

```
<cours annee= "L2" filiere="Informatique"  
titre="Génie Logiciel">
```

```
<!-- BEGIN NAVIGATION -->  
<nav id="nav" role="navigation">  
  <ul>  
    <li><a href="index.html">Home</a></li>  
    <li><a href="home-events.html">Home Events</a></li>  
    <li><a href="multi-col-menu.html">Multiple Column Men</a></li>  
    <li class="has-children"> <a href="#" class="current">Home</a>  
      <ul>  
        <li><a href="tall-button-header.html">Tall But</a></li>  
        <li><a href="image-logo.html">Image Logo</a></li>  
        <li class="active"><a href="tall-logo.html">Ta</a></li>  
      </ul>  
    </li>  
    <li class="has-children"> <a href="#">Carousels</a>  
      <ul>  
        <li><a href="variable-width-slider.html">Variab</a></li>  
        <li><a href="variable-width-slider.html">Testimoni</a></li>  
      </ul>  
    </li>  
  </ul>  
</nav>
```

Versioning

T.Brouard – T.Devogele – M.Boulakbech

En bref...

- Versioning ?
- [Types de systèmes de versioning](#)
- [Git](#)
 - Caractéristiques et principes
 - [Utilisation locale](#)
 - [Gestion des versions](#)
 - [Utilisation en réseau](#)
- [Github](#) et d'autres

Versioning ?

- La gestion de version consiste à maintenir dans un état cohérent un ensemble de documents.
- Un système de gestion de version conserve les évolutions d'un ensemble de fichiers, de telle façon à ce qu'on puisse à tout moment restaurer l'ensemble tel qu'il était à une date donnée.

V.C.S. : Version Control System

Domaines d'application

- TOUS
- Les VCS sont applicables à tous types de documents :
- Fichiers textes (codes sources java, php, html, etc. mais aussi LaTeX, RTF...)
- Fichiers binaires (images, librairies...)
- Ils sont donc particulièrement adaptés à la gestion des codes sources d'une application, permettant d'intégrer de façon isolée de nouvelles fonctions ou des correctifs

Avantages

- Assurer la cohérence d'un ensemble de *documents* (codes sources, mais pas que)
- Retrouver, de façon simple, une version précédente d'un document
- Comparer différentes versions d'un document
- Partager les versions au sein d'une communauté
- Participer à leur élaboration

Première approche



AVANTAGES / INCONVENIENTS ?

S1 : duplication complète

- Duplication de la version précédente dans un répertoire, travail sur la version en cours
- Duplique la totalité : cohérent, mais gaspillage d'espace disque
- Aucune protection contre l'effacement accidentel ou l'écrasement de données
- Difficile de visualiser les évolutions
- Très simple à réaliser, droits gérés par SGF (système de gestion de fichiers)

S2 : base de code



- Base de données locale
- Conservation des modifications d'un fichier (on mémorise les différences entre fichiers)
- Bien appliqué aux codes sources, plus difficile pour les binaires
- Systèmes anciens – ex. *RCS* (Revision Control System), 1982, projet GNU depuis 1995, toujours livré avec Linux, OSX...

<https://www.gnu.org/software/rcs/>

S3 : gestion centralisée

- Gestion centralisée
- Permet à plusieurs personnes, éloignées géographiquement, de partager un ensemble de fichiers
- Un dépôt central contient tous les fichiers
- Des clients peuvent se connecter pour modifier les contenus des dépôts
- Systèmes très courants : *CVS*, *Subversion*...
C.V.C.S. : Centralized Version Control System

S3 : gestion centralisée

- Gestion centralisée
-  Contrôle fin des permissions d'accès
Vue d'ensemble de qui fait quoi
-  **Grande fragilité** due à la centralisation, en cas de panne du serveur ou de corruption du disque central
- Des clients peuvent se connecter pour modifier les contenus des dépôts
- Systèmes très courants : *CVS, Subversion...*

C.V.C.S. : Centralized Version Control System

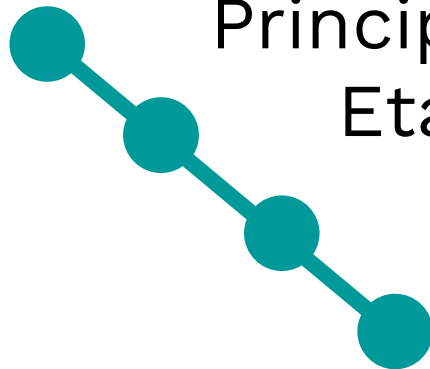
S4 : gestion distribuée

- Gestion distribuée
- Le client extrait la copie du dépôt, pas seulement les fichiers modifiés
- En cas de crash serveur, on peut reconstruire d'après au moins un dépôt client
- Systèmes très courants : *Git*, *Mercurial*...

D.V.C.S. : Distributed Version Control System



Exemple de DVCS : Git



Principes de fonctionnement
Etats des données
Installation
Configuration

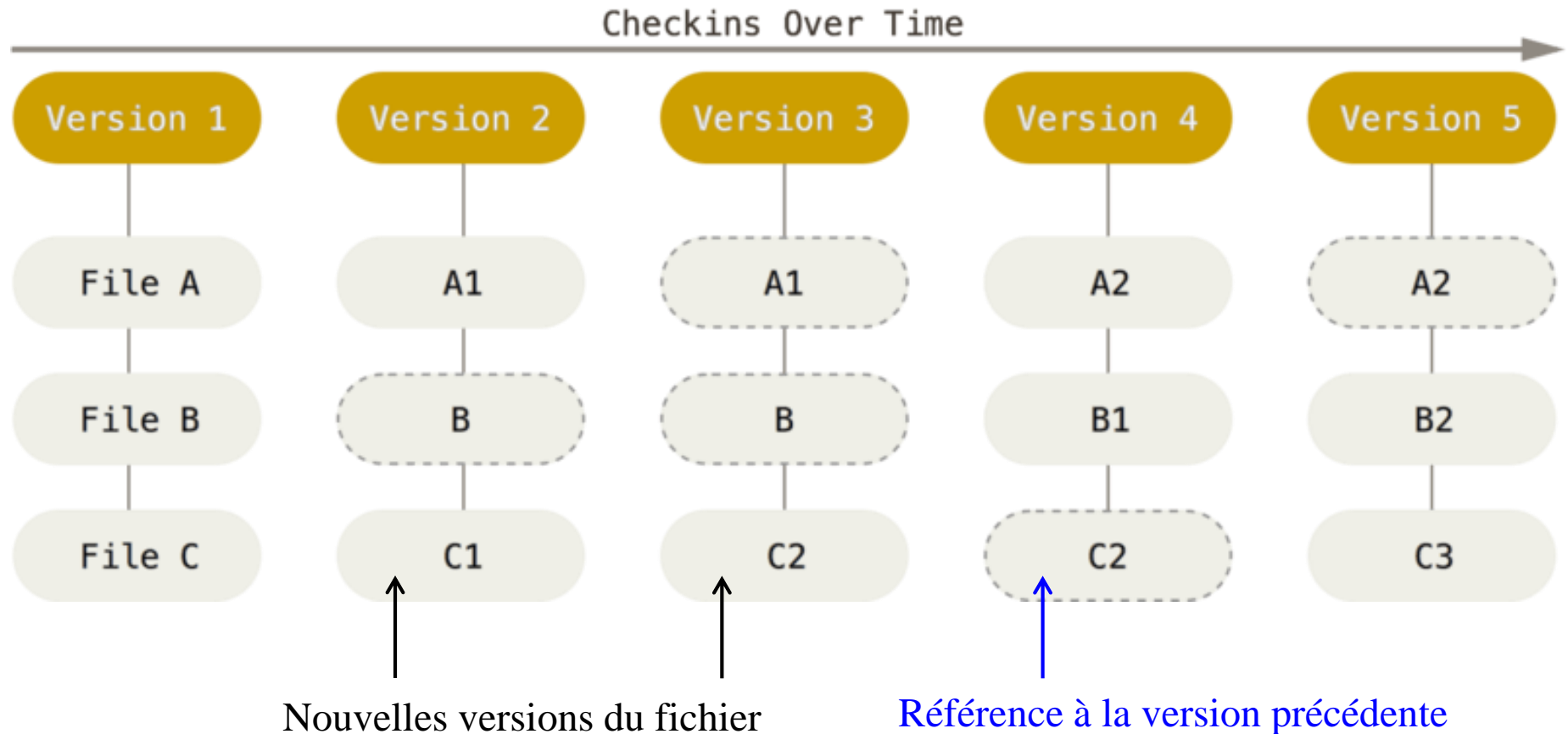
Git

- DVCS né en 2005 [2]
- Prononcer [ɡɪt] en anglais comme {g}et+b{it}
- Conçu / développé par **L.Torvalds** pour la gestion du noyau linux
- Suite à 3 années d'utilisation d'un VCS propriétaire, BitKeeper : nombreux enseignements à intégrer
- Rapide, adapté aux grands projets, distribué, développements non linéaires, fiable
- Vidéo à voir : <https://youtu.be/4XpnKHJAok8>

Git gère des *instantanés*

- *Les* VCS considèrent le changement
 - Ils stockent une liste de modifications par rapport à une situation antérieure
- Git considère une photo du projet à un instant donné (appelée *instantané* ou *commit*)
- Il stocke donc *a priori* tous les fichiers
- Chacun est *taggé*, permettant de savoir s'il est différent de la version antérieure
- S'il est identique, il renvoie à la version précédente

Git : cohérence et parcimonie



Source : [1]

Git : distribué mais local !

- Avec Git, on a une copie **locale** du projet
- On accède à l'historique des évolutions
- Les opérations peuvent se faire en local (notamment retourner à une version antérieure ou calculer une différence)
- On peut travailler plus vite (pas de latence réseau) et hors connexion
- De nouveau connecté, on peut resynchroniser le tout

Git & l'intégrité

- Tout fichier entrant fait l'objet d'un calcul d'une somme de contrôle (SHA1, [3]).
- Le résultat sert d'identifiant pour Git
- Il est donc impossible de modifier un fichier sans que *Git s'en aperçoive*
- *Protection contre la corruption*
- *Protection contre la perte*
- *Détection des modifications*

