Git

pour la

Gestion des Configurations

Préparé & présenté Par : Wajdi ZAIRI
LinkedIn : linkedin.com/in/wajdi-zairi-b1793531

➢ *Intégration continue*

❑ *L'intégration classique*

1. Définir avec le client un certain nombre de fonctionnalités
2. Codage (sans communication avec client ou livraison intermédiaires)
3. Intégration

○***Inconvénients :***

- Code testé juste avant la livraison
- Code non maintenable
- Plein de bugs qui remontent => des retards

⇒***Client non satisfait***

❑*L'intégration continue:*

Quoi :« Ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée »

- Chacun de ces outils a pour but d'améliorer la qualité globale du code
- C'est un concept Agile
- L'intégration continue fait partie des 12 méthodes d'eXtrême Programming (XP)
- **eXtreme Programming** : Méthode agile basée sur des bonnes pratiques de développement informatique.

Valeurs de XP

Communication
Simplicité
Feedback
Respect

Méthodes de XPs

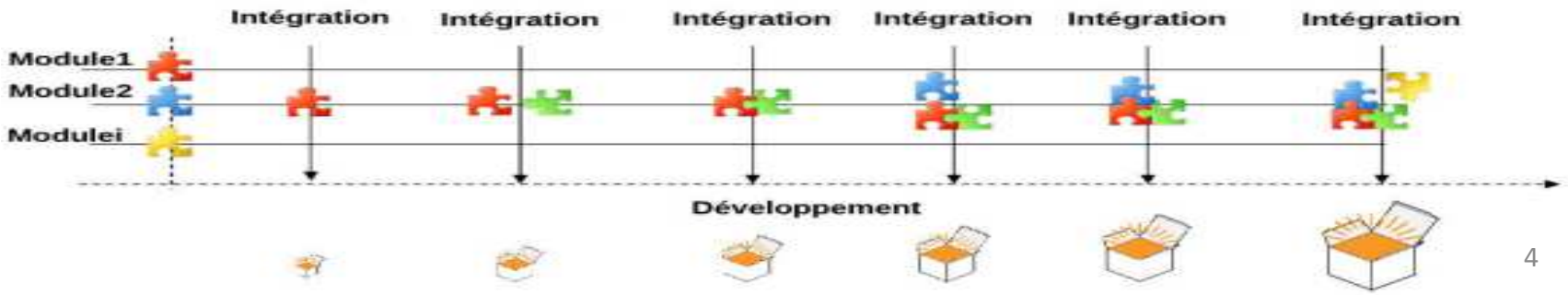
Client sur site
Jeu du Planning
Intégration continue
Petites livraisons
Rythme soutenable
Tests de recette (ou tests fonctionnels)
Tests unitaires
Conception simple
Refactoring
Appropriation collective du code
Convention de nommage
Programmation en binôme

Pourquoi :

- Coût : Plus un défaut est détecté tard et plus il couterait cher à corriger.
- Des demandes de démonstrations ou des releases urgentes non planifiées ?

Comment :

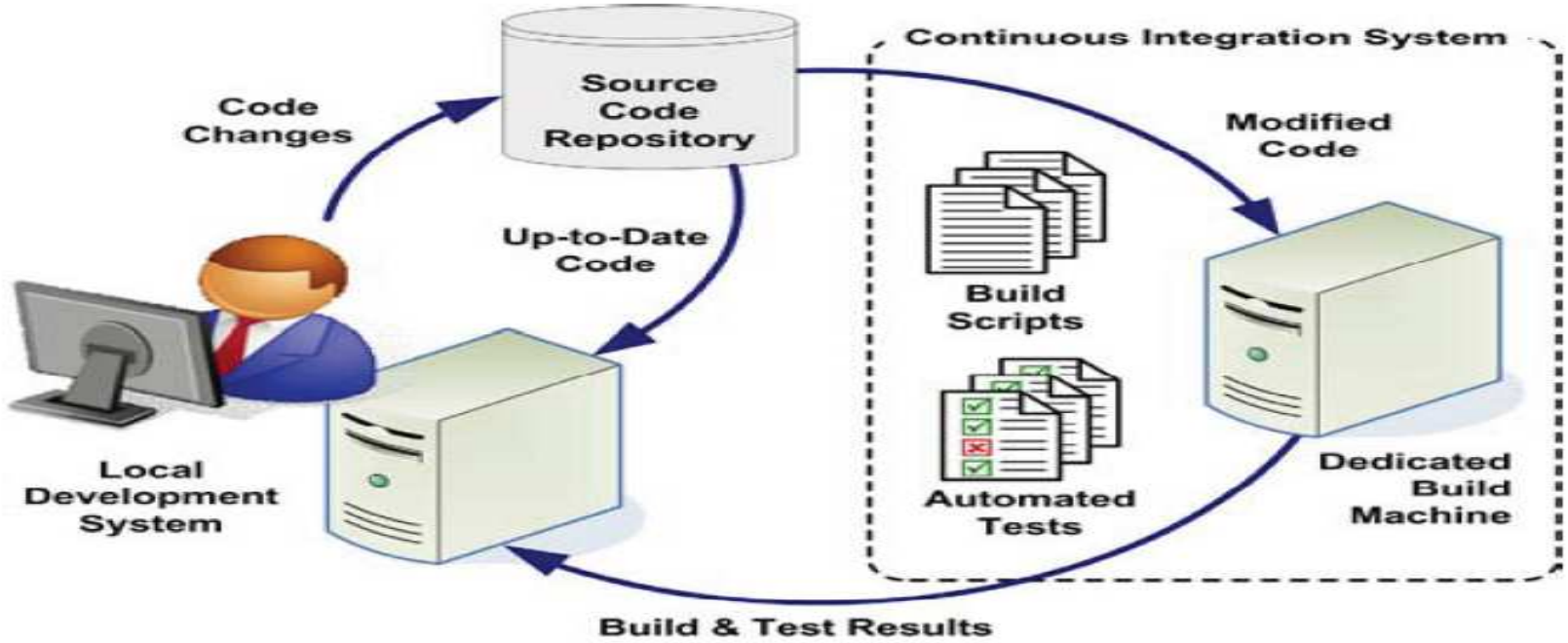
- Tester à chaque modification
- Assurer la non régression
- Reporter
- Automatiser (exp : les builds, les tests ..)



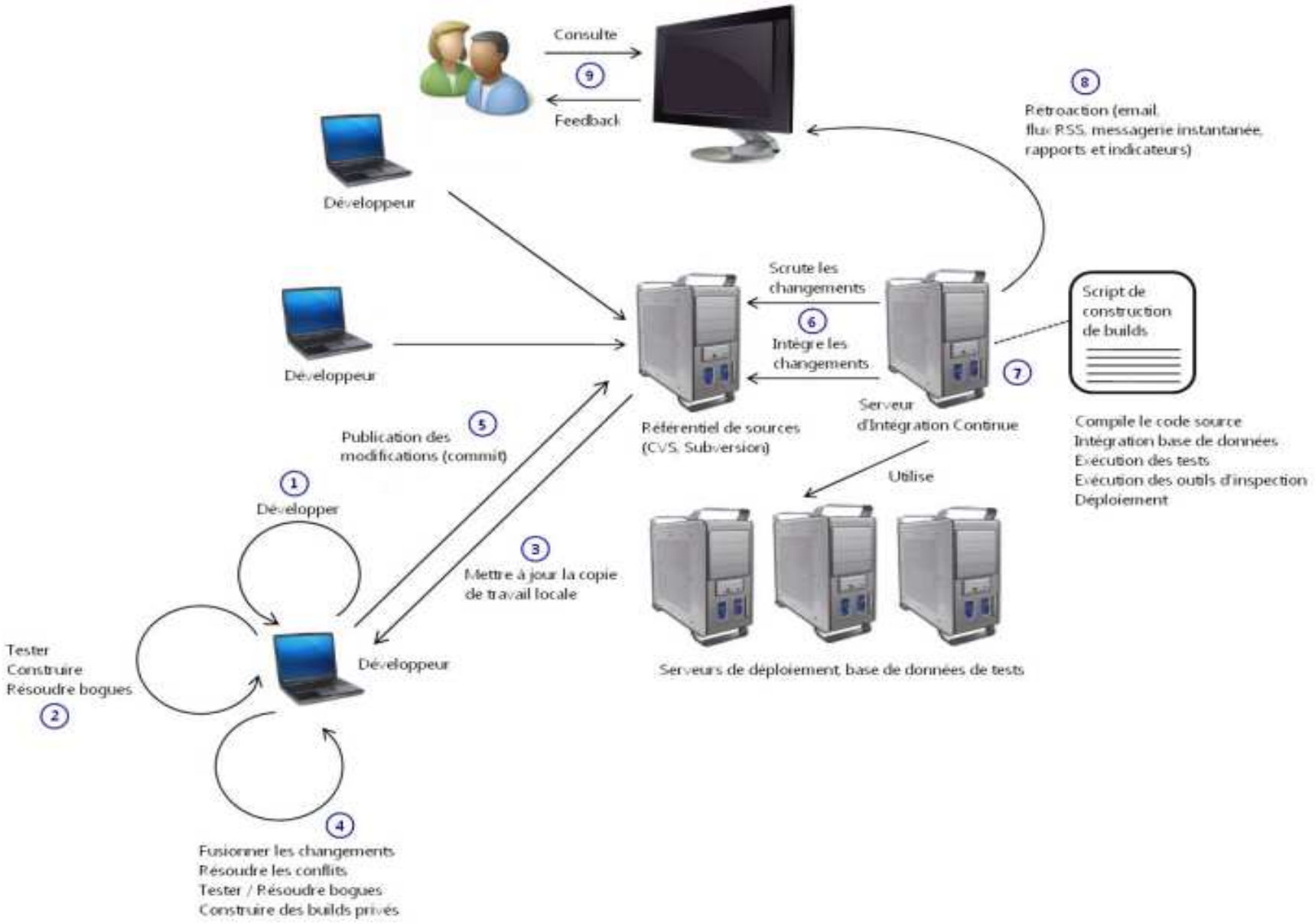
Architecture :

L'architecture comporte différents éléments n'intervenant pas uniquement dans l'intégration continue mais aussi dans le développement :

- Un gestionnaire de code source
- Un serveur de build
- Un serveur d'analyse de code
- Une équipe de développeurs
- Un outil de reporting ou/et d'un serveur de suivi de bug



Etapes :


Consulte
9
Feedback
8
Retroaction (email,
flux RSS, messagerie instantanée,
rapports et indicateurs)
Développeur
Développeur
Scrute les
changements
6
Intègre les
changements
Script de
construction
de builds
7
Référentiel de sources
(CVS, Subversion)
Serveur
d'Intégration Continue
Compile le code source
Intégration base de données
Exécution des tests
Exécution des outils d'inspection
Déploiement
Publication des
modifications (commit)
5
1
Développer
Utilise
3
Mettre à jour la copie
de travail locale
Tester
Construire
Résoudre bogues
2
Développeur
Serveurs de déploiement, base de données de tests
4
Fusionner les changements
Résoudre les conflits
Tester / Résoudre bogues
Construire des builds privés

○ *Avantages :*

- Plus de productivité
- Time to market réduit
- Gagner au niveau coût
- Garantir la qualité de code
- Maintenabilité
- Testabilité
- Rapidité et réactivité pour faire face aux divers problèmes pouvant être présents dans les différentes phases du projet.

➢ *VCS : Version Control System*

Quoi:

Un gestionnaire de version est un système qui :

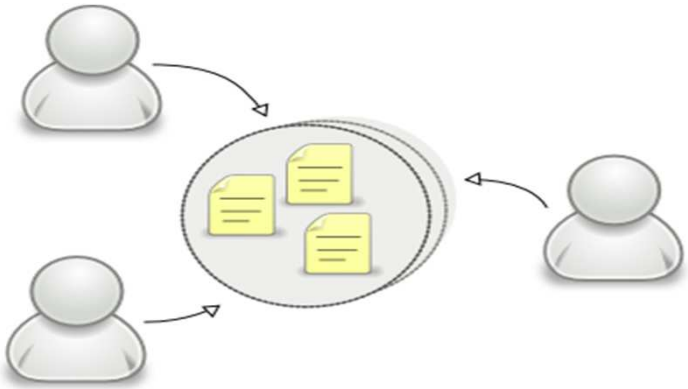
- Enregistre l'évolution d'un fichier ou d'un ensemble de fichiers au cours du temps de manière à ce qu'on puisse rappeler une version antérieure d'un fichier à tout moment.
- Ramener un fichier à un état précédent
- Ramener le projet complet à un état précédent
- Visualiser les changements au cours du temps
- Qui a modifié quoi
- Quand la modification est introduite
- Permet facilement de revenir à un état stable.

- **Types**
- **VCS Locale :**
- Copier les fichiers dans un autre répertoire (sorte de base de données locale)
- Un utilisateur puisse contrôler les révisions d'un fichier

EXP: RCS
Revision Control System
Crée en 1982

- **VCS Centralisé:**

• L'enjeu majeur : besoin de collaborer avec d'autres
• Centraliser toutes les versions des fichiers dans un seul serveur central
-> Plusieurs clients interagissent avec ce dernier pour récupérer (faire des checkout) leurs fichiers et contrôler leurs versions.

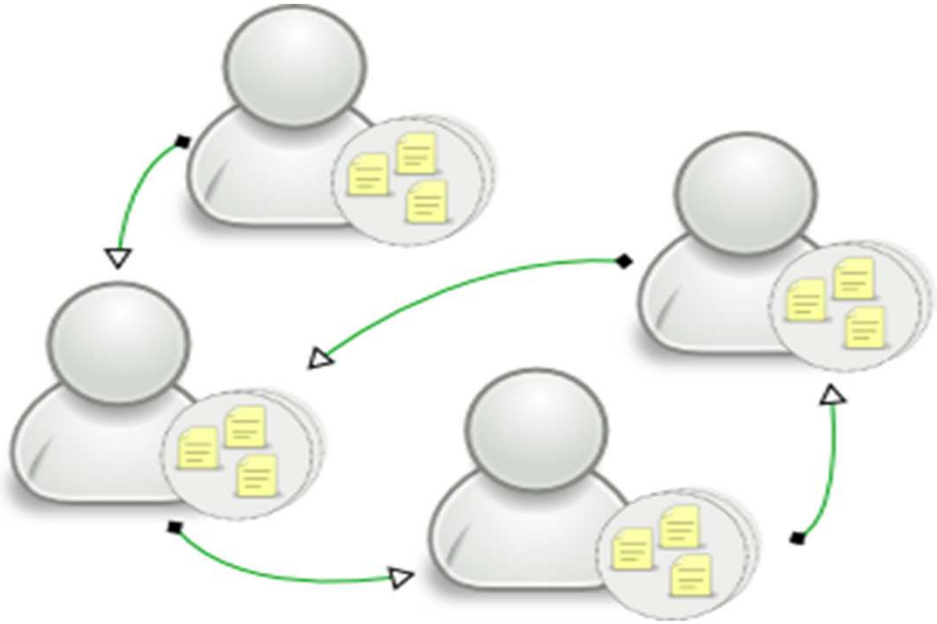


EXP:
CSV(*Concurre nt Versions System)*
En 1990

EXP: SVN (Subversion)
En 2000

- **VCS Distribué:**

- Chaque « checkout » est vraiment une sauvegarde complète de toutes les données
- Les dépôt distribués reflètent complètement le dépôt central.
- Si un serveur tombe en panne, l'un des dépôts des clients pourra être copié sur le serveur pour le restaurer.
- Opérations communes (commit, historique, et l'annulation des modifications) sont rapides, car il n'y a pas besoin de communiquer avec un serveur central.
- La communication est seulement nécessaire lors du partage de changements parmi d'autres pairs.



➢ *Introduction GIT*

❑ Historique

- **En 2002**, le projet du noyau Linux commença à utiliser un DVCS propriétaire appelé BitKeeper
- En 2005, les relations entre la communauté développant le noyau linux et la société en charge du développement de BitKeeper furent rompues

- **En 2005** la communauté open source développe leur propre outil

❑ Objectifs

- ✓ vitesse
- ✓ conception simple
- ✓ support pour les développements non linéaires (milliers de branches parallèles)
- ✓ complètement distribué
- ✓ capacité à gérer efficacement des projets d'envergure tels que le noyau Linux (vitesse et compacité des données).

