Systèmes de gestion de versions
R2.03

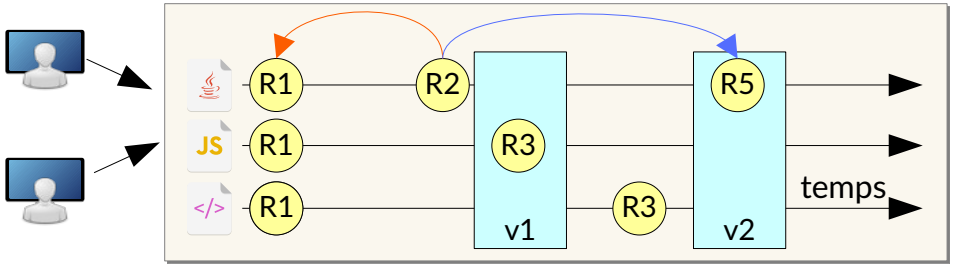
Nicolas Le Sommer
Nicolas.Le-Sommer@univ-ubs.fr
Université Bretagne Sud, IUT de Vannes, Département Informatique

Plan du cours

- Les systèmes de gestion de versions
- Subversion (SVN)
- Git

Rôle et utilité d'un système de gestion de versions

- Gérer plusieurs révisions/versions de fichiers dans le temps
  - Il est possible de revenir à des révisions/versions antérieures
- Maintenir un état cohérent d'un ensemble de fichiers dans le temps
- Assurer le partage de fichiers entre plusieurs utilisateurs
- Gérer les conflits liés à des modifications concurrentes d'un (ou plusieurs) fichier(s)

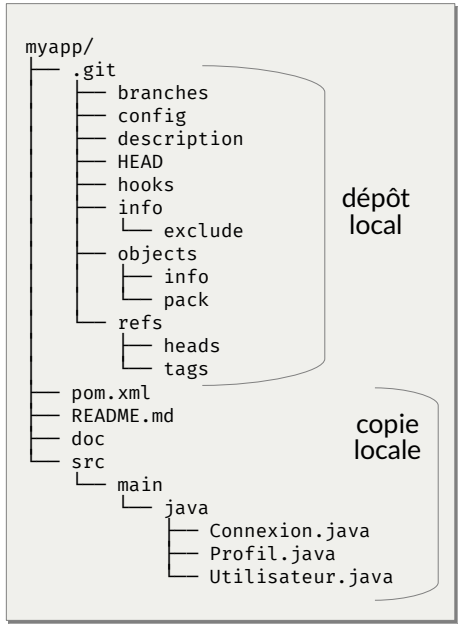


Un peu de terminologie…

- Plusieurs termes pour désigner un système de gestion de versions
  - Système de contrôle de versions
  - En anglais on trouve la notion de SCM (*System Control Manager*), VCS (*Version Control System*) ou de RCS (*Revision Control System*)
- Révision
  - État qui fait suite à l'ajout, la modification ou la suppression d'un fichier ou ensemble de fichiers
  - Étape d'avancement
- Version
  - État cohérent d'un ensemble de fichiers à un instant donné
  - État identifié par une étiquette (1.0.1, v1, alpha1)
  - Une version d'un logiciel correspond à différentes révisions d'un ensemble fini de fichier(s) à une date donnée.

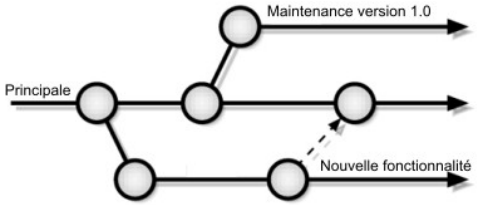
Un peu de terminologie…

- Copie de travail/Copie locale
  - Ensemble de fichiers versionnés dans un état de révision donné
  - Les opérations d'ajout, de modification, de suppression et de gestion (changement de branche ou de révision) sont appliquées sur la copie de travail
  - Propre à un utilisateur
- Dépôt local
  - Lieu de stockage des différentes données : fichiers, révisions, étiquettes, etc.
  - Propre à un utilisateur
- Dépôt distant
  - Lieu de stockage des différentes données : fichiers, révisions, étiquettes, etc.
  - Souvent public et partagé



Un peu de terminologie...

- Branche
  - Utilisées pour
    - maintenir d'anciennes versions du logiciel (sur les branches) tout en continuant le développement des futures versions (sur le tronc)
    - développer en parallèle de plusieurs fonctionnalités sans bloquer le travail quotidien sur les autres fonctionnalités
    - maintenir une branche principale stable et à jour (i.e. intégrant les dernières fonctionnalités)
  - Une branche peut être fusionnée avec une autre



- Étiquette
  - Des étiquettes (tags) peuvent être associées à certains états de révision pour identifier des versions du logiciel à un instant donné.
  - Les étiquettes peuvent aussi être des noms donnés à un logiciel à un instant donné.