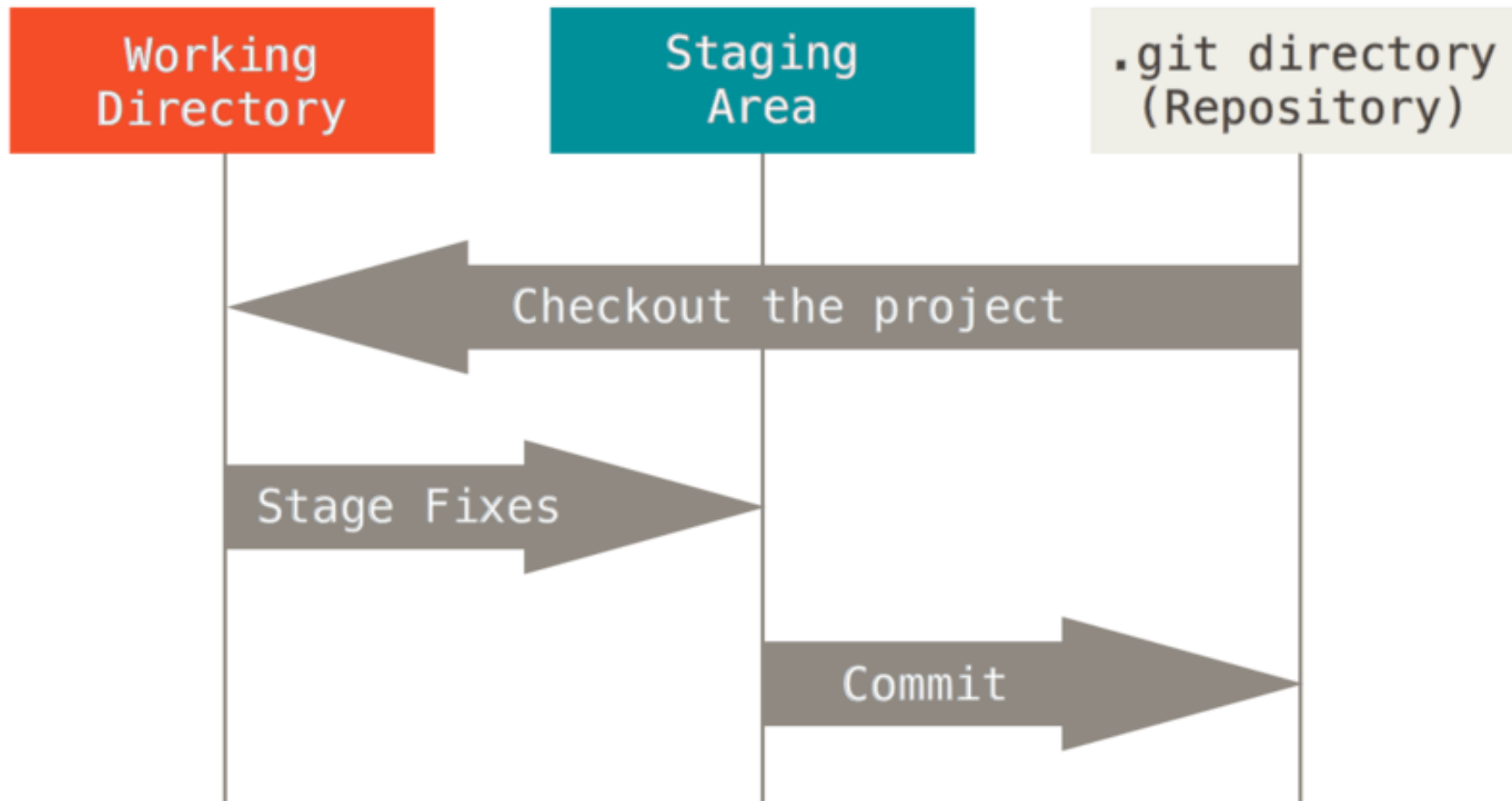
Git & l'intégrité

- ❑ Paul vient de découvrir Git et trouve ça génial. Il gère tout avec, même son rapport de projet et ses slides pour la soutenance. Il va jusqu'à faire figurer la clé SHA1 attribuée par Git à ses slides sur la première page de ses slides, pensant impressionner le jury.
- ❑ Pourtant, lors de la soutenance, Hervé, membre du jury lui affirme que la clé est fausse.
- ❑ **Git se serait trompé ?**

Statut des données

- Tout fichier géré par Git passe par 3 états :
- Le fichier est dans le répertoire de travail, où il peut être modifié sans interférer avec qui que ce soit [modifié]
- Ensuite, il est indexé, c'est à dire préparé pour intégrer le prochain instantané Git [indexé]
- Enfin, il est intégré dans l'instantané Git et conservé, pour pouvoir être distribué par la suite, restauré, comparé, etc. [validé]

Statut des données



Source : [1]

Installation

- Votre machine doit disposer de l'application, au choix :
- **En ligne de commande, depuis un shell**
- Au moyen d'un client graphique [4]
- Intégrée à votre IDE (par ex. Eclipse) [5]

<https://git-scm.com/book/fr/v2/Démarrage-rapide-Installation-de-Git>

Configuration

- Quand :
- Avant le premier lancement
- Sur demande, suite à changement important
- Conservée en cas de mise à jour
- Quoi :
- Votre identité (nom, email) nécessaire pour valider les fichiers de l'instantané
- l'éditeur de texte par défaut, utilisé par Git pour ajouter un message lors de la validation

Configuration

- Exemple (sous OSX) :

```
$ git config --global user.name "Thierry BROUARD"  
$ git config --global user.email brouard@univ-tours.fr  
$ git config --global core.editor vim
```

- Cela définit
- Le nom de l'utilisateur local
- Son email
- l'éditeur de texte lors d'échanges avec Git
- Visualisation : `git config --list`

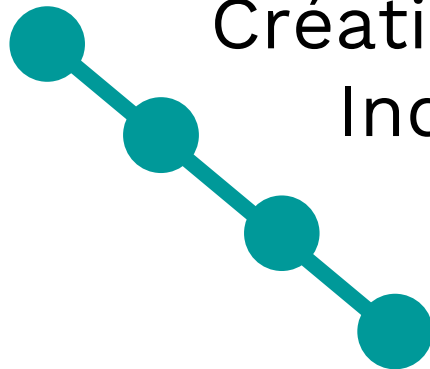
Aide en ligne

- Internet, [1] notamment
- IRC : *irc.freenode.net*, #git ou #github
- Shell :

```
$ git help <commande>  
$ git <commande> --help  
$ man git-<commande>
```




Utilisation de Git en local



Création de dépôt
Indexation de fichiers
Validation de dépôt
Taggage d'instantané

Utilisation en local

- Git s'utilise beaucoup en local, car il assure, avant tout, la cohérence de la version située sur le poste de travail
- Il ne fait que dupliquer l'instantané vers le réseau (ou depuis le réseau)
- Cela lui confère une grande vitesse d'exécution ainsi que la capacité à travailler hors ligne

Créer un nouveau dépôt

- Nouveau projet non publié (sinon cf. *clone*)
- La démarche est simple :
- On se place dans le répertoire contenant les fichiers
- On crée le dépôt initial
- Git installe ses fichiers pour suivre les modifications, mais pour le moment ne fait rien

Pour effacer toutes les données de suivi, on supprime le répertoire .git

Répertoire de travail

- Le répertoire du projet est maintenant devenu répertoire de travail.
- Il faut bien comprendre que le contenu de ce répertoire est *contrôlé* par git
- Une validation fait une « photo » de ce répertoire
- Changer de version *via* git **remplace** le contenu de ce répertoire

Créer un nouveau dépôt

- Exemple de création de dépôt :

```
$ cd ~/dev/projet1
```

```
$ ls -al
```

```
drwxr-xr-x   3          102 11 oct 23:18 .  
drwxr-xr-x+ 65          2210 11 oct 23:17 ..  
-rw-r--r--   1           27 11 oct 23:18 LISEZMOI.TXT
```

```
$ git init
```

```
Initialized empty Git repository in /Users/brouard/dev/.git/
```

```
$ ls -al
```

```
drwxr-xr-x   3          102 11 oct 23:18 .  
drwxr-xr-x+ 65          2210 11 oct 23:17 ..  
drwxr-xr-x  10          340 11 oct 23:18 .git  
-rw-r--r--   1           27 11 oct 23:18 LISEZMOI.TXT
```

Git est là !



Créer un nouveau dépôt

- Organisation interne du dépôt

```
$ ls -al .git
```

drwxr-xr-x	10	340	11	oct	23:18	.	
drwxr-xr-x	4	136	11	oct	23:18	..	
-rw-r--r--	1	23	11	oct	23:18	HEAD	← Dernière version
drwxr-xr-x	2	68	11	oct	23:18	branches	← Versions
-rw-r--r--	1	137	11	oct	23:18	config	
-rw-r--r--	1	73	11	oct	23:18	description	
drwxr-xr-x	13	442	11	oct	23:18	hooks	
drwxr-xr-x	3	102	11	oct	23:18	info	
drwxr-xr-x	4	136	11	oct	23:18	objects	← Fichiers
drwxr-xr-x	4	136	11	oct	23:18	refs	

Etat du dépôt

- Etat du dépôt, fichiers modifiés depuis le dernier instantané :

```
$ git status
```

```
On branch master
```

```
No commits yet
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
LISEZMOI.TXT
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

Ajout de fichiers à l'instantané

- Mise en place du suivi pour ce projet
- Désigner les fichiers (un par un, ou par lot...)
- Inclusion de tout ce qui est déjà suivi
- Les fichiers sont **indexés**, mais **non validés** : on peut détecter les modifications, mais pas les restaurer depuis un état antérieur
- Il faut les valider (*commit*) pour bénéficier de la restauration

Ajout de fichiers à l'instantané

```
$ git add LISEZMOI.TXT
```

```
$ git status
```

```
On branch master
```

```
No commits yet
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)
```

```
new file:   LISEZMOI.TXT
```

← Fera partie de l'instantané

```
$ ls -al .git/objects/52
```

```
total 8
```

```
drwxr-xr-x  3    102  .
```

```
drwxr-xr-x  5    170  ..
```

```
-r--r--r--  1     43  afcd8817386a2ce217ea674cf5c43f2185b74e
```

Clé SHA1 du fichier



Création de l'instantané

`$ git commit` (ouverture de l'éditeur pour saisir le message accompagnant la validation...saisie, enregistrement puis sortie)

```
[master (root-commit) a3b73ba] Mon premier commit !  
1 file changed, 2 insertions(+)  
create mode 100644 LISEZMOI.TXT
```

Message saisi dans l'éditeur

Clé SHA1 abrégée de l'instantané

`$ git status`

```
On branch master  
nothing to commit, working tree clean
```

`$ cat .git/logs/refs/heads/master`

```
0000000000000000000000000000000000000000000000000000000000000000  
a3b73ba4f29ece5a3e02ba276732de7a81b5788f Thierry BROUARD  
<brouard@univ-tours.fr> 1539294885 +0200 commit (initial):  
Mon premier commit !
```

Identité qui a validé

Clé SHA1 de l'instantané

Modification du dépôt et suivi

(Après modification du fichier LISEZMOI.TXT)

```
$ git status
```

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: LISEZMOI.TXT

Fichier modifié mais non indexé



no changes added to commit (use "git add" and/or "git commit -a")

(Nouvel instantané)

```
$ git commit -a (Ré-indexe les fichiers précédemment suivis et valide)
```

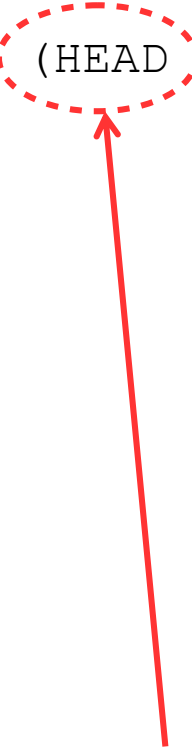
[master e3fe63b] Une nouvelle version : 2.0 !!!

1 file changed, 1 insertion(+)

Liste des instantanés

```
$ git log
```

```
commit e3fe63bf93c1d54f747ad143a8c67123856dca83 (HEAD -> master)
Author: Thierry BROUARD <brouard@univ-tours.fr>
Date:   Fri Oct 12 00:07:55 2018 +0200
```



Une nouvelle version : 2.0 !!!

```
commit a3b73ba4f29ece5a3e02ba276732de7a81b5788f
Author: Thierry BROUARD <brouard@univ-tours.fr>
Date:   Thu Oct 11 23:54:45 2018 +0200
```

Mon premier commit !

DERNIERE VERSION

Qu'est-ce qui a changé ?

```
$ git log -p
commit e3fe63bf93c1d54f747ad143a8c67123856dca83 (HEAD -> master)
Author: Thierry BROUARD <brouard@univ-tours.fr>
Date:   Fri Oct 12 00:07:55 2018 +0200
```

Une nouvelle version : 2.0 !!!