➢ Concept et Architecture

❑ Stockage et Intégrité

- Tous les objets sont stockés comme du contenu comprimés par leur empreintes SHA-1 :

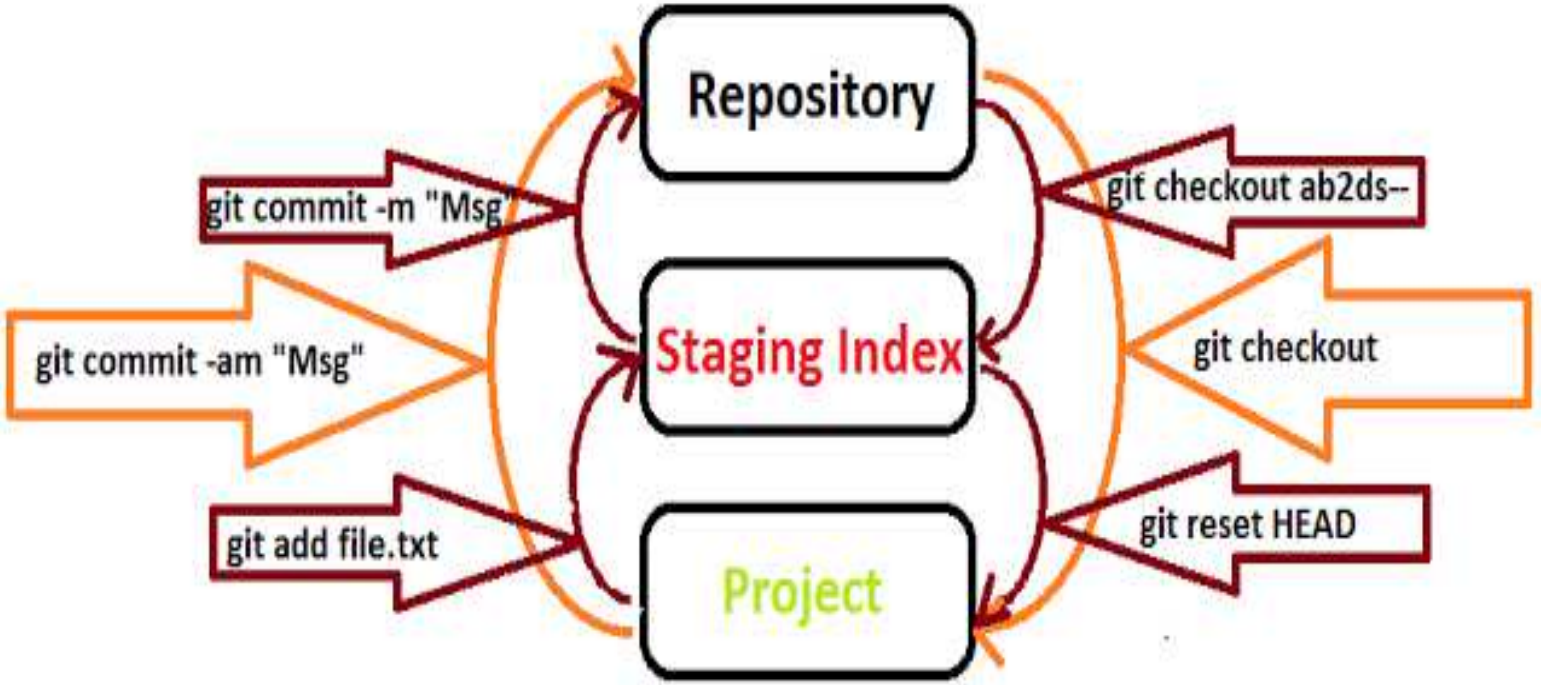
  Exp SHA : ab04d884140f7b0cf8bbf86d6883869f16a46f65

- Cette signature unique, sert comme référence
- Très difficile de le perdre les modifications commitées ou lorsque on synchronise la base de données locale avec un dépôt distant.

❑ Les Etats

- **Validé** :données stockées en sécurité dans votre base de données locale.
- **Modifié** :fichier modifié mais pas encore validé en base.
- **Indexé :** marqué un fichier modifié dans sa version actuelle pour qu'il fasse partie du prochain instantané du projet.


Local Operations
working directory
staging area
git directory (repository)
checkout the project
stage files
commit


Repository
Staging Index
Project
git commit -m "Msg"
git commit -am "Msg"
git add file.txt
git checkout ab2ds--
git checkout
git reset HEAD

❑ *Protocole de GIT*

Git utilise des protocoles pour transporter les données :
(Local, *Secure Shell*(SSH), HTTP).

➢ *Local*

Le dépôt distant est un autre répertoire dans le système de fichiers
Clonage /update rapide
$ git init /path/locale/projet.git

Inconvénients

- Pour les PC distants, ils doivent faire un montage NFS (transfert plus long des fichiers)
- Les dépôts sont hébergés le même ordinateur => toute défaillance catastrophique.

➢ *SSH*

C'est le plus utilisé , il permet :
Plus rapide
Un accès authentifié
Données sont chiffrées
$ git clone ssh://utilisateur@serveur/projet.git

Inconvénients

S'oppose à l'optique open source (même en lecture seule il faut s'authentifier)

➢ *HTTP/S*

- Plus simple à mettre en place
- Donner un accès public en lecture à votre dépôt Git ne nécessite que quelques commandes

$ git clone http://link.com/projetgit.git

Inconvénients

- Prend généralement beaucoup plus longtemps de cloner ou tirer depuis le dépôt
- Plus de trafic réseau et de plus gros volumes de transfert que pour les autres protocoles

➢ *Installer GIT*

- Systèmes Linux (Ubuntu)

```
$ sudo apt-get install git
```

- Systèmes Windows

Le build le plus officiel est disponible sous :

http://git-scm.com/download/win

➢ *Créer un repo GIT*

- Positionner dans le répertoire du projet et saisir :

$ git init

$ git init <directory>

$ git init --bare <directory> : Initialiser un dépôt Git vide, mais sans le répertoire de travail.

✓ crée un nouveau sous-répertoire nommé .git

✓ contient un squelette de dépôt Git

✓ Rien n'est encore suivi en version

```
ll .git/
COMMIT_EDITMSG
config
description
FETCH_HEAD
HEAD
hooks/
index
info/
logs/
objects/
ORIG_HEAD
refs/
```

➢ *Configurer GIT*

Git supporte 3 niveaux/types de configurations :

Types :

- **Locale :** Spécifique à chaque dépôt local, elle est stockée dans le **.git/config**.
- **Globale :** Spécifique à l'utilisateur, elle est stockée à la racine de son compte, dans le fichier **~/.gitconfig**.
- **Système :** Généralement stockée dans **/etc/gitconfig**, elle est partagée par tous les utilisateurs.

$git config (--local , --global,--system)

Les plus utiles :

➢ **Qui :**

- $ git config --global user.name "Your Name“
- $ git config --global user.email your@email.address

➢ **Editeur :**

- $ git config --global core.editor « editor_name »

➢ **Color (Diff ,status …)**

- $git config --global color.status.changed yellow

➢ **whitespace**

- $ git config --global apply.whitespace nowarn

➢ Alias config

$ git config --global alias.co checkout

$ git config --global alias.ci commit

$ git config --global alias.st status

$ git config --global alias.br branch

➢ Diff tool config

Git passe 7 arguments au logiciel de comparaison :

path old-file old-hex old-mode new-file new-hex new-mode

On a besoin que des deux valeurs : old-file et new-file

```
$ cat path/MergeTool
#!/bin/sh
« path_to_merge_tool »/mymergeTool.exe $*


$cat path/DiffTool
#!/bin/sh
 [ $# -eq 7 ] && path/MergeTool "$2" "$5"
```

$ sudo chmod +x MergeTool
$ sudo chmod +x DiffTool

- $ git config --global merge.tool path/**`MergeTool`**
- $ git config --global mergetool.**MergeTool**.cmd \ 'MergeTool \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"'
- $ git config –global mergetool.**MergeTool**.trustExitCode false
- $ git config --global diff.external path/**`DiffTool`**

➢ **Annuler une configuration**

$ git config --global --unset diff.external

➢ **Lister les configurations**

$ git config --list

➢ Ignorer des fichiers (.gitignore)

- Utilisé par Git pour determiner les fichiers ou les repertoires à ignorer

  (exp : dans les commits , status, add ..)
- Ce fichier doit être commité (partagé) pour être utilisé par les développeurs qui clone votre projet.

Comment :

- Créer un fichier .gitignore

```
# Les ligne commençant par '#' sont des commentaires.
# Ignorer tous les fichiers nommés foo.txt
foo.txt
# Ignorer tous les fichiers html
*.html
# à l'exception de foo.html qui est maintenu à la main
!foo.html
# Ignorer les objets et les archives
*.[oa]
```

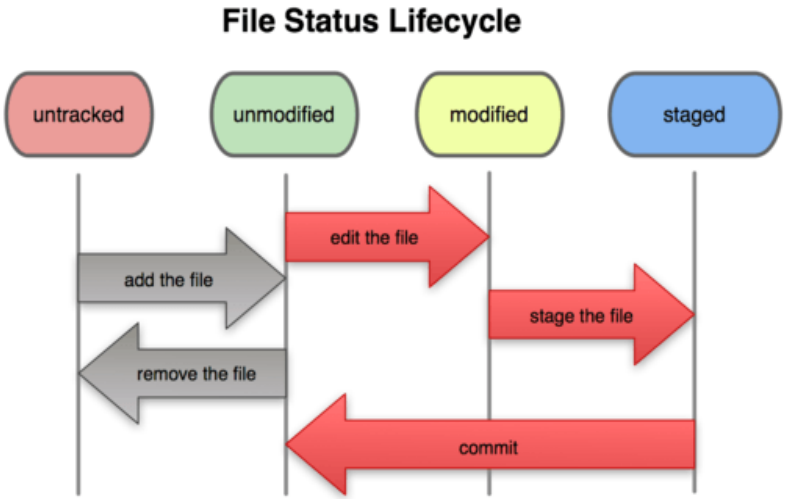
- Configurer

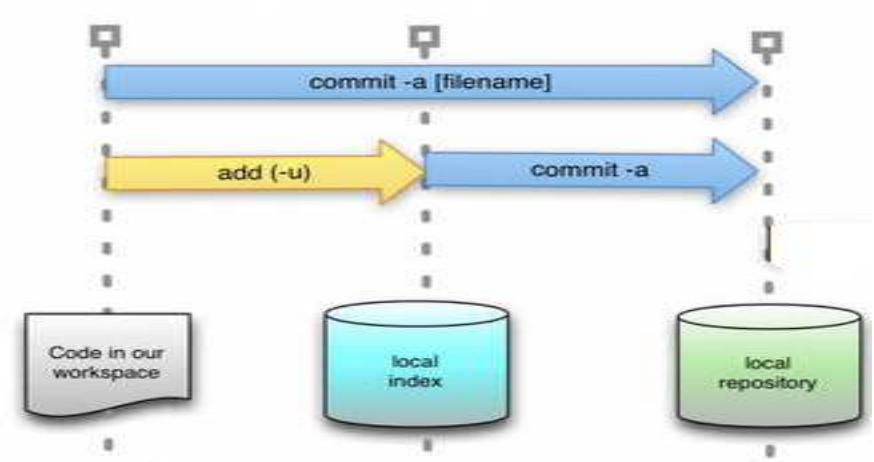
$ git config --global core.excludesfile ~/.gitignore_global

➢ *Introduction au notion de Commit/Patch*

❑ Commit :

- Git se base sur la composition et l'enregistrement des instantanés(Snapshot) de votre projet et ensuite travaille avec.
- Le cycle de vie des états des fichiers :





Créer un commit :

step 1 : Modifier/créer le fichier

step 2 :

$ git status (pour vérifier avant d'indexer)

$ git add file (indexer le fichier modifé)

$ git commit –m « commit message »

Configurer le Template de Commit

$ vi ~/.myCommitTemplate.txt and Save

$ git config --global commit.template ~/.myCommitTemplate.txt

$ git commit

❑Patch:

- Un morceaux de code qui permet de résoudre un problème existant
- Facilement testé, examiné et documenté.
- Un patch est un fichier structuré qui consiste en une liste des différences entre un ensemble de fichiers et un autre.
- Facilite le développement (car au lieu de fournir un fichier de remplacement, pouvant être constitué de milliers de lignes de code, le patch ne comprend que les changements exacts qui ont été faites)

➢ Un patch est une liste de toutes les modifications apportées à un fichier, qui peuvent être utilisées pour recréer ces changements sur une autre copie de ce fichier.

Créer un patch :

$ git format-patch -1 « SH1 » --stdout > MyPatchName.patch

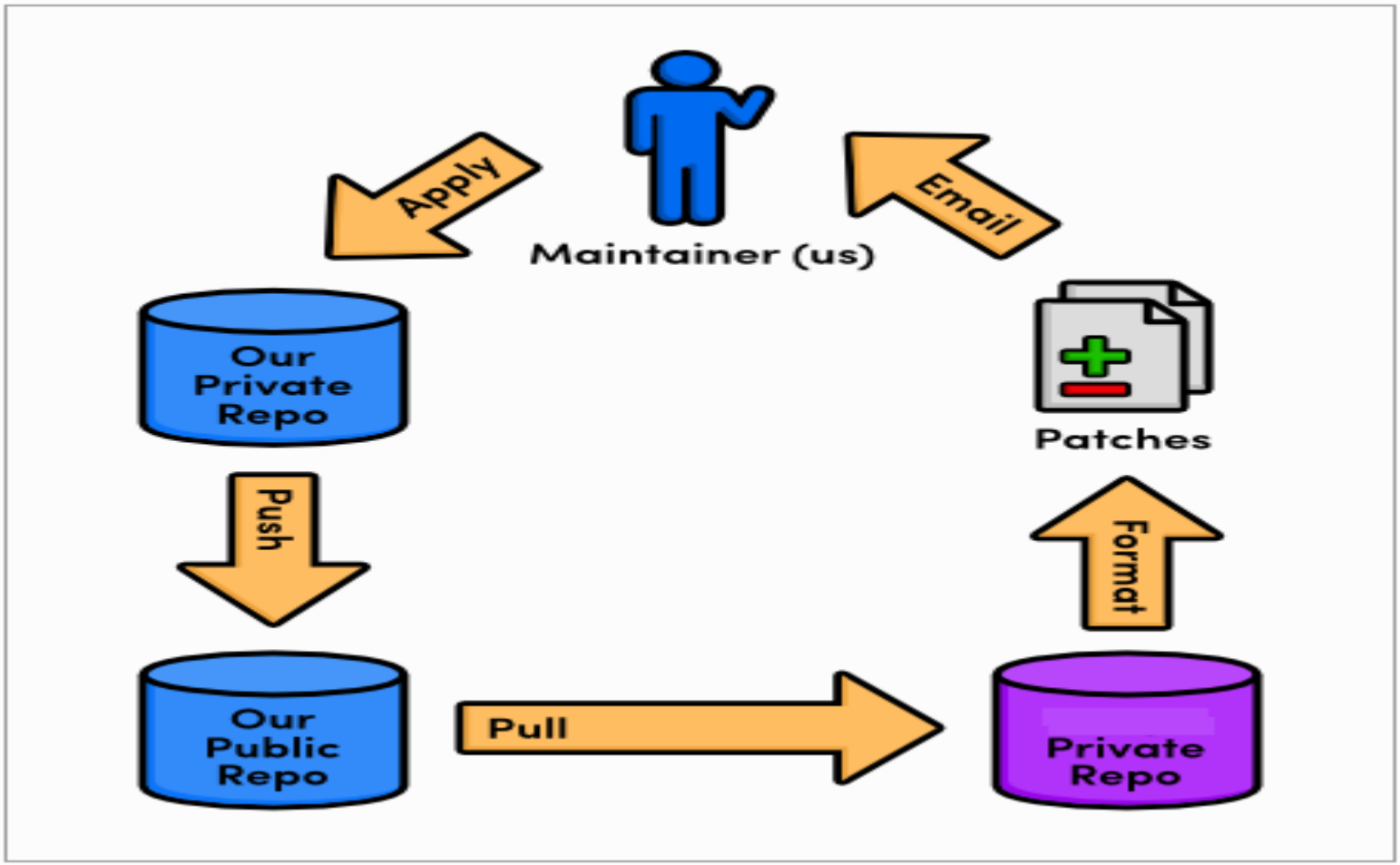
Appliquer un patch

$ git apply --stat <name>.patch (check diff)

$ git apply --check <name>.patch (check applicability)

$ git am --signoff < myPatch.patch

Patch workflow



➢ *Annuler les changements :*

Pour annuler des changements Git fournit ces outils :

Checkout :

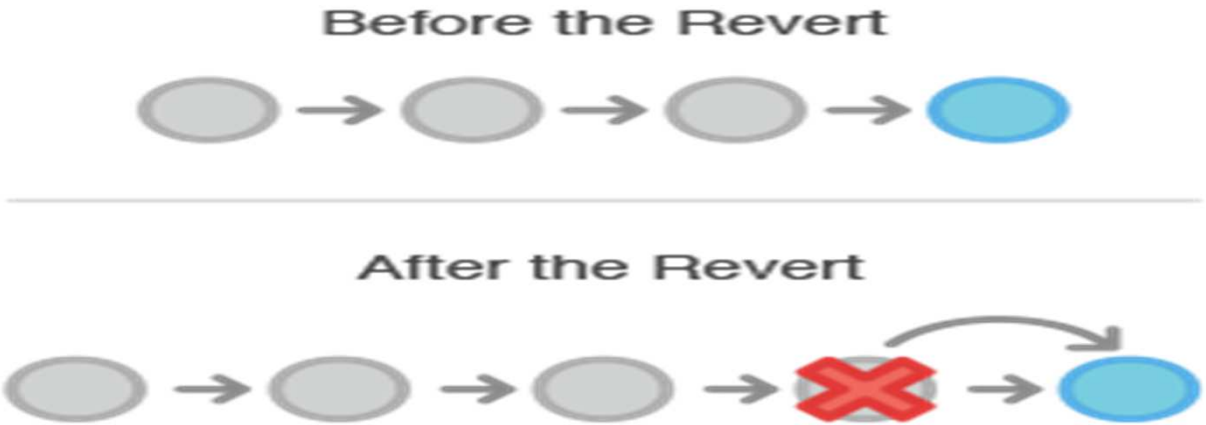
Sert pour checkouter des commits , files , branches

$ git checkout « nom de fichier »

Checkout le fichier de HEAD, écrase les changements.

