si elles sont bonnes ou mauvaises
  - Au bout d'un certain nombre de versions git identifie la version d'introduction du bug
- Commandes :
  - `git bisect start` : démarre le bisect
  - `git bisect bad [<ref>]` : marque le commit en bad
  - `git bisect good [<ref>]` : marque le commit en good
  - `git bisect skip [<ref>]` : passe le commit
  - `git bisect visualize` : affiche les suspects restant (graphique)
  - `git bisect reset` : arrête le bisect

# Grep

- Permet de rechercher du texte ou une regexp dans les fichiers du repository
- Permet également de préciser dans quel commit faire la recherche
- `git grep <texte> [<ref>]`

# Hunk

- Plusieurs modifications dans le même fichiers qui correspondent à des commits différents ?
- Ajoute un fragment des modifications du fichier à l'index
- `git add -p ou git gui`

# Cherry pick

- Prend uniquement les modifications d'un commit (sans historique) et l'applique à la branche
- `git cherry-pick id_du_commit`
- A utiliser avec parcimonie (branches sans liens)

# Patch

- Permet de formater et d'appliquer des diffs sous forme de patch (ex : pour transmettre des modifications par mail)
- `git format-patch [-n]` : prépare n patches pour les n derniers commits (incluant le commit pointé par HEAD)
- `git apply <patch>` : applique un patch

# Rebase interactif

- Contrôle total sur l'historique
- git rebase -i HEAD~3 (rebase les 3 derniers commits)
- Inversion des commits (inverser les lignes)
- Modification du message de commit ( r )
- Suppression d'un commit ( supprimer la ligne)
- Fusionner un commit avec le précédent ( s )
- Fusionner un commit avec le précédent sans garder le message ( f ) (exemple correctif sur un correctif)
- Editer un commit : revenir avant le commit proprement dit pour ajouter un fichier par exemple ( e )
- Comme toujours les commits ne sont pas vraiment modifiés, des nouveaux commits sont créés et pointés par HEAD mais les anciens existent toujours (cf reflog)

# Scénarios classiques

# BugFix sur *master* (1 commit)

- Je suis sur *master* (sinon git checkout master)
- Je fais mon commit : ajout des fichiers dans l'index via git add puis git commit -m "mon commit"
- Je récupère les modifications des autres en rebasant *master* : git fetch && git rebase :
- Je résous les éventuels conflits puis git rebase --continue (ou git rebase --abort)
- Mes modifications se retrouvent au sommet de *master*
- Je publie mon (ou mes) commit(s) : git push

# Nouvelle fonctionnalité sur *master* (n commits, un seul développeur)

- Exemple : nouvel écran, nouveau batch → plusieurs commits
- Je mets à jour *master* : git fetch && git rebase
- Je crée et je me place sur ma *feature branch* : git checkout -b nouvel\_ecran
- Je fais mon développement et plusieurs commits sur ma branche
- Je me place sur *master* et je fais git fetch && git rebase
- Je merge ma branche dans *master (sans fast forward)* : git merge --no-ff nouvel\_ecran
- Je publie : git push
- Cas particulier : quelqu'un a pushé entre mon *merge* et mon *push* → je dois refaire un git fetch && git rebase -p sinon le *rebase* va linéariser mon *merge*
- Je supprime ma *feature branch* : git branch -d nouvel\_ecran

## Correction d'anomalie en production (1 commit)

- Je me place sur la branche de prod : git checkout prod-1.10
- Je mets à jour ma branche locale de prod : git fetch && git rebase
- Je fais ma correction et je commite
- Je mets à jour ma branche local de prod : git fetch && git rebase (**conflits éventuels**)
- Je publie mon commit : git push
- Je me place sur *master* pour reporter ma modification :
- Je mets à jour *master* : git fetch && git rebase
- Je merge ma branche de prod dans *master* : git merge prod-1.10
- Dans des cas TRES particuliers (on ne veut qu'un seul *commit* sans les précédents) on peut faire un *cherry-pick* plutôt qu'un *merge*
- Je publie mon report de commit : git push

## Création d'une branche de prod

- Je me place sur le *tip* (sommet de la branche) de *master* (ou sur le commit qui m'intéresse) : git checkout master
- Je crée ma branche locale et je l'emprunte : git checkout -b prod-1.10
- Je pushe ma branche : git push -u origin prod-1.10

## Création d'un tag

- Je checkoute le commit où je veux faire mon *tag* (ou le *tip* d'une branche) : git checkout id\_du\_commit
- Je crée le *tag local* : git tag -a 1.10 -m "tag prod 1.10"
- Je pushe le tag : git push origin 1.10

- La *cheatsheet*  
<http://ndpsoftware.com/git-cheatsheet.html>
- La documentation  
<https://www.kernel.org/pub/software/scm/git/docs/>
- Le livre Pro Git  
<http://git-scm.com/book/>
- Le site Git Magic  
<http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/fr/>
- Les tutoriels Atlassian  
<https://www.atlassian.com/fr/git/tutorial/>
- Les articles GitHub  
<https://help.github.com/articles/>