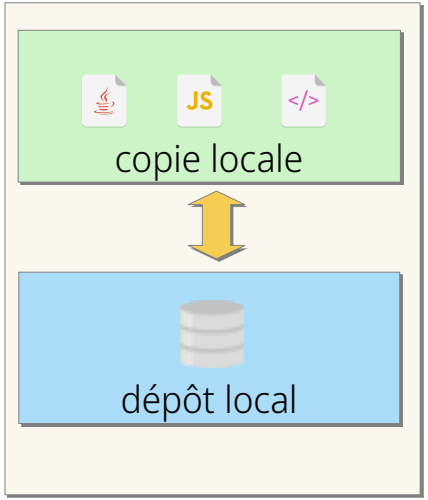
Numérotation des versions

- Schéma
  - major.mini[.micro] [-qualifier[-build_numer]]
- Incrément
  - Major : changement majeur
    - pas de retro-compatibilité (descendante) garantie
  - Mini : ajouts fonctionnels
    - retro-compatibilité garantie
  - Micro : maintenance corrective (bug fix)
- Qualificateurs
  - SNAPSHOT : version en évolution
  - alpha1 : version alpha numéro 1 (très instable et incomplète)
  - beta1 : version béta numéro 1 (instable)
  - rc2 : seconde release candidate
  - ...

Les différents types de systèmes de gestion de versions

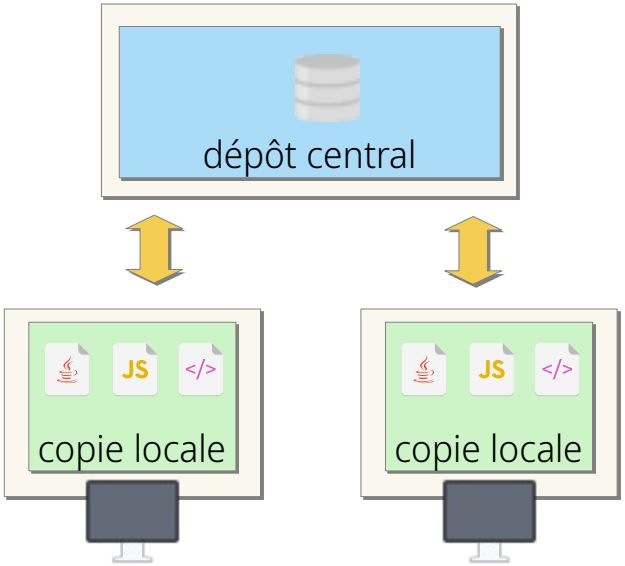
- 3 types : locaux, centralisés et distribués

locaux



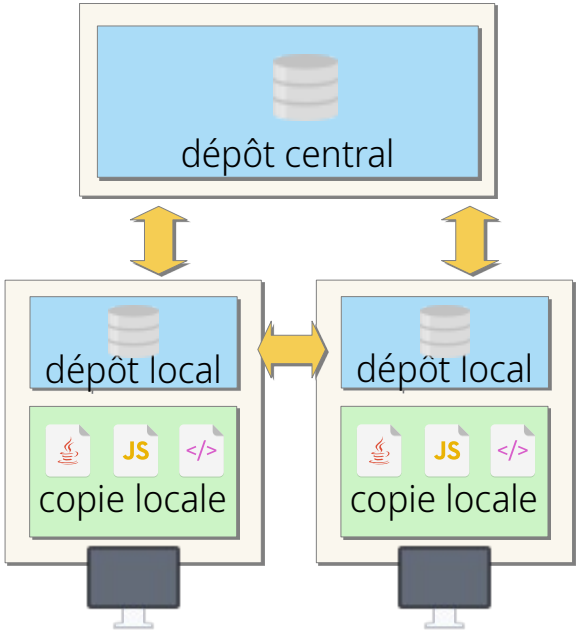
Exemple : RCS

centralisés



Exemples : SVN, CSV

distribués



Exemples : Git, Mercurial, Bazaar

Fonctionnement par différences ou par instantanés

- Par différences
  - On ne stocke que les différences entre l'état R-1 et R
    - Nouveau fichier, diff (unix) entre 2 fichiers existants dans R-1 et R
  - Avantage : moins d'espace de stockage utilisé
  - Inconvénient : plus lent pour basculer entre les révisions