Revert :

- Défait un commit engagé.
- Au lieu d'enlever le commit de l'historique du projet,il ajoute un nouveau commit avec le contenu résultant.
- Cela évite de perdre l'historique des changments
- Usage **: $ git revert « commit »**



Reset :

- Permet d'enlever des snapshot/commit engagés.
- souvent utilisé pour annuler des changements dans le staging area ou dans répertoire de travail.
- Il ne doit être utilisé pour annuler les modifications/commits déjà partagés avec d'autres développeurs***. IL PEUT ETRE UTILISE PAR QUELQU'UN D'AUTRE***
- ***Usage :***
- ***$git reset « file »***
- ***$git rest –hard « commit »***

Si vous avez besoin de fixer un COMMIT publique => Utiliser « **git revert** » qui est spécialement conçu à cet effet.

Clean :

- Supprime les fichiers non suivis à partir de votre répertoire de travail.
- Assurez-vous vraiment supprimer les fichiers non-suivis avant de l'exécuter.

Usage :

$ git clean –n : Liste les fichiers qui vont être enlevés.

$ git clean –f : Supprimer les fichiers non-suivis depuis le répertoire courant

$ git clean –df (files and directories)

$ git clean –xf (n’utilise pas le .gitignore)

➢ *Les objets GIT*

❑ *SHA*

Toutes les informations nécessaires pour décrire l'historique d'un projet sont stockées dans des fichiers référencés par un « nom d'objet » :

SHA-1 : secure hash algorithme : c'est une focntion de hachage cryptographique

& vi test_SHA.txt & add text

$ git hash-object -w test_SHA.txt

$ ls ./git/objects/ ou $ find .git/objects -type f

$ git cat-file –p « le SHA1 »

Avec SHA1 il est virtuellement impossible de trouver deux objets différents avec le même nom.

Chaque objet se compose :un **type**, une **taille** et le **contenu**.

❑ *Type d'Objet:*

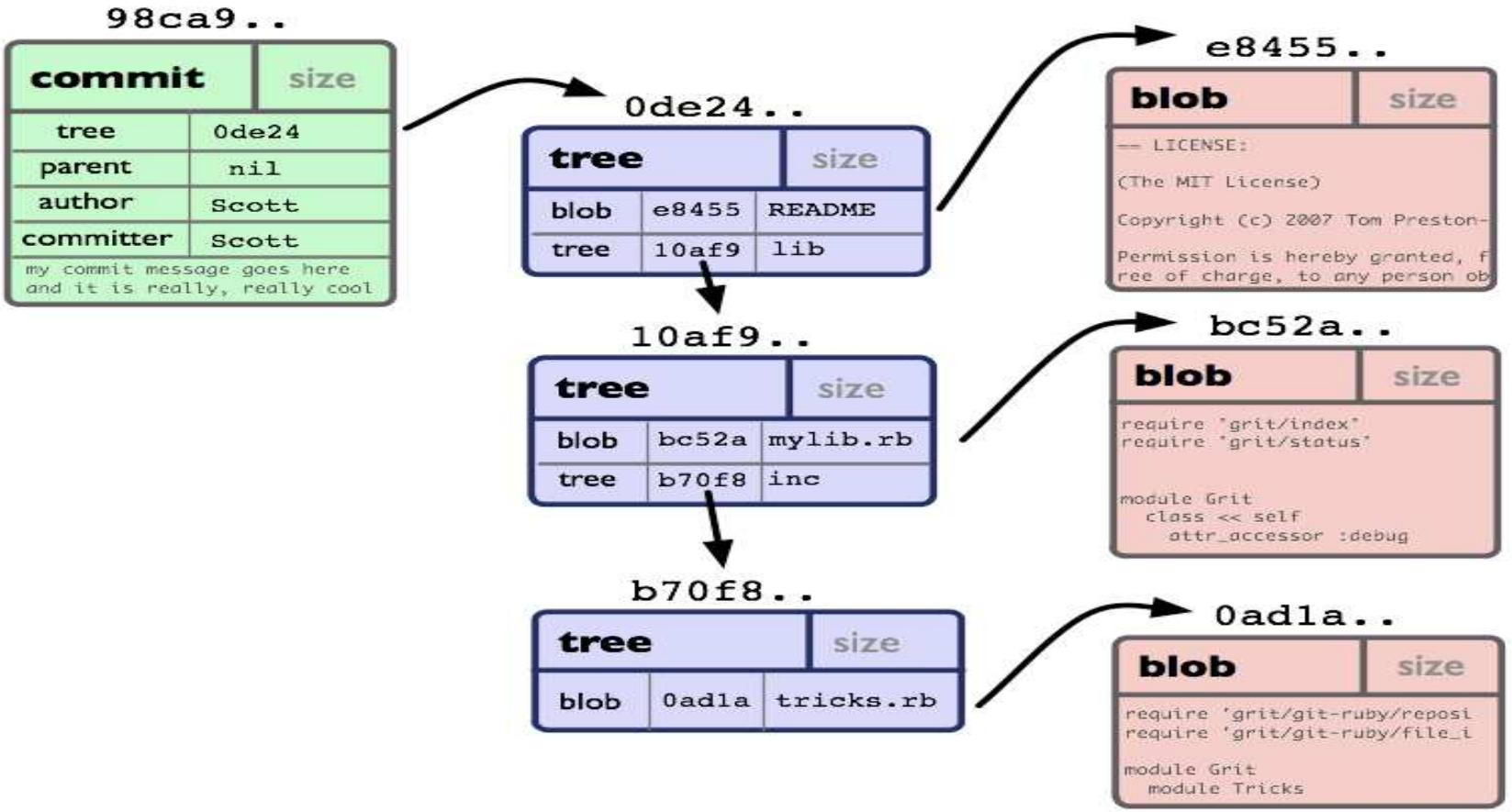
- ***B*lob:** utilisé pour stocker les données d'un fichier — il s'agit en général d'un fichier.
  Remarque: Si deux fichiers dans un répertoire (ou dans différentes versions du dépôt) ont le même contenu, ils partageront alors le même objet blob

- **Tree :** C'est comme un répertoire — il référence une liste d'autres « tree » et/ou d'autres « blobs »

- **Commit** pointe vers un unique "tree" et le marque afin de représenter le projet à un certain point dans le temps. Il contient des méta-informations à propos de ce point dans le temps.

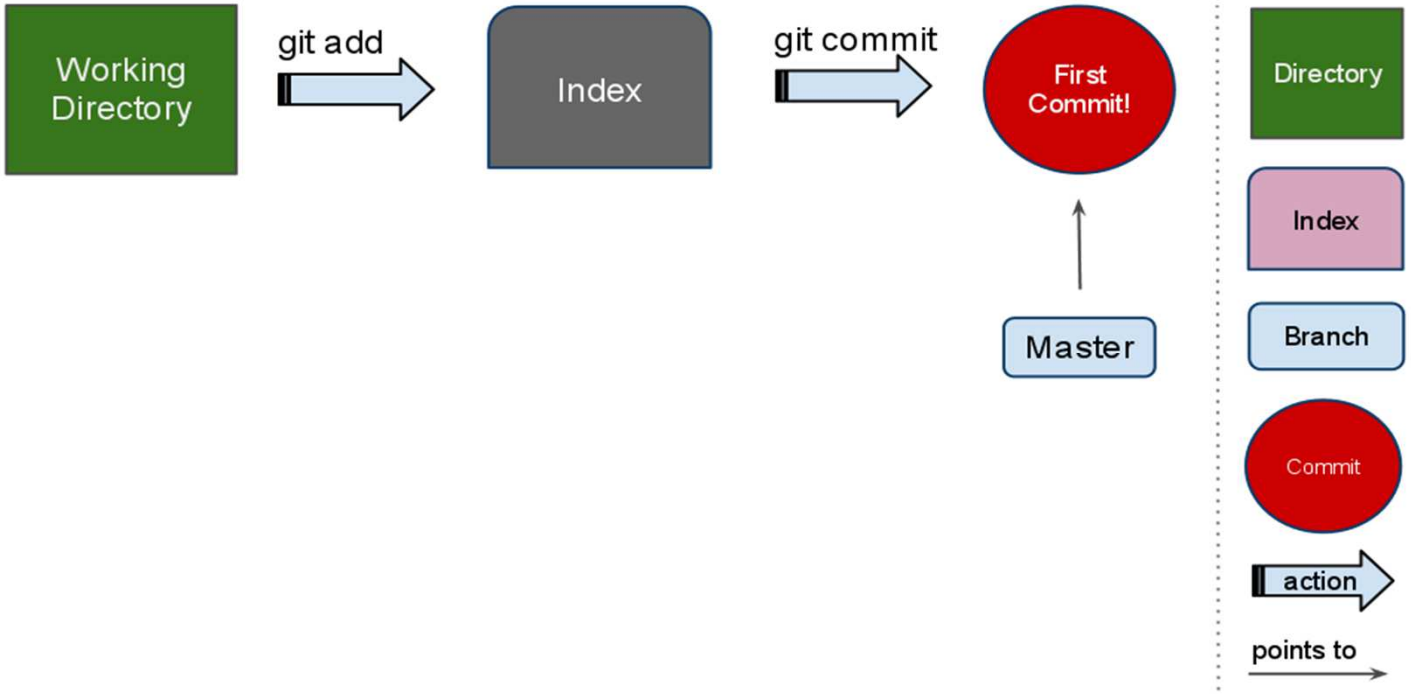
- **Tag :** Est une manière de représenter un commit spécifique un peu spécial. Il est normalement utilisé pour tagger certains commits en tant que version spécifique ou quelque chose comme ça.

$ git show -s --pretty=raw “commit”
$ git ls-tree « tree SHA » ou git show « tree SHA »
$ git show blob



❑ *Index*

- L' index est un fichier binaire temporaire et dynamique
- Il décrit la structure de répertoire de l'ensemble du référentiel.
- Il prend une version de la structure globale du projet à un certain moment dans le temps.
- L' index permet une séparation entre les étapes de développement supplémentaires et le renvoi de ces changements.



➢ *Présenter les éléments d'un repo GIT*

Git crée un répertoire .git qui contient presque tout ce que Git stocke et manipule.

- **HEAD** : Pointe sur la branche qui est en cours dans votre répertoire de travail (*checkout*)
- **config** : Contient les options de configuration spécifiques à votre projet
- **description** : Utilisé uniquement par le programme GitWeb
- **hooks/** : Contient les scripts de procédures automatiques côté client ou serveur.
- **index** : Est l'endroit où Git stocke les informations sur la zone d'attente
- **info/** : Contient un fichier listant les motifs que vous souhaitez ignorer et que vous ne voulez pas mettre dans .gitignore
- **objects/** : Stocke le contenu de votre base de données
- **refs/** : Stocke les pointeurs vers les objets commit de ces données (branches)

Remarque : Si vous voulez sauvegarder ou cloner votre dépôt, copier ce seul répertoire

➢ *Les branches sous GIT*

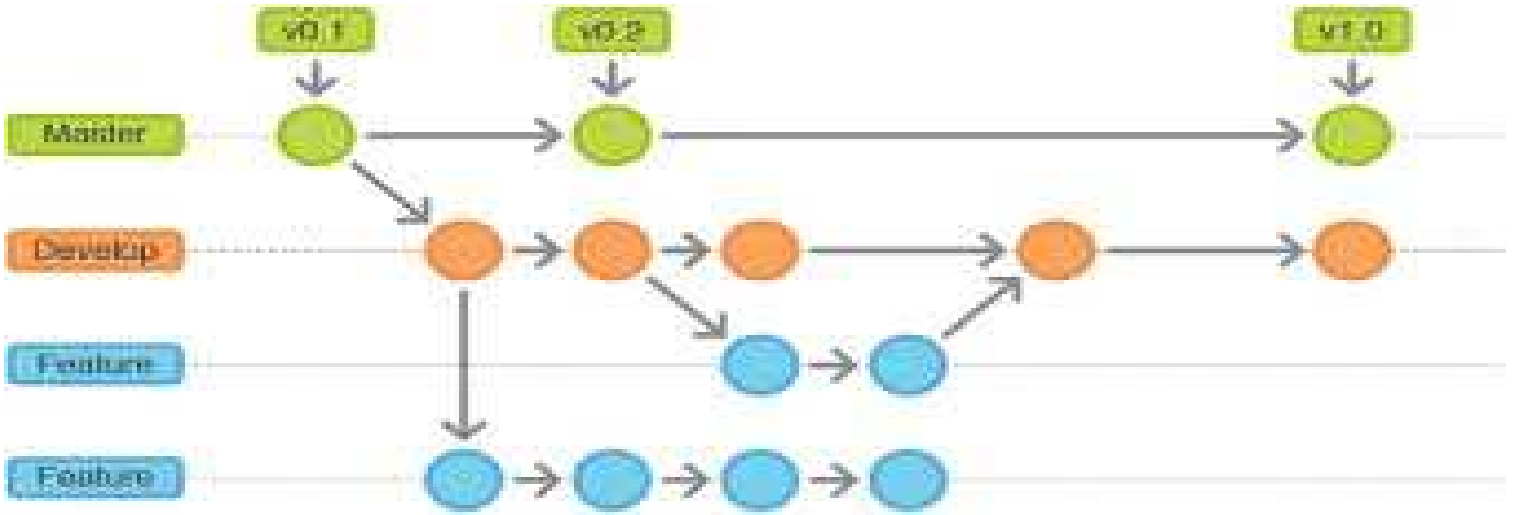
- ***Pourquoi :***

- Dans le développement d'un projet, il arrive souvent qu'on souhaite travailler sur une fonctionnalité en limitant les interactions avec le reste de l'équipe.

- **Quoi :**

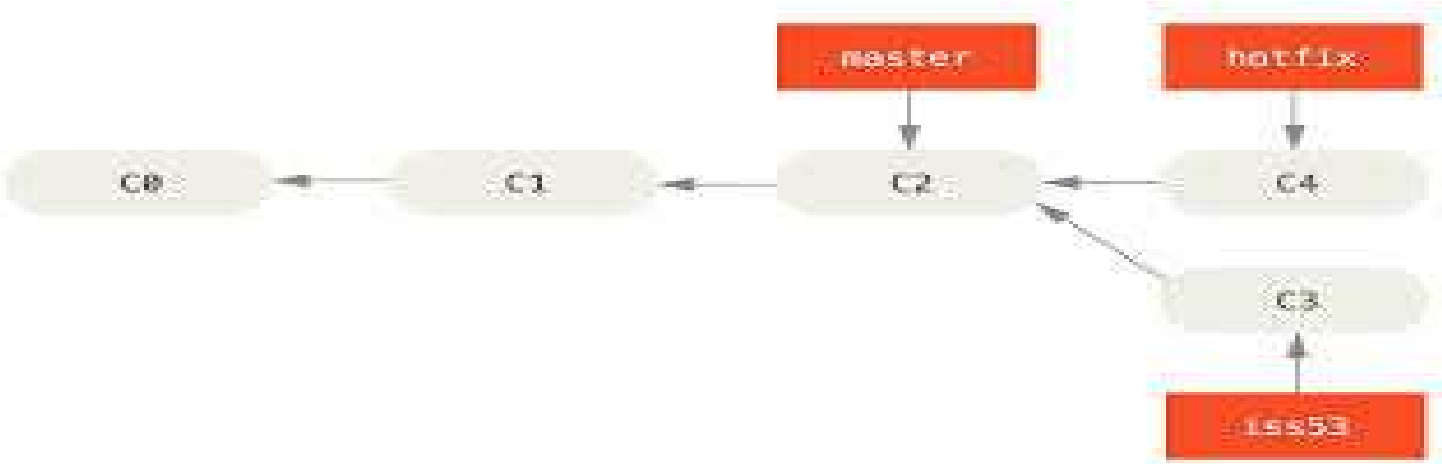
- Faire une branche signifie diverger de la ligne principale de développement et continuer à travailler sans se préoccuper de cette ligne principale
- Dans de nombreux outils de gestion de version, cette fonctionnalité est souvent chère en ressources et nécessite souvent de créer une nouvelle copie du

répertoire de travail, ce qui peut prendre longtemps dans le cas de grands projets.



- Une branche dans Git est tout simplement un pointeur mobile léger vers un de ces objets commit.

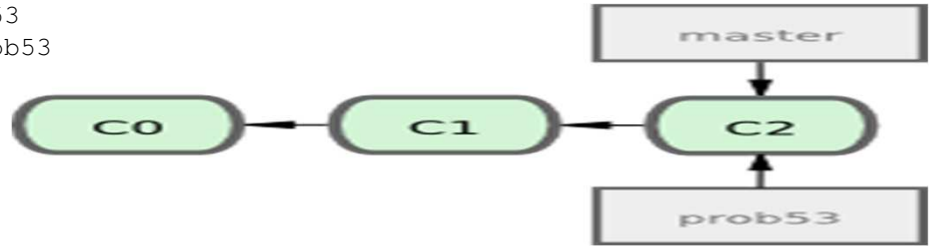
- La branche par défaut dans Git s'appelle master. Au fur et à mesure des validations, la branche master pointe vers le dernier des commits réalisés
- À chaque validation, le pointeur de **la branche ACTIVE** avance automatiquement.