```
diff --git a/LISEZMOI.TXT b/LISEZMOI.TXT
index 84a0160..039f1f3 100644
--- a/LISEZMOI.TXT
+++ b/LISEZMOI.TXT
@@ -1,2 +1,3 @@
  Bienvenue dans le projet 1
  Tiens ! une modif !
+Encore une autre modif !
```



**Git affiche les différences
constatées entre les fichiers
de l'instantané**

Ajouter un tag

- Le tag est une étiquette pour repérer une version particulière d'un instantané
- Par ex. le numéro de version d'un ensemble de codes sources
- S'ajoute lors de la validation, ou *a posteriori*, en référençant le code SHA1 de l'instantané
- Le tag sert de pointeur sur un état particulier des fichiers

Ajout d'un tag

```
$ git log --pretty=oneline
e3fe63bf93c1d54f747ad143a8c67123856dca83 (HEAD -> master) Une
nouvelle version : 2.0 !!!
a3b73ba4f29ece5a3e02ba276732de7a81b5788f Mon premier commit !
```

(message *via* l'éditeur par défaut)

```
$ git tag -a v1.0 a3b73b
```

(message *via* la ligne de commande)

```
$ git tag -a v1.2 e3fe63 -m "Informations sur la
version 1.2"
```

(liste des versions)

```
$ git tag
v1.0
v1.2
```

Informations sur une version

```
$ git show v1.2
```

```
tag v1.2
```

```
Tagger: Thierry BROUARD <brouard@univ-tours.fr>
```

```
Date:   Fri Oct 12 07:33:21 2018 +0200
```

Informations sur la version 1.2

```
commit e3fe63bf93c1d54f747ad143a8c67123856dca83 (HEAD -> master,  
tag: v1.2)
```

```
Author: Thierry BROUARD <brouard@univ-tours.fr>
```

```
Date:   Fri Oct 12 00:07:55 2018 +0200
```

Une nouvelle version : 2.0 !!!