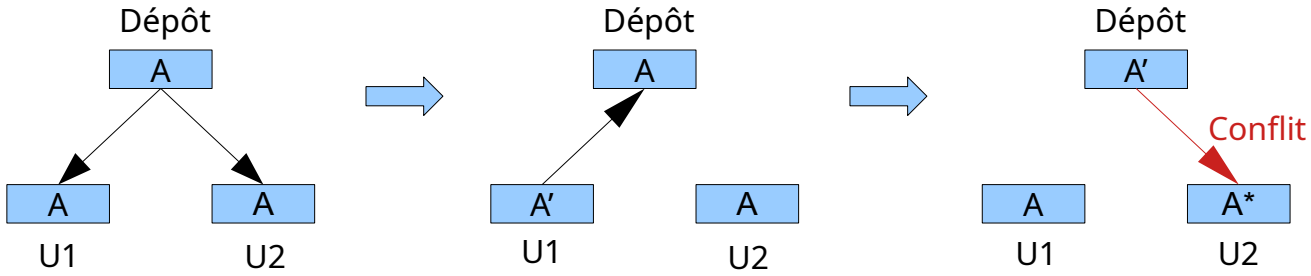
- Par instantanés
  - On stocke les fichiers dans leur intégralité entre l'état R-1 et R
    - Un fichier existant à R-1, sera intégralement stocké à R, même pour une différence mineure
  - Avantage : bascule rapide entre les révisions
  - Inconvénient : beaucoup d'espace de stockage utilisé

Gestion des conflits

- Deux modes de gestion des conflits :
  - Gestion « préventive »
    - Pause d'un verrou avant modification du fichier
    - Levée du verrou explicite ou lors d'une propagation de la modification sur le dépôt
  - Gestion « optimiste »
    - « On croise les doigts pour que personne ne modifie le fichier en parallèle »
    - On doit gérer les éventuels conflits lors de la propagation des modification sur le dépôt
      - diff

Quand un conflit apparaît ?

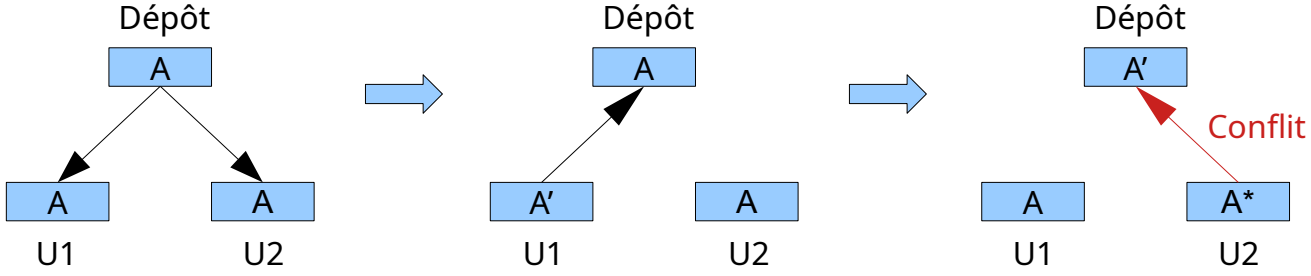
- Lorsque des fichiers n'ont pas les mêmes parents dans le graphe temporel des révisions
- Lors de la mise à jour de la copie locale



Transition incorrecte
Dépôt → U2 : A → A*
Dépôt : A→A'

Transition attendue
U2 : A → A' → A*

- Lors de la mise à jour du dépôt distant



Transition incorrecte
U2→ dépôt : A → A*
Dépôt : A→ A'

Transition attendue
U2 : A → A' → A*
Dépôt : A → A' → A*

- Lors de la fusion de deux branches

Bonne pratique de résolution des conflits

- C'est le développeur qui identifie le conflit qui doit le résoudre
- Toujours sauvegarder les fichiers sur lesquels portent le conflit avant de le résoudre, par exemple en acceptant les modifications des autres développeurs
  - Perte possible du travail effectué

- Deux démarches possibles pour résoudre un conflit :
  - Démarche 1 : avec l'assistance du système de gestion de version
  - Démarche 2 : par le développeur sans assistance du système de gestion de version

Bonne pratique de résolution des conflits

- Démarche 1 :
  1) Sauvegarde des fichiers sur lesquels portent le conflit
  2) Visualisation des différences entre les fichiers sur lesquels portent le conflit au moment de la résolution du conflit par l'outil de gestion de version
  3) Choix du stratégie de résolution :
     1) acceptation des modifications des autres développeurs
     2) Conservation de ses modifications
     3) Fusion des modifications
  4) Si la stratégie de résolution choisie est la 2 ou la 3, alors les modifications doivent être remontées sur le dépôt partagé
  - Attention aux incohérences dans le code si la stratégie 3 est appliquée sans vérifications préalables

