

# DevOps

Gestionnaires de versions

Thomas Ropars

`thomas.ropars@univ-grenoble-alpes.fr`

2018

# Agenda

Introduction

GIT

Utilisation de Git

Les bonnes pratiques

Synchronisation avec des dépôts distants

# Agenda

Introduction

GIT

Utilisation de Git

Les bonnes pratiques

Synchronisation avec des dépôts distants

# Motivations

Une équipe de développeurs participe à la réalisation d'une application:

- Comment conserver un historique?
- Comment revenir en arrière?
- Comment travailler à plusieurs en parallèle sur le même code?
- Comment gérer plusieurs versions du code à la fois?
- Comment savoir ce qui a été modifié et par qui (et pourquoi)?

# Motivations

Une équipe de développeurs participe à la réalisation d'une application:

- Comment conserver un historique?
- Comment revenir en arrière?
- Comment travailler à plusieurs en parallèle sur le même code?
- Comment gérer plusieurs versions du code à la fois?
- Comment savoir ce qui a été modifié et par qui (et pourquoi)?

Utilisation d'un VCS (Version Control Software)

# Ce qu'on y stocke

Essentiellement des fichiers texte.

## Ce qu'on y met

- Fichier sources (.java, .c, .html, etc)
- Certains fichiers binaires non dérivés des sources (images)
- Fichiers de configuration, compilation (Makefile)

## Ce qu'on n'y met pas

- Fichiers temporaires
- Fichiers générés

# diff & patch

Un VCS repose sur un mécanisme permettant de calculer les différences entre 2 versions d'un fichier.

## diff

- Comparaison de fichiers ligne par ligne
- Indique les lignes ajoutées ou supprimées
- Peut ignorer les casses, les tabulations, les espaces

## patch

- Utilise la différence entre deux fichiers pour passer d'une version à l'autre

# diff & patch

## Illustration

- Sauvegarder dans un patch les modifications d'un fichier

```
$ diff toto.c toto-orig.c > correction.patch
```

- Appliquer le patch à une autre version du fichier

```
$ patch -p 0 mytoto.c < correction.patch
```

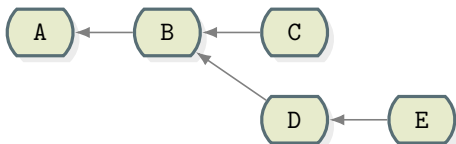
diff et patch peuvent être appliqués à une arborescence de fichiers



# La notion d'historique

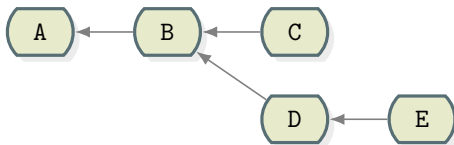
En plus de calculer la différence entre deux versions d'un fichier, il faut gérer un historique des diffs:

- L'**historique** est un graphe orienté acyclique composé d'un ensemble de versions pouvant être recalculées à partir des versions adjacentes en appliquant les patches modélisés par les arcs sortants



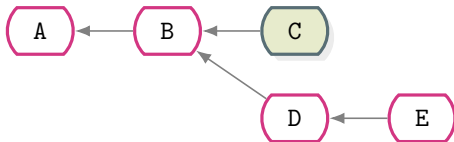
## Historique: les branches

La **branche** de la version  $v_i$  d'un historique est le sous-graphe composé de l'ensemble des versions accessibles depuis  $v_i$ .



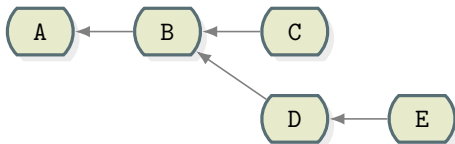
## Historique: les branches

La **branche** de la version  $v_i$  d'un historique est le sous-graphe composé de l'ensemble des versions accessibles depuis  $v_i$ .



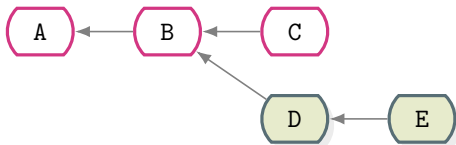
## Historique: la branche principale

La **branche principale** de l'historique est la branche issue de la dernière version stable.



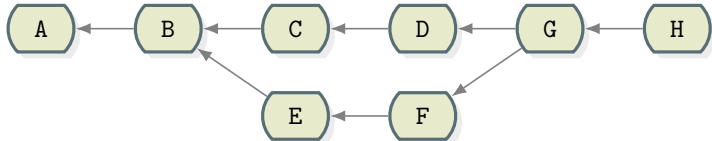
## Historique: la branche principale

La **branche principale** de l'historique est la branche issue de la dernière version stable.



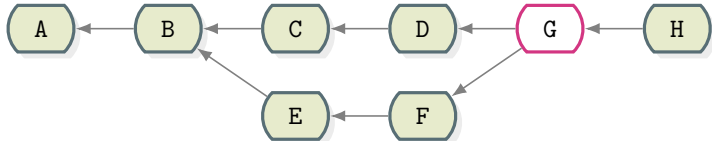
## Historique: les merges

On appelle **merge** toute version ayant un degré sortant strictement supérieur à 1. Cette version correspond alors à la fusion des **patches** de plusieurs branches.



## Historique: les merges

On appelle **merge** toute version ayant un degré sortant strictement supérieur à 1. Cette version correspond alors à la fusion des **patches** de plusieurs branches.



# Gestion des accès concurrents

## Gestion pessimiste

- Un seul contributeur à accès en écriture à un fichier
- Pas de conflits
- Pas pratique

## Gestion optimiste

- Chaque développeur peut modifier sa copie locale en parallèle
- Risques de conflits
  - Modifications concurrentes de la même zone de texte
- Tous les VCS actuels ont une approche optimiste



# Modèle centralisé/distribué

## Modèle centralisé

- Un serveur gère l'intégralité des version (le dépôt)
- Les utilisateurs y ajoutent leurs modifications
- Les utilisateurs y récupèrent les modifications des autres

# Modèle centralisé/distribué

## Modèle centralisé

- Un serveur gère l'intégralité des version (le dépôt)
- Les utilisateurs y ajoutent leurs modifications
- Les utilisateurs y récupèrent les modifications des autres

## Modèle distribué

- Chaque utilisateur possède un dépôt entier
- Les dépôts peuvent s'échanger des modifications

# Agenda

Introduction

**GIT**

Utilisation de Git

Les bonnes pratiques

Synchronisation avec des dépôts distants

# Avant de commencer ...

## Warning

Remarque introductive, présentation de GIT @ Google par Linus Torvalds, 2007.

*[Linus] is a guy who delights being cruel to people. His latest cruel act is to create a revision control system which is expressly designed to make you feel less intelligent than you thought you were. [...] So Linus is here today to explain to us why on earth he wrote a software tool which, eh, only he is smart enough to know how to use.*

Il existe des dizaines de documentations/tutoriels disponibles en ligne.

- La meilleure chose est d'apprendre par vous même.

# Fondé sur une fonction de hachage

## SHA-1

- Secure Hash Algorithm (cryptographie)
- Génère une empreinte des données d'entrée
  - ▶ Contenu du fichier
  - ▶ en-tête
- Propriétés:
  - ▶ Hash de 160 bits
  - ▶ Très faible probabilité de collision
- Identifie de manière unique chaque objet

## Exemple

```
$ echo a > toto
$ sha1sum toto
3f786850e387550fdab836ed7e6dc881de23001b  toto
$ echo b >> toto
05dec960e24d918b8a73a1c53bcbbaac2ee5c2e0  toto
```

# Les objets dans Git

- Blobs
- Tree
- Commit
- Tag

## Content-adressable file system

- Chaque objet est accessible à partir de sa clé.

# Blob

## Définition

On appelle **Blob**, l'élément de base qui permet de stocker le contenu d'un fichier.

- Chaque Blob est identifié à partir de manière unique par sa clé
- À chaque révision du fichier correspond un nouveau Blob
- Le Blob ne dépend pas du nom ou de l'emplacement :
  - ▶ Si un fichier est renommé, pas de nouveau Blob
  - ▶ Si un fichier est déplacé, pas de nouveau Blob
- Le contenu du Blob est compressé avec zlib. Il contient:
  - ▶ Le type d'objet (blob)
  - ▶ La taille du fichier initial
  - ▶ Le contenu du fichier

# Tree

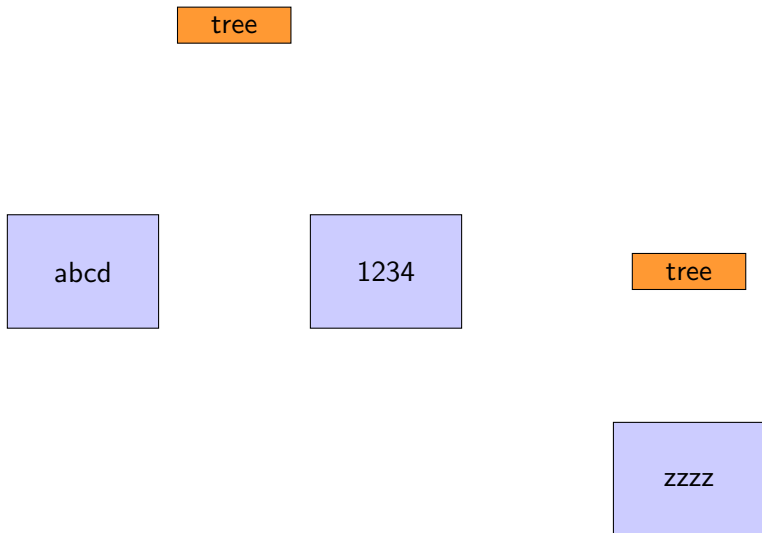
## Définition

Un **Tree** stocke la liste des fichiers d'un répertoire.

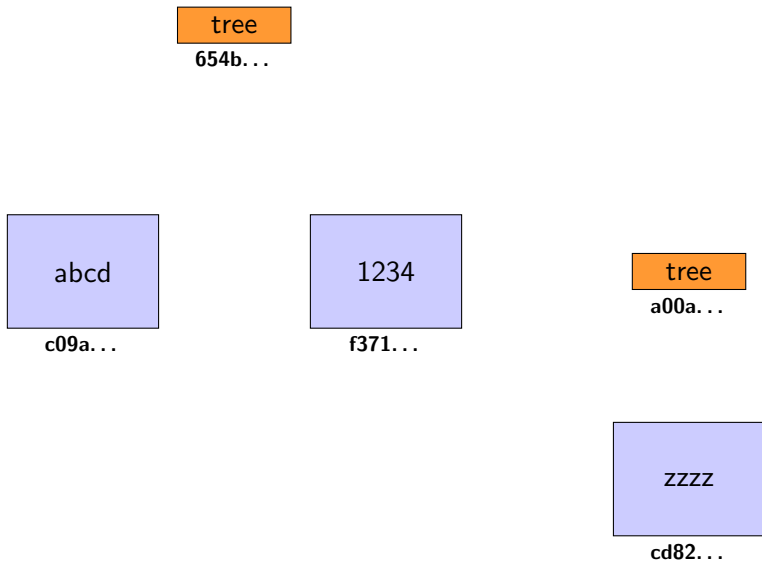
- Un Tree est un ensemble de pointeurs vers des Blobs et d'autres Trees.
- Un Tree associe un nom de fichier (resp. répertoire) à chacun des pointeurs de Blobs (resp. Trees).
- Un ensemble de Trees permet de décrire l'état d'une hiérarchie de dossiers à un moment donné.