```
diff --git a/LISEZMOI.TXT b/LISEZMOI.TXT
```

```
index 84a0160..039f1f3 100644
```

```
--- a/LISEZMOI.TXT
```

```
+++ b/LISEZMOI.TXT
```

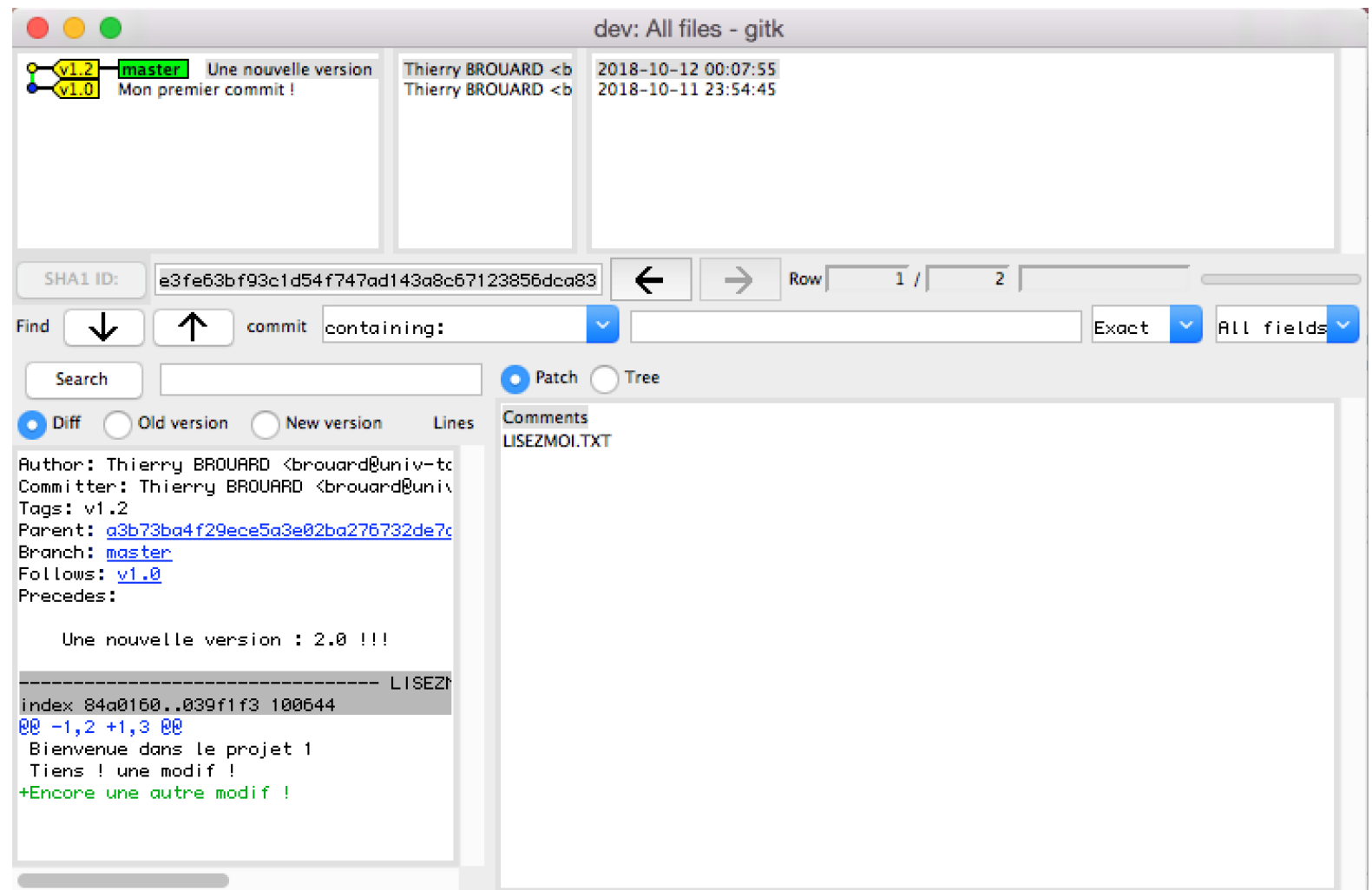
```
...
```



**Liste des validations
et des différences par
rapport à la version n-1**

La même chose en graphique...

\$ gitk



Et si on se trompe ?

- On a oublié des fichiers dans un commit
 - On corrige (`git add...`) puis on revalide
 - ❑ `git commit --amend`
- Le contraire : un fichier est prévu pour le commit, mais il ne faut pas :
 - ❑ `git rm -- cached nomfichier`
- Ne pas oublier : *git status* pour contrôler l'état du dépôt

Et si on se trompe ?

- Annuler les modifications faites sur un fichier (= restaurer dans l'état du dernier commit)
 - ❑ `git checkout -- nomfichier`
- Tout ce qui est *commité* peut être restauré en cas de perte ou de corruption
- Tout ce qui est perdu, sans avoir été *commité*, est effectivement perdu.

Recommandations

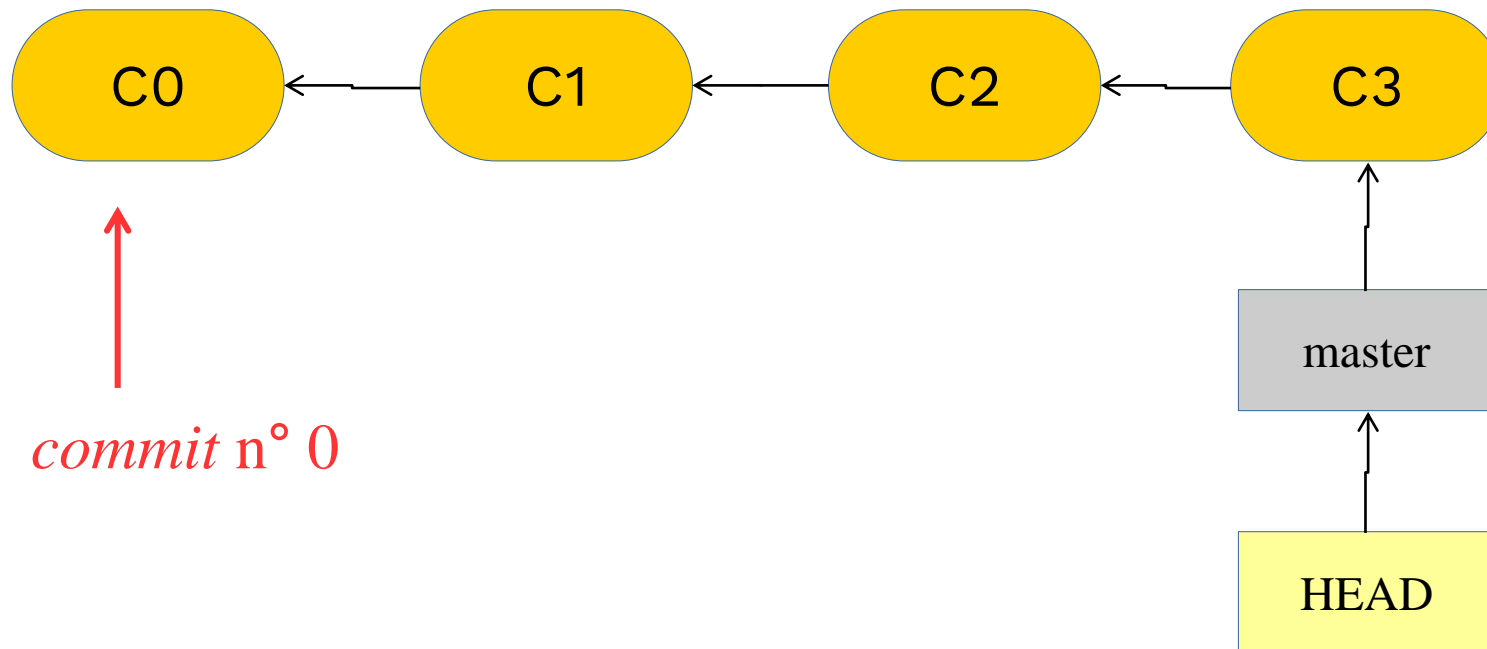
- Être précis dans les commentaires, lors des indexations et des validations, Git est fait pour collaborer
- Faire plusieurs petites validations plutôt qu'une seule, cela aidera
- À la fusion des dépôts en cas de travail à plusieurs
- À l'annulation de modifications ou à la reprise d'une version antérieure

Un mot sur le stockage

- Les instantanés sont « chaînés » de telle sorte que Git peut reconstruire tout l'historique des versions
- La liste principale, par défaut (appelée branche) est désignée par *master*
- On peut avoir autant de branches que l'on veut (*cf. section sur les branches*)
- On travaille dans la branche désignée par le pointeur *HEAD*

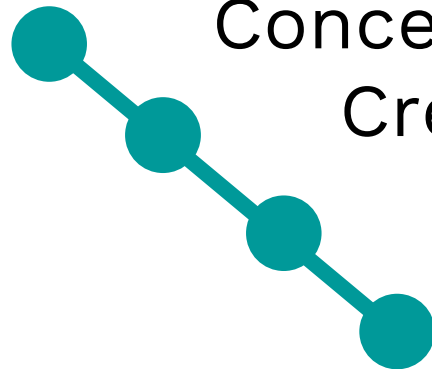
Un mot sur le stockage

Quatre instantanés, le pointeur de branche (master) et le pointeur de branche de travail (HEAD)





Gérer plusieurs versions d'un même dépôt



Concept de branche
Création

Fusion

Visualisation

Ex de scénario [1]

- vous travaillez sur un site web ;
- vous créez une branche, locale, pour un nouvel article en cours ;
- vous commencez à travailler sur cette branche.
- ❑ **À cette étape, vous recevez un appel pour vous dire qu'un problème critique a été découvert et qu'il faut le régler au plus tôt.**
- vous basculez sur, et synchronisez, la branche de production ;
- vous créez une branche pour y ajouter le correctif ;
- après l'avoir testé, vous fusionnez la branche du correctif et poussez le résultat en production ;
- vous rebasculez sur la branche initiale et continuez votre travail.

Branches

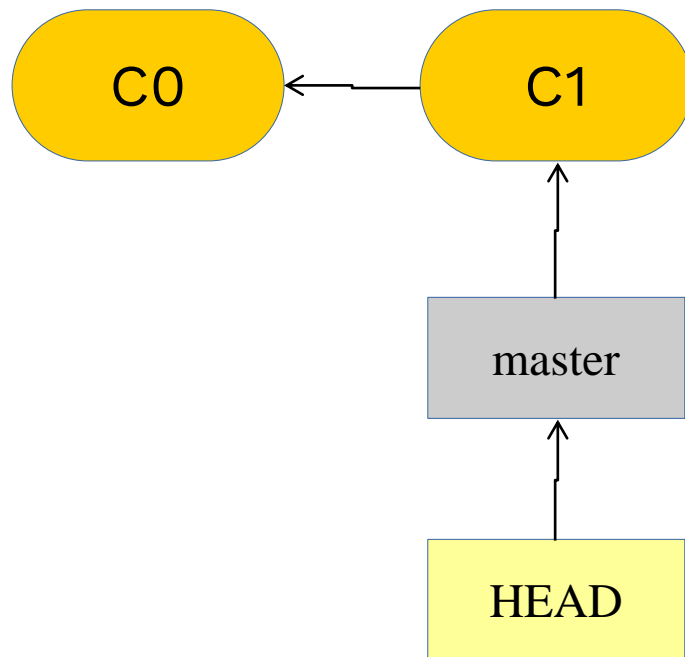
- Une branche désigne une liste d'instancés, un historique de versions
- Elle permet d'isoler une fonctionnalité, ou une orientation particulière, ou un patch...
- Lorsque le projet est stabilisé, on peut :
- conserver la branche (par ex. une version française et une version anglaise)
- ou la fusionner avec une autre branche (en général, avec *master*)

Branches

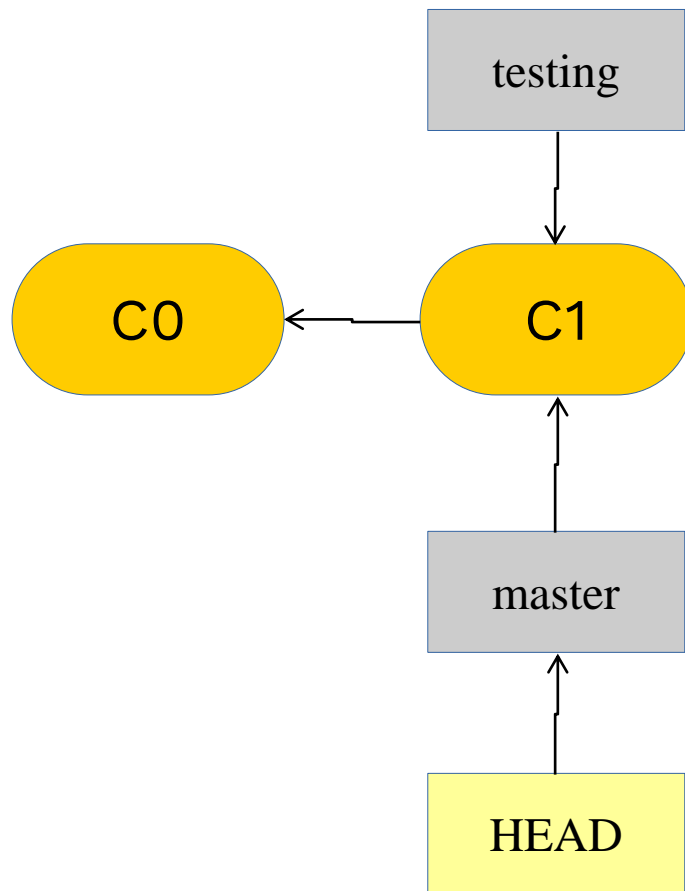
- Sous Git, les branches sont très utilisées
- Leur gestion par Git est très facile
- Par défaut, le dépôt utilise une branche *master*
- Créer une branche revient à créer une nouvelle tête de liste à partir de laquelle on pourra chaîner les instantanés.
- Le chaînage se fait toujours via HEAD

Chaînage des versions

Version en cours : C1



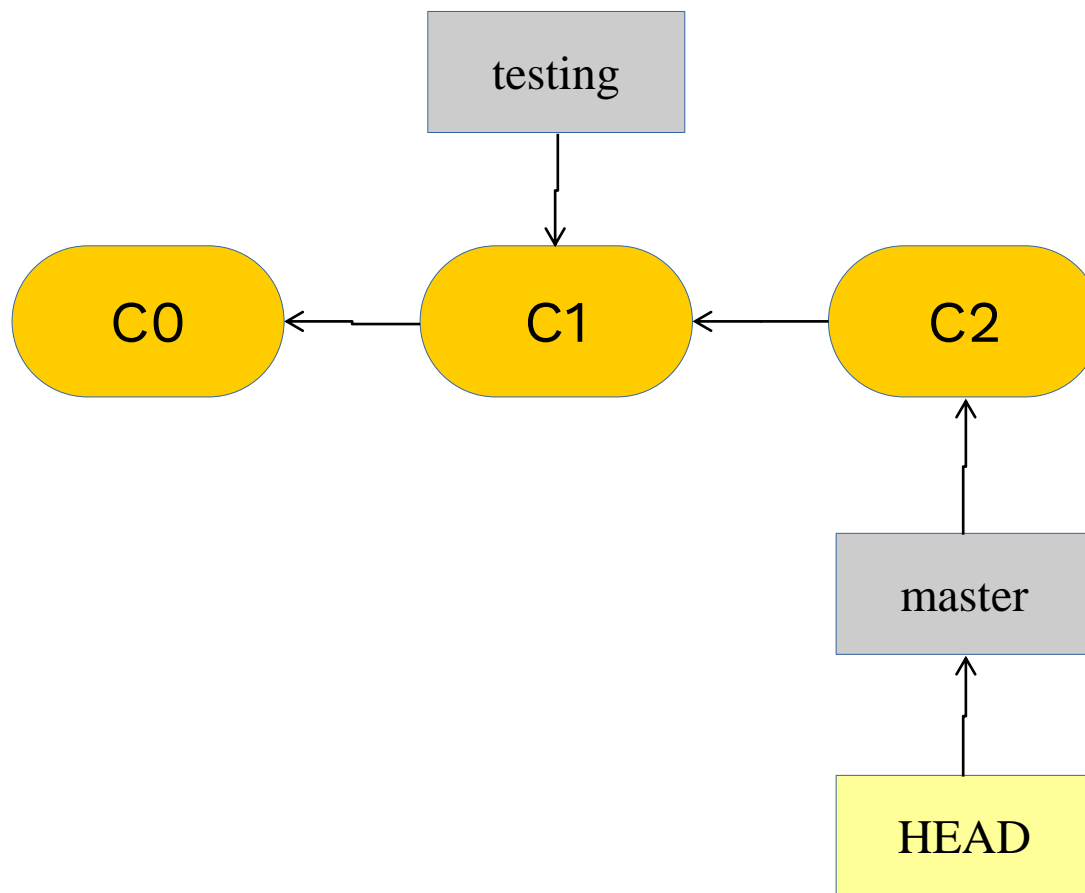
Création d'une branche



Création d'une branche :
`$ git branch testing`

On travaille toujours dans la
branche *master* car c'est
elle qui est désignée par
HEAD

Travail dans *master*

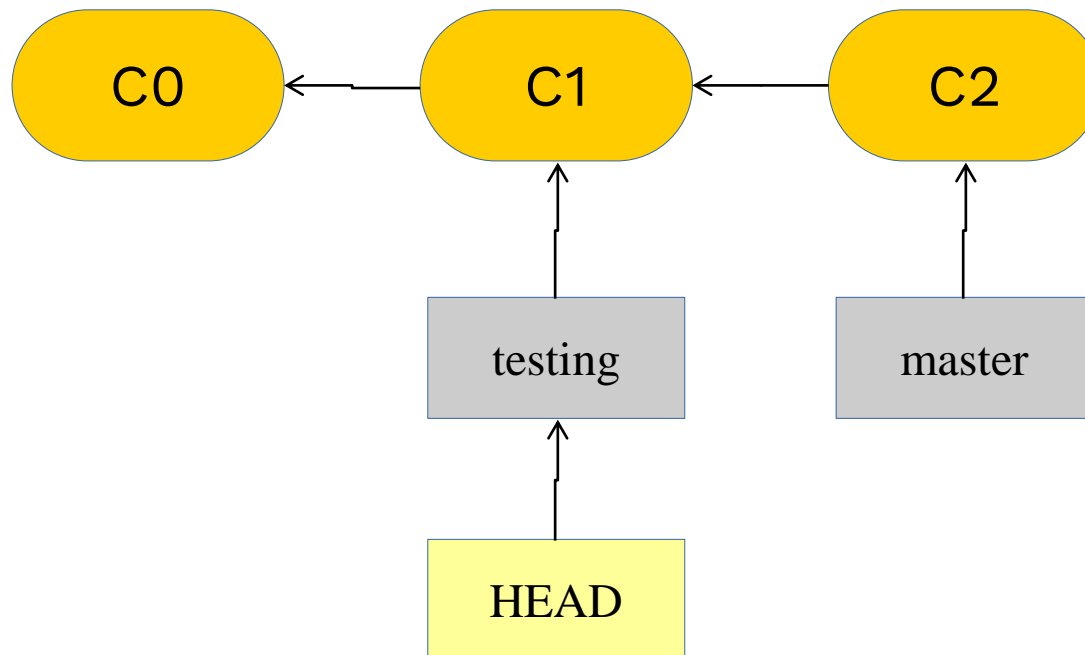


Validation d'un instantané :
\$ git commit

Il est donc ajouté à la branche *master* et les pointeurs sont déplacés pour refléter l'état de la liste

Changement de branche

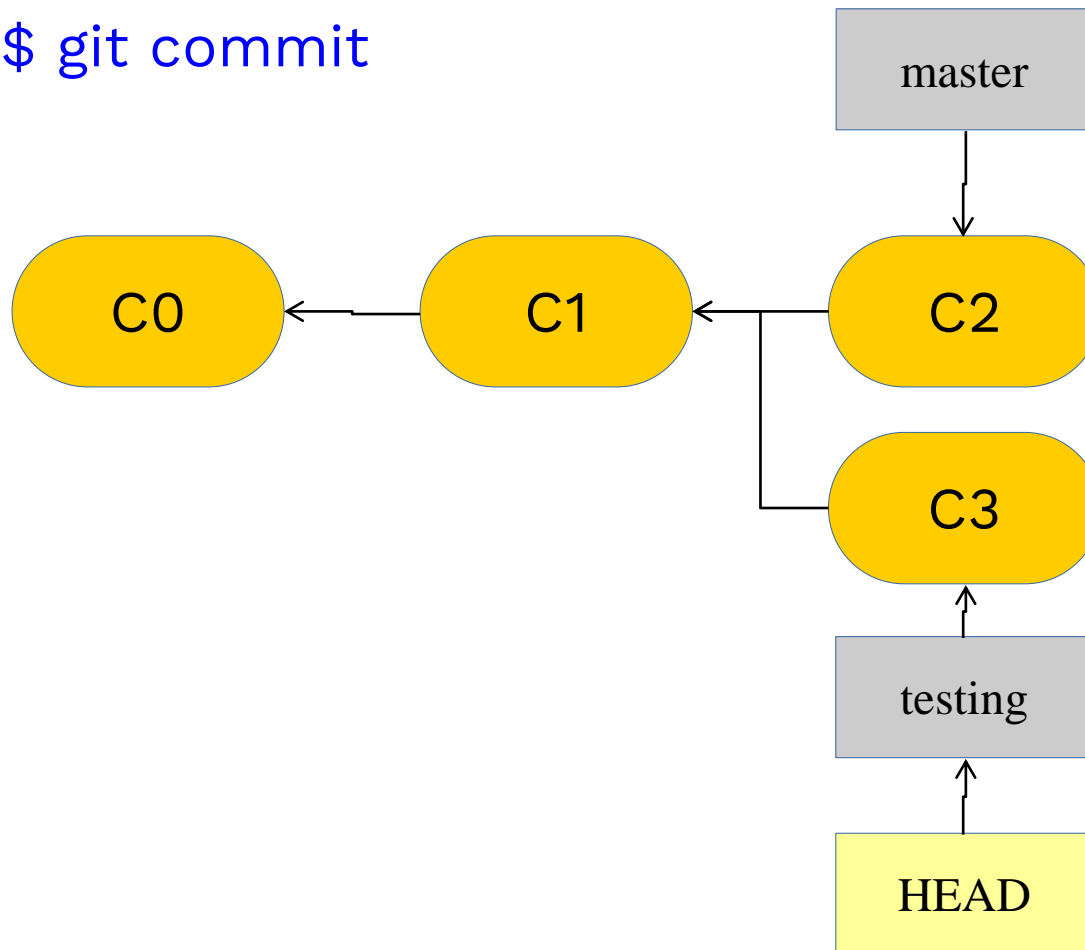
Changement de branche :
\$ git checkout testing



IMPORTANT :
git a également synchronisé les fichiers de votre répertoire de travail avec la version pointée par *testing*
Il faut donc avoir fait un commit avant de changer de branche, sinon...

Travail dans la branche

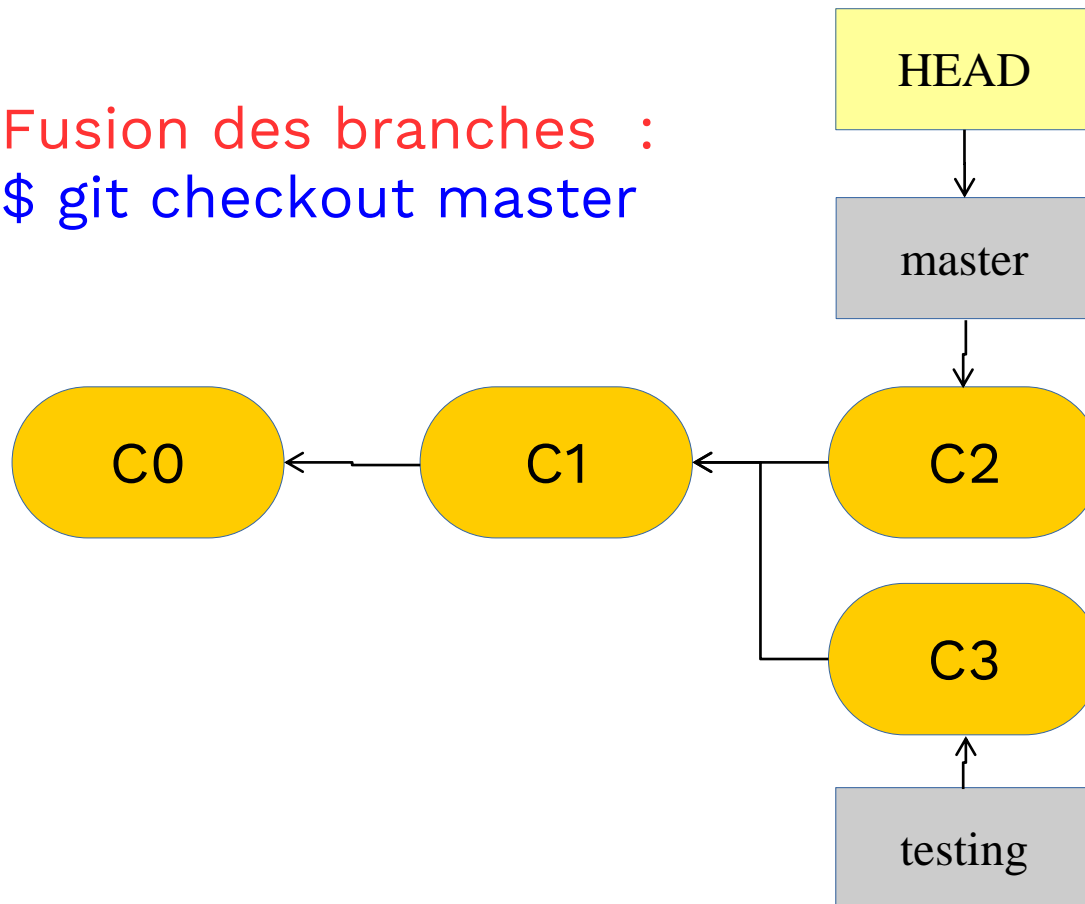
Validation d'un instantané :
\$ git commit



Cette fois-ci HEAD désigne *testing*, c'est donc cette branche qui accueille le commit

Fusion de branches (v1)

Fusion des branches :
\$ git checkout master



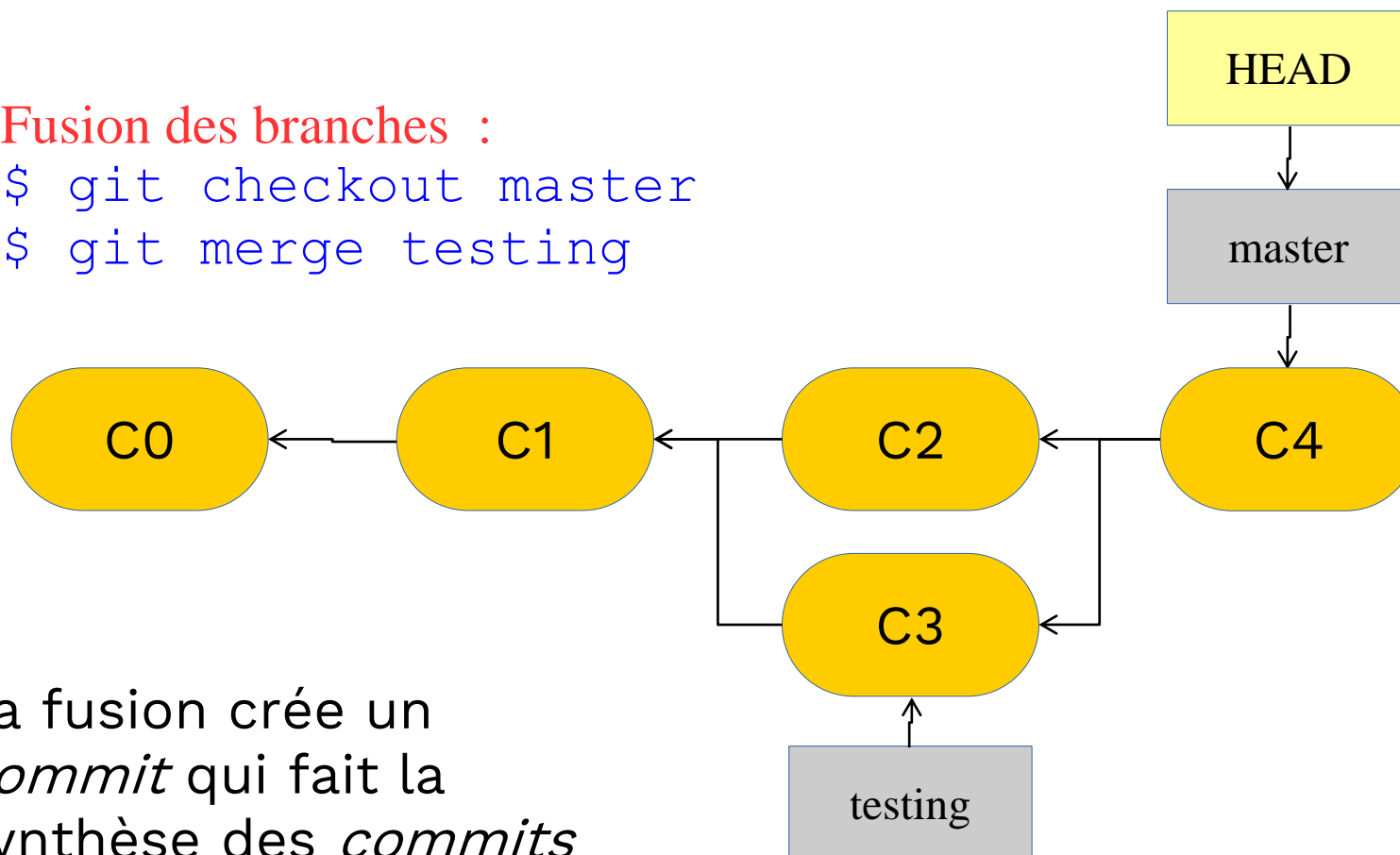
On se place sur la
branche **d'arrivée**

Fusion de branches (v1)

Fusion des branches :