Bonne pratique de résolution des conflits

- Démarche 2 :
  1) Sauvegarde des fichiers sur lesquels portent le conflit
  2) Acceptation des modifications des autres développeurs
  3) Identification des différences par l'utilisateur à l'aide d'un outil (e.g. diff unix, meld, ...)
  4) Résolution des conflits par le développeur
  5) Remontée des modifications sur le dépôt partagé

Gestion préventive des conflits

- Démarche de gestion des conflits reposant sur le principe « verrouiller, modifier, libérer »
- Fonctionnement :
  - L'utilisateur 1 verrouille un fichier (ou un ensemble de fichiers) sur le dépôt partagé (centralisé) avant de mettre à jour sa copie et de travailler
  - Tant que le(s) fichier(s) est(sont) verrouillé(s), l'utilisateur 2 ne peut pas les modifier
  - L'utilisateur 2 doit attendre que l'utilisateur 1 libère le verrou
    - en propageant ses modifications sur le dépôt
    - en libérant explicitement le verrou sans faire de modifications

Fusion automatique des modifications

- La plupart des systèmes de gestion de versions intègrent des algorithmes de fusion des modifications pour aider à résoudre automatiquement les conflits
- En général, la fusion automatique fonctionne bien si les développeurs n'ont pas modifié les mêmes lignes d'un même fichier
- La fusion automatique peut entraîner des incohérences

Le système de gestion de version Subversion

Subversion (SVN)

- Histoire de Subversion (SVN)
  - Première version publiée en 2000 par la société CollabNet
  - En 2010 SVN devient un projet de la fondation Apache
  - Dernière version 1.14.2 en 2022

- SVN est un système de gestion de versions
  - centralisé
  - fonctionnant par différences
  - reposant sur un modèle client/serveur

État de la copie locale

- La commande « `svn status` » permet d'obtenir l'état des fichiers dans la copie locale
- Les états sont :

indicateur	Signification
A	Le fichier sera ajouté au dépôt
D	Le fichier sera supprimé du dépôt
M	Le fichier a été modifié localement depuis le dernier update
C	Le fichier est source de conflits
X	Le fichier est eXterne à cette copie de travail
?	Le fichier n'est pas géré par subversion
!	Le fichier est manquant, sans doute supprimé par un autre outil que subversion

Les principales commandes

Commande	Rôle
add	Déclare l'ajout d'une nouvelle ressource pour le prochain commit.
checkout (co)	Récupère en local une version ainsi que ses méta-données depuis le dépôt.
commit (ci)	Enregistre les modifications locales dans le dépôt créant ainsi une nouvelle version.
copy (cp)	Copie des ressources à un autre emplacement (localement ou dans le dépôt).
delete (rm)	Déclare la suppression d'une ressource existante pour le prochain commit.
diff (di)	Calcule la différence entre deux versions (permet de créer un correctif à appliquer sur une copie locale).
import	Envoie une arborescence locale vers le dépôt.
list (ls)	Donne la liste des entrées d'un répertoire du dépôt.
lock/unlock	Verrouille/déverroille un fichier.
log	Donne les messages de commit d'une ressource.
diff (di)	diff (di)
move (mv)	Déclare le déplacement d'une ressource.
switch (sw)	Bascule sur une version/branche différente du dépôt.
update (up)	Met à jour la copie locale existante depuis la dernière version disponible sur le dépôt.

Exemple d'utilisation des commandes

```
# Création d'un dépôt central
#> svnadmin create /tmp/myrepo

# Import d'un ensemble de fichiers dans le dépôt central
#> svn import myapp file:///tmp/myrepo/ -m "import de myapp"

# Suppression des fichiers non versionnés
#> rm -rf myapp

# Création d'une copie locale (fichiers versionnés)
#> svn checkout file:///tmp/myrepo/

# Ajout de nouveaux fichiers au dépôt
#> cd myrepo
#> svn add README.md
#> svn commit -m "Ajout du fichier README"
#> svn add docs
#> svn commit -m "Ajout du répertoire contenant les fichiers de documentation"


# Mise à jour toute la copie locale
#> svn update

# Mise à jour du fichier README uniquement
#> svn update README.md

# Mise à jour du fichier README à une révision antérieure
#> svn update -r 36 README.md
```

