

Formation



git

Première partie

Principes de base

Avant propos

Une bonne partie des planches de cette formation sont issues (ou fortement inspirées) de la formation GIT proposée par la société Delicious Insight de Christophe Porteneuve (décembre 2016).

Merci donc de ne pas diffuser ces planches au delà d'un usage interne au service IL.

Horaires

- 9h30 – 17h30
- Pause déjeuner : 12h45 – 13h45
- Pause matin 11h
- Pause Après midi 15h30



Plan : première partie

- La gestion de versions : pourquoi, comment ?
- Concepts fondamentaux
 - Working Directory, Stage, Repository
- Travailler avec son dépôt localement
 - Créer, cloner, commandes git principales
- Soucis classiques et solutions optimales
 - J'ai oublié un truc dans mon commit !
 - J'ai stagé un truc que j'aurais pas dû !
 - Et voilà! j'ai paumé mon commit !
 - J'ai fait un reset foireux !
 - J'ai commisé un truc en trop !
 - J'ai commisé un truc en trop.. Y'a un bail
 - J'ai supprimé un fichier à tord il y a un bail...
 - J'ai oublié de configurer mon nom...
- Branches, stash et checkout

Plan : seconde partie

- Merges et rebases
 - Commit de merge/fusion, Fast forward, Tête détachée, pourquoi et comment soigner son historique, etc.
- Cherry-picking
- Dépôts distants
 - Push, pull, fetch, comment annuler un push
- Soucis classiques et solutions optimales sur dépôt distant
 - Mon git pull me dit « You have unstaged changes » !
 - Mon git push me dit « [rejected]...
 - Je voudrai créer une copie de mon repos sur un média USB local
- (Rerere : optimiser les merges)** Si on a le temps
- (Chasse aux bugs accélérées :Le bisecting)*
- Les Workflows
 - Centralized workflow, Feature Branch workflow, Git Flow workflow, Forking workflow

Index des commandes git abordées

- [git add](#)
- [git bisect](#)
- [git cherry-pick](#)
- [git clean](#)
- [git clone](#)
- [git commit](#)
- [git diff](#)
- [git difftool](#)
- [git fetch](#)
- [git help](#)
- [git init](#)
- [git log \(alias git lg ou git l\)](#)
- [gitk](#)
- [git stash](#)
- [git branch](#)
- [git checkout](#)
- [git merge](#)
- [git pull](#)
- [git push](#)
- [git rebase](#)
- [git remote](#)
- [git show](#)
- [git status \(alias git st\)](#)
- [git tag](#)
- [git reset](#)
- [git reflog](#)
- [git rerere](#)
- [git rev-list](#)

Index de quelques termes utilisés

- [Bitbucket](#)
- [Beyond Compare](#)
- [Branche](#)
- [Cache](#)
- [Commit de merge/fusion](#)
- [Dépôt distant](#)
- [Fast forward](#)
- [GitBash](#)
- [.gitconfig](#)
- [HEAD](#)
- [Index](#)
- [Master](#)
- [Protocoles](#)
- [Repository](#)
- [Stage](#)
- [Tête détachée](#)
- [Working Directory](#)
- Workflow
 - [Centralized workflow](#)
 - [Feature Branch workflow](#)
 - [Git Flow workflow](#)
 - [Forking workflow](#)
- [.gitconfig](#)

Quelques définitions

Commit / Révision / Version	Série de modifications faisant l'objet d'une soumission unique, idéalement thématique et atomique. Reçoit un identifiant unique.
HEAD	Dernier commit en date dans l'historique de référence actuel
Tag / Label	Étiquette sémantique apposée sur un commit (ex. « v1.0 »). Un tag n'est pas censé être repositionné après coup.
Dépôt / Repository	Ensemble de révisions pour un même projet, local ou distant (sur un serveur), avec les méta-données associées.
Branche	Historique thématique (série de révisions). Un même dépôt peut avoir plein de branches, comme un arbre.
Master	La branche de référence d'un dépôt
Fusion / Merge	Réconciliation des travaux de plusieurs branches pour un résultat consolidé.
Conflit	Situation empêchant une fusion automatique ; se résout manuellement
Working Directory	L'état actuel sur le disque des dossiers et fichiers