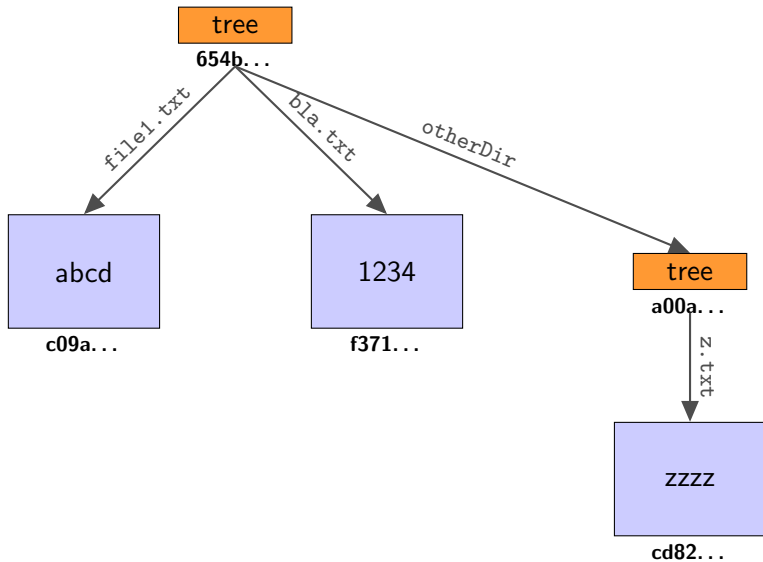
## Tree et Blob: Exemple



# Tree et Blob: Exemple



## Tree et Blob: Exemple



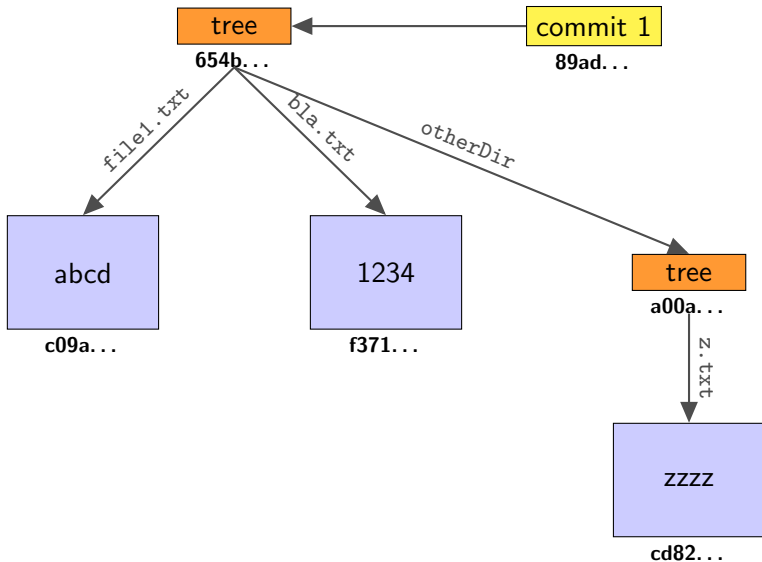
# Commit

## Définition

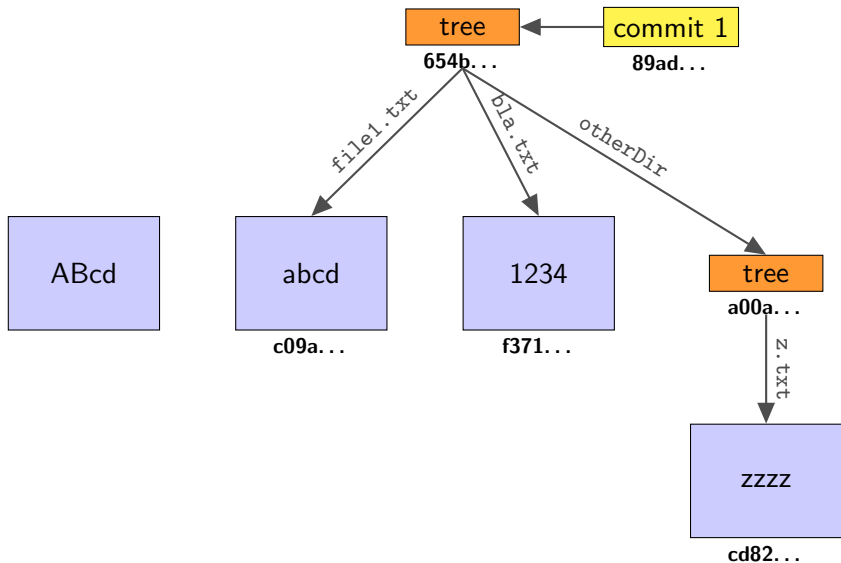
Un **Commit** stocke l'état d'une partie du dépôt à un instant donné.  
Il contient :

- Un pointeur vers un Tree (arbre racine) dont on souhaite sauver l'état.
- Un pointeur vers un ou plusieurs autres Commits pour constituer un historique.
- Les informations sur l'auteur du Commit.
- Une description sous forme d'une chaîne de caractères.

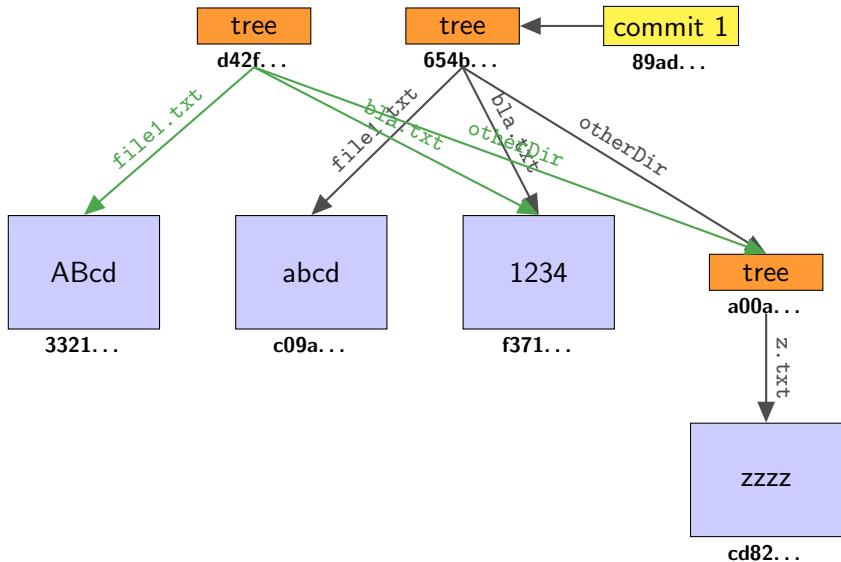
## Exemple avec Commit



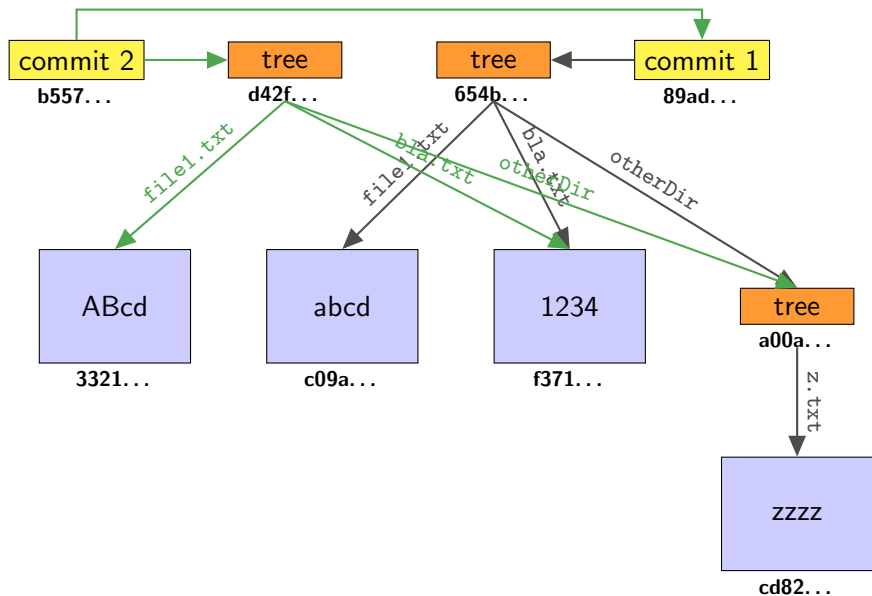
## Exemple avec Commit: modification de file1.txt



## Exemple avec Commit: modification de file1.txt



## Exemple avec Commit: modification de file1.txt





# Les Tags

## Définition

Un **Tag** permet d'identifier un des objets précédents à l'aide d'un nom.

- Il contient un pointeur vers un Blob, un Tree ou un Commit.

# Agenda

Introduction

GIT

Utilisation de Git

Les bonnes pratiques

Synchronisation avec des dépôts distants

# Les commandes

Git est un ensemble de commandes. Les commandes sont de la forme:

```
git commande options
```

## Exemple

```
git add file1.txt
```

# Création d'un dépôt

## Création d'un dépôt *serveur*

```
$ mkdir projet.git  
$ cd projet.git  
$ git --bare init
```

- Pas de répertoire `xxx/.git` mais directement un `xxx.git/`
- Ne contient pas les fichiers versionnés mais juste l'historique
- A cloner pour travailler dessus

# Création d'un dépôt

## Initialisation d'un dépôt

```
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/XX/XX/project.git
$ git push origin master
```

- Créer un répertoire local `myproject` pour stocker notre version du projet.
- Associer le dépôt local avec le dépôt distant
- Envoyer l'état initial du dépôt vers le serveur
- A partir de ce moment, tout le monde peut obtenir sa copie locale du dépôt en utilisant `git clone`

# Cloner un dépôt existant

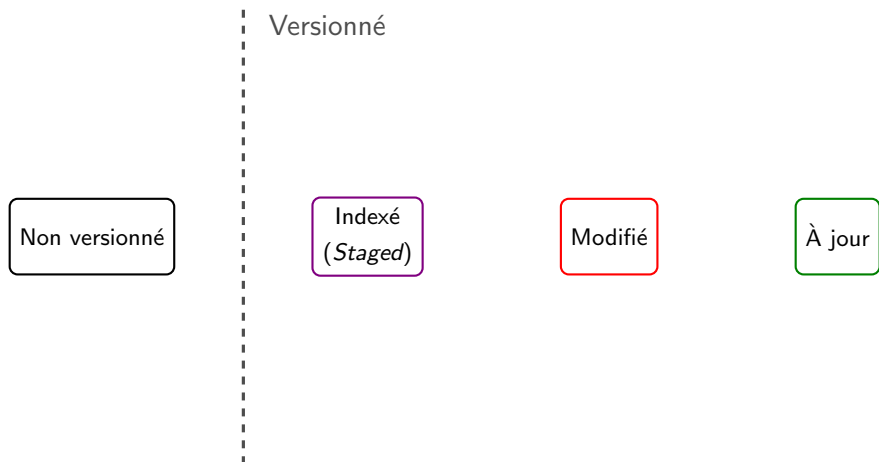
Très souvent, un dépôt existe déjà. On veut alors récupérer une copie de ce dépôt.

## Cloner un dépôt

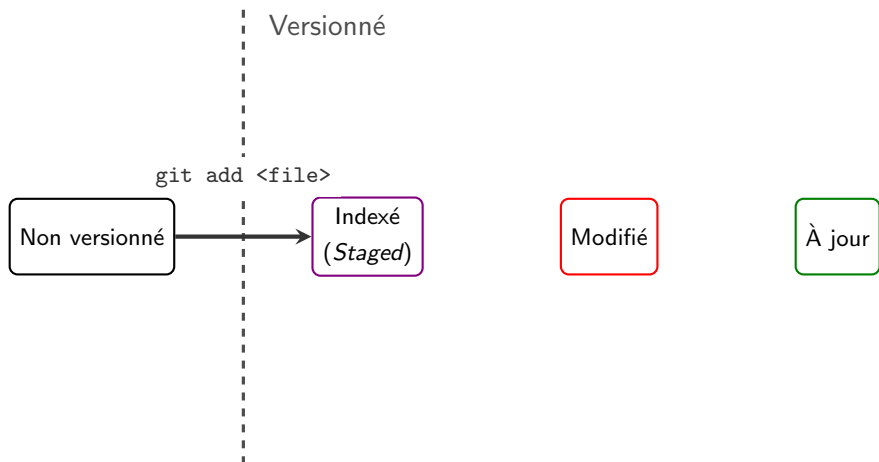
```
$ git clone URL
```

- Crée une copie locale du dépôt entier.
- L'URL peut être de la forme:
  - ▶ `file:///./myproject/project.git`
  - ▶ `http://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git`
  - ▶ `git://github.com/schacon/git.git`

# Le cycle de vie d'un fichier

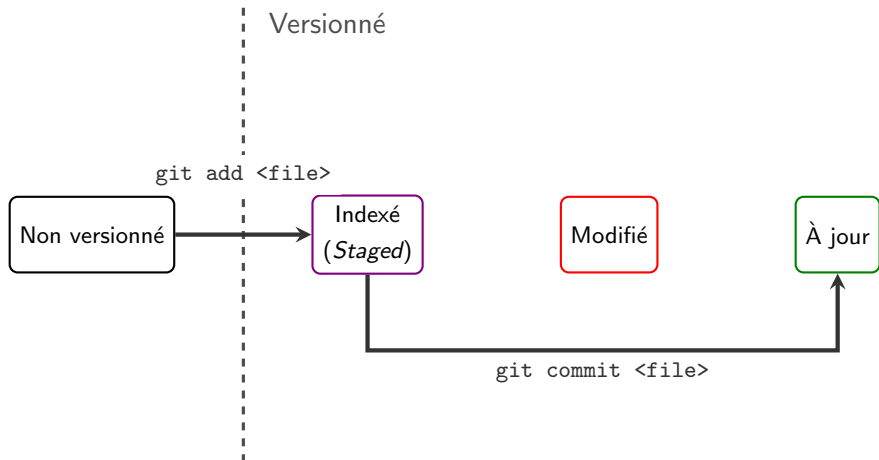


# Le cycle de vie d'un fichier

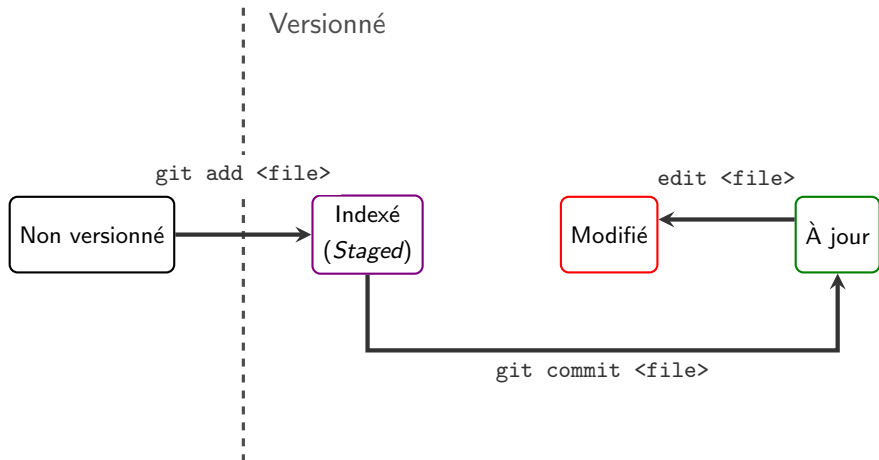




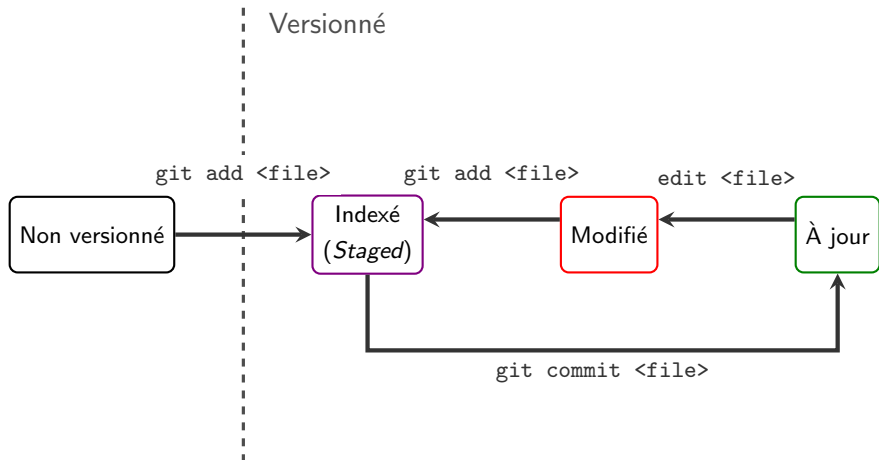
# Le cycle de vie d'un fichier



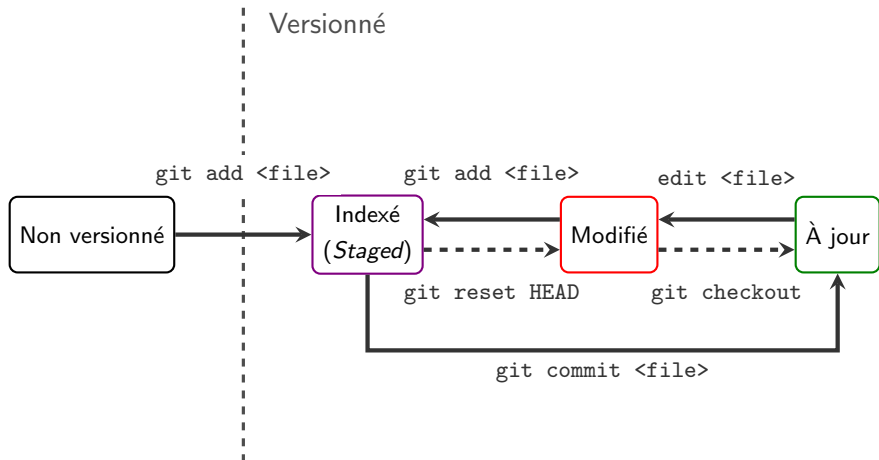
# Le cycle de vie d'un fichier



# Le cycle de vie d'un fichier



# Le cycle de vie d'un fichier



## Quelques commandes

**add** : Ajoute dans l'**index** un fichier à commiter dans son état actuel.

**commit** : enregistre dans le dépôt local les modifications qui ont été ajoutées dans l'**index** par une commande add

**reset HEAD** : supprime la référence d'un fichier de l'**index** ajouté par une commande add.

L'index est aussi appelé **staging area**.