Le système de gestion de version Git

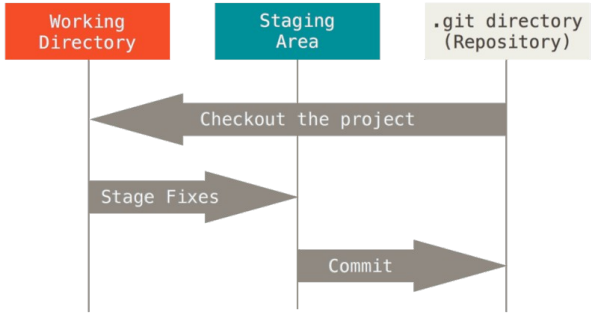
Git

- Histoire de Git
  - Conçu initialement par Linus Torvald en 2005 pour supporter le développement du noyau Linux
  - Dernière version : 2.40.0 (mars 2023)
  - Largement utilisé dans les forges logicielles comme GitHub, GitLab et Bitbucket

- Git est un système de gestion de versions
  - distribué
  - fonctionnant par instantanés

Principe de fonctionnement de Git

- 3 « zones locales » considérées par Git
  - Le répertoire contenant la copie locale (*Working directory*)
  - La zone d'index
  - Le dépôt local Git
    - répertoire « .git » dans le répertoire contenant la copie locale
- 3 états locaux gérés par Git
  - Validé
    - Les fichiers sont sauvegardés dans le dépôt local
  - Modifié
    - Les fichiers ont été modifiés, mais ne sont pas encore sauvegardés dans le dépôt local
  - Indexé
    - Les fichiers modifiés (ou ajoutés ou supprimés) sont marqués comme devant faire partie du prochain instantané



Source : https://git-scm.com/

Configuration de Git

- Les propriétés de configuration de Git sont définies dans le fichier « `~/.gitconfig` »
- Elles peuvent être modifiées à l'aide de la commande « `git commit` »

```
# Définition de l'identité du développeur
#> git config --global user.name "Tryphon Tournesol"
#> git config --global user.email "tryphon.tournesol@gmail.com"

# Éditeur et outil de différences
#> git config --global core.editor emacs
#> git config --global merge.tool meld

# Couleurs
#> git config --global color.ui true
```

Les principales commandes pour une gestion locale

Commande	Rôle
clone	Cloner un dépôt dans un nouveau répertoire.
init	Créer un dépôt Git vide ou réinitialiser un existant. Dépôt local ou distant et partagé
add	Ajouter le contenu de fichiers dans l'index
mv	Déplacer ou renommer un fichier, un répertoire, ou un lien symbolique.
restore	Restaurer les fichiers l'arbre de travail.
rm	Supprimer des fichiers de la copie de travail et de l'index.
diff	Afficher les changements entre les validations, entre validation et copie de travail, etc.
status	Afficher l'état de la copie de travail.
branch	Lister, créer ou supprimer des branches.
log	Afficher l'historique des validations.
commit	Enregistrer les modifications dans le dépôt.
merge	Fusionner deux ou plusieurs historiques de développement ensemble.
switch	Bascule sur branche différente du dépôt.
tag	Créer, lister, supprimer ou vérifier un objet d'étiquette.

Exemples de commandes

```
# Création d'un dépôt local
#> git init

# Ajout de fichiers dans le dépôt
#> git add *.c
#> git commit -m "Ajout du code source de l'application"
#> git add README.md
#> git commit -m "Ajout du fichier README"

# Suppression de fichier
#> git rm README.md
#> git commit -m "Suppression du fichier README"

# Annlation du dernier commit
#> git add *.c
#> git commit -m "Ajout du code source de l'application"
#> git add *.h
#> git commit –amend

# Annulation des modifications de la copie de travail et restauration d'un fichier modifié et non indexé
#> git restore README.md

# Annulation des modifications de la copie de travail et restauration d'un fichier modifié et indexé
#> git restore --staged README.md
#> git restore README.md
```