Cf. [git help glossary](https://git-scm.com/docs/gitglossary.html) : <https://git-scm.com/docs/gitglossary.html>



**La gestion de
versions : pourquoi,
comment ?**

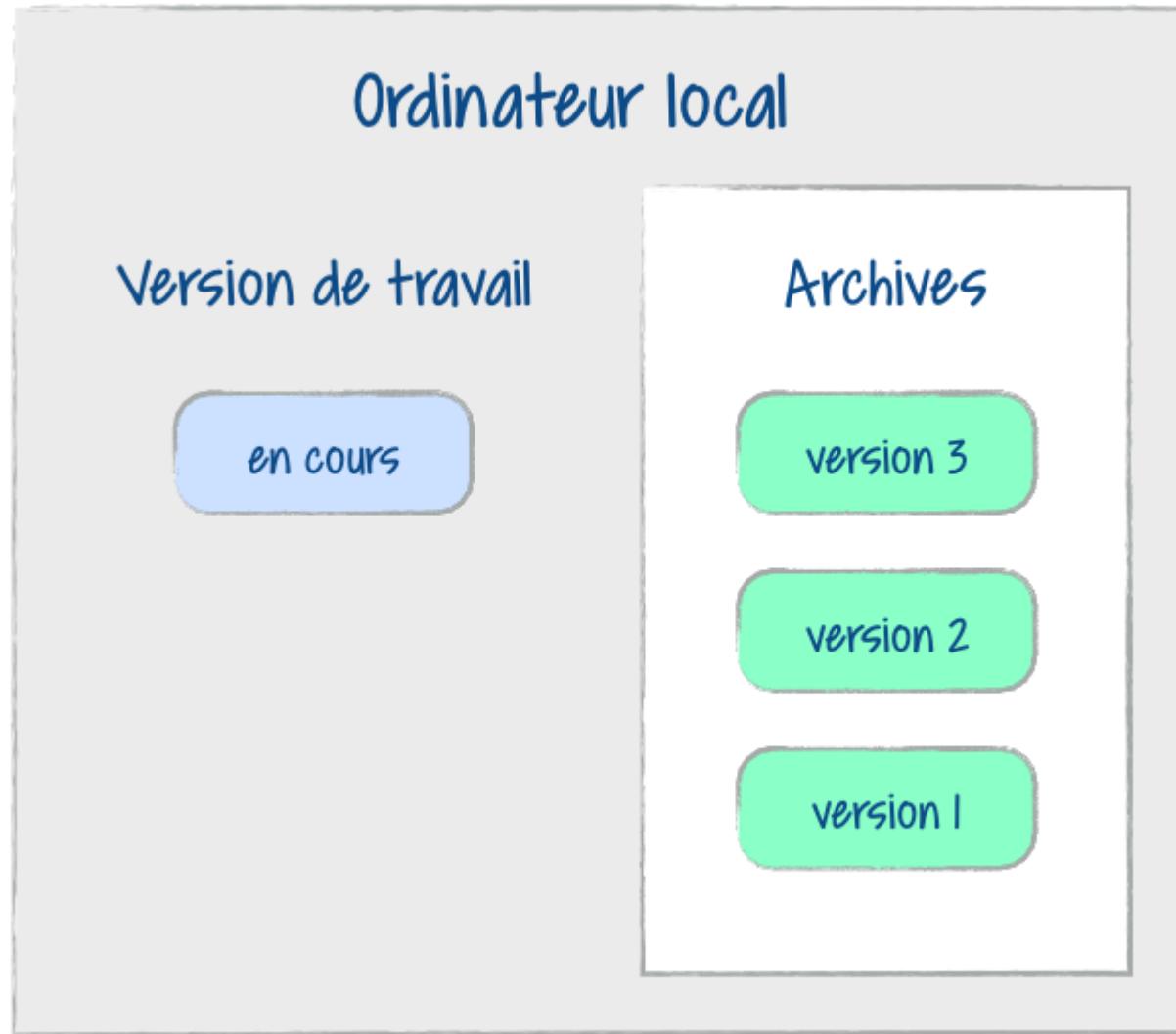
La gestion de versions : pourquoi, comment ?