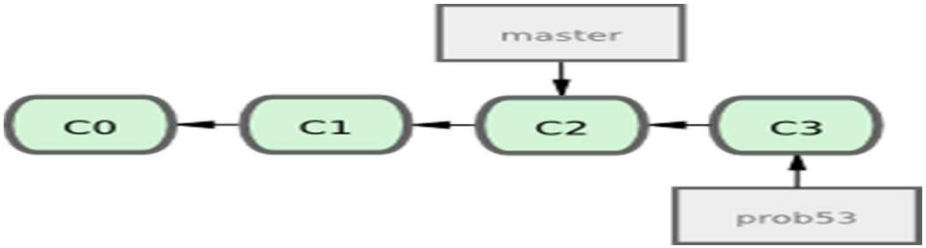
- ***La branche ACTIVE est connue par HEAD (updater par « git checkout branchName »***

- Exemple Utilisation Classique:

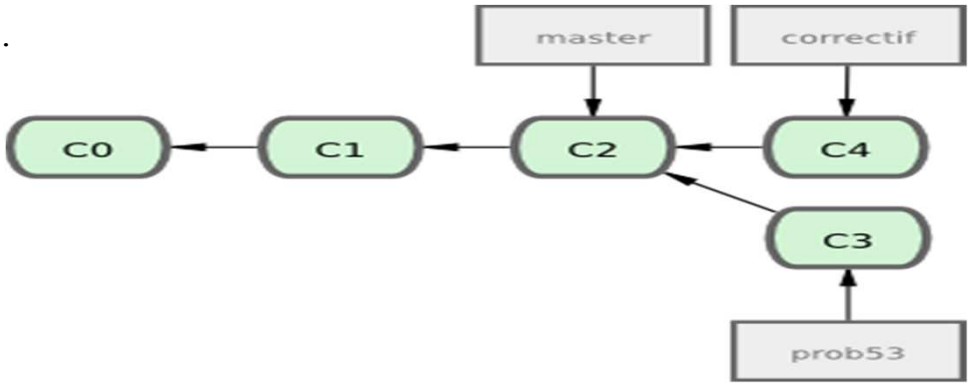

```
$ git branch prob53
$ git checkout prob53
```



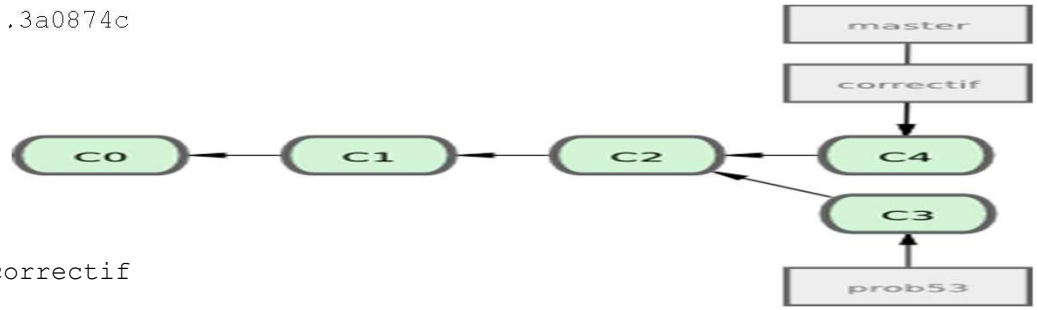

```
$ vi ..
$ git commit -am ..
```



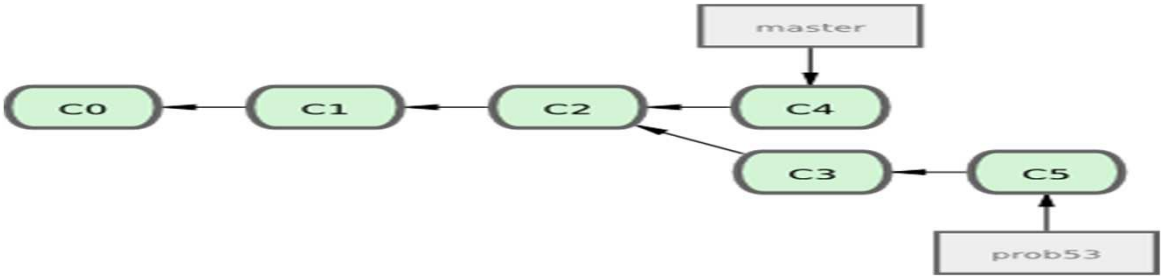

```
$ git checkout master
$ git checkout -b correctif
$ vi ….
$ git commit -am ….
```



```
$ git checkout master
$ git merge correctif
Updating f42c576..3a0874c
Fast forward
```

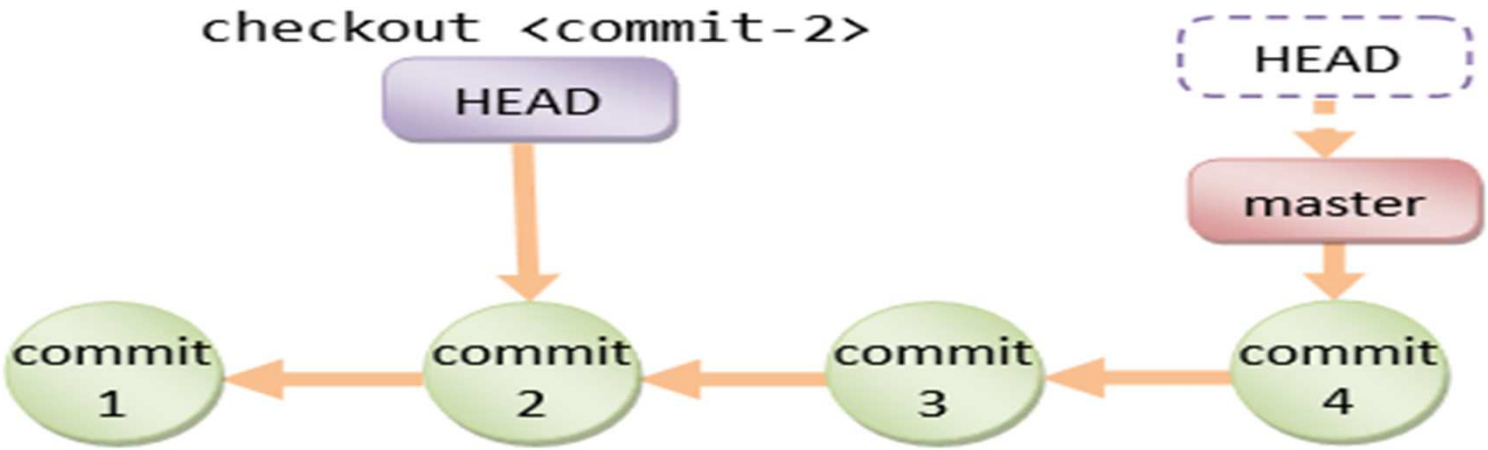



```
$ git branch -d correctif
```



❑Detached HEAD :

- Signifie que vous n'êtes plus sur une branche
- Signifie que HEAD se réfère à un commit mais pas à une branche



Attention : Le developpement doit être toujours fait sous une branche et pas en detached HEAD

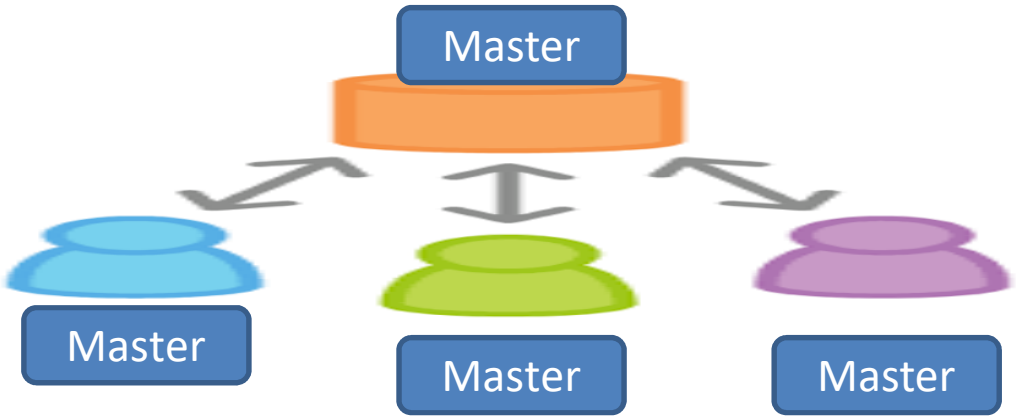
➢ les différents Workflows

❑ Workflow Centralisé

Utilise un repo central pour servir de point d'entrée-de-unique pour toutes les modifications apportées au projet.

La branche de développement par défaut est appelé « master » et tous les changements se sont engagés dans cette branche.

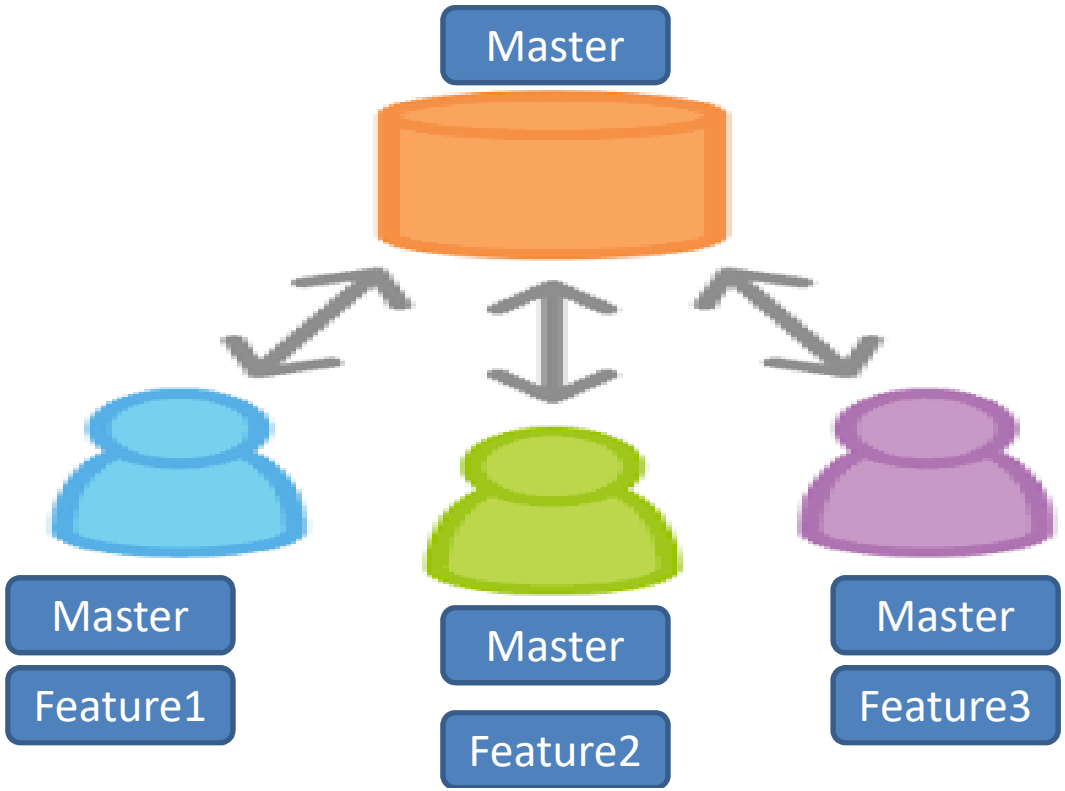
Ce flux de travail ne nécessite pas d'autres branches en dehors la « master ».
Pour publier des modifications au projet officiel, les développeurs «push» de leur « master » branche locale au « master » repo central.



Similaire à SVN sauf que on peut faire des commits locals

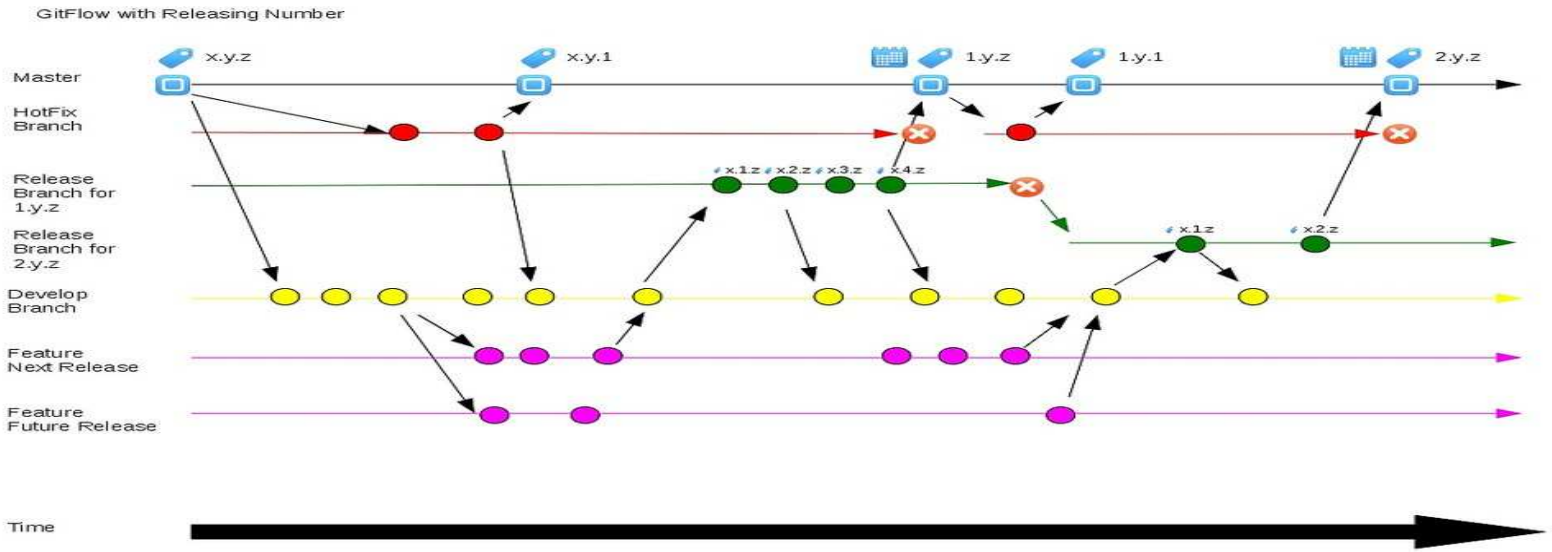
❑ *Feature Branch Workflow*

- L'idée de base derrière ce Workflow est que tout développement de fonctionnalité devrait avoir lieu dans une branche dédiée à la place de la branche master
- cette encapsulation rend facile pour plusieurs développeurs de travailler sur une fonctionnalité particulière sans perturber le code de base principale
- « Pull Request » avant de merger dans la master



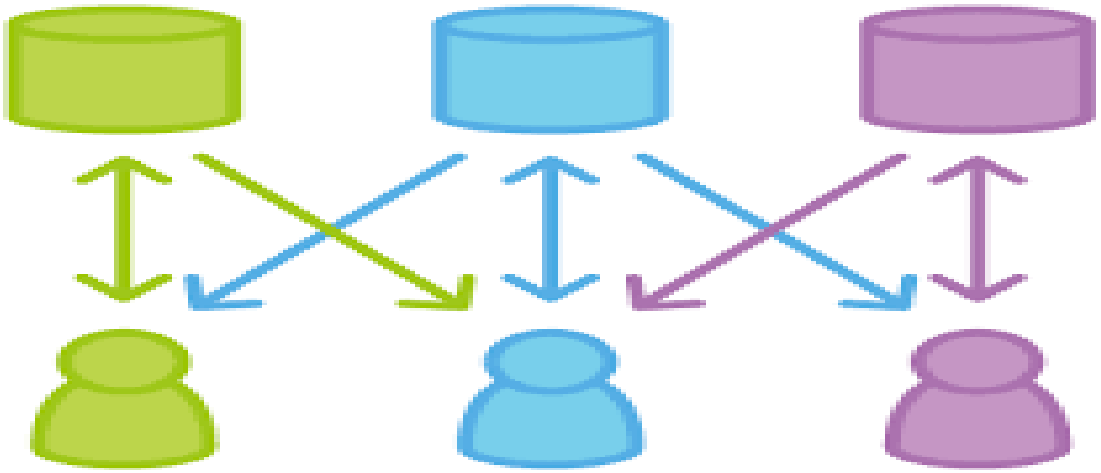
❑Gitflow Workflow

- Le Gitflow Workflow est un modèle commun pour gérer le développement de fonctionnalités, préparation à libération, et l'entretien.
- Le Git workflow définit un modèle de branche stricte conçu autour des livraisons projet
- fournit un framework solide pour la gestion des grands projets.
- Il donne à chaque branche des roles specifiques



❑*Forking Workflow*

- Au lieu d'utiliser un repo unique côté serveur pour agir en tant que code «central» de la base, il donne à chaque développeur un repo côté serveur.
- Cela signifie que chaque contributeur a deux dépôts Git (un local privé et un public côté serveur).
- Développeurs poussent à leurs propres Repos côté serveur, et seul le MAINTAINER/INTEGRATEUR de projet peuvent pousser au repo officiel.
- fournit un moyen souple pour les grandes équipes organiques de collaborer en toute sécurité.
- Un flux de travail idéal pour les projets open source.



MAINTAINER/INTEGRATEUR

➢ *Les commandes relatives aux branches*

- ***Créer***

$ git branch nom_nouvelle_branche

- ***Créer et y basculer***

$ git checkout -b nom_nouvelle_branche

- ***Supprimer***

$ git branch -d branche_à_supprime

La suppression sera refusée si la branche indiquée n'a pas été fusionné avec la branche actuelle.

Pour forcer la suppression remplacer l'option ***-d*** par ***-D***. Tout le travail de la branche supprimée est alors perdu.