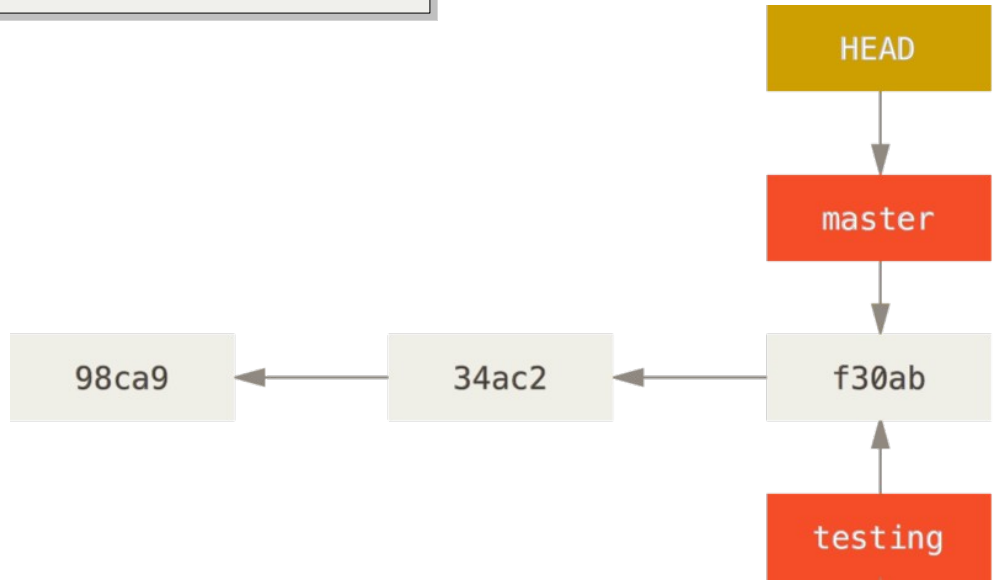
Les branches

- Les commandes à utiliser pour créer et gérer les branches sont les suivantes :

```
# Création d'une branche
#> git branch ma_branche_de_test

# Pour changer de branche
#> git switch ma_branche_de_test

# Pour créer et changer de branche en une seule opération
#> git switch -c ma_branche_de_test

# Pour fusionner 2 branches (exemple fusion branche test dans master)
#> git switch master
#> git merge test

# Pour supprimer une branche
#> git branch -d ma_branche_de_test
```

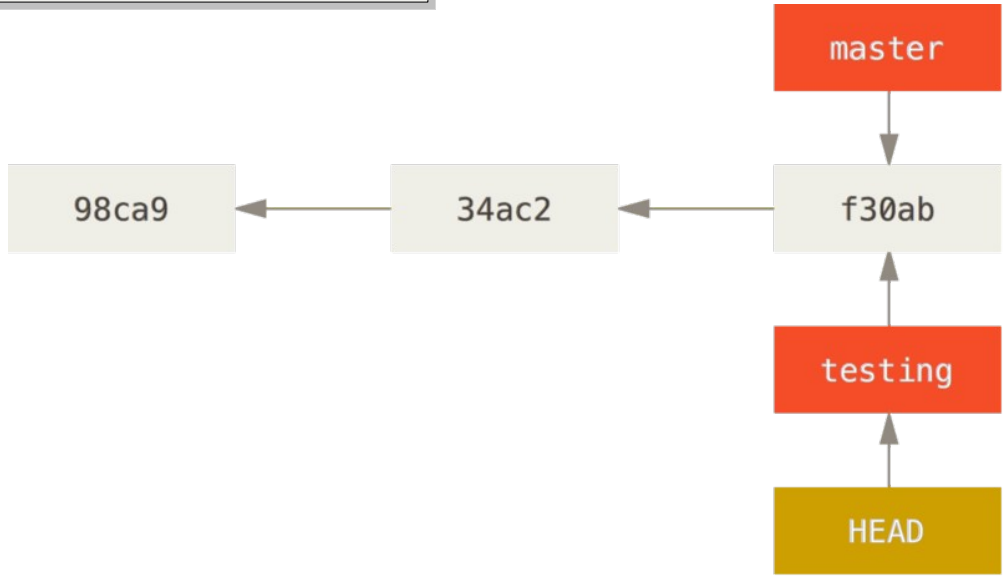
Les branches : illustration du fonctionnement

```
#> git branch testing
```