Souvent on veut simplement commiter toutes les modifications en cours (= fichiers de l'index + fichiers modifiés):

```
$ git commit -a
```

# Exemple de commit

## Commit

```
$ echo "coucou" >hello.txt  
$ git add hello.txt  
$ git commit -m "description du commit"
```

En l'absence de message décrivant le commit, un fichier décrivant le commit est ouvert, vous invitant à compléter la description.

## Sélectionnez votre éditeur favori

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

# État courant de votre répertoire de travail

```
$ git status
```

- Permet de connaître l'état courant de l'index
  - Les modifications indexées
  - Les modifications non indexées
  - Les fichiers non versionnés
- Git vous indique même comment effectuer les actions principales
  - Commiter
  - Ajouter des fichiers à l'index
  - Annuler des modifications

# Les branches

## Rappel

La **branche** de la version  $v_i$  d'un historique est le sous-graphe composé de l'ensemble des versions accessibles depuis  $v_i$ .

## Dans Git

- Une branche est un pointeur sur un commit
- Chaque commit pointe vers son prédécesseur
- La variable **HEAD** pointe sur la branche sur laquelle on travaille actuellement.



# Branche : les commandes.

**branch** : liste les branches avec une **\*** pour la branche active.

**branch <nom>** : crée une nouvelle branche <nom>.

**branch -m** : permet de renommer une branche.

**branch -d** : permet de supprimer une branche.

**checkout** : change (ou/et crée) de branche active.

**show-branch** : affiche les branches et leurs commits.

## Exemple

```
$ git branch
* master
$ git branch maBranche
$ git branch
  maBranche
* master
$ git checkout maBranche
$ git branch
* maBranche
  master
```

# Les merges

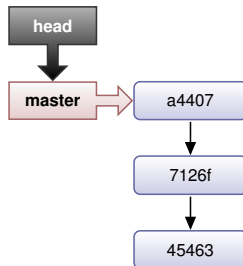
```
$ git checkout brancheDestination
```

```
$ git merge brangeSource
```

- Créé un commit qui a pour parent les deux branches
- La branche courante avance à ce commit
- La source ne bouge pas, mais devient un fils du nouveau commit

# Branche : structure interne des commits.

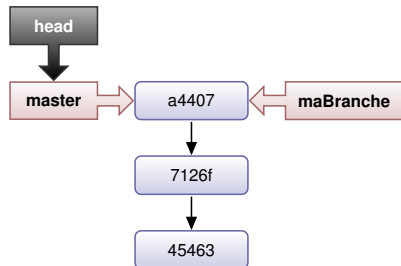
```
ls  
foo.txt dir
```



# Branche : structure interne des commits.

```
ls
  foo.txt dir

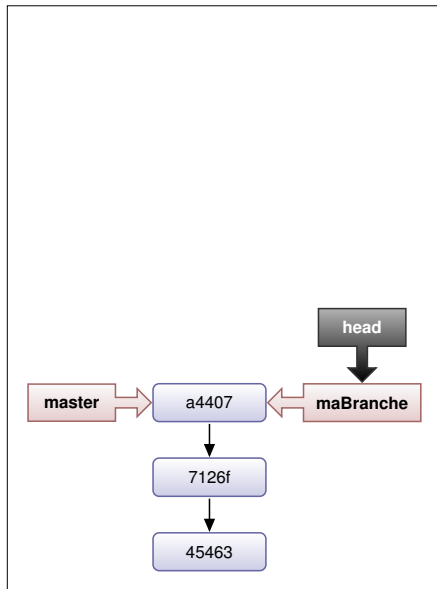
git branch maBranche
```



# Branche : structure interne des commits.

```
ls
  foo.txt dir

git branch maBranche
git checkout maBranche
```



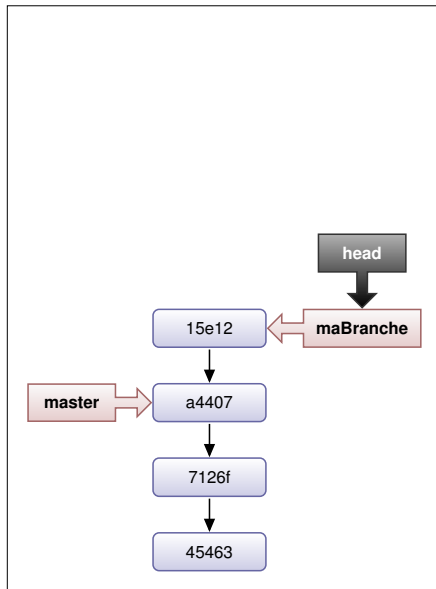
# Branche : structure interne des commits.

```
ls
foo.txt dir

git branch maBranche

git checkout maBranche

touch fichier1.txt
ls
dir fichier1.txt foo.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"
```



# Branche : structure interne des commits.

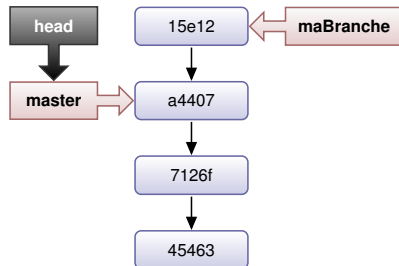
```
ls
foo.txt dir

git branch maBranche

git checkout maBranche

touch fichier1.txt
ls
dir fichier1.txt foo.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"

git checkout master
ls
dir foo.txt
```



# Branche : structure interne des commits.

```
ls
  foo.txt dir

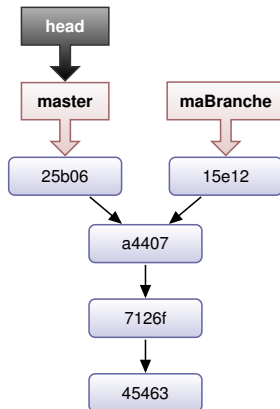
git branch maBranche

git checkout maBranche

touch fichier1.txt
ls
  dir fichier1.txt foo.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"

git checkout master
ls
  dir foo.txt

touch fichier2.txt
git add fichier2.txt
git commit -m "Add fichier2.txt"
```

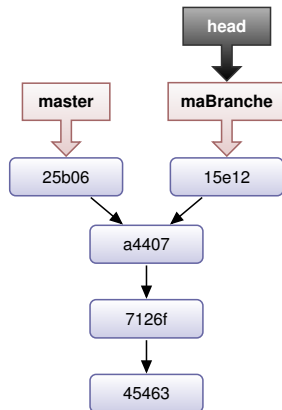




# Merge : exemple sur deux branches distinctes.

```
ls
  dir fichier2.txt foo.txt

git checkout maBranche
ls
  dir fichier1.txt foo.txt
```



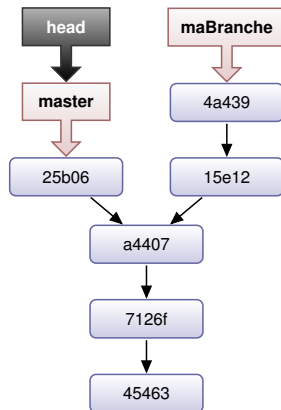
# Merge : exemple sur deux branches distinctes.

```
ls
  dir fichier2.txt foo.txt

git checkout maBranche
ls
  dir fichier1.txt foo.txt

echo "titi" > fichier1.txt
git commit -am "Modif
fichier1.txt"

git checkout master
```



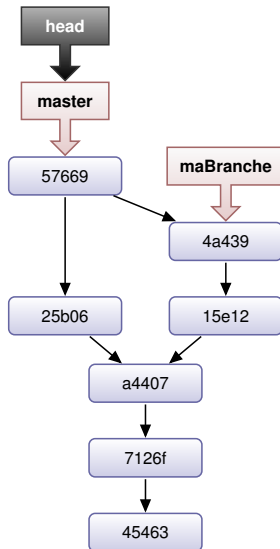
# Merge : exemple sur deux branches distinctes.

```
ls
  dir fichier2.txt foo.txt

git checkout maBranche
ls
  dir fichier1.txt foo.txt

echo "titi" > fichier1.txt
git commit -am "Modif
fichier1.txt"

git checkout master
git merge maBranche
ls
  dir fichier1.txt fichier2.txt
  foo.txt
```



# Merge : exemple sur deux branches distinctes.

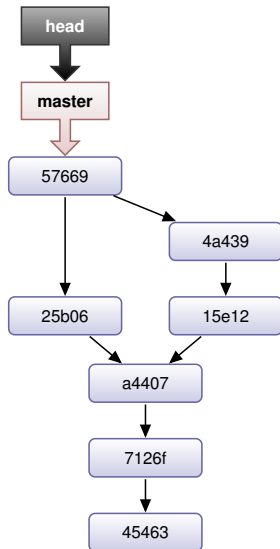
```
ls
  dir fichier2.txt foo.txt

git checkout maBranche
ls
  dir fichier1.txt foo.txt

echo "titi" > fichier1.txt
git commit -am "Modif
fichier1.txt"

git checkout master
git merge maBranche
ls
  dir fichier1.txt fichier2.txt
  foo.txt

git branch -d maBranche
```



# Identifier des commits

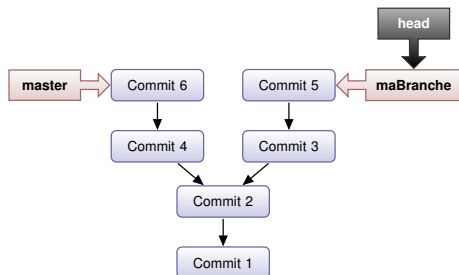
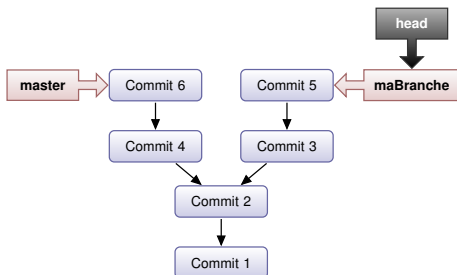
- Le dernier commit de la branche courante ou d'une autre
  - HEAD
  - maBranche
- L'avant dernier et les précédents
  - HEAD<sup>^</sup>, mabranche<sup>^^</sup>, ...
  - HEAD<sup>~3</sup>, mabranche<sup>~12</sup>, ...
- D'autres manières
  - HEAD@yesterday
  - mabranche@June.1

# Rebase

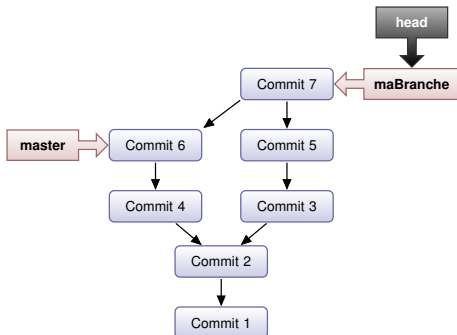
- Autre manière de fusionner 2 branches
- Fusionne entièrement la branche source dans la branche destination
- Permet de simplifier l'historique
- **Ne jamais rebaser des commits qui ont déjà été poussés sur un dépôt public**

Pour plus de détails, regardez par vous même.

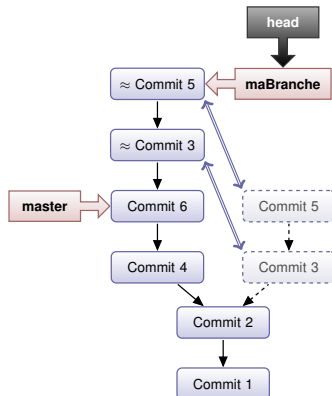
# Rebase vs Merge.