- ***Afficher la liste des branches existantes***

$ git branch

L'option --merged permet de voir les branches qui ont déjà fusionné avec celle actuelle,

l'option --no-merged donnant le résultat inverse.

- ***Basculer vers une branche existante.***

$ git checkout nom_branche_visée

- ***Afficher l'identifiant du commit sur lequel pointe une branche.***

$ git rev-parse nom_de_la_branche

- ***Récupérer un commit donné présent sur une autre branche.***

$ git cherry-pick somme_de_contrôle_du_commit

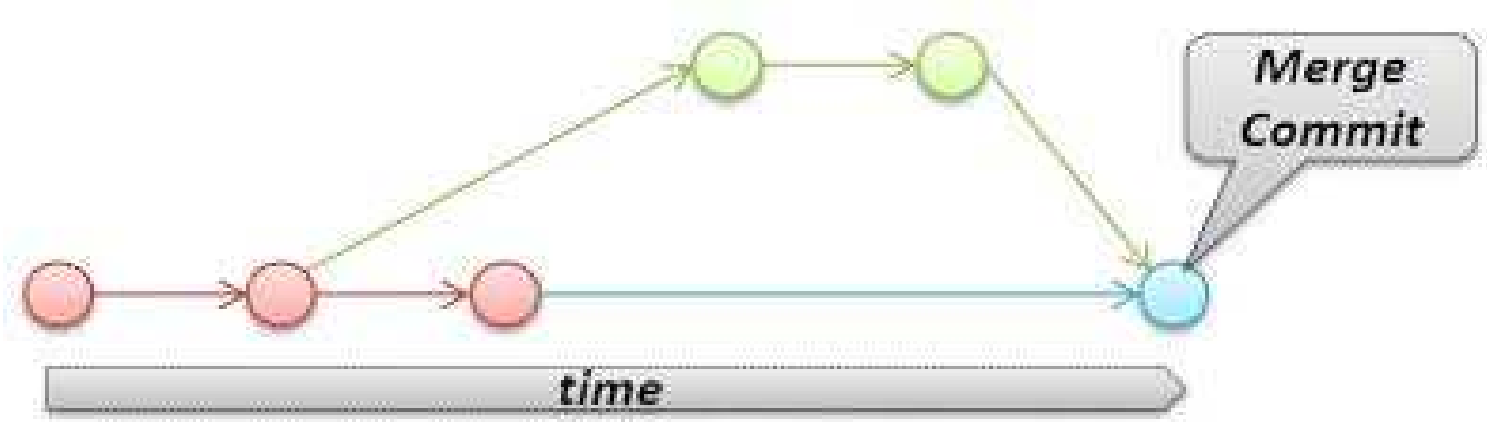
➢ Notion de Merge et Rebase

❑ Merge

Quand ?

- Lorsque vous aurez besoin de faire de nouvelles mises à jour (fetch/pull) du repo central.
- Fusionner d'autres branches dans votre branche.

Quoi ?

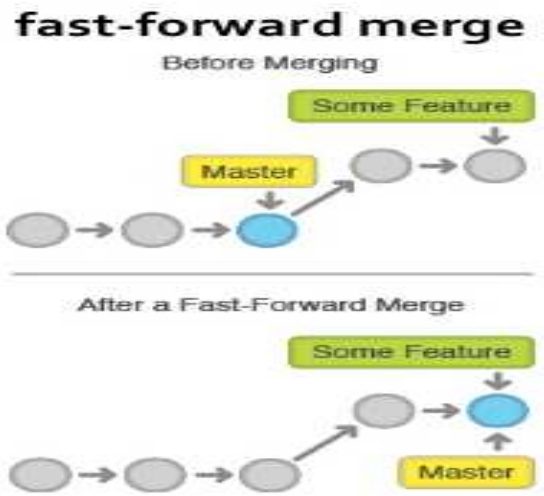


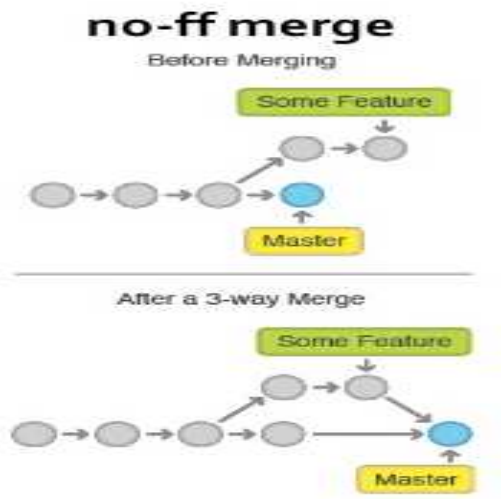
✓ La branche courante sera mis à jour pour refléter la fusion,
✓ La branche cible sera totalement insensible.
✓« git merge » est souvent utilisé en conjonction avec « git checkout » pour sélectionner la branche et « -d » pour la suppression de la branche mergé.

Comment ?

git merge <branch> : Merge la <branch> dans la branche courante

git merge --no-ff <branch> : Toujours générer un « merge commit » même si un fast-forward merge .





❑Rebase

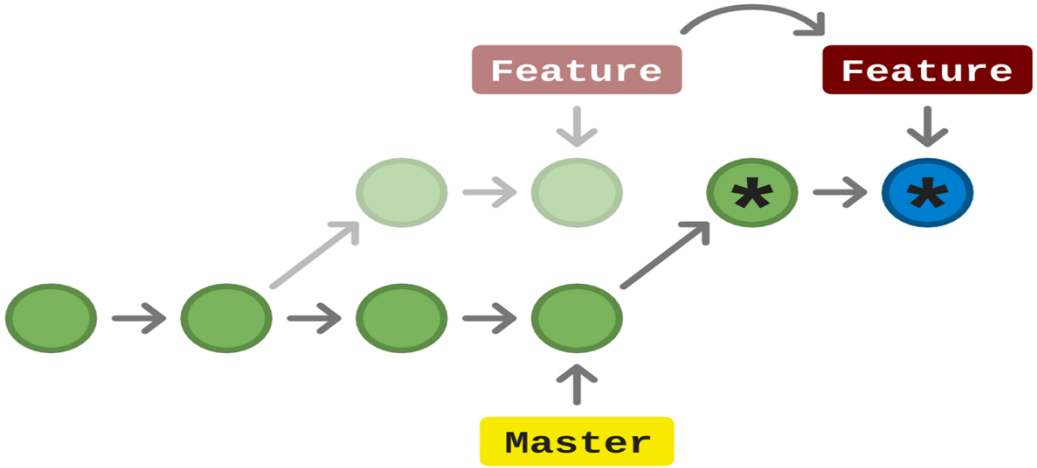
Quand:

Intégrer les modifications d'une branche dans une autre

Obtenir un historique de projet beaucoup plus propre.

Quoi :

Prend toutes les modifications qui ont été validées sur une branche et les rejouez sur u



- Chercher l'ancêtre commun le plus récent des deux branches
- Récupérer toutes les différences introduites par rapport à la branche active
- Sauver dans des fichiers temporaires
- Réappliquer chaque modification dans le même ordre

Comment :

$ git checkout feature

$ git rebase master

Rebase interactif: Plus puissant que d'un rebase automatisé, car il offre un contrôle complet sur les commits de la branche

- Edit previous commit messages
- Combine multiple commits into one
- Delete or revert commits that are no longer necessary

$ git rebase -i master

$ git rebase -i HEAD~n : rebase les n derniers commits de la branche courante

```
pick 1fc6c95 Patch A
pick 6b2481b Patch B
pick dd1475d something I want to split
pick c619268 A fix for Patch B
pick fa39187 something to add to patch A
pick 4ca2acc i cant' typ goods
pick 7b36971 something to move before patch B

# Rebase 41a72e6..7b36971 onto 41a72e6
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

- **pick**, permet de d'inclure le commit. On peut en profiter pour changer l'ordre des différents commit
- **reword**, permet d'inclure le commit tout en ayant la possibiliter de changer le message
- **edit**, permet d'éditer le commit. En séparant en plusieurs commits par exemple
- **squash**, combine le commit avec le commit du dessus et permet de changer le message du commit
- **fixup**, comme **squash** mais utilisera le message du commit situé au dessus
- **exec**, permet de lancer des commandes shell sur le commit

➢ Le Sync avec GIT

❑ Remote

- Pour partager des données avec vos collègues => Un repo distant entre en jeu.
- La commande « ***git remote*** » est vraiment un moyen plus facile de passer des URL aux commandes de partage (PULL/PUSH)
- On peut ajouter , supprimer ,renommer .. Des remotes

Usage

$ git remote add <name> <URL/remote_repo_name.git>

$ git remote –v : lister les remotes

$ git remote rm <name> : supprime la connexion au repo distant

$ git remote rename <old-name> <new-name>

❑ *Fetch*

- Lorsque on veut voir ce que les autres ont commité avant de merger .
- Fetch n'a pas d'effet sur le developpement local
- Importe les commits du repo distant au repo local

Usage

$ git fetch <remote> Fetch all of the branches from the repository

$ git fetch <remote> <branch>

Exp:

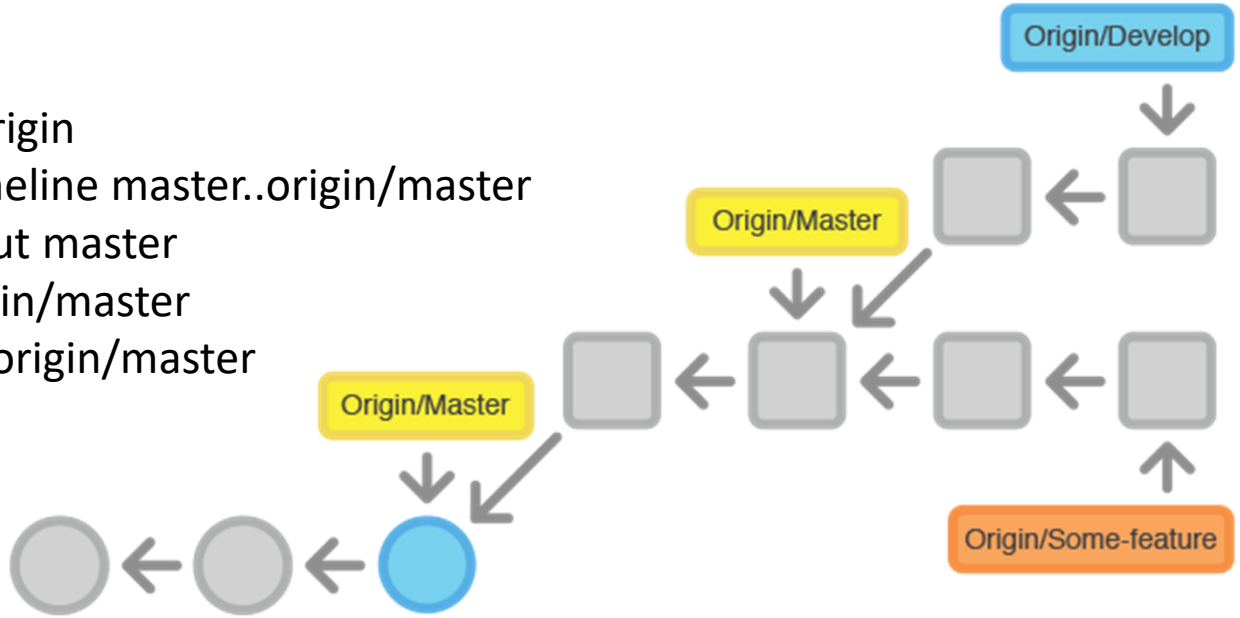
$ git fetch origin

$ git log --oneline master..origin/master

$ git checkout master

$ git log origin/master

$ git merge origin/master



❑ *Pull*

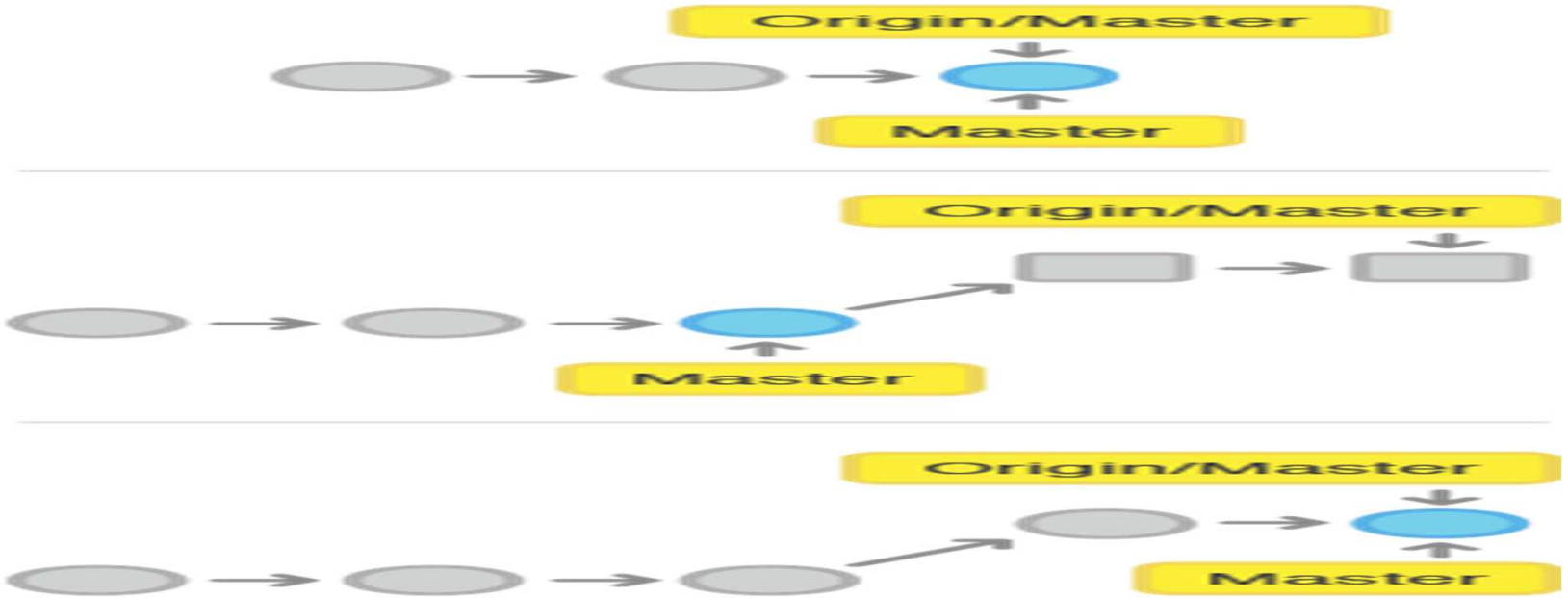
Permet de merger les « commits » des autres contributeurs dans le repo local

Pull = Fetch + Merge

Usage

$ git pull <remote>

$ git pull --rebase <remote> : Cela déplace tout simplement vos modifications locales sur le dessus de ce que tout le monde a déjà contribué.



❑*Push :*

Transfert les commits à partir de votre repo local à un repo distant

C'est l'inverse de « fetch »

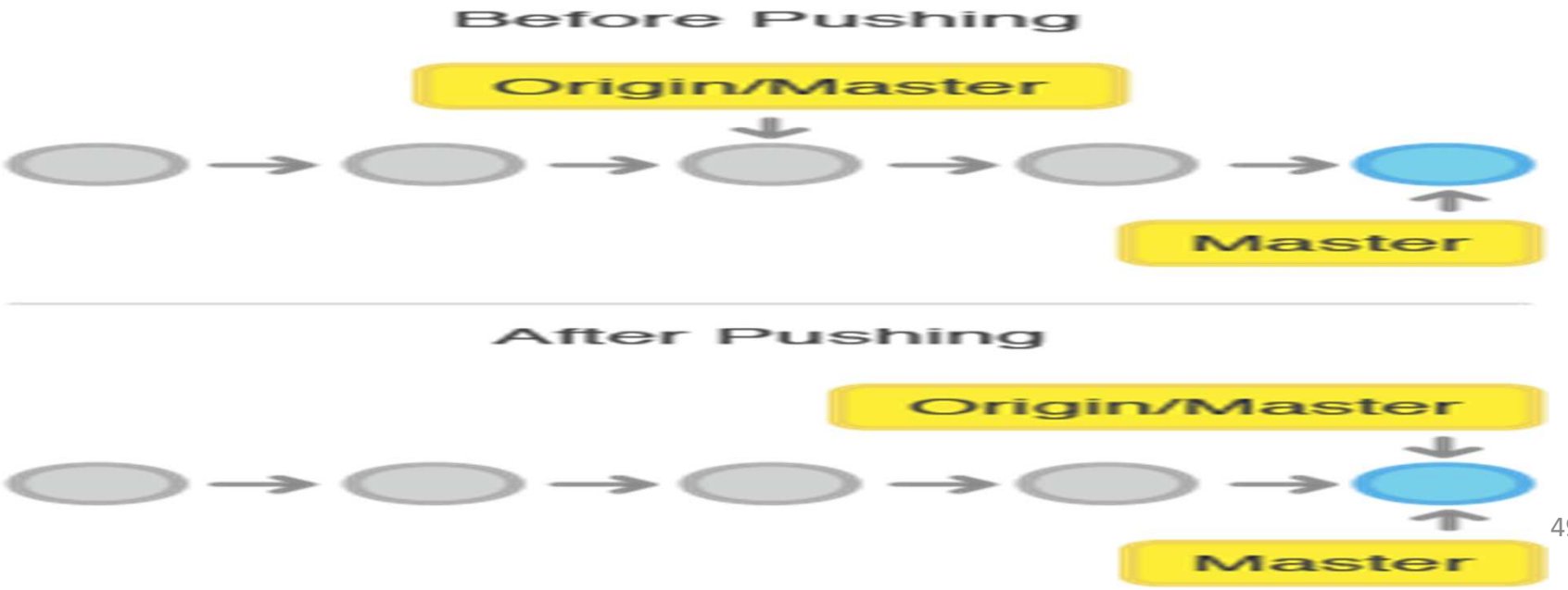
-On fait de push que vers des repos crées avec –bare flag

Usage

$ git push <remote> <branch>

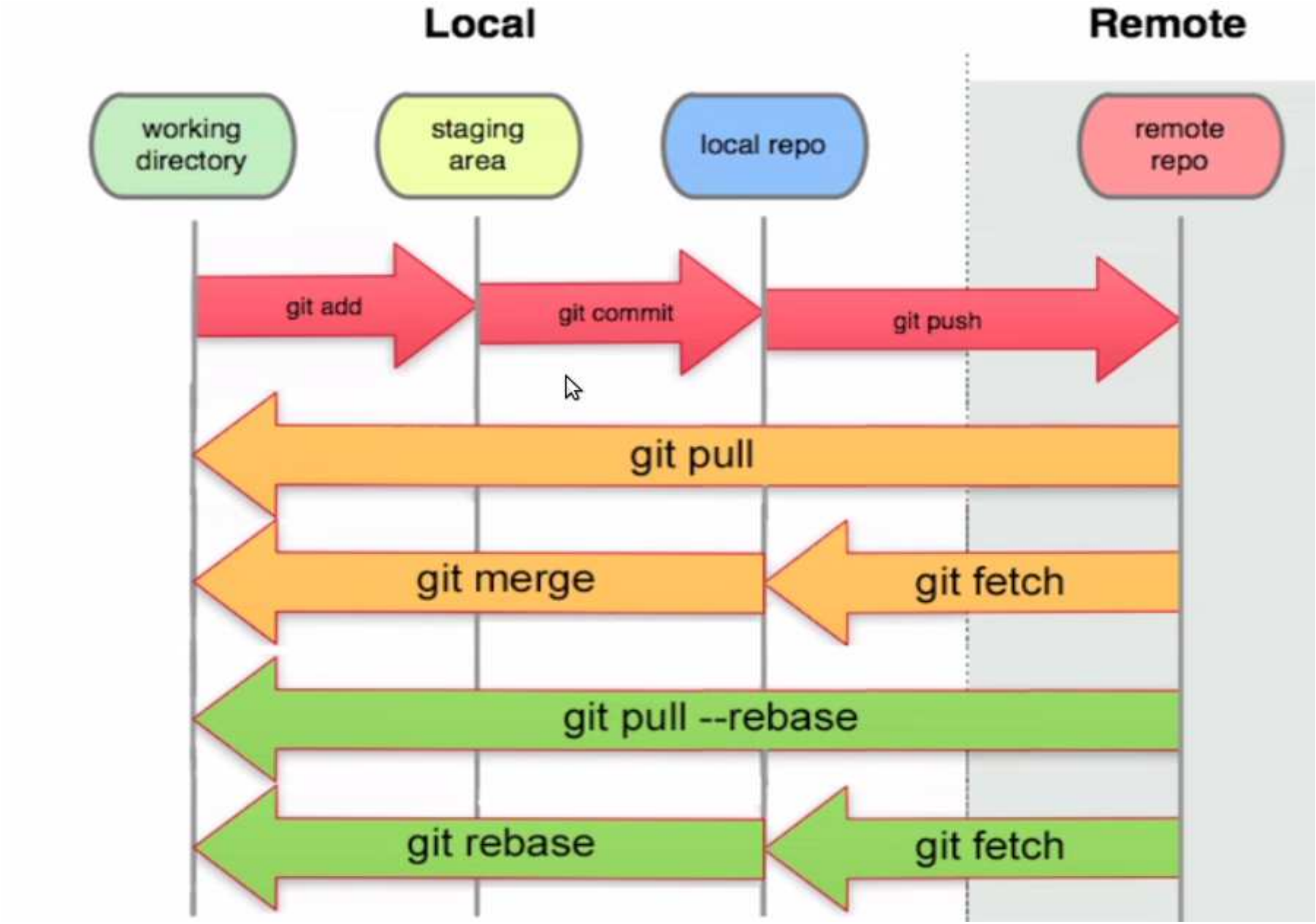
$ git push <remote> --all

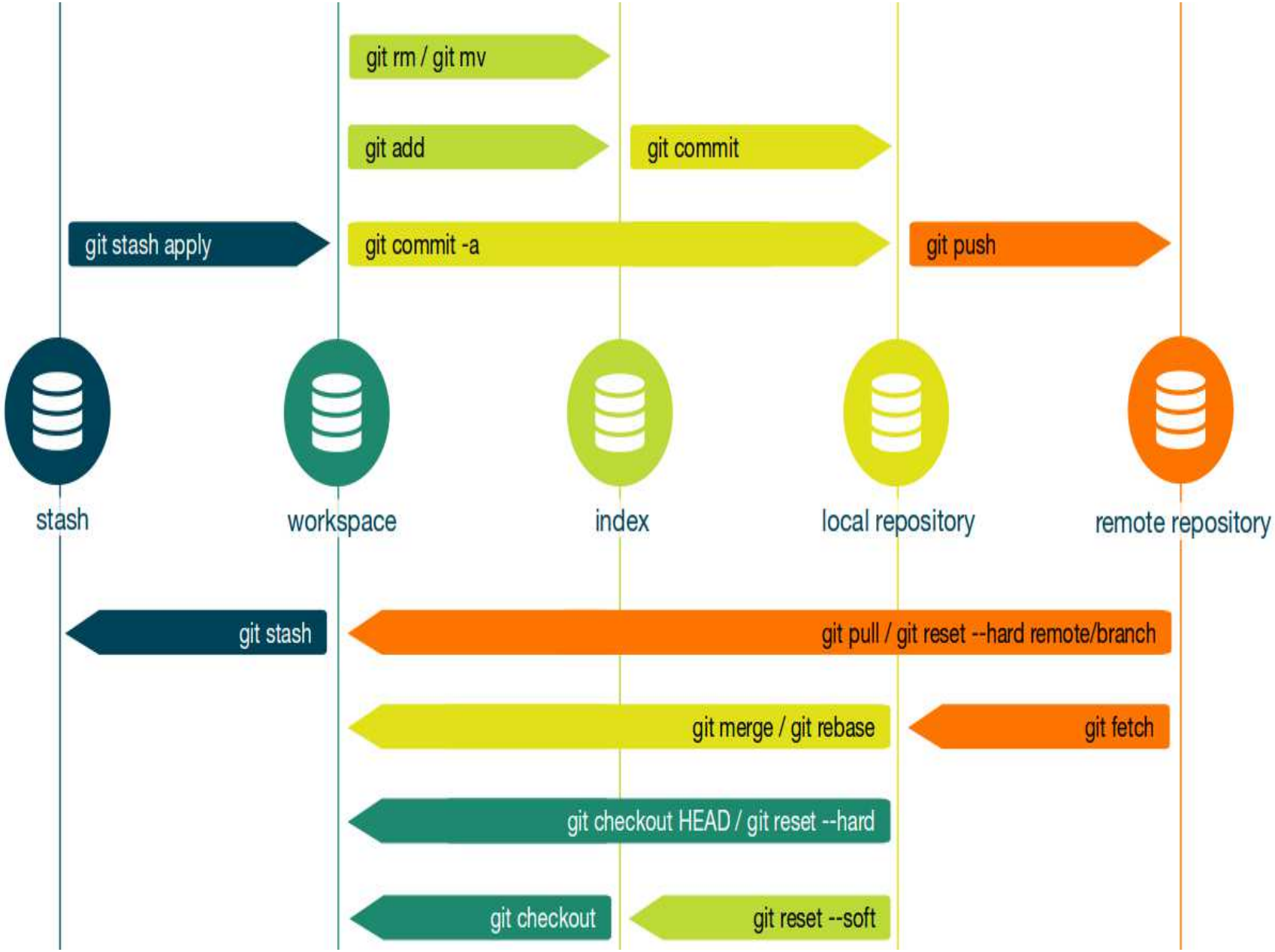
$ git push <remote> --tags (envoie tous vos balises locales pour le dépôt distant)