```
$ git checkout master
```

```
$ git merge testing
```



La fusion crée un *commit* qui fait la synthèse des *commits* fusionnés

Elle a conservé C2 et C3, deux versions divergentes. On parle de **MERGING**

Exercice : merging

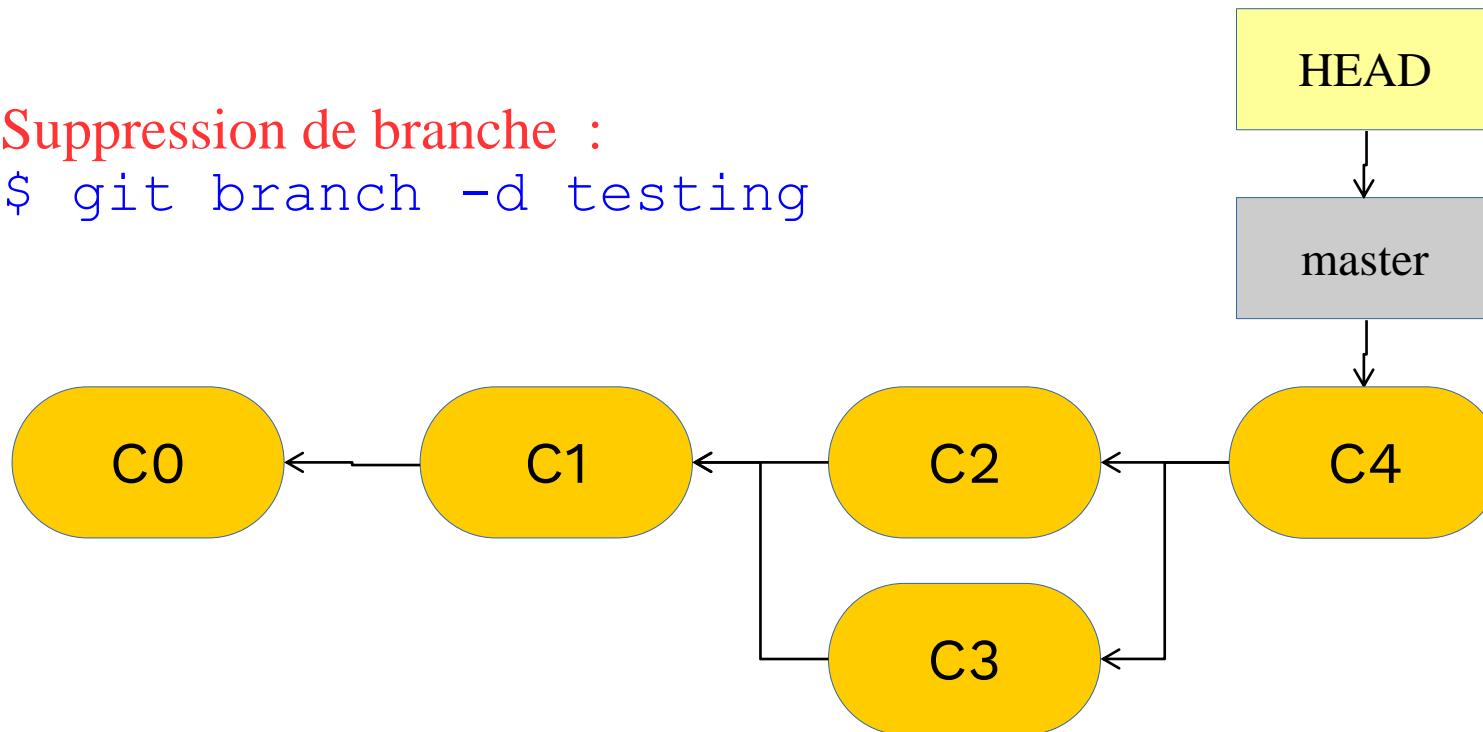
- Dessiner les branches et les pointeurs, d'après les principales étapes mentionnées (dernier commit : C3) :