# Rebase vs Merge.



```
git checkout maBranche  
git merge master
```



```
git checkout maBranche  
git rebase master
```



# Revenir en arrière

## Cas de modifications non commitées

- Restaurer mon fichier dans la dernière version de l'index:  
`git checkout -- monfichier`
  - ▶ Utilisation de "--": spécifie que `monfichier` désigne un fichier et pas une branche.
- Restaurer mon fichier dans la dernière version commitée:  
`git checkout HEAD monfichier`
- Restaurer tous les fichiers du répertoire courant:  
`git checkout .`

# Revenir en arrière

## Cas de modifications commitées

Trois commandes disponibles:

`amend` : modifier le dernier commit

- Ajoute des fichiers au commit
- Changer le message de commit

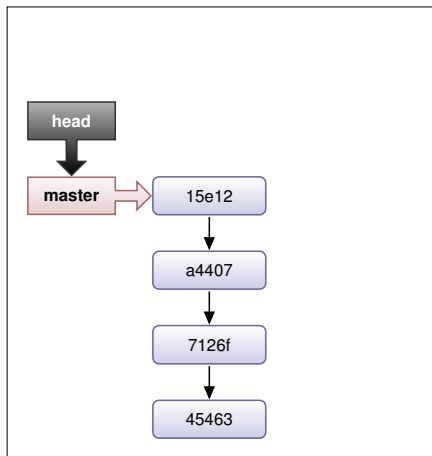
`revert` : annuler un commit par un autre commit

`reset` : rétablir la situation d'un ancien commit

**Si l'erreur a été rendue publique, la seule bonne pratique est `revert`.**

# Amend : modification du dernier commit.

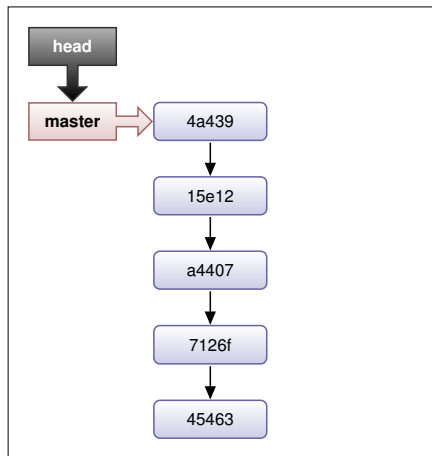
```
ls  
foo.txt dir
```



# Amend : modification du dernier commit.

```
ls
  foo.txt dir

touch bar.txt
git commit -m "Ajou d'un fichier."
```



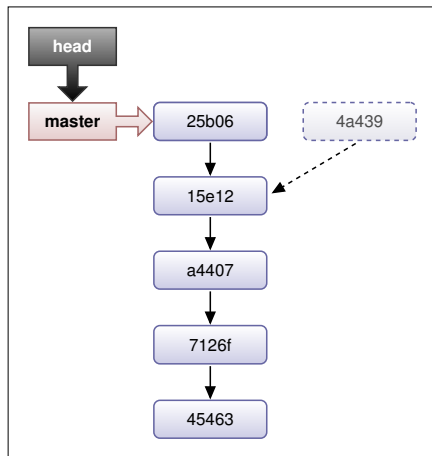
# Amend : modification du dernier commit.

```
ls
  foo.txt dir

touch bar.txt
git commit -m "Ajou d'un fichier."

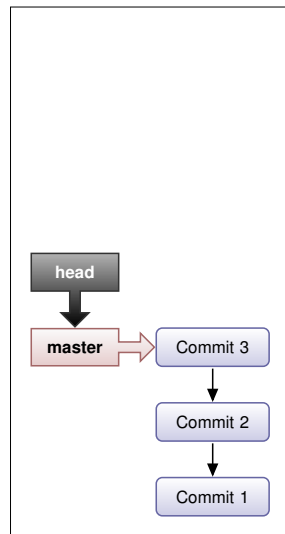
git add bar.txt

git commit --amend -m "Ajout d'un
fichier."
```



# Git revert : annulation par commit.

```
git branch master
cat fichier1.txt
  Premiere version de F1
cat fichier2.txt
  Premiere version de F2
```

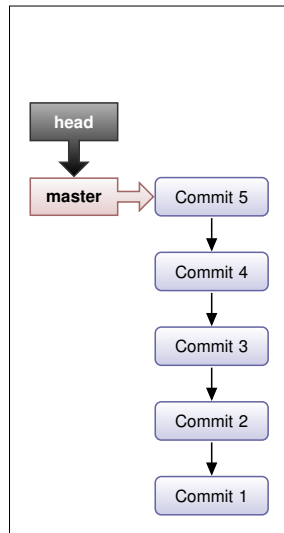


# Git revert : annulation par commit.

```
git branch master
cat fichier1.txt
  Premiere version de F1
cat fichier2.txt
  Premiere version de F2

echo "Deuxieme version de F1" > fichier1.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"

echo "Deuxieme version de F2" > fichier2.txt
git add fichier2.txt
git commit -m "Add fichier2.txt"
```



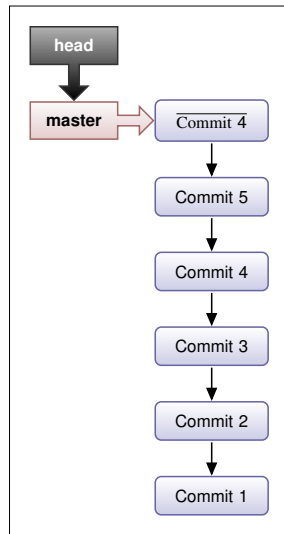
# Git revert : annulation par commit.

```
git branch master
cat fichier1.txt
  Premiere version de F1
cat fichier2.txt
  Premiere version de F2

echo "Deuxieme version de F1" > fichier1.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"

echo "Deuxieme version de F2" > fichier2.txt
git add fichier2.txt
git commit -m "Add fichier2.txt"

git revert HEAD^
cat fichier1.txt
  Premiere version de F1
cat fichier2.txt
  Deuxieme version de F2
```





# La commande reset

- Annuler des ajouts dans l'index

```
git reset monfichier
```

- Restaurer un ancien commit (mais en conservant toutes les modifications des fichiers et l'index)

```
git reset --soft commitID
```

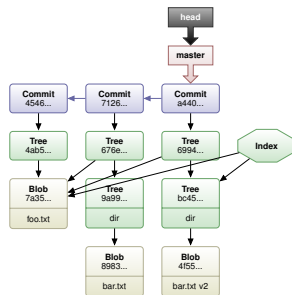
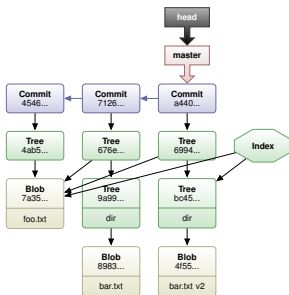
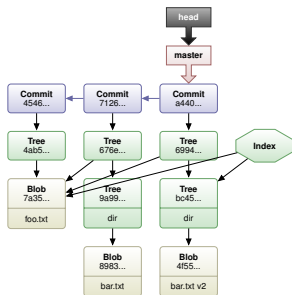
- Restaurer un ancien commit et l'index (mais en conservant toutes les modifications des fichiers)

```
git reset commitID
```

- Restaurer un ancien commit, l'index, et le contenu des fichiers correspondants

```
git reset --hard commitID
```

# Les différents type de reset.

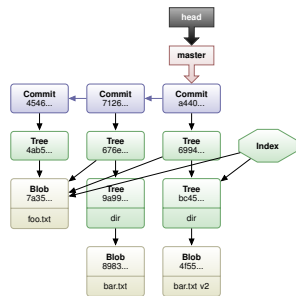
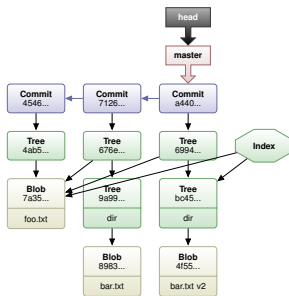
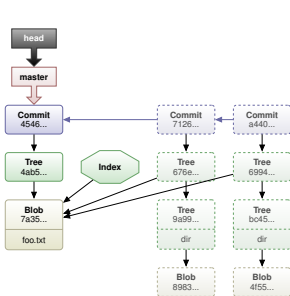


```
ls -R
.: foo.txt
dir: bar.txt
```

```
ls -R
.: foo.txt
dir: bar.txt
```

```
ls -R
.: foo.txt
dir: bar.txt
```

# Les différents type de reset.

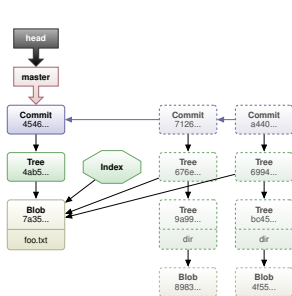


```
ls -R
.: foo.txt
dir: bar.txt
git reset --hard 4546
ls -R
.: foo.txt
```

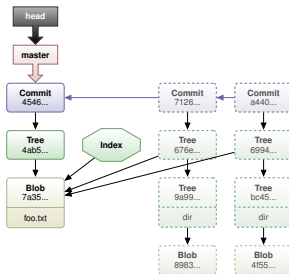
```
ls -R
.: foo.txt
dir: bar.txt
```

```
ls -R
.: foo.txt
dir: bar.txt
```

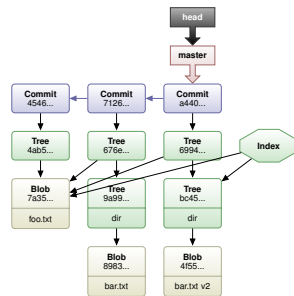
# Les différents type de reset.



```
ls -R
.: foo.txt
dir: bar.txt
git reset --hard 4546
ls -R
.: foo.txt
```

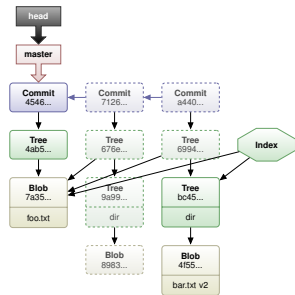
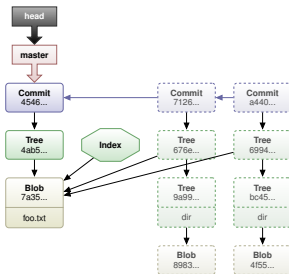
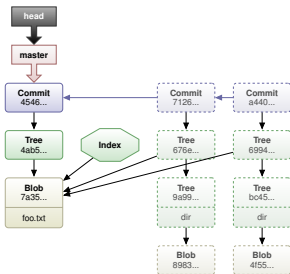


```
ls -R
.: foo.txt
dir: bar.txt
git reset 4546
ls -R
.: foo.txt
dir: bar.txt
```



```
ls -R
.: foo.txt
dir: bar.txt
```

# Les différents type de reset.



```
ls -R
.: foo.txt
dir: bar.txt
git reset --hard 4546
ls -R
.: foo.txt
```

```
ls -R
.: foo.txt
dir: bar.txt
git reset 4546
ls -R
.: foo.txt
dir: bar.txt
```

```
ls -R
.: foo.txt
dir: bar.txt
git reset --soft 4546
ls -R
.: foo.txt
dir: bar.txt
```

# Consulter l'historique des commits

Affiche l'historique des commits en remontant à partir de `commitID`.

```
git log commitID
```

- Par défaut, `commitID` est HEAD

De multiples possibilités. On peut voir:

- le log entre 2 versions
- le log d'un fichier
- le log sur une durée
- ...

# Consulter des changements

Afficher les détails sur un commit:

```
git show commitID  
git show commitID -- monfichier monrepertoire
```

Afficher les différences entre des versions:

```
git diff commitID1..commitID2 -- monfichier monrepertoire
```

Savoir qui a modifié un fichier (voir une ligne):

```
git blame file.txt  
git blame L80,+20 file.txt
```

# Agenda

Introduction

GIT

Utilisation de Git

Les bonnes pratiques

Synchronisation avec des dépôts distants



# Configurer git

- Commencer par renseigner son nom et email

```
$ git config --global user.name "Thomas Ropars"
$ git config --global user.email thomas.ropars@imag.fr
```

# Modifications d'un historique public

## Règle générale

Ne jamais réécrire un historique publiquement accessible

- Attention à l'utilisation des commandes `rebase` et `reset`
- Ne pas utiliser `push --force` à moins de vraiment savoir ce que vous faites
- Nettoyer son historique avant de le pousser si confus

# Ignorer des fichiers: gitignore

`gitignore` spécifie les fichiers non versionnés que git doit ignorer:

- Les fichiers déjà versionnés ne sont pas affectés.
- Définition à l'aide de *patterns*

Les fichiers *gitignore*:

- Des fichiers `.gitignore` peuvent être placés n'importe où dans la hiérarchie.
  - ▶ Les règles des fichiers plus bas dans la hiérarchie se substituent aux règles définies plus haut.
    - Priorités: Même répertoire, parent, ..., racine du dépôt
  - ▶ Ces fichiers `.gitignore` sont à ajouter au dépôt
- `$GIT_DIR/info/exclude`
  - ▶ Règles spécifiques à un utilisateur
- Fichier spécifié par l'option de configuration `core.excludesFile` dans `(homedir)/.gitconfig`

# Ignorer des fichiers: gitignore

## Définir des règles

```
# a comment - this is ignored
# no .a files
*.a
# but do track lib.a, even though you're ignoring .a files above
!lib.a
# only ignore the root TODO file, not subdir/TODO
/TODO
# ignore all files in the build/ directory
build/
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
```

# Messages de commit

## Le plus important

### Décrire quoi et pourquoi et pas comment

- Ne pas décrire les modifications qui sont faites (informations disponibles avec un diff)
- Décrire les fonctionnalités ajoutées

## Exemple

- Bad: *Modifie la fonction  $f$  pour tester la variable  $a$*
- Good: *Vérifie les droits de l'utilisateur avant d'exécuter l'action  $X$*

# Messages de commit

## Bonnes pratiques pour le format des messages

- Séparer le sujet du corps du message par une ligne vide
  - ▶ Adapté à l'affichage des logs (`git log --oneline; git shortlog`)
- Utiliser 50 caractères max pour le sujet
  - ▶ Pas une limite réelle
  - ▶ Bon conseil à suivre pour conserver une historique lisible
- Commencer le sujet par une majuscule et ne pas terminer par un point
- Formuler les messages au présent

# Interface graphique

Quelque soit votre environnement de travail, il existe des interfaces graphiques pour gérer vos projets:

- Avec git:
  - ▶ **gitk**: Visualisation de l'historique
  - ▶ **git-gui**: Interface pour la création de commits
- Linux: **gitg**, **giggle**, ...
- Windows, Mac: look here  
<https://git-scm.com/download/guis>
- Eclipse: EGit, ...