```
$ git checkout -b feature_branch_name
 $ git push -u origin feature_branch_name

$ git checkout -b branchB
$ git push origin branchB:branchB
```


Local
Remote
working directory
staging area
local repo
remote repo
git add
git commit
git push
git pull
git merge
git fetch
git pull --rebase
git rebase
git fetch


git rm / git mv
git add
git commit
git stash apply
git commit -a
git push
stash
workspace
index
local repository
remote repository
git stash
git pull / git reset --hard remote/branch
git merge / git rebase
git fetch
git checkout HEAD / git reset --hard
git checkout
git reset --soft

➢ *Les Tags*

Quoi :

- C’est une étiquette qui référence un commit particulier
- Permet d'étiqueter un certain état dans l'historique comme
- Permet de créer des versions et référencer des versions stables

Types

Tag annotée

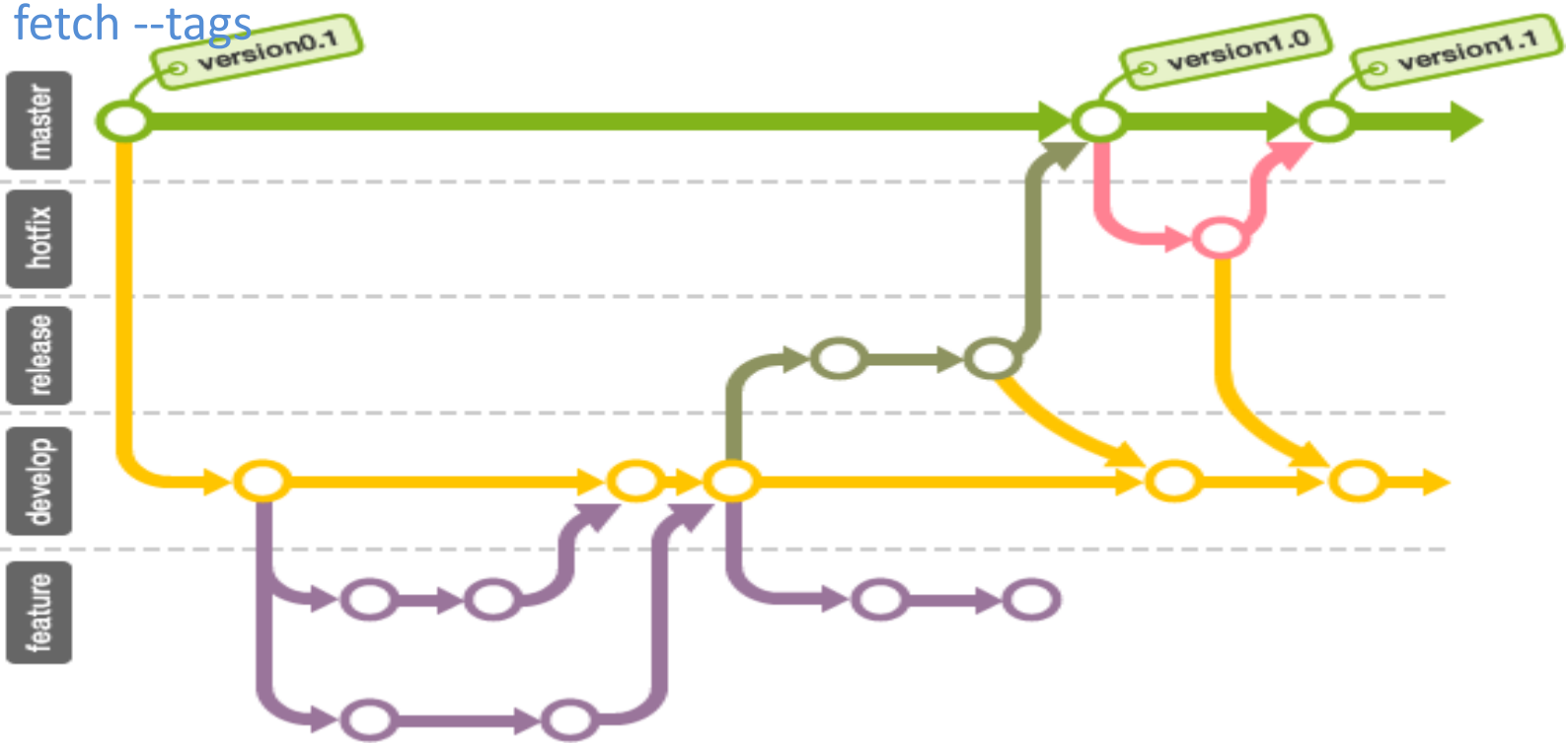
- Sont créés avec un message de sorte qu'il ya une certaine contexte autre que le nom de la balise à aller avec la balise.
- En plus du message, les balises annotées comprennent également le nom de la tagger, et la date de la balise a été créé.

Tag Non annotée

- Balises non-annotés (aussi appelés "tags légers») sont créés sans un message

Commandes

$ git tag v1
$ git tag (liste les tags)
$ git tag -a v1.4 -m 'my version 1.4' (Tag annoté)
$ git show v1.4 (visualiser une tag)
$ git push origin tag v1.5
$ git push --tags
$ git fetch --tags



➢ *Les Refs*

- Sont des pointeurs vers des objets (commit , tags , branches)
- Se sont des fichiers contenant des empreintes SHA-1 dans le répertoire .git/refs/
- **Exp :**
- refs/heads/master ->
- refs/heads/master -> commit fec6ed...
- refs/heads/branche1 -> commit ce5c1e...
- refs/tags/v2.6.8 -> commit e8ce2f...

- Git donne la possibilité de changer ces refs par la commande « **git update-re** »

$ git update-ref refs/heads/master « numéro commit »

- Ceci permet de positionner une branche/tag sur le commit voulu

➢ Gestion de l'historique

❑ Quoi

- Le but de tout système de contrôle de version est d'enregistrer des modifications à votre code.
- L'historique permet de revenir en arrière dans le projet pour voir qui a contribué, et les changements apportés .

❑ Comment :

- Git propose la commande **« git log »**
- Pour voir l'historique d'une branche :

$ git log

$ git log branch

- Trouver tous les commits dont les messages contiennent une chaîne de caractères:

$ git log --grep="le text à chercher «

- Trouver tous les commits dans lequel une chaîne a été retirée ou introduite dans un des fichiers:

$ git log -S"le text a chercher"

- Limiter la recherche dans le temps en utilisant --before et --after.

$ git log --before="2 weeks ago" --after=« YY-MM-DD"

- Chercher entre deux commits:

$ git log 710f0f..8a5cbc

- Voir les commits qui sont sur une branche et pas l'autre:

$ git log master..develop

- Lister les commits d'un utilisateur:

$ git log --author «name »

- Lister les fichiers qui ont été changés dans un commit

$ git log –stat

- Formater le Log

$ git log --pretty=oneline

❑reflog

- La commande **reflog** donne une bonne histoire de ce qui se passe sur la tête de vos branches.
- Permet de trouver/récupérer un état/commit perdu après une mauvaise manip.

Exp :

$ git reflog

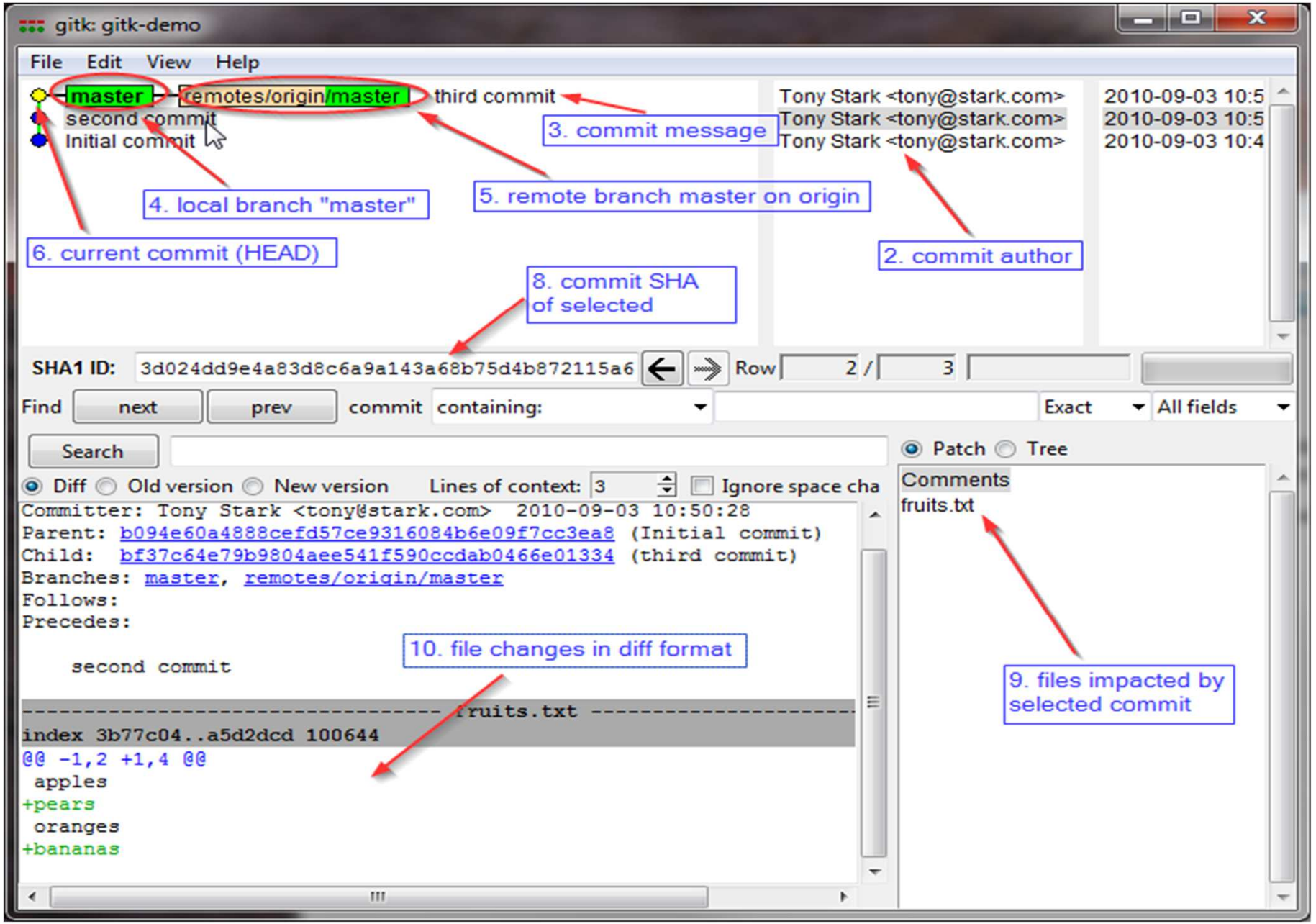
- 17e2785 HEAD@{0}: « commit message 1 »
- 964db49 HEAD@{1}: « commit message 2 »
- bd4fc16 HEAD@{2}: « commit message 3 »
- 7aae59e HEAD@{3}: « commit message 4 »
- 7aae59e HEAD@{n}: « commit message n »

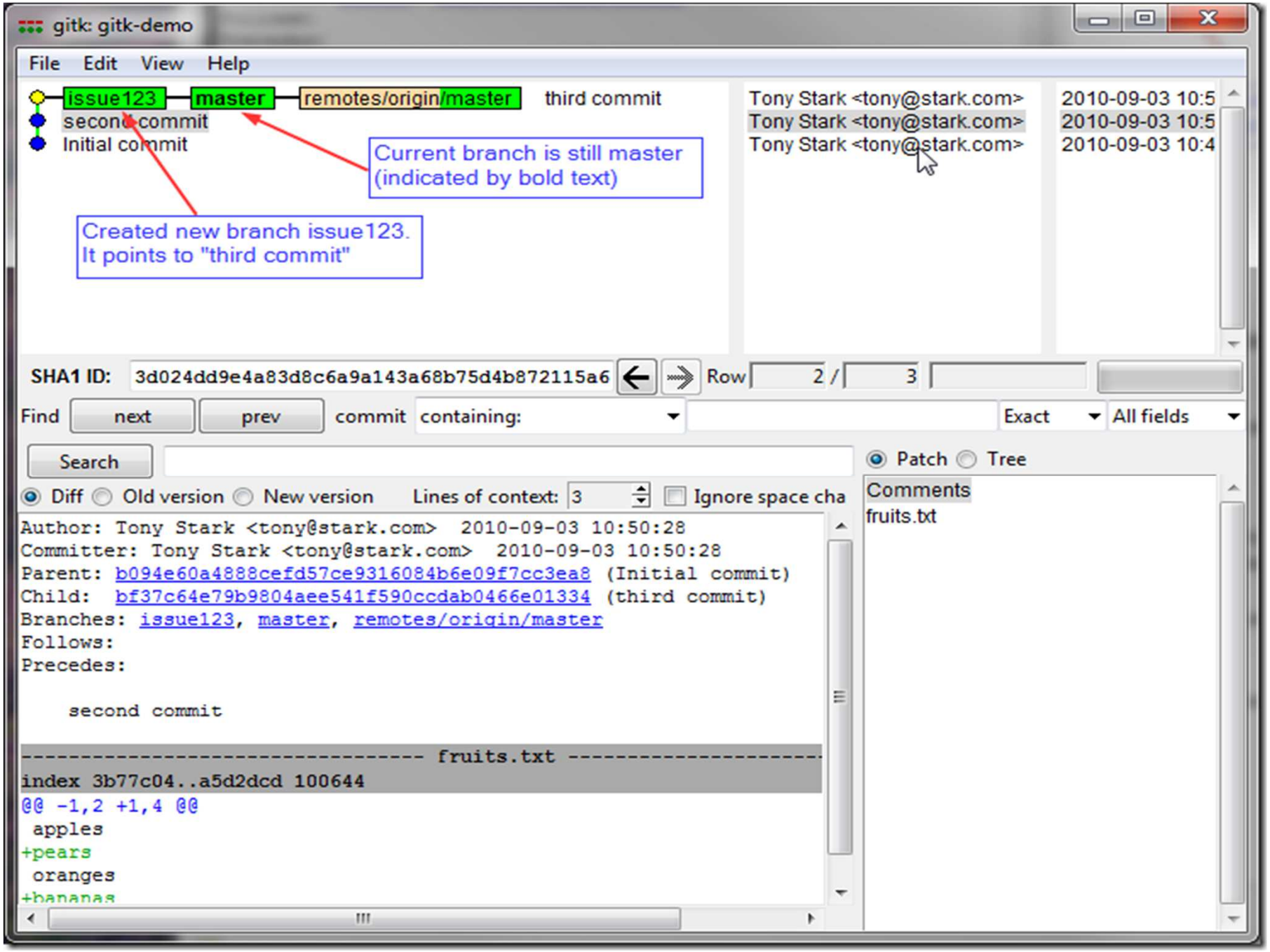
Pour revenir à un état on lance :

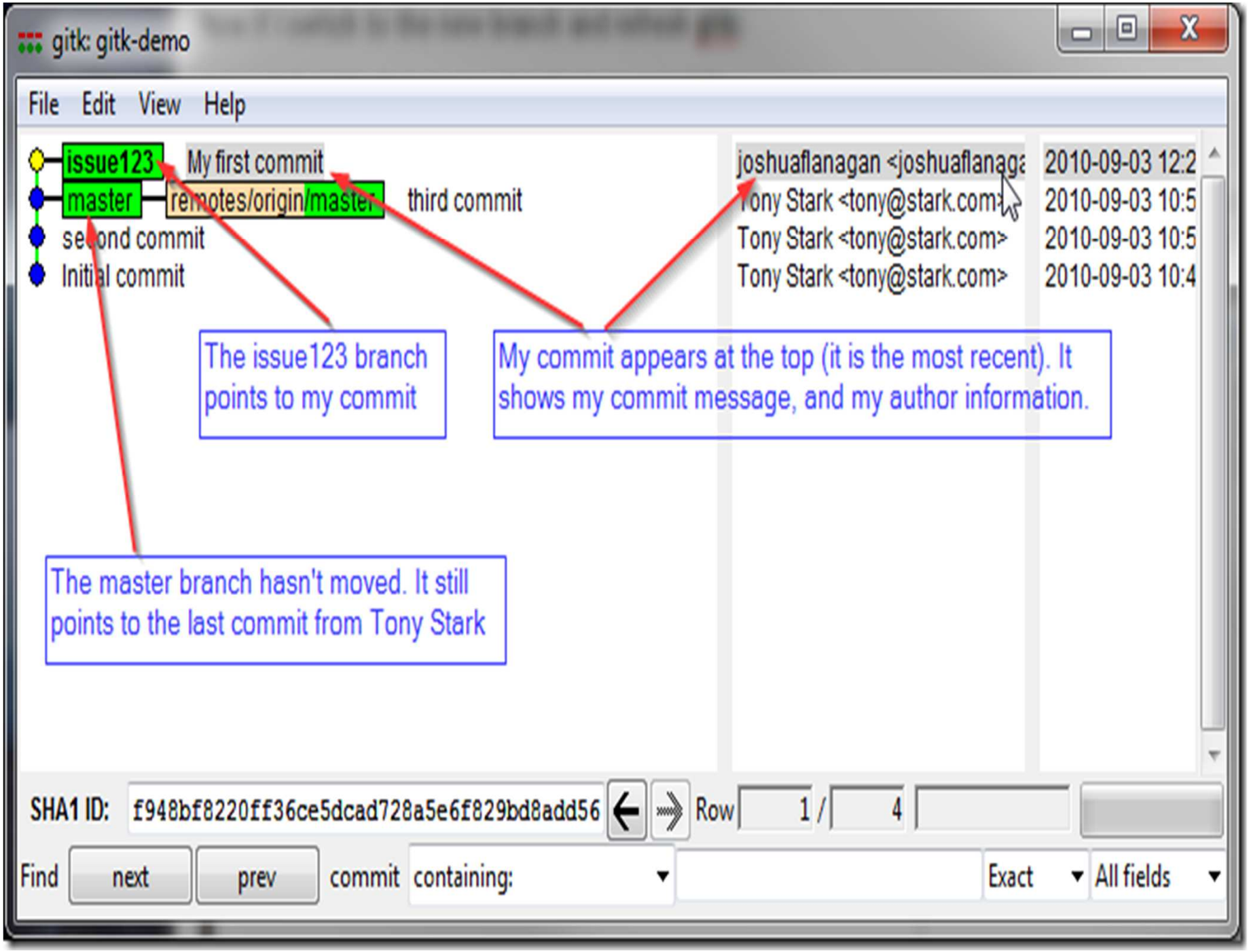
$ git reset –hard « commit » : positionner le HEAD sur le commit choisit.

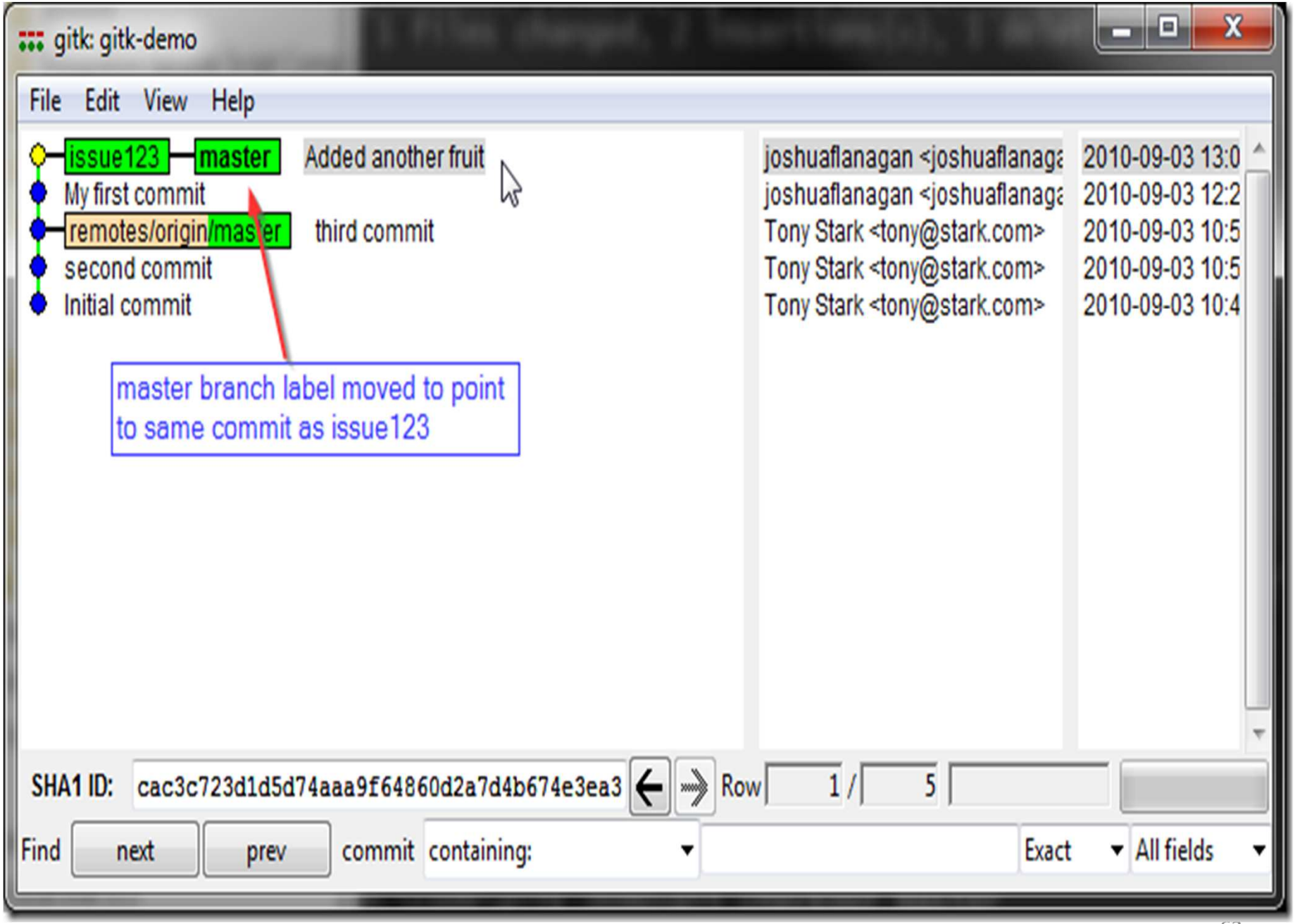
- $ git reflog HEAD@{1.week.ago}
- $ git reflog master@{1.week.ago}
- ***Fromat :***
- 1.minute.ago
- 1.hour.ago
- 1.day.ago
- yesterday
- 1.week.ago
- 1.month.ago
- 1.year.ago

Gitk

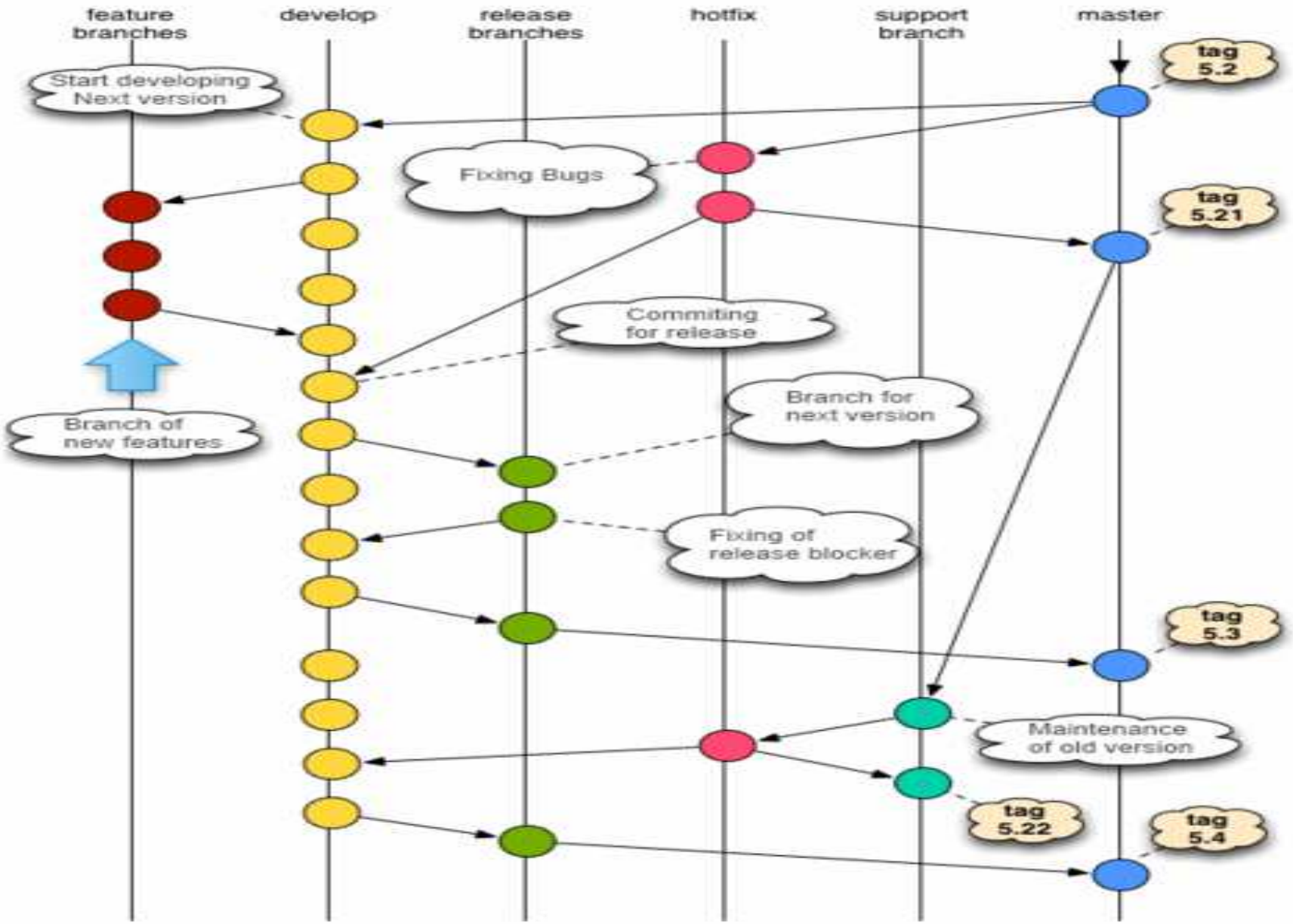