- **Parce qu'on veut toujours pouvoir revenir en arrière...**
 - Parce qu'on s'est trompé
 - Parce qu'on veut comparer
 - Parce qu'on veut sauvegarder
 - ...

La gestion de versions : pourquoi, comment ?

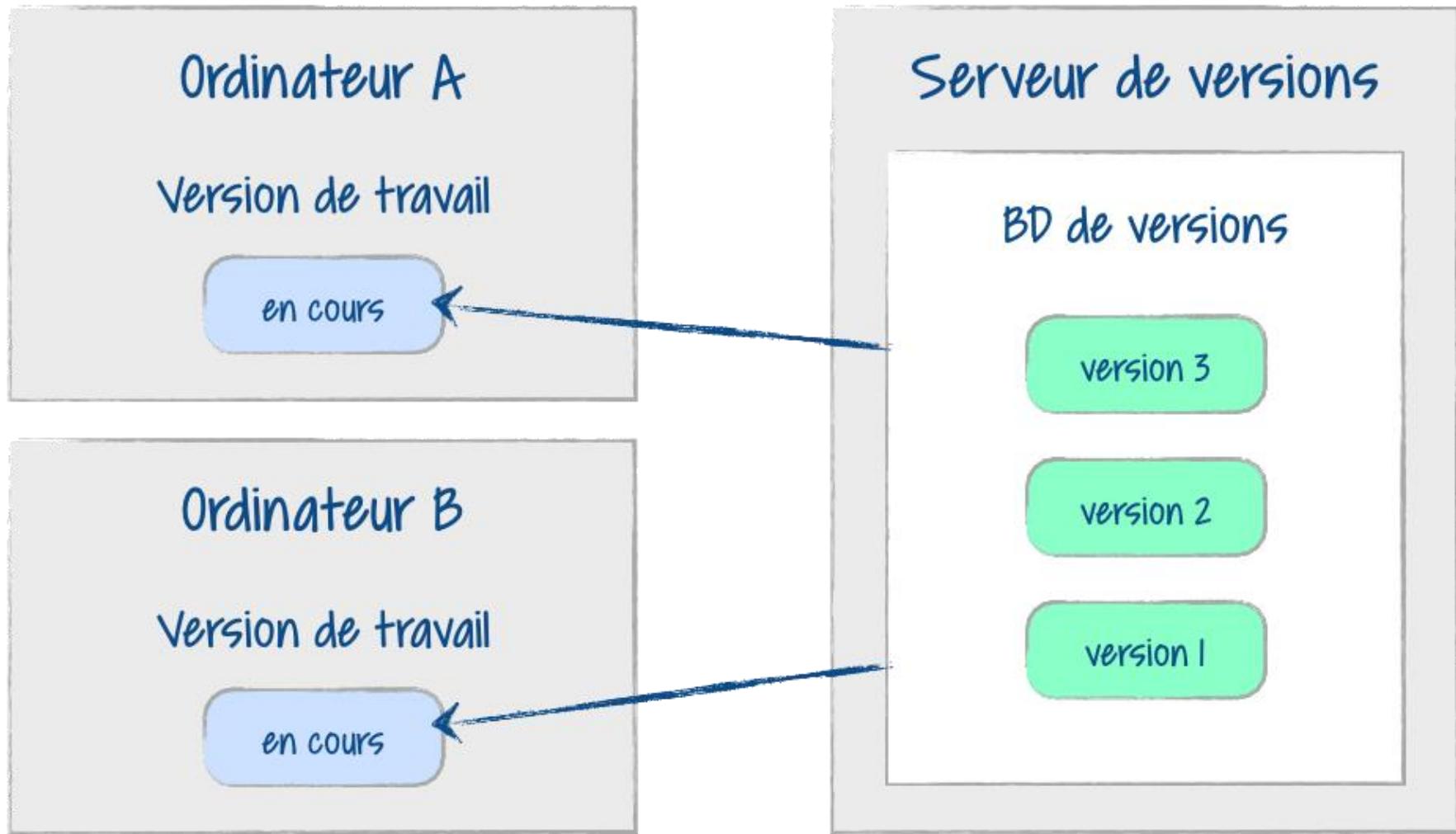
- **Parce qu'on veut pouvoir travailler à plusieurs...**
 - sans se marcher dessus
 - sans devoir tout consolider manuellement
 - sans se bloquer les uns les autres
 - en réseau... ou hors-ligne !