# Agenda

Introduction

GIT

Utilisation de Git

Les bonnes pratiques

Synchronisation avec des dépôts distants



# Rappel

## Modèle centralisé

- Un serveur gère l'intégralité des version (le dépôt)
- Les utilisateurs y ajoutent leurs modifications
- Les utilisateurs y récupèrent les modifications des autres

## Modèle distribué

- Chaque utilisateur possède un dépôt entier
- Les dépôts peuvent s'échanger des modifications

# Modèle distribué

- Chaque client a l'ensemble des fichiers dans son dépôt local
  - ▶ Travailler off-line
  - ▶ Changer de branche est rapide. On peut abuser des branches.
- Actions nécessitant un accès au dépôt distant
  - ▶ Mise à jour du dépôt local depuis l'extérieur
  - ▶ L'envoi d'informations
- Le client peut versionner en local.

Un projet décentralisé possède deux types de branches :

## Définition

*On appelle **branche distante**, une branche qui pointe sur des dépôts distants en lecture et/ou écriture. Ces dépôts distants peuvent être référencés par une ou plusieurs personnes.*

## Définition

*On appelle **branche locale**, une branche propre au dépôt local. Pour être envoyées, les données d'une telle branche doivent être fusionnées avec une branche distante.*

# Fetch, Merge et Pull

**fetch** : Importe les commits d'un dépôt distant dans le dépôt local.

- Utiliser `merge` pour importer les changements dans une branche locale

**pull** : Fusionne les changements d'un dépôt distant directement dans une branche locale.

- Équivalent d'un `fetch` suivi d'un `merge`.
- Peut être configuré pour utiliser `rebase` au lieu de `merge`
- `Fetch` permet d'observer les changements avant de les intégrer dans sa branche de travail