gitk: gitk-demo
File Edit View Help
master
remotes/origin/master
third commit
second commit
Initial commit
Tony Stark <tony@stark.com>
Tony Stark <tony@stark.com>
Tony Stark <tony@stark.com>
2010-09-03 10:5
2010-09-03 10:5
2010-09-03 10:4
3. commit message
4. local branch "master"
5. remote branch master on origin
6. current commit (HEAD)
2. commit author
8. commit SHA of selected
SHA1 ID: 3d024dd9e4a83d8c6a9a143a68b75d4b872115a6
Row 2 / 3
Find next prev commit containing: Exact All fields
Search
Patch Tree
Diff Old version New version Lines of context: 3 Ignore space cha
Committer: Tony Stark <tony@stark.com> 2010-09-03 10:50:28
Parent: b094e60a4888cefd57ce9316084b6e09f7cc3ea8 (Initial commit)
Child: bf37c64e79b9804aee541f590ccdab0466e01334 (third commit)
Branches: master, remotes/origin/master
Follows:
Precedes:
second commit
10. file changes in diff format
fruits.txt
index 3b77c04..a5d2dcd 100644
@@ -1,2 +1,4 @@
apples
+pears
oranges
+bananas
Comments
fruits.txt
9. files impacted by selected commit


gitk: gitk-demo
File Edit View Help
issue123
master
remotes/origin/master
third commit
second commit
Initial commit
Tony Stark <tony@stark.com>
Tony Stark <tony@stark.com>
Tony Stark <tony@stark.com>
2010-09-03 10:5
2010-09-03 10:5
2010-09-03 10:4
Current branch is still master
(indicated by bold text)
Created new branch issue123.
It points to "third commit"
SHA1 ID: 3d024dd9e4a83d8c6a9a143a68b75d4b872115a6
Row 2 / 3
Find next prev commit containing:
Exact
All fields
Search
Patch Tree
Diff Old version New version Lines of context: 3 Ignore space cha
Author: Tony Stark <tony@stark.com> 2010-09-03 10:50:28
Committer: Tony Stark <tony@stark.com> 2010-09-03 10:50:28
Parent: b094e60a4888cefd57ce9316084b6e09f7cc3ea8 (Initial commit)
Child: bf37c64e79b9804aee541f590ccdab0466e01334 (third commit)
Branches: issue123, master, remotes/origin/master
Follows:
Precedes:
second commit
fruits.txt
index 3b77c04..a5d2dcd 100644
@@ -1,2 +1,4 @@
apples
+pears
oranges
+bananas
Comments
fruits.txt


gitk: gitk-demo
File Edit View Help
issue123
My first commit
master
remotes/origin/master
third commit
second commit
Initial commit
joshuaflanagan <joshuaflanaga
2010-09-03 12:2
Tony Stark <tony@stark.com>
2010-09-03 10:5
Tony Stark <tony@stark.com>
2010-09-03 10:5
Tony Stark <tony@stark.com>
2010-09-03 10:4
The issue123 branch points to my commit
My commit appears at the top (it is the most recent). It shows my commit message, and my author information.
The master branch hasn't moved. It still points to the last commit from Tony Stark
SHA1 ID: f948bf8220ff36ce5dcad728a5e6f829bd8add56
Row 1 / 4
Find next prev commit containing: Exact All fields


gitk: gitk-demo
File Edit View Help
issue123
master
Added another fruit
My first commit
remotes/origin/master
third commit
second commit
Initial commit
joshuaflanagan <joshuaflanaga
joshuaflanagan <joshuaflanaga
Tony Stark <tony@stark.com>
Tony Stark <tony@stark.com>
Tony Stark <tony@stark.com>
2010-09-03 13:0
2010-09-03 12:2
2010-09-03 10:5
2010-09-03 10:5
2010-09-03 10:4
master branch label moved to point to same commit as issue123
SHA1 ID: cac3c723d1d5d74aaa9f64860d2a7d4b674e3ea3
Row 1 / 5
Find next prev commit containing: Exact All fields

➢ Model de Branche



❑ Branche Master

- C’est la branche qui est toujours maintenu.
- Le plus récent
- Le code plus stable
- Généralement pas de commit mergé.
- Les versions majeures, la version est mergée dans la Master et Tagée avec le numéro de version.
- **Durée de vie infinie**

❑ Branche Develop

- Le développement majeur de corrections de bugs et de nouvelles fonctionnalités sont effectuées et mergé au premier lieu sur cette branche
- Contient les commits les plus récents.
- **Durée de vie infinie**

❑ Branche Feature

- Ce type de branche est dédié au développement de nouvelle fonctionnalité
- Peut être checkouté du Develop ou Master branche
- **Durée de vie du développement du Feature**

Création et Utilisation