```
$ git commit
$ git branch bug113
$ git commit
$ git checkout bug 113
$ git commit
$ git commit
$ git checkout master
$ git commit
$ git branch bug115
$ git merge bug113
```

Suppression après fusion

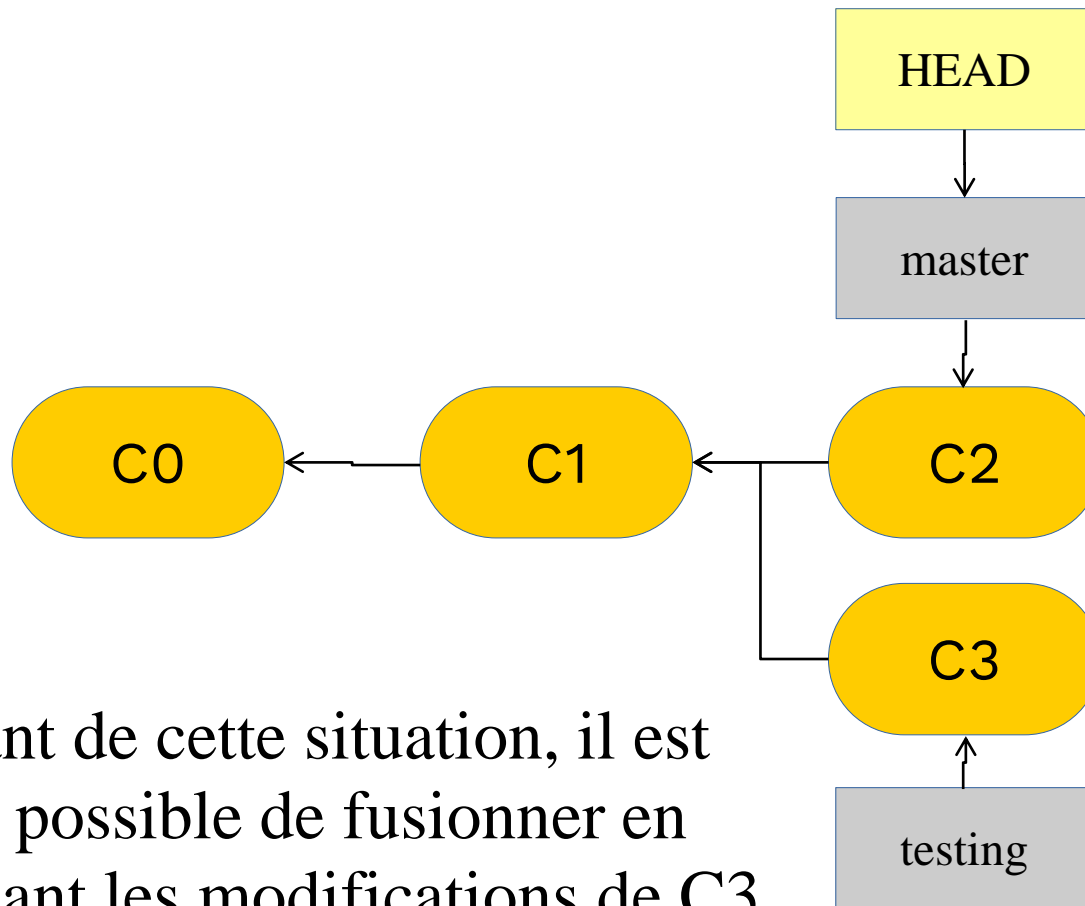
Suppression de branche :

```
$ git branch -d testing
```



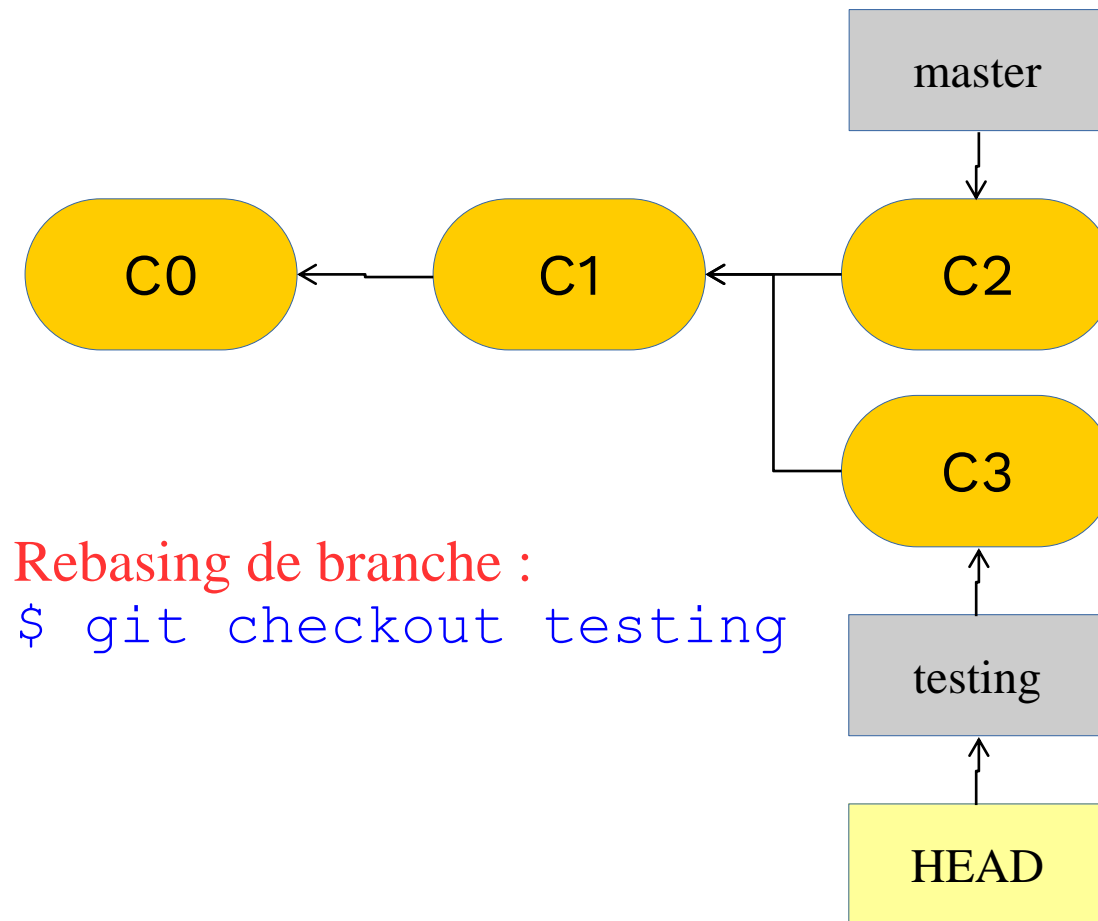
On peut, si on le souhaite, conserver un accès direct à *testing*, ou non. On peut aussi noter l'identifiant SHA1 (les 6 premiers caractères suffisent) de C3, il sera toujours possible d'y accéder par ce biais.

Fusion de branches (v2)



Partant de cette situation, il est aussi possible de fusionner en rejouant les modifications de C3 sur C2 (le master). C'est le **REBASING**

Fusion de branches (v2)



Rebasing de branche :

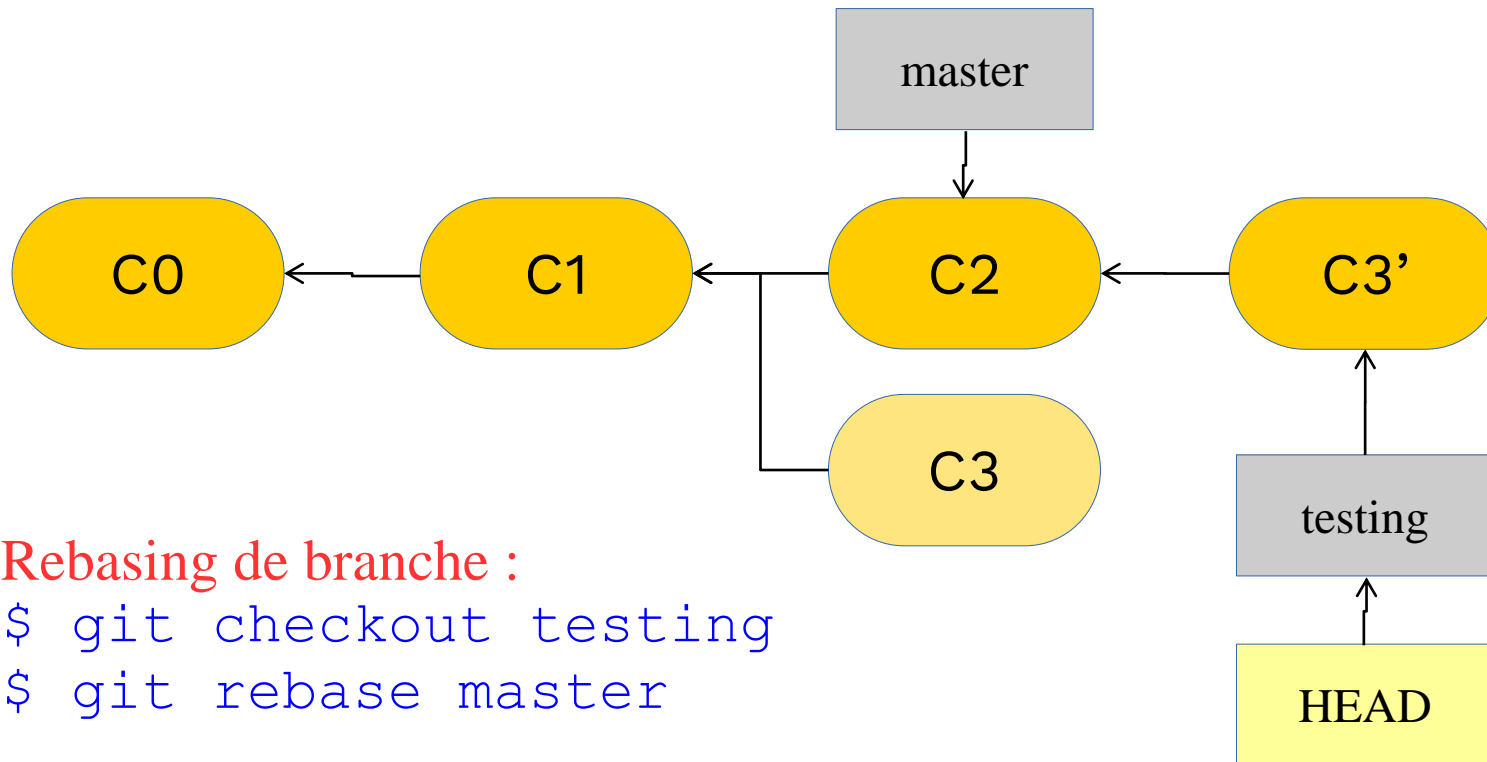
`$ git checkout testing`

ATTENTION

on se place sur la
branche qui **disparaît**, car
c'est elle qui contient les
différences par rapport à
la branche à conserver

Fusion de branches (v2)

Phase 1 : inventorie toutes les modifications, depuis l'ancêtre commun, entre *testing* et *master*



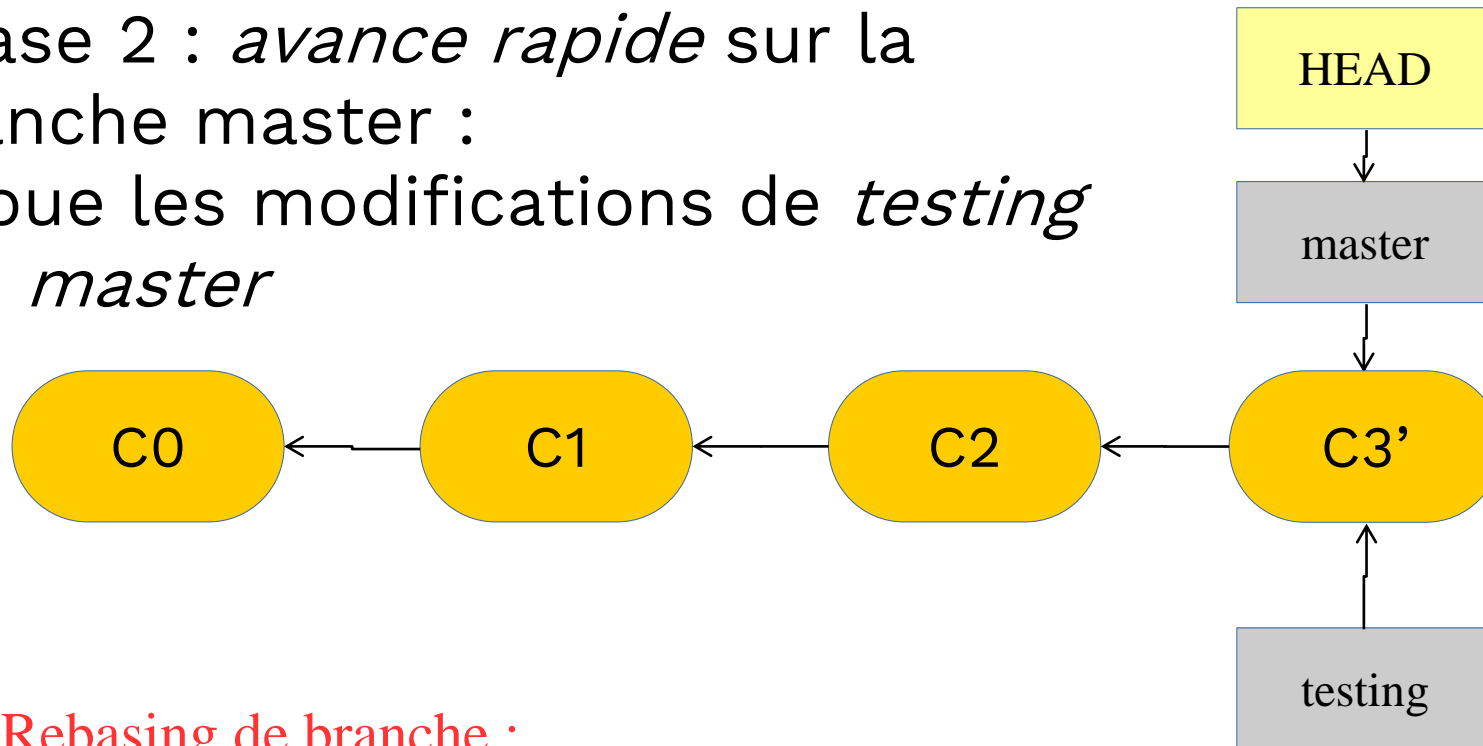
Rebasing de branche :