# Modèle distribué



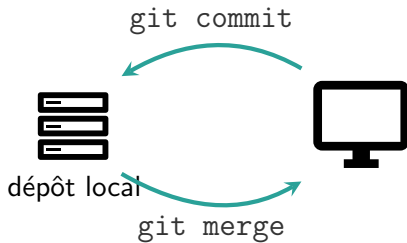
dépôt distant



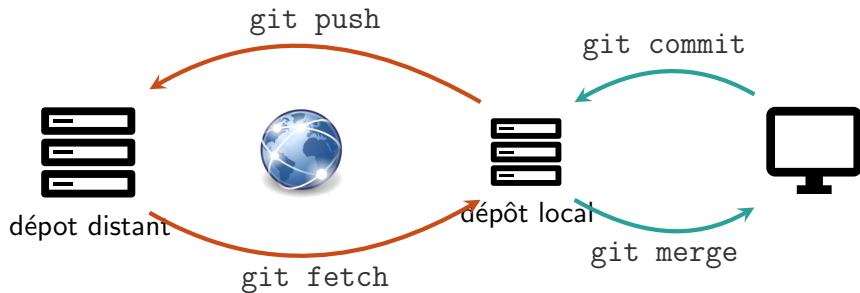
dépôt local



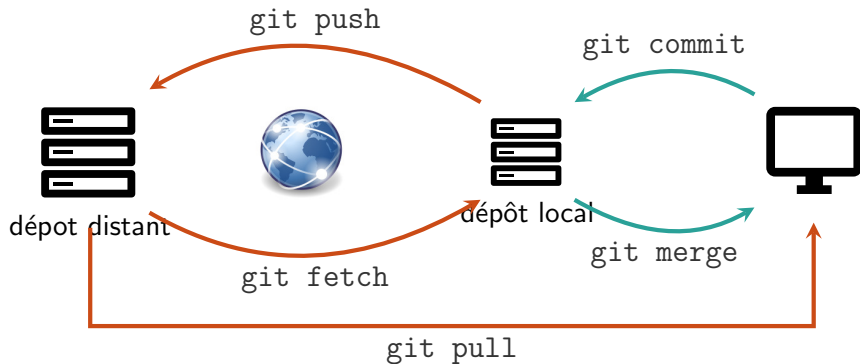
# Modèle distribué



# Modèle distribué

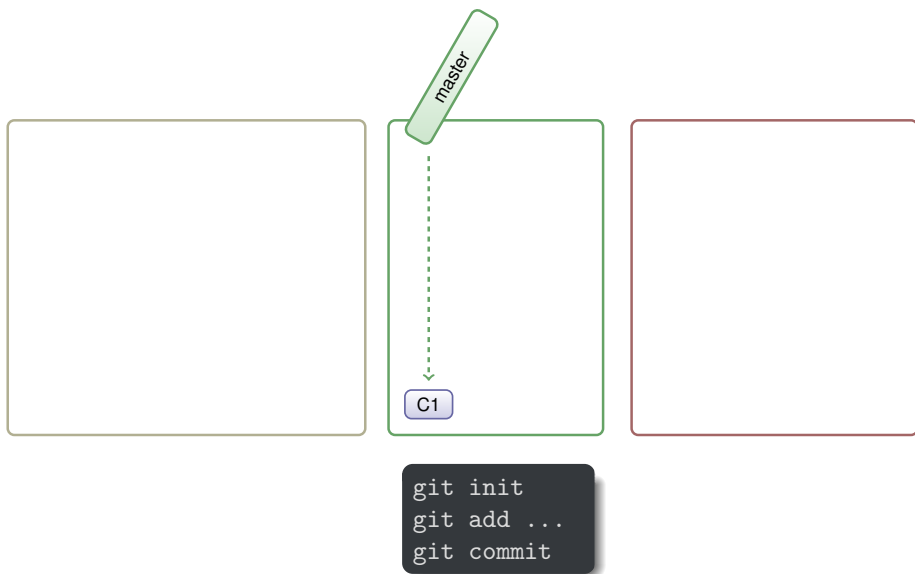


# Modèle distribué

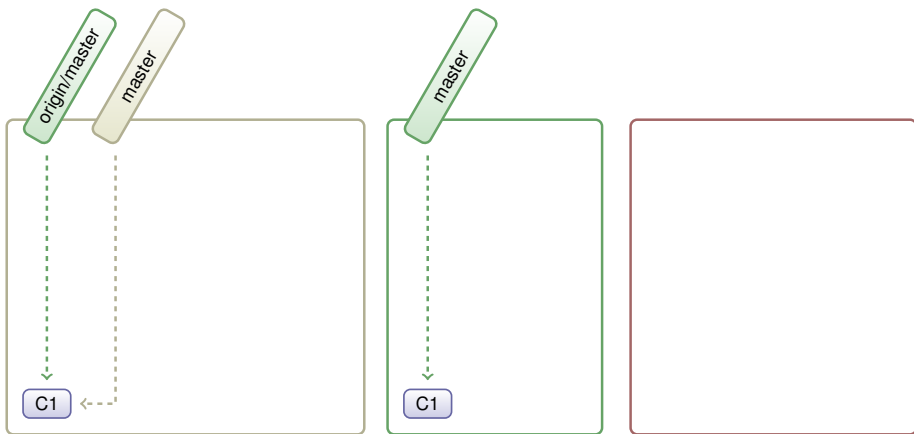




# Dépôts distants et gestion de la concurrence.

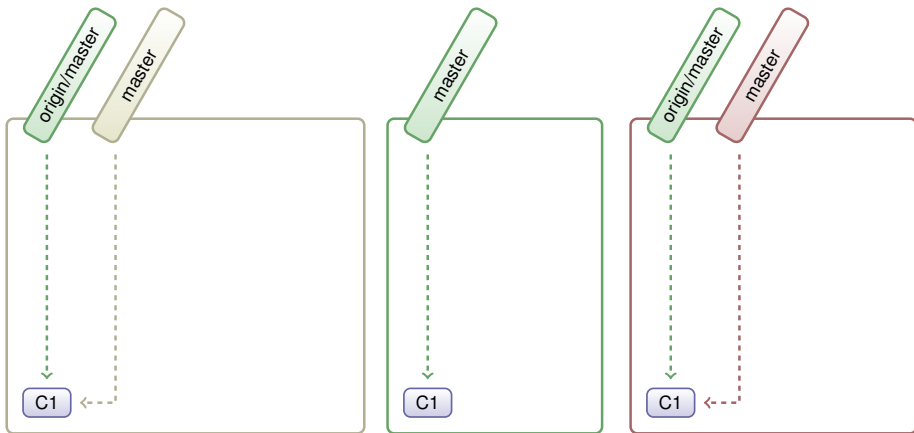


# Dépôts distants et gestion de la concurrence.



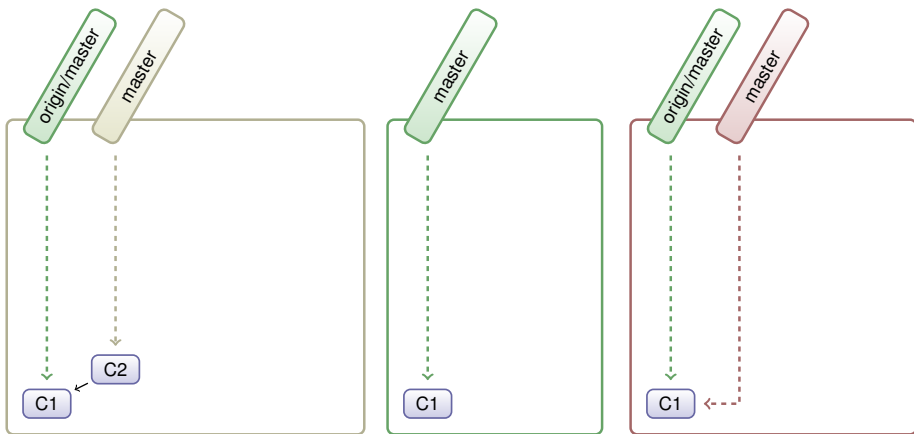
```
git clone <@serveur>
```

# Dépôts distants et gestion de la concurrence.



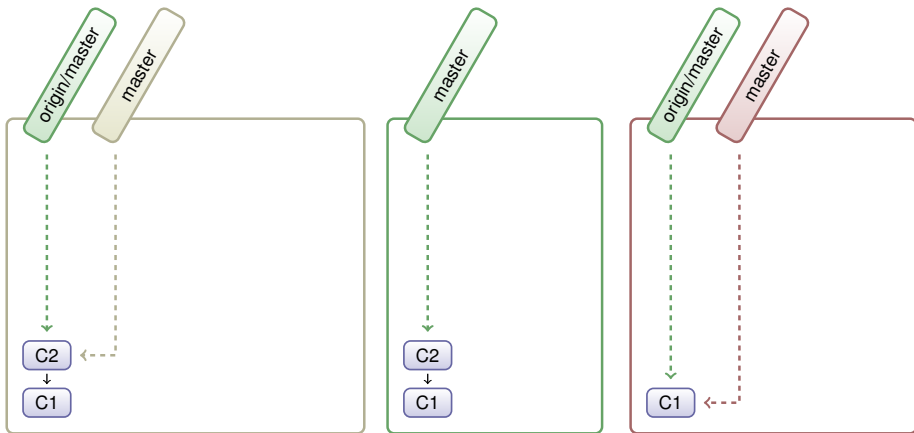
```
git clone <@serveur>
```

# Dépôts distants et gestion de la concurrence.



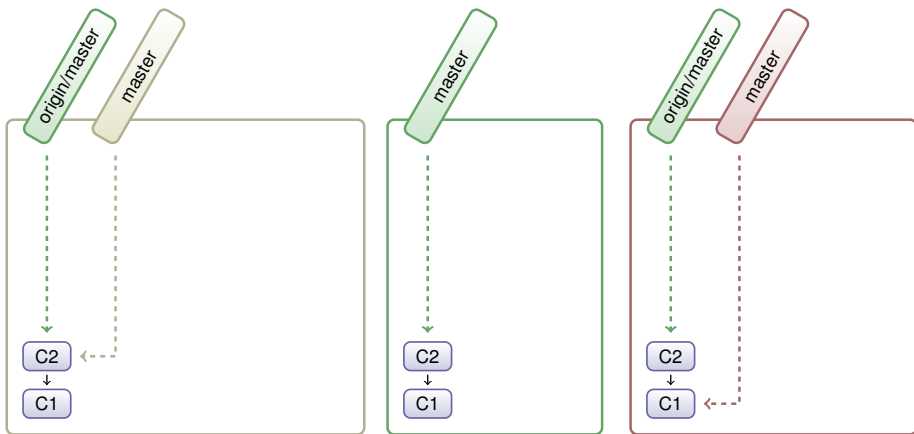
```
git commit
```

# Dépôts distants et gestion de la concurrence.



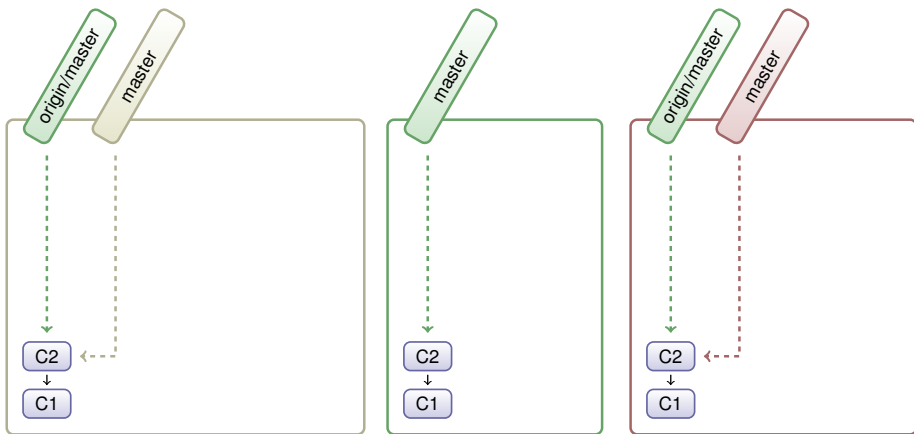
```
git push origin
```

# Dépôts distants et gestion de la concurrence.



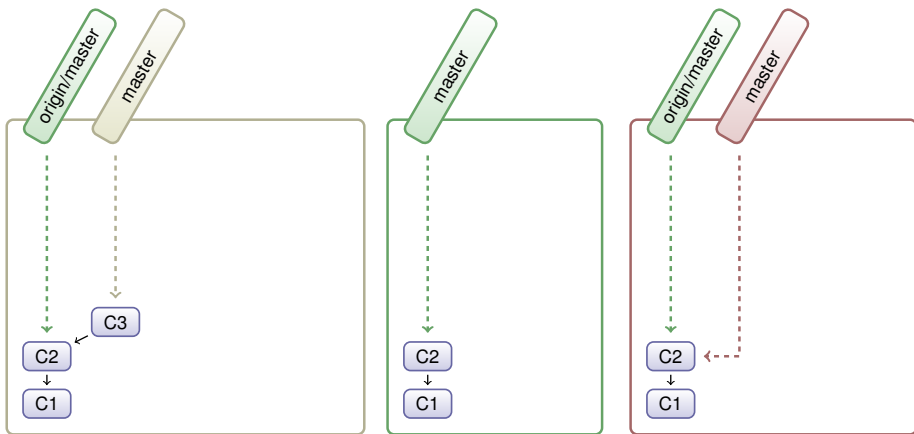
```
git fetch origin
```

# Dépôts distants et gestion de la concurrence.



```
git merge
```

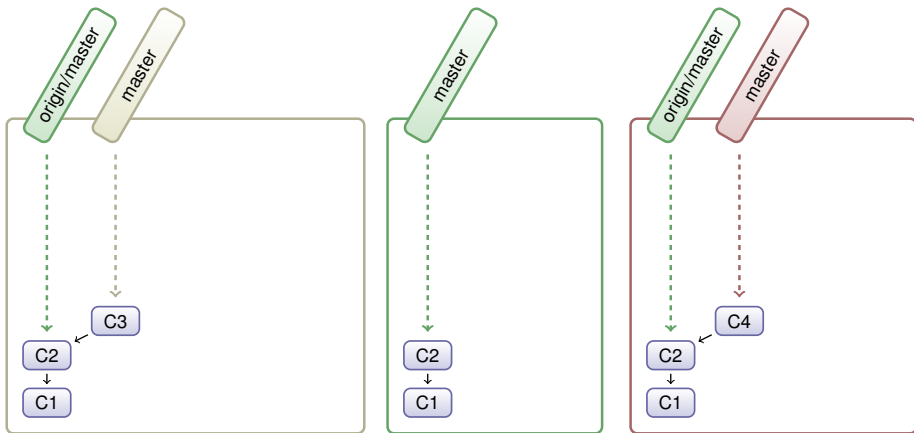
# Dépôts distants et gestion de la concurrence.



```
git commit
```

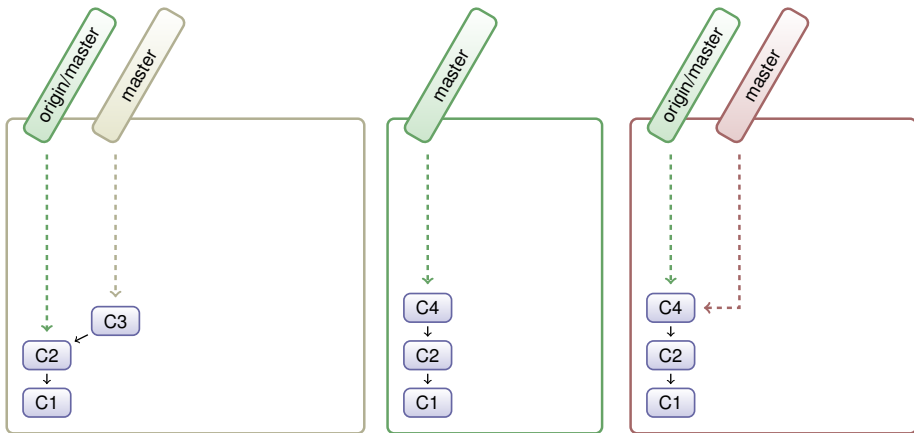


# Dépôts distants et gestion de la concurrence.



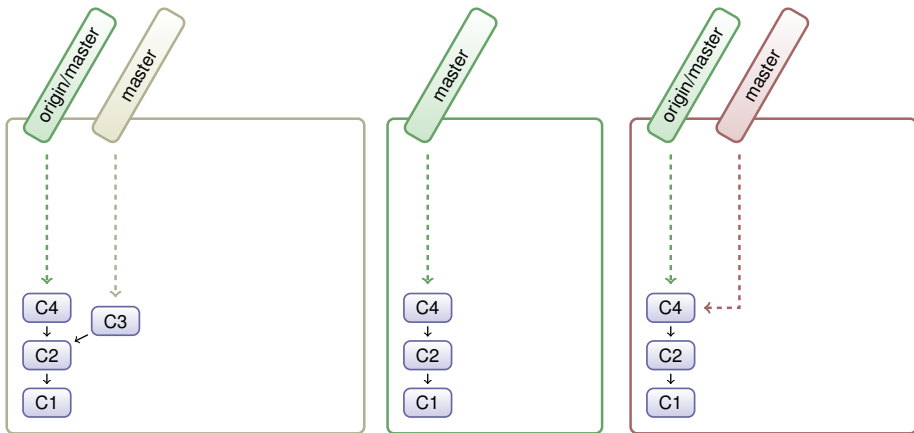
```
git commit
```

# Dépôts distants et gestion de la concurrence.



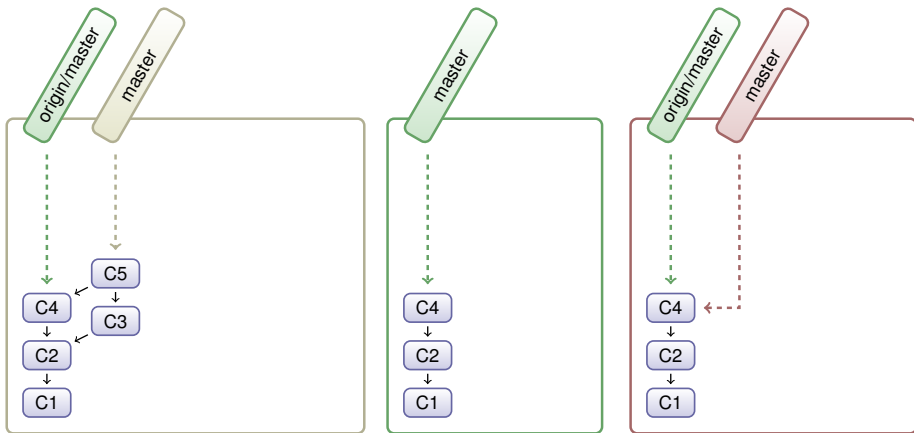
```
git push origin
```

# Dépôts distants et gestion de la concurrence.



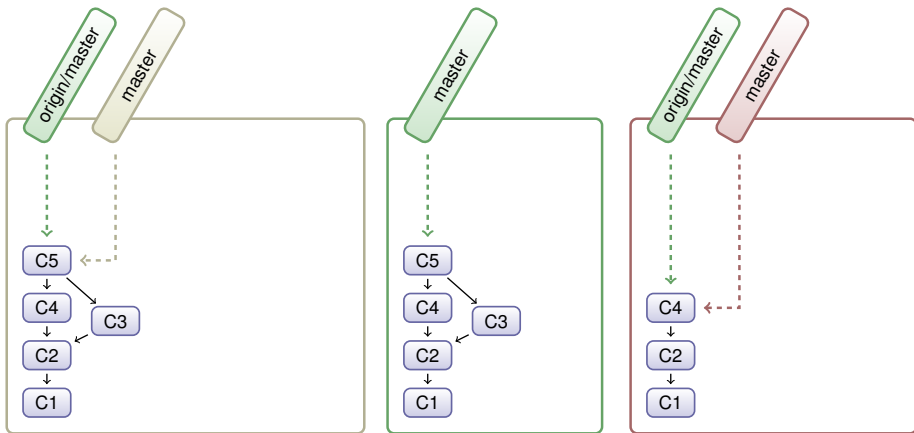
```
git fetch origin
```

# Dépôts distants et gestion de la concurrence.



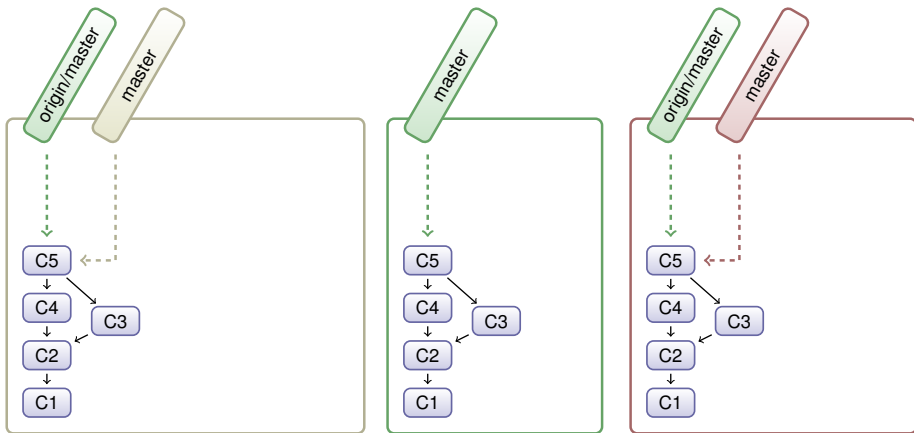
```
git merge origin
```

# Dépôts distants et gestion de la concurrence.



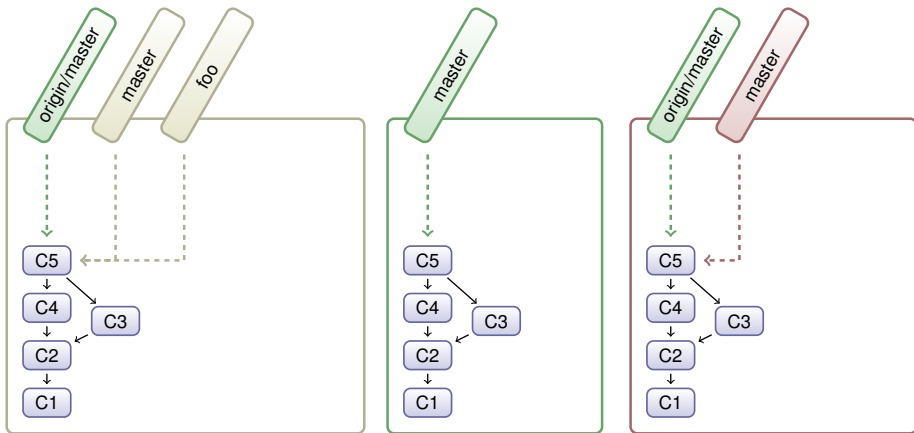
```
git push origin
```

# Dépôts distants et gestion de la concurrence.



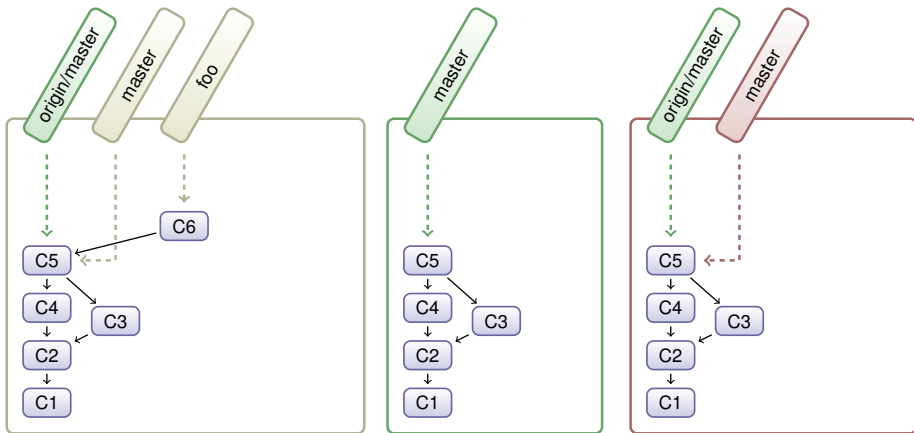
```
git pull origin
```

# Dépôts distants et gestion de la concurrence.



```
git checkout -b foo
```

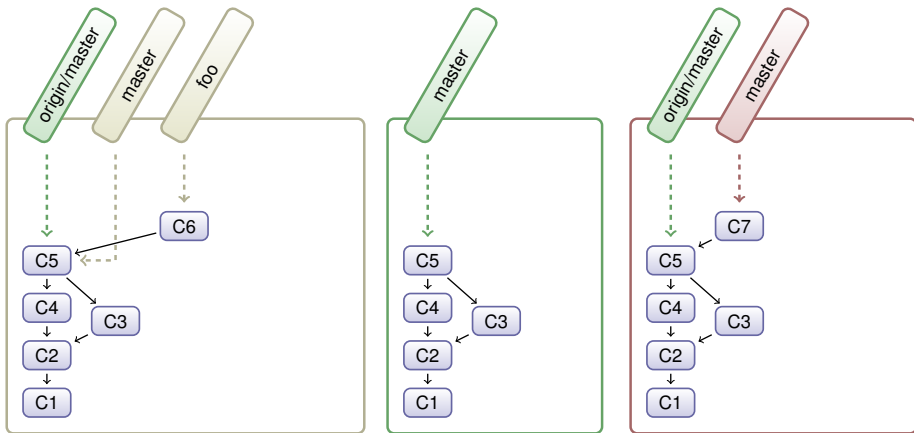
# Dépôts distants et gestion de la concurrence.



```
git commit
```