```
$ git checkout -b myfeature develop
$ git checkout develop
$ git merge --no-ff myfeature
$ git branch -d myfeature
$ git push origin develop
```

❑ Branche Release

- Soutenir la préparation d'une nouvelle version de production.
- Sont créés à partir de la branche développer
- Cette nouvelle branche peut exister pendant un certain temps, jusqu'à ce que la production peut être déployé définitivement.
- Pendant ce temps, des corrections de bogues peuvent être appliqués dans cette branche.
- Ajouter de grandes nouvelles fonctionnalités ici est strictement interdite.
- Doit être mergé dans la master lorsqu'elle est stable
- Doit être tagée
- On supprime après le merge dans develop branche
- **Durée de vie déterminé**

Création et Utilisation

```
$ git checkout -b release-x.x develop
$ git checkout master
 Switched to branch 'master'
$ git merge --no-ff release-x.x
$ git tag -a x.x
```

```
$ git checkout develop
$ git merge --no-ff release-x.x
```

```
$ git branch -d release-x.x
```

❑ Branche HotFix

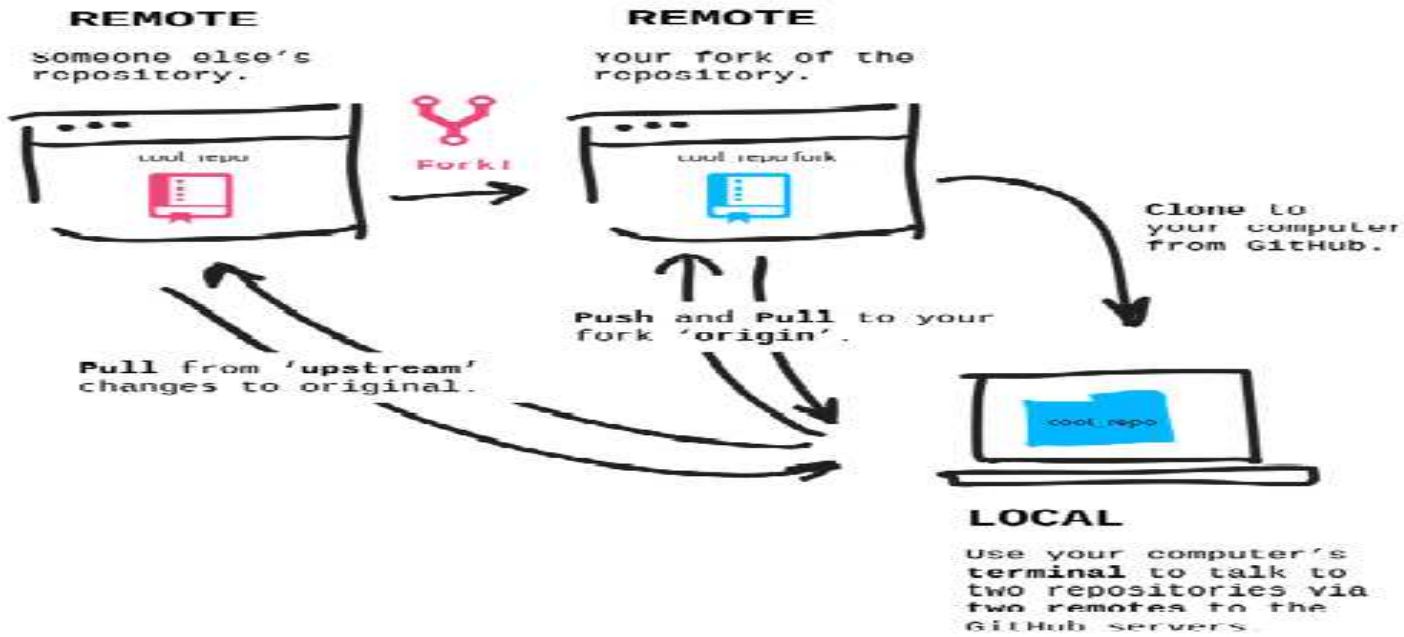
- C’est la branche de corrections de bugs avant les versions de maintenance.
- Cette branche est attachée à la branche master lorsque la version de maintenance est prévue.
- Après la sortie est terminée, cette branche est mergé dans la « Develop »
- Cette branche est supprimée après le merge.
- **Durée de vie déterminé**

Création et Utilisation

```
$ git checkout -b hotfix master
$ git checkout master
$ git merge --no-ff hotfix
$ git tag -a x.x.x
$ git checkout develop
$ git merge --no-ff hotfix
$ git branch -d hotfix
```

➢ *Fork*

- C’est une opération qui est utilisé par un certain flux de git
- Rendu populaire par GitHub, appelé « Fork and Pull Workflow »
- Permet à quiconque de Forker un repo existant et pousser des changements à leur repo personnelle sans nécessiter l'accès soit accordé au repo source.
- Populaire auprès des projets open source, car il permet aux gens de travailler de façon autonome, sans coordination en amont.



➢ Git & Bonnes pratiques

❑ Commiter les modifications en relation

- La fixation de deux différents bugs devrait produire deux commits séparés.
- Petits commits rendre plus facile pour les autres membres de l'équipe à comprendre ces changements.

❑ Commiter souvent

- Pensez à souvent commiter afin de créer des points de repères dans votre arbre local/distant.
- Cela n'affecte personne et vous permettra de facilement revenir en arrière en cas de besoin.

❑ Message du commit

- Essayer d'être le plus précis possible dans vos message de commit cela vous sera utile ainsi qu'a tout les autres collaborateurs ..
- Le mieux serait d'avoir un template de commit

❑ Tester avant de commiter

- Testez-le soigneusement pour vous assurer que le changement est terminé .
- Vérifier si pas d'effets secondaires ou regressions.

❑ Ne Jamais forcer un push

- Si vous vous trouvez dans une situation où vos changements ne peuvent pas être poussés en amont, quelque chose cloche.
- Ne forcer pas le push sans comprendre le problème.

❑ Attention au rebase

- Ne jamais rebaser une branche que vous avez poussé, ou que vous avez tiré d'une autre personne .
- Rebaser des branches publiées peut conduire à changer l'historique dans le référentiel partagé.

❑ Développement sur Master

- Ne pas effectuer le développement dans la branche master
- Mettre à jour périodiquement avec un « pull » puis pousser votre branche locale

❑ Push en Fast-Forword

- **git pull --ff-only** : peut être utilisé pour assurer qu'un pull est seulement du fast-forward.
- Si un pull fast-forward n'est pas possible, ce flag force git à quitter avec une erreur, et de laisser la branche locale intacte.

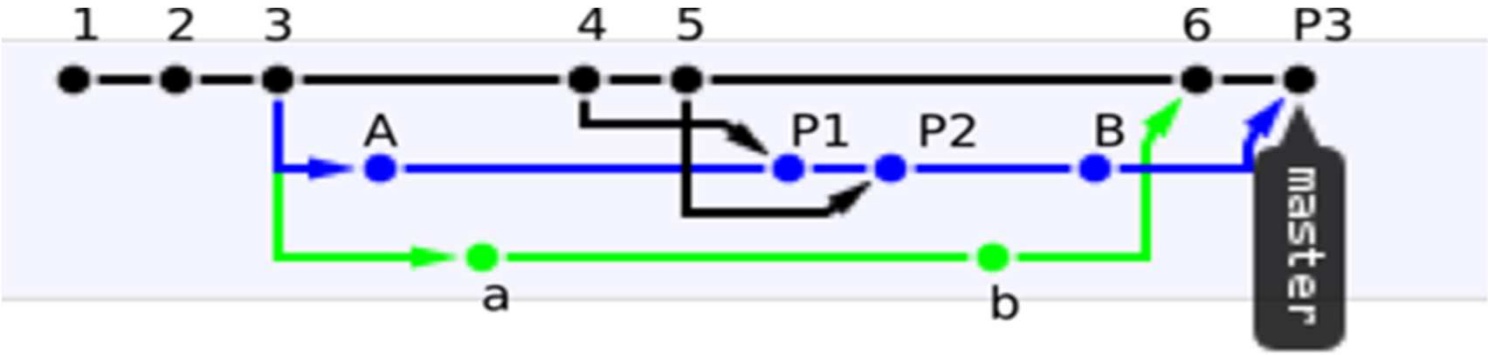
➢ *Le flux de production*

❑ *Etapes:*

1. Créer une branche du master
2. Faire votre travail
3. Tester et valider vos modifications
4. Pousser votre branche jusqu'à le dépôt distant (d'origine)
5. Checkout Master
6. Assurez-vous qu'il est mis à jour
7. Merger votre branche dans master
8. Tester à nouveau
9. Pousser votre copie locale de master jusqu'à le master de dépôt distant (origin /master)
10. Supprimer votre branche locale (et à distance, aussi, si vous l'avez publié)

❑*Méthode 1 : Branche local & Pull/Merge*

- (master) git pull
- (master) git branch MyBranch origin/master –track
- «**–track** : git pull va fusionner de manière appropriée de la branche de suivi à distance »
- (MyBranch) git commit -m "A"
- (MyBranch) git pull (results in a silent, automatic merge commit P1 since this merge is not fast-forward)
- (MyBranch) git pull (results in a silent, automatic merge commit P2 since this merge is not fast-forward)
- (MyBranch) git commit -m "B"
- (master) git pull (Is fast-forward. No merge commit created)
- (master) git merge MyBranch (results in an explicit merge commit P3}
- (master) git push



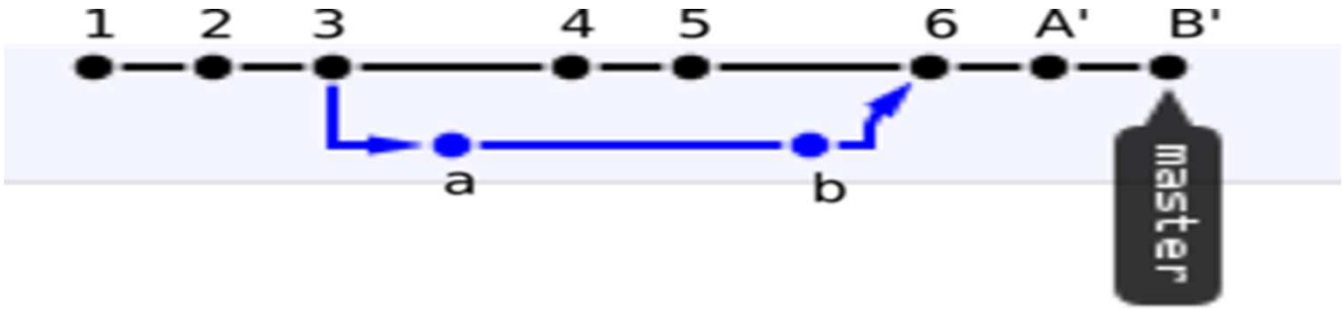
❑ *Méthode 2 : Branche local & rebase*

- Le développement dans une branche locale **non-publiée**, avec l'utilisation de « **git rebase** » à la place de pull ou merge pour mettre à jour la branche locale avec des changements de master est aussi valable.

- Cette technique se traduit par l'élimination de l'histoire de branche locale

- **Attention :** Ne pas rebaser une branche publiée -> Gi t refuse de pousser « push » une branche dont l'historique a changé avec rebase.

- (master) git pull
- (master) git branch Mybranch origin/master --track
- (Mybranch) git commit -m "A"
- (Mybranch) git fetch; git rebase origin/master
- (Mybranch) git commit -m "B"
- (Mybranch) git fetch; git rebase origin/master
- (master) git pull (fast-forward)
- (master) git merge Mybranch (fast-forward)
- (master) git push (master distante = local master)



➢ *Process d'Intégration*

❑ Les deadlines

- On définie les date des livraison officielle avec le client
- On définie des livraison intermédiaires et des releases candidates en interne en tenant compte des jalons officiels
- Un planning doit être maintenu pour chaque version produit

❑ Les contenu

- Pour chaque livraison on doit définir le contenu attendu
- L'intégrateur doit respecter le contenu demandé et n'accepte que les « pull request » qui implémentent les fonctions/Bugs décrites dans le contenu.

❑ Gestion de releases

- Chaque release doit être tagée avec un message
- Le contenu de release peut être ajouter au tag
- Le nom du Tag ou version doit respecter une règle de nommage définie au préalable

❑ Gestion des branches

- Pour chaque version produit on peut maintenir un branche spécifique.
- Une branche spécifique peut être maintenue pour chaque client .

❑ Tests d'intégration

- Avant de tagér et livrer une version , un ensemble de tests doit être éfectuer
- **Smoke tests :** tests non avancés pour vérifier les services de base d'une fonctionalité sont présents
- **Test de non-régression** : Assurer qu'on fait pas des régressions majeurs qui stoppent la laivraison.
- **Test du contenu** : Assurer que le contenu demandé dans une livraison est bien présent

❑ Reporting

- Des rapports de qualité de code peuvent aussi être maintenu pour chaque release