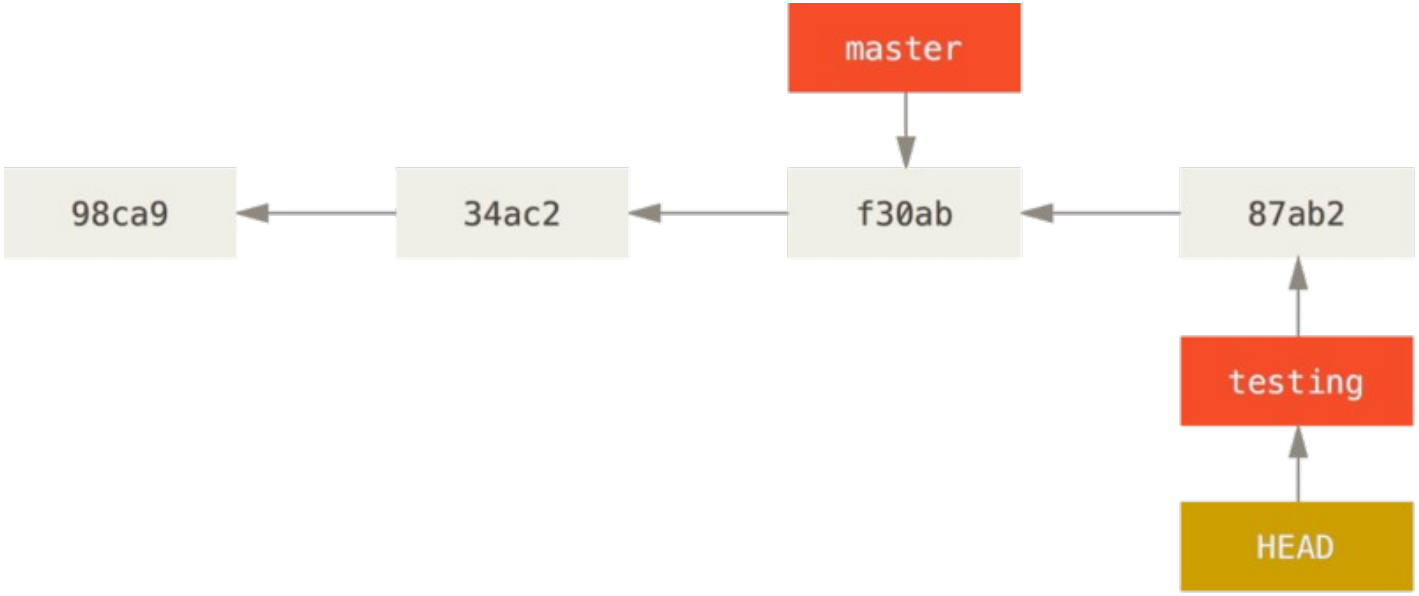
Les branches : illustration du fonctionnement

```
#> git branch testing
#> git switch testing
```



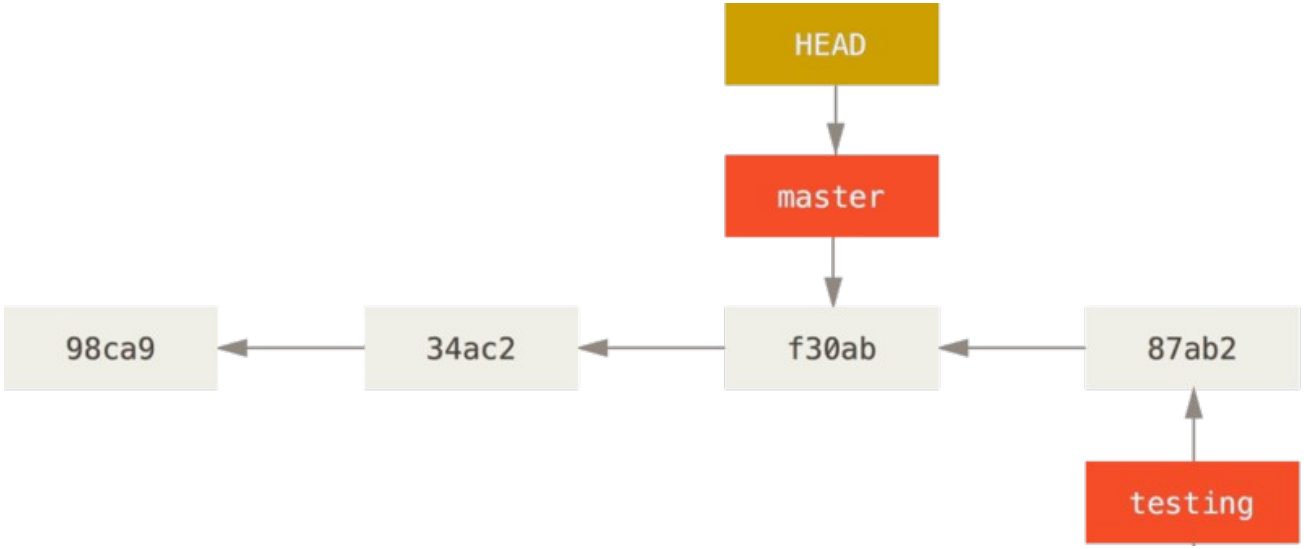
Les branches : illustration du fonctionnement

```
#> git branch testing
#> git switch testing
#> vim test.txt
#> git commit -a -m 'made other changes'
```