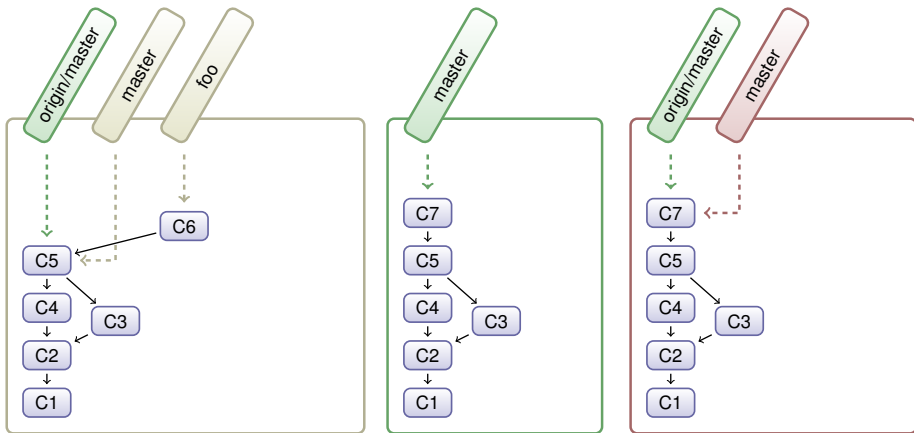


# Dépôts distants et gestion de la concurrence.



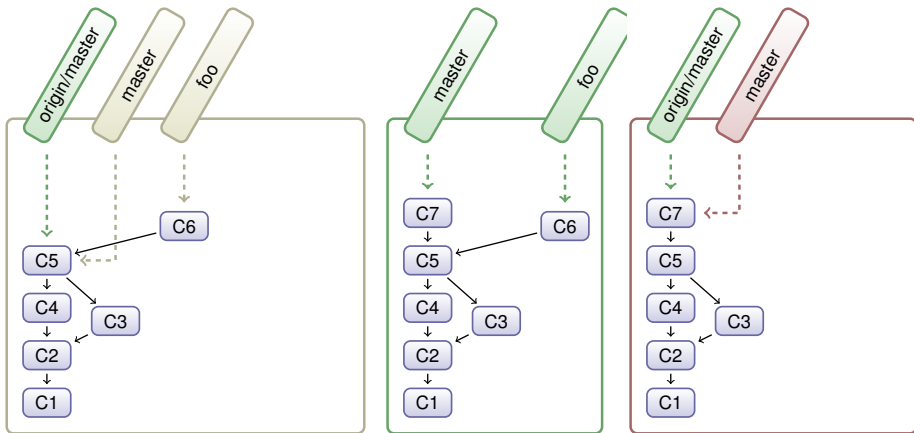
```
git commit
```

# Dépôts distants et gestion de la concurrence.



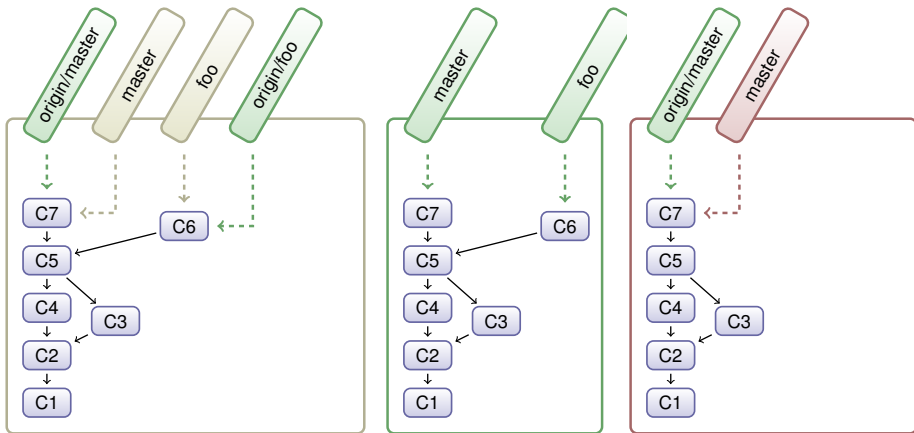
```
git push origin
```

# Dépôts distants et gestion de la concurrence.



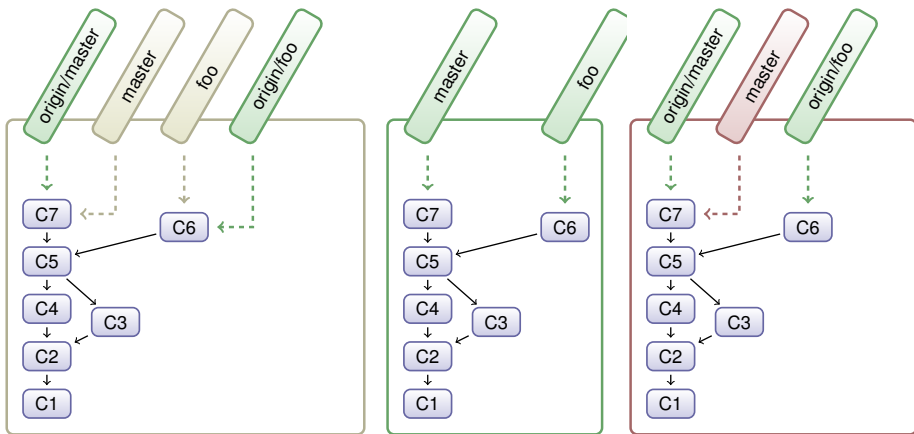
```
git push origin foo
```

# Dépôts distants et gestion de la concurrence.



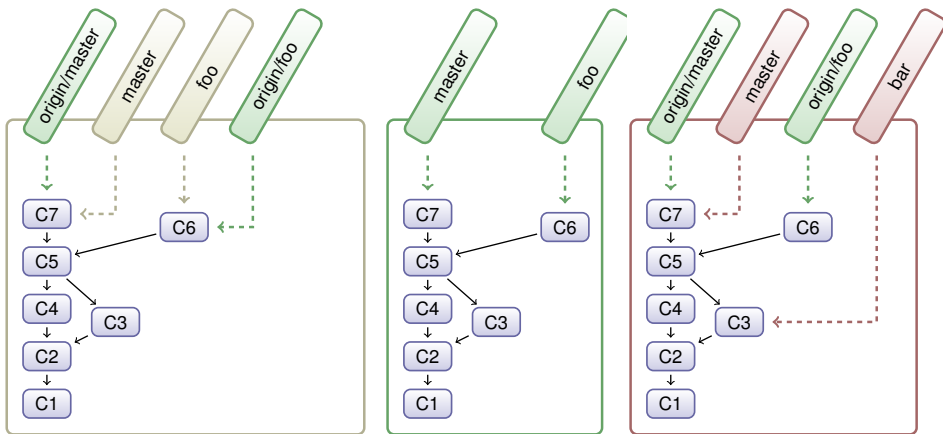
```
git pull origin
```

# Dépôts distants et gestion de la concurrence.



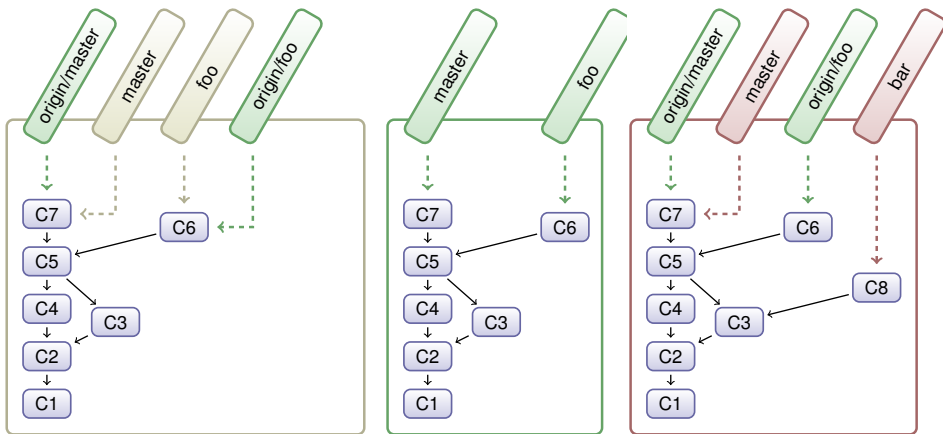
```
git pull origin
```

# Dépôts distants et gestion de la concurrence.



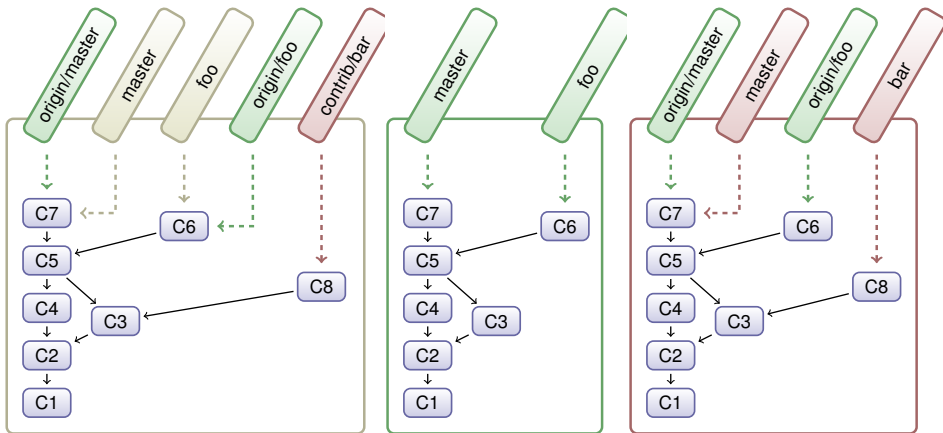
```
git branch bar  
git checkout bar
```

# Dépôts distants et gestion de la concurrence.



```
git commit
```

# Dépôts distants et gestion de la concurrence.



```
git remote add contrib git://...  
git fetch contrib/bar
```

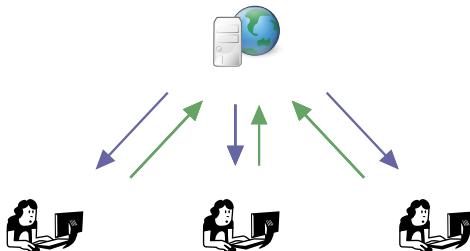


# Modèles de coopération

- Un dépôt centralisé (à la SVN) – mais avec les avantages des dépôts locaux
- Un modèle complètement distribué: Un dépôt pour chaque développeur, chacun se synchronise chez les autres
- Le modèle avec un gestionnaire d'intégration
- Le modèle dictateur et ses lieutenants (noyau linux)

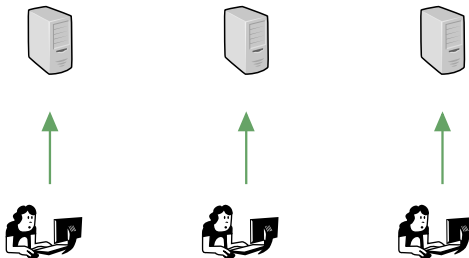
# Modèle avec dépôt centralisé

*"Centralized Workflow."*



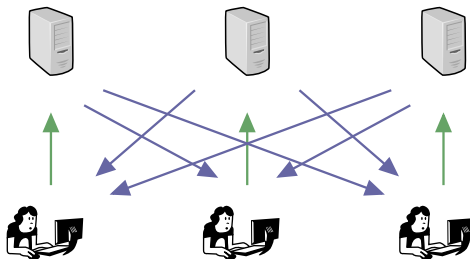
# Modèle décentralisé avec dépôts publics.

*"Cooperative and Decentralized Workflow."*



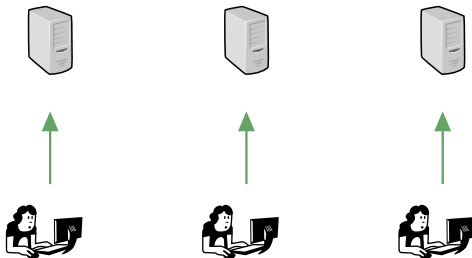
# Modèle décentralisé avec dépôts publics.

*"Cooperative and Decentralized Workflow."*



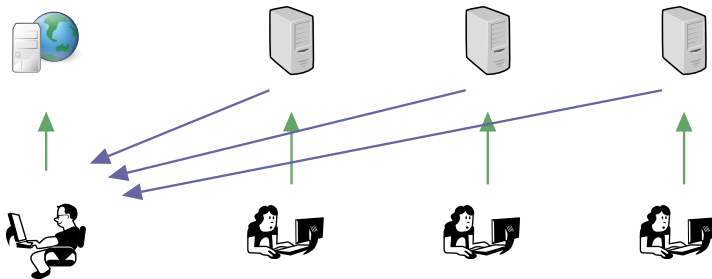
# Modèle avec manager de dépôt

*"Integration-Manager Workflow."*



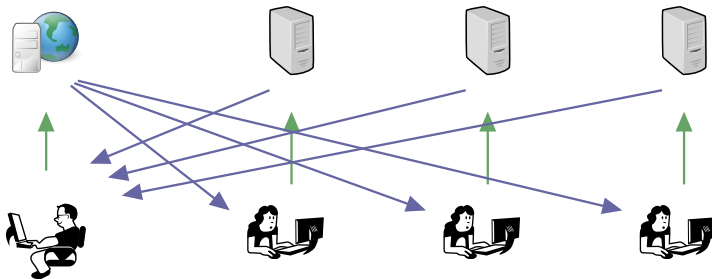
# Modèle avec manager de dépôt

*"Integration-Manager Workflow."*



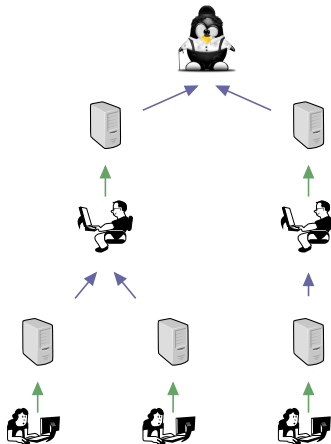
# Modèle avec manager de dépôt

*"Integration-Manager Workflow."*



# Modèle avec dictateur et lieutenants.

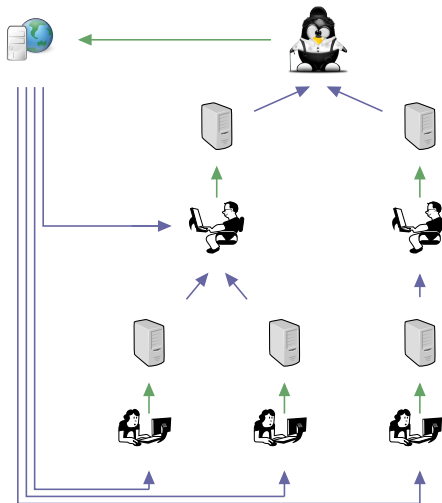
*"Dictator and Lieutenants Workflow."*





# Modèle avec dictateur et lieutenants.

*"Dictator and Lieutenants Workflow."*



## Quelques liens utiles

- <https://git-scm.com/book/fr/v2>
- <http://julien.sopena.fr/enseignements/M2-SAR-Git/cours/01-Git/01-Git.pdf>
- <https://www.kernel.org/pub/software/scm/git/docs/giteveryday.html>
- <https://alexgirard.com/git-book/index.html>

# Références

- Notes de D. Donsez
- Notes de J. Sopena
- Notes de B. Goglin
- Notes de B. Florat

Plus de slides ...