```
$ git checkout testing
```

```
$ git rebase master
```

Fusion de branches (v2)

Phase 2 : *avance rapide* sur la
branche master :
rejoue les modifications de *testing*
sur *master*

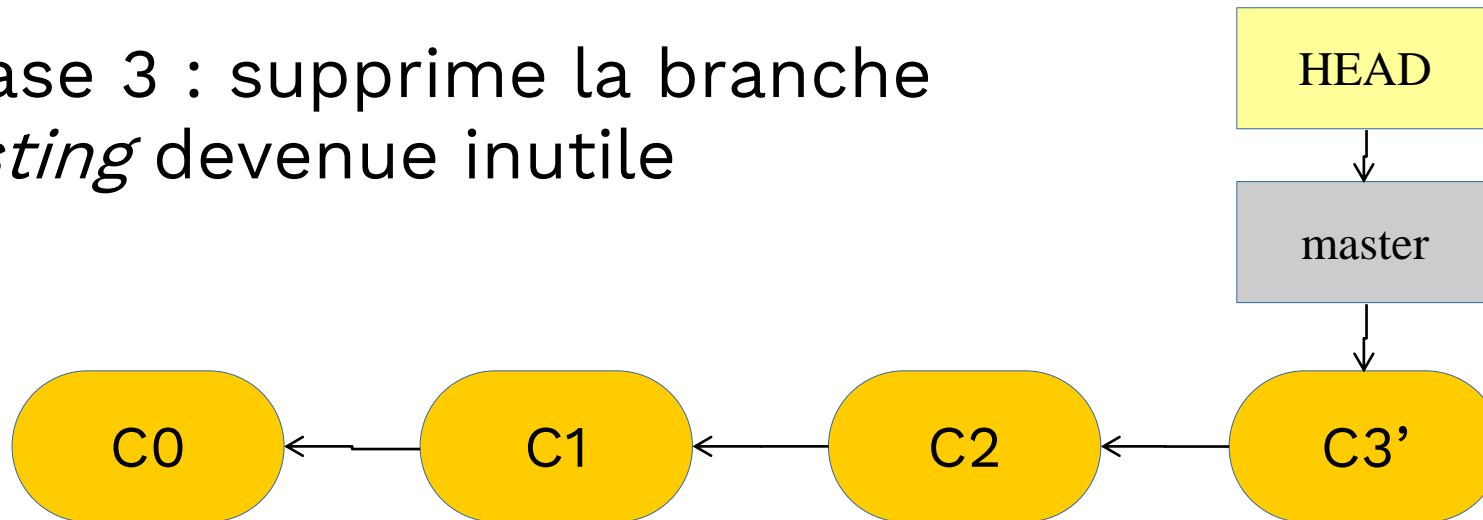


Rebasing de branche :

```
$ git checkout testing  
$ git rebase master  
$ git checkout master  
$ git merge testing
```

Fusion de branches (v2)

Phase 3 : supprime la branche *testing* devenue inutile



Rebasing de branche :

```
$ git checkout testing
$ git rebase master
$ git checkout master
$ git merge testing
$ git branch -d testing
```


Exercice : rebasing

- Dessiner les branches et les pointeurs, d'après les principales étapes mentionnées (dernier commit : C3) :