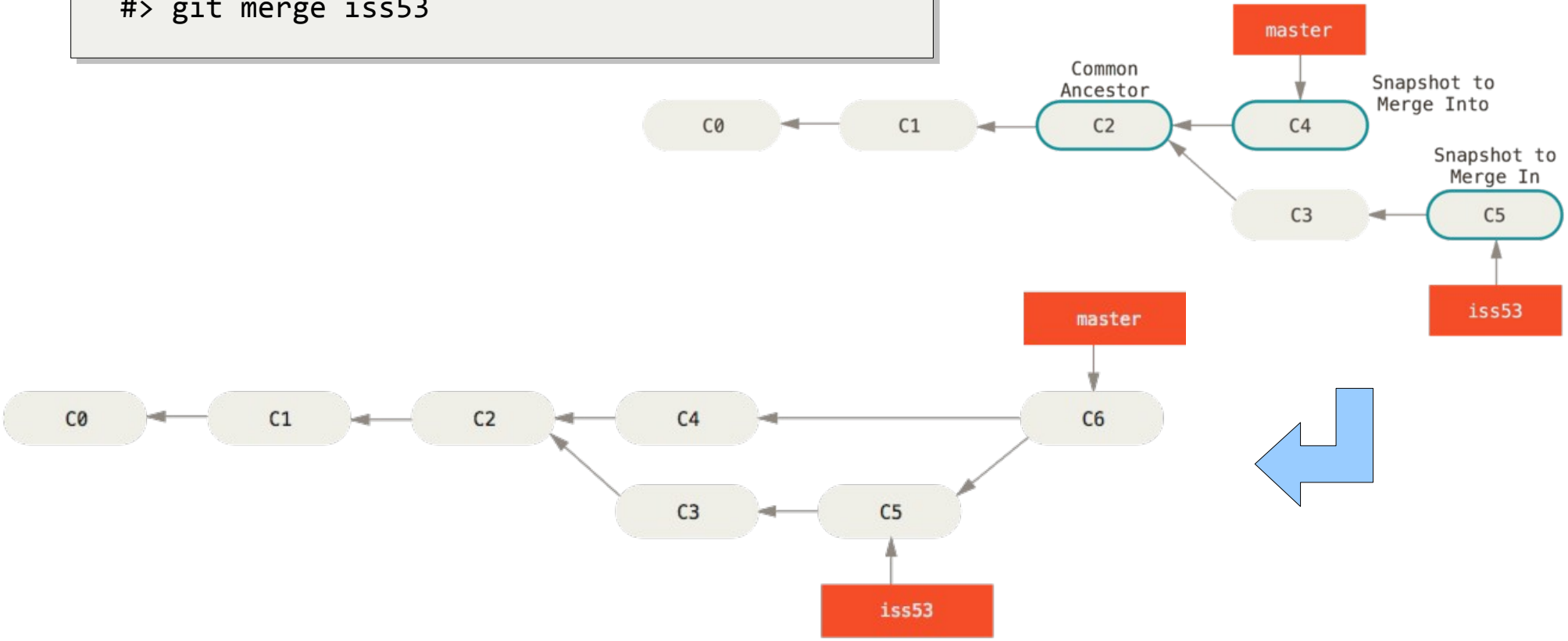
Les branches : illustration du fonctionnement

```
#> git branch testing
#> git switch testing
#> vim test.txt
#> git commit -a -m 'made other changes'
#> git switch master
```



Les branches : illustration du fonctionnement

```
#> git switch master
#> git merge iss53
```



Gestion des dépôts distants

- Les commandes pour la gestion des dépôts distants sont :

Commande	Rôle
remote	Liste les dépôts distants.
fetch	Télécharge les objets et références depuis un autre dépôt.
pull	Rapatrie et intégre un autre dépôt ou une branche locale
push	Met à jour les références distantes ainsi que les objets associés.

Exemple de commandes pour les dépôts distants

```
# Ajout d'un dépôt distant
#> git remote add autre_depot http://gitlab.com/autre_depot.git

# Récupération des données depuis un dépôt
#> git fetch autre_depot

# Récupération des données d'un dépôt disant et fusion avec la copie locale
#> git pull autre_depot

# Récupération des données d'une branche donnée d'un dépôt disant et fusion avec la copie locale
#> git pull autre_depot ma_branche

# Mise à jour du dépôt distant (d'origine)
#> git push
#> git push origin master

# Suppression d'une branche
#> git push autre_depot --delete ma_branche

# Renommage de la branche master en main
#> git branch --move master main
#> git push --set-upstream origin main
#> git push origin --delete master

# Liste de toutes les branches
#> git branch --all
```

Pour en savoir plus...

- http://git-scm.com/book/fr
- https://svnbook.red-bean.com/