# Étude des objets générés par un exemple simple.

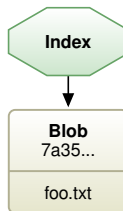
```
mkdir project  
cd project  
git init
```



# Étude des objets générés par un exemple simple.

```
mkdir project
cd project
git init

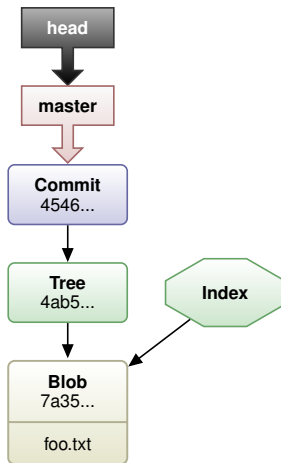
echo "toto" > foo.txt
git add foo.txt
```



# Étude des objets générés par un exemple simple.

```
mkdir project
cd project
git init

echo "toto" > foo.txt
git add foo.txt
git commit -m "Add foo.txt"
```



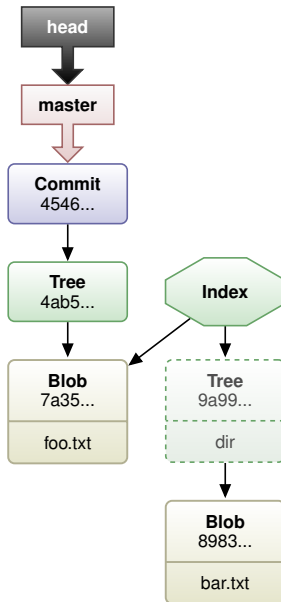
# Étude des objets générés par un exemple simple.

```
mkdir project
cd project
git init

echo "toto" > foo.txt
git add foo.txt

git commit -m "Add foo.txt"

mkdir dir
echo "titi" > dir/bar.txt
git add dir/bar.txt
```





# Étude des objets générés par un exemple simple.

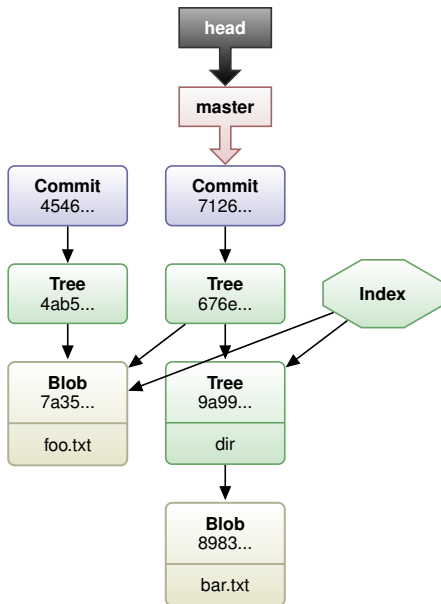
```
mkdir project
cd project
git init

echo "toto" > foo.txt
git add foo.txt

git commit -m "Add foo.txt"

mkdir dir
echo "titi" > dir/bar.txt
git add dir/bar.txt

git commit -m "Add dir/bar.txt"
```



# Étude des objets générés par un exemple simple.

```
mkdir project
cd project
git init

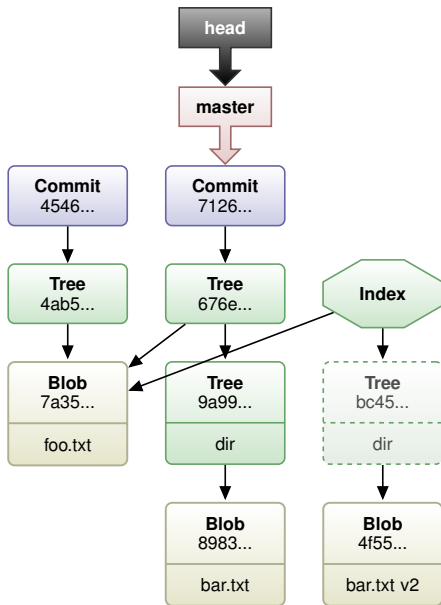
echo "toto" > foo.txt
git add foo.txt

git commit -m "Add foo.txt"

mkdir dir
echo "titi" > dir/bar.txt
git add dir/bar.txt

git commit -m "Add dir/bar.txt"

echo "tutu" > dir/bar.txt
git add dir/bar.txt
```



# Étude des objets générés par un exemple simple.

```
mkdir project
cd project
git init

echo "toto" > foo.txt
git add foo.txt

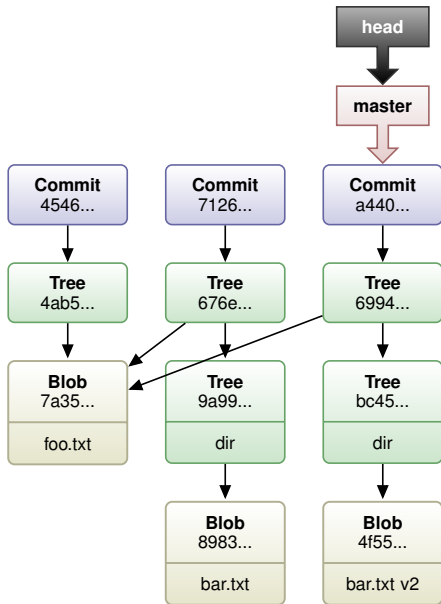
git commit -m "Add foo.txt"

mkdir dir
echo "titi" > dir/bar.txt
git add dir/bar.txt

git commit -m "Add dir/bar.txt"

echo "tutu" > dir/bar.txt
git add dir/bar.txt

git commit -m "Modif dir/bar.txt"
```



# Étude des objets générés par un exemple simple.

```
mkdir project
cd project
git init

echo "toto" > foo.txt
git add foo.txt

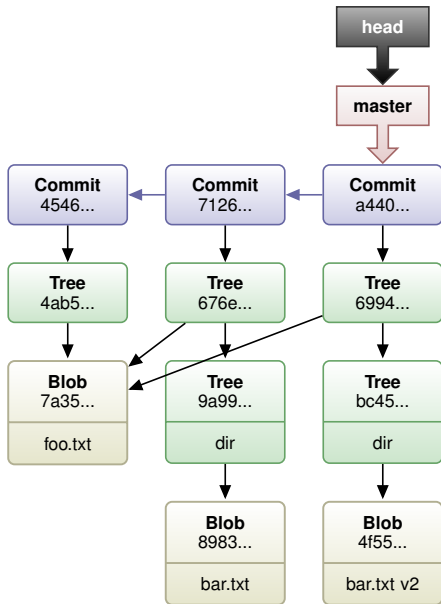
git commit -m "Add foo.txt"

mkdir dir
echo "titi" > dir/bar.txt
git add dir/bar.txt

git commit -m "Add dir/bar.txt"

echo "tutu" > dir/bar.txt
git add dir/bar.txt

git commit -m "Modif dir/bar.txt"
```

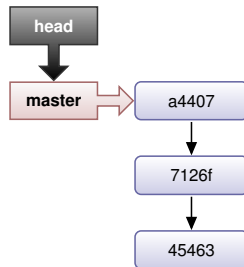


# Merge avec fast forward

- Merge avec une branche `maBranche` qui est **en avance** sur la branche `master`
- La branche `master` ne contient pas de commits non présents dans l'historique de `maBranche`
- **Fast Forward**: Pas besoin d'un nouveau commit, la branche `master` peut simplement être déplacée vers le dernier commit de la branche `maBranche`
- Ce comportement peut être désactivé avec l'option **`-no-ff`**

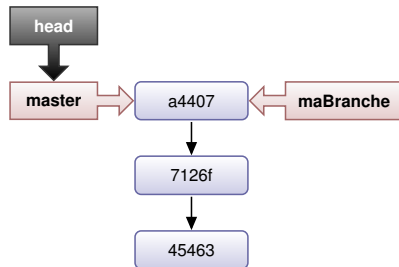
# Merge : exemple sur un historique unique.

```
ls  
foo.txt dir
```



# Merge : exemple sur un historique unique.

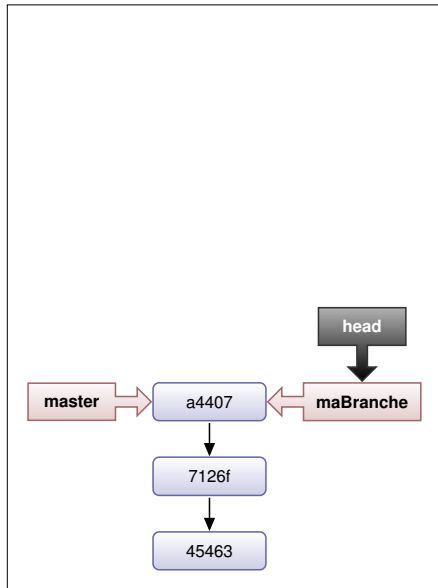
```
ls  
foo.txt dir  
  
git branch maBranche
```



# Merge : exemple sur un historique unique.

```
ls
foo.txt dir

git branch maBranche
git checkout maBranche
```





# Merge : exemple sur un historique unique.

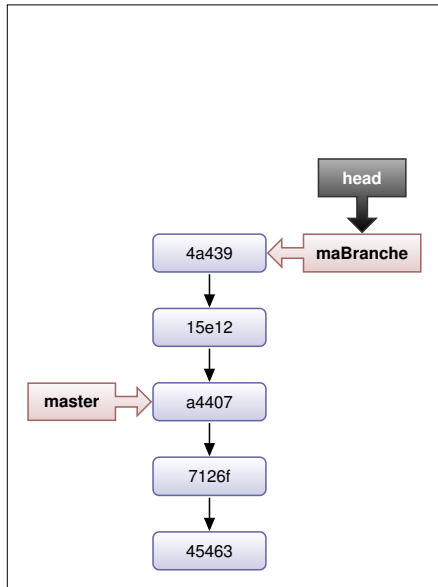
```
ls
foo.txt dir

git branch maBranche

git checkout maBranche

echo "toto" > fichier1.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"

echo "titi" > fichier1.txt
git commit -am "Modif
fichier1.txt"
```



# Merge : exemple sur un historique unique.

```
ls
foo.txt dir

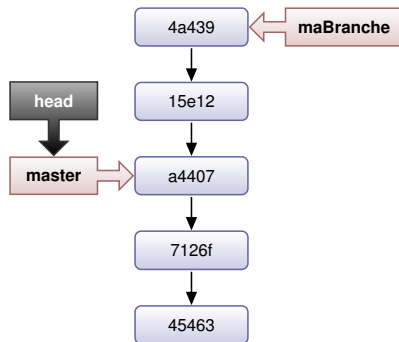
git branch maBranche

git checkout maBranche

echo "toto" > fichier1.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"

echo "titi" > fichier1.txt
git commit -am "Modif
fichier1.txt"

git checkout master
ls
dir foo.txt
```



# Merge : exemple sur un historique unique.

```
ls
foo.txt dir

git branch maBranche

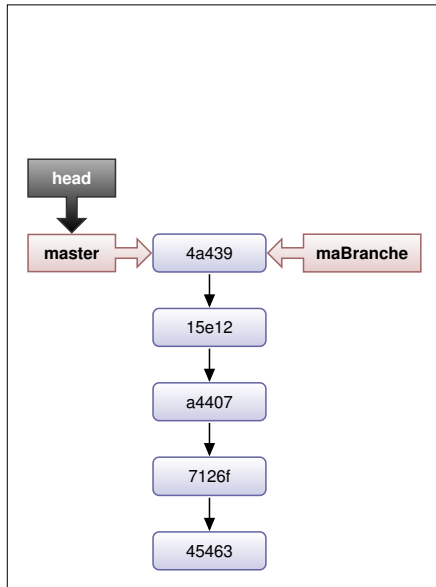
git checkout maBranche

echo "toto" > fichier1.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"

echo "titi" > fichier1.txt
git commit -am "Modif
fichier1.txt"

git checkout master
ls
dir foo.txt

git merge maBranche
cat fichier1.txt
titi
```



# Merge : exemple sur un historique unique.

```
ls
  foo.txt dir

git branch maBranche

git checkout maBranche

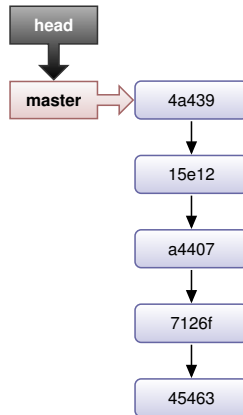
echo "toto" > fichier1.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"

echo "titi" > fichier1.txt
git commit -am "Modif
fichier1.txt"

git checkout master
ls
  dir foo.txt

git merge maBranche
cat fichier1.txt
  titi

git branch -d maBranche
```



# Comparaison : git diff

- ▶ Différences entre le répertoire de travail et l'index :

```
$ git diff
```

- ▶ Différences entre HEAD et l'index :

```
$ git diff --staged
```

- ▶ Différences entre répertoire de travail et HEAD :

```
$ git diff HEAD
```

- ▶ Différences entre répertoire de travail et un autre commit :

```
$ git diff <commit_1>
```

- ▶ Différences entre deux commit :

```
$ git diff <commit_1> <commit_2>
```