```
$ git commit
$ git branch bug113
$ git commit
$ git checkout bug 113
$ git commit
$ git commit
$ git checkout master
$ git commit
$ git branch bug115
$ git checkout bug113
$ git rebase master
```

Merging vs. Rebasing

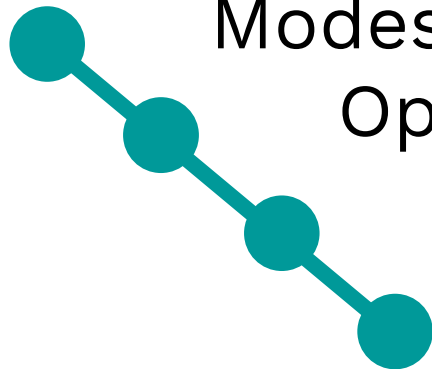
- L'instantané final est le même dans les deux cas
- L'historique du rebasing est plus lisible car linéaire
- En réseau, généralement, on pousse sur le serveur la version tirée, modifiée, puis rebasée
- On ne rebase JAMAIS des commits déjà poussés (cf. [1] section 3.6) car on risque de modifier, après poussage, un historique utilisé par quelqu'un, qui aura une grosse surprise quand il rebasera ou fusionnera

Merging ou Rebasing ?

- Le *merging* conserve l'historique, et donc la manière dont le code a évolué, avec les brouillons, les égarements, etc.
- Le *rebasing* modifie cet historique de façon à raconter la meilleure histoire qui soit, et donc effacer les traces de la construction du résultat final
- Lequel choisir ? Il n'y a pas de réponse toute faite, tout dépend de ce que vous souhaitez laisser dans l'historique.



Utilisation en réseau



Modes de gestion des dépôts
Opérations de synchronisation
Import / Export

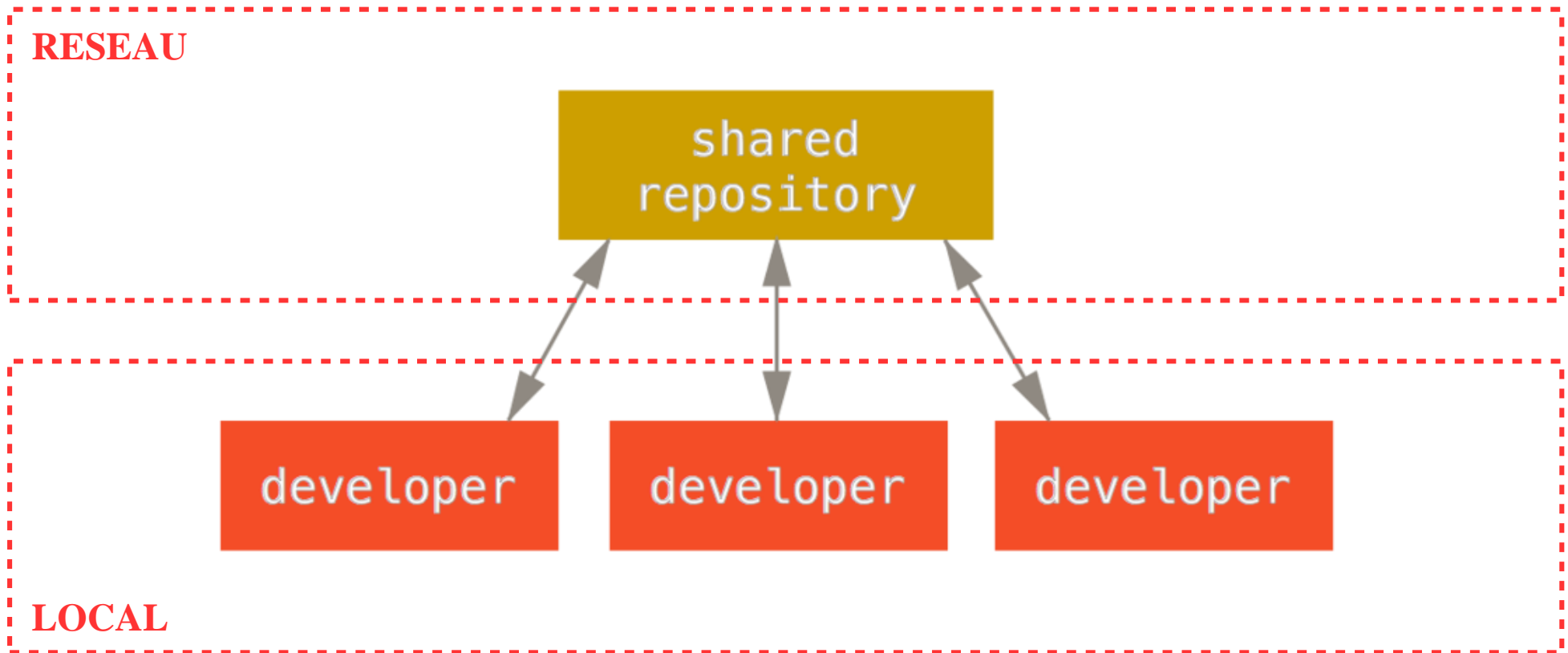
Utilisation en réseau

- Permet de partager un dépôt avec plusieurs autres personnes
- Permet d'avoir une sauvegarde en ligne
- Permet de travailler à plusieurs, simultanément, sur le même projet
- Rappel : le travail est en local, il faut partager ensuite en *poussant* sur le réseau

Mode centralisé traditionnel

- Fonctionnement identique aux CVCS
- Un dépôt sert de référence
- Les développeurs copient le projet depuis ce dépôt et modifient le projet à leur rythme
- Chacun pousse ses modifications vers le dépôt : premier arrivé premier servi !
- Git gère les conflits : la fusion des modifications doit avoir lieu en local avant. Si Git ne peut pas résoudre le conflit, il faut le résoudre à la main, indexer et valider

Mode centralisé traditionnel

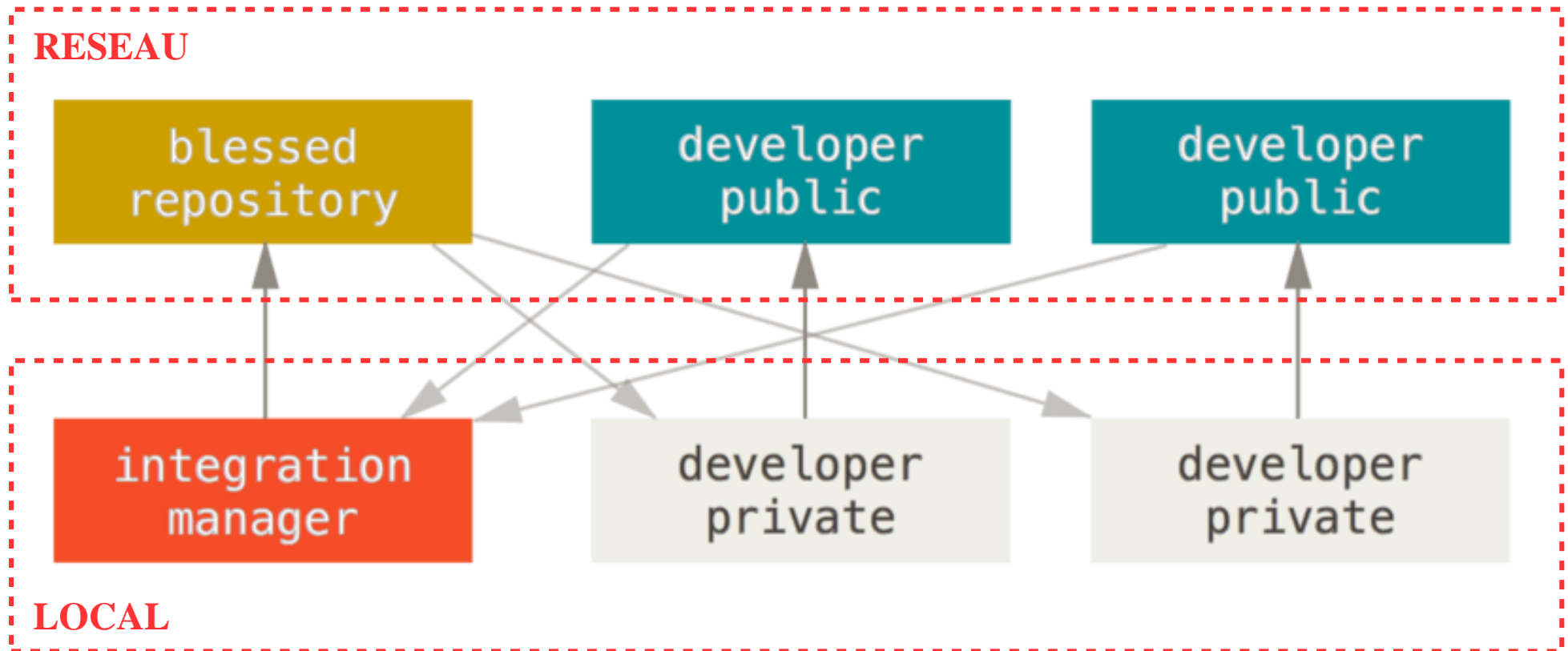


Source : [1]

Mode gestion d'intégration

- Le projet est déposé sur un dépôt de référence, en lecture pour les développeurs
- Chaque développeur dispose de son propre dépôt à distance, où il poussera ses modifications
- Ces dernières seront intégrées au projet par l'administrateur du projet après les avoir *tirées* des dépôts des développeurs

Mode gestion d'intégration



Source : [1]

Mode dictateur-lieutenants

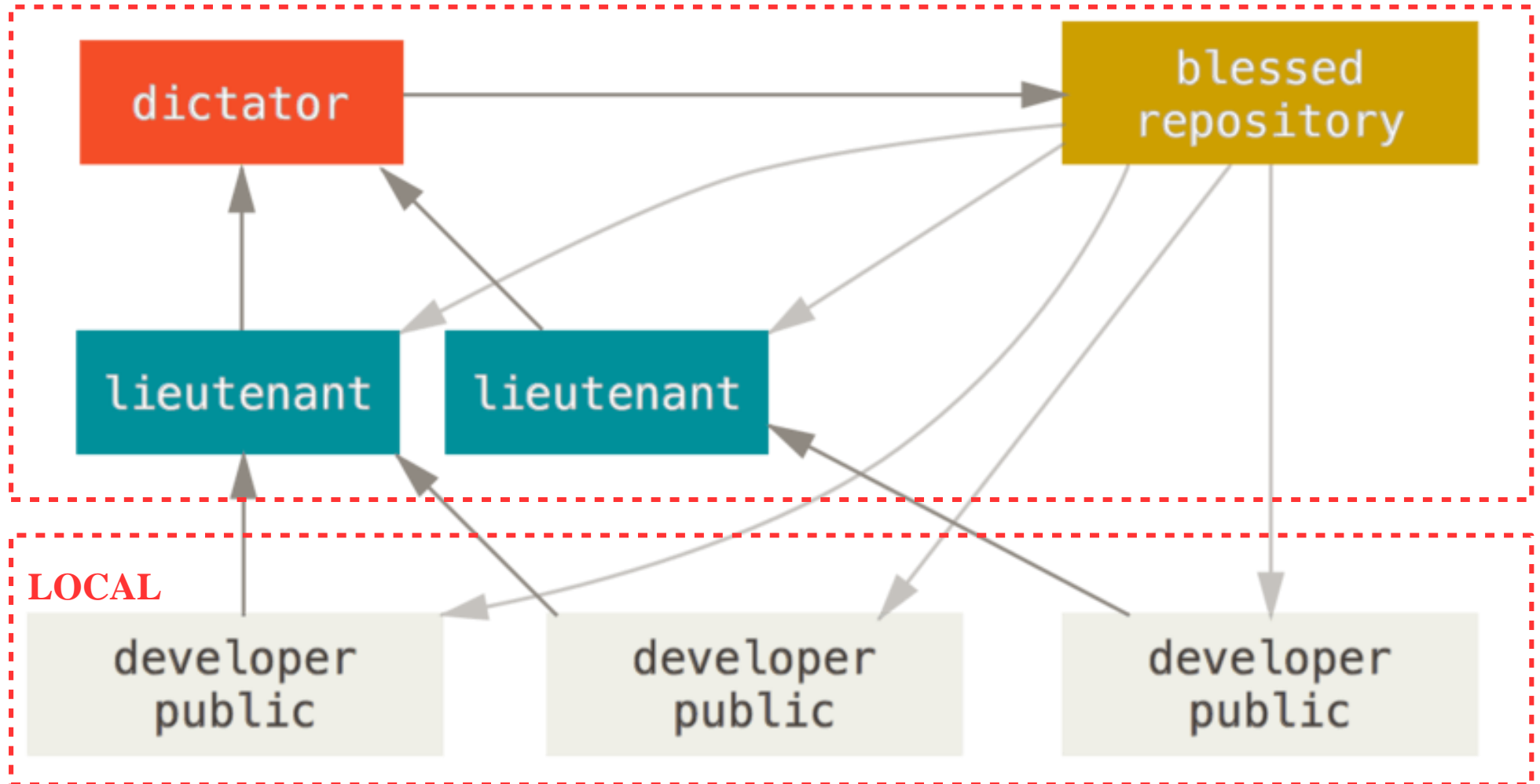
- Même fonctionnement qu'en gestion d'intégration, avec un niveau intermédiaire
- Certaines parties sont déléguées à des lieutenants, chargés de la fusion des modifications de leur partie
- Les lieutenants servent de source pour le gestionnaire global, appelée dictateur
- Approche « diviser pour régner » adaptée aux gros projets (ex. noyau linux)

Mode dictateur-lieutenants

- Les simples développeurs travaillent sur la branche thématique et *rebasent* leur travail sur master. La branche master est celle du dictateur.
- Les lieutenants fusionnent les branches thématiques des développeurs dans leur propre branche master.
- Le dictateur fusionne les branches master de ses lieutenants dans sa propre branche master.
- Le dictateur pousse sa branche master sur le dépôt de référence pour que les développeurs se *rebasent* dessus.

Mode dictateur-lieutenants

RESEAU



Source : [1]

Import d'un projet distant

- Copie du projet distant *project.git*
 - `git clone serveur:project.git`
 - Création du pointeur *origin* (nom par défaut) qui désigne le serveur
 - Le répertoire local *project* est créé
 - La branche distante *master* importée est désignée par *origin/master*
 - La branche locale d'import est *master*

Mise à jour depuis un projet

- Ré-import, en local, du projet.
 - Les derniers instantanés présents sur le serveur sont importés
 - Cela ajuste *origin/master*
 - On ré-importe toujours avant de fusionner localement de façon à faciliter la tâche de l'intégrateur distant

❑ `git pull`

- Import de la *rev2* du serveur pointé par *origin* dans la branche locale (pointée par HEAD)

❑ `git pull origin rev2`

Quelques commandes

- Envoi, vers le serveur par défaut du projet, dans la branche distante master, de la branche locale pointée par HEAD

- ❑ `git push origin master`

- Connexion à un autre serveur

- ❑ `git remote add origin serveur`

Exemple de structure de dépôt

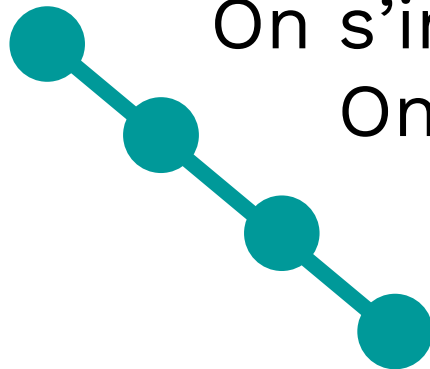
- Retour d'expérience à lire en ligne :
 - A successful Git branching model (V.Driessen), voir référence [6]
- [1] contient également de nombreux cas d'organisation de dépôt git, pour des situations locales, centralisées, distribuées, etc.

Quelques conseils

- La branche *master* doit rester propre et servir à conserver la version stable
- On crée une branche, on modifie, on teste, et on finit par rebaser (ou merger)
- On fait de petits commits, il est plus facile de localiser les changements en cas de bug
-



GitHub et les autres



On s'inscrit
On partage
On collabore !

Github ?

- <http://github.com>
- Près de 30 millions de comptes en 2018
- Gratuit pour les comptes personnels, les étudiants, les projets open source...
- Permet de mettre à disposition, publiquement ,et gratuitement, des dépôts
- Propriété de Microsoft, depuis juin 2018

Github et la collaboration

- Lorsqu'on a participé à un projet et qu'on souhaite que notre apport soit intégré au projet original
- On *commit* sur notre compte distant
- Github permet d'ouvrir une « pull request »
- Une notification, de notre apport, au propriétaire du projet
- Ce canal permet de discuter, de nos modifications, avec le propriétaire
- Le propriétaire peut récupérer notre apport et le fusionner dans son projet puis re-publier le résultat

Y a pas que Github dans la vie !

- [SourceForge](#) utilisait CVS, puis SVN, il supporte aussi Git
- [GitLab](#) est une autre plateforme, semblable à Github
- [FramaGit](#) est une plateforme libre, gérée par Framasoft et basée sur GitLab
- [Savannah](#), la forge de la Free Software Foundation, hébergeant les projets GNU
- ...

C'est pas bientôt fini ?

- Et non ! Git est très riche, et dispose de beaucoup de commandes. La lecture de [1] n'est pas optionnelle
- Il y a de multiples façons de collaborer, voir [6] notamment. À vous de découvrir les vôtres
- C'est facile à utiliser en local, pour tout ce qui a vocation à évoluer : projets (codes, docs, rapports), notes de cours, versions graphiques de sites...

Alors, qu'attendez vous ?

Références

- ❑ [1] Pro Git ; S.Chacon & B.Straub, Apress, 512 p., 2018
- ❑ [2] <https://git-scm.com> (site officiel)
- ❑ [3] <http://www.ietf.org/rfc/rfc3174.txt>
- ❑ [4] <https://git-scm.com/downloads/guis>
- ❑ [5] <https://www.eclipse.org/egit/>
- ❑ [6] <https://nvie.com/posts/a-successful-git-branching-model/>