À la base : des sauvegardes locales



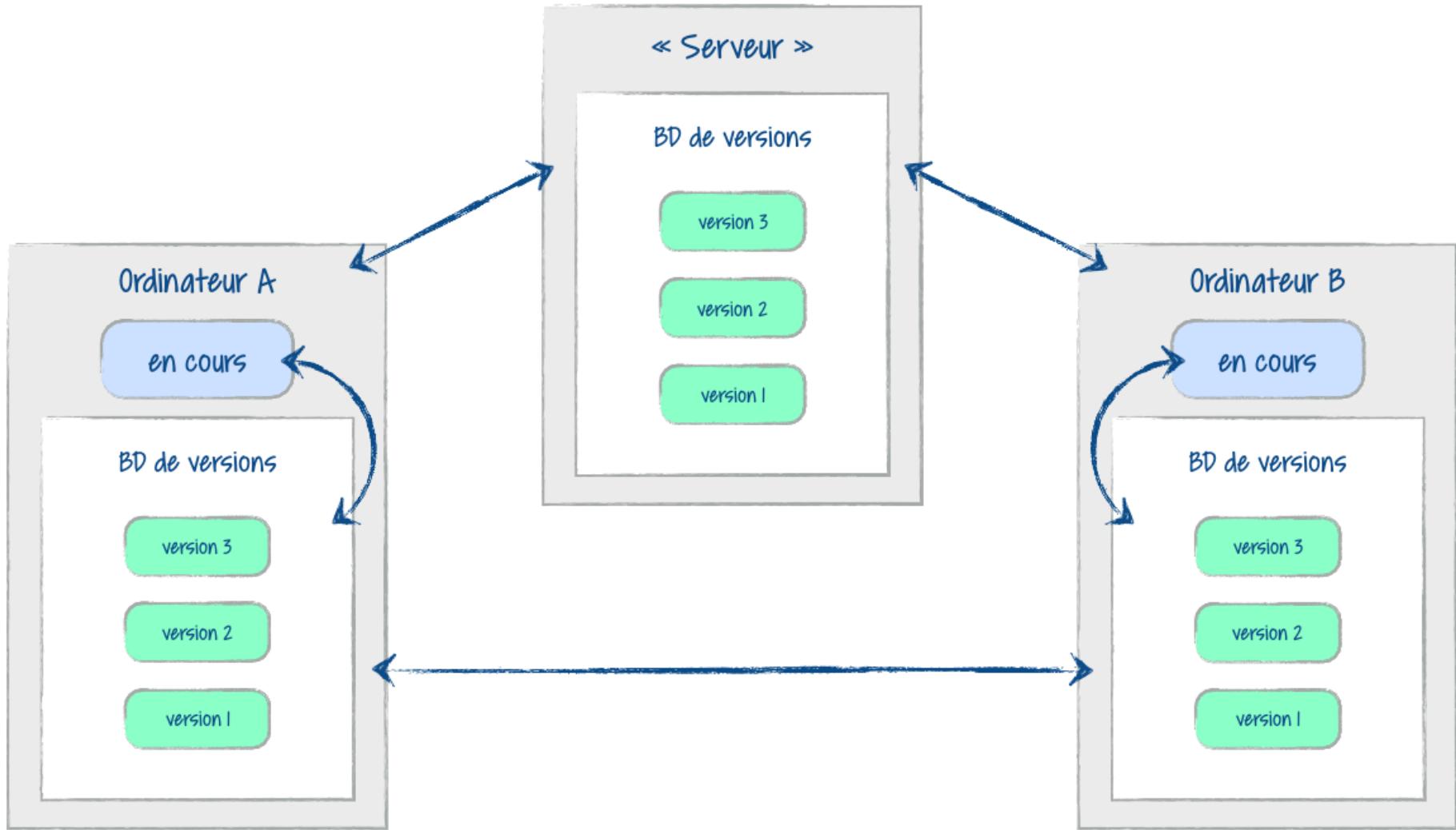
Puis vinrent les gestions centralisées

(SVN, Source Safe, ClearCase, PVCS...)



Et enfin les gestions distribuées

(Git, GNU Arch, Mercurial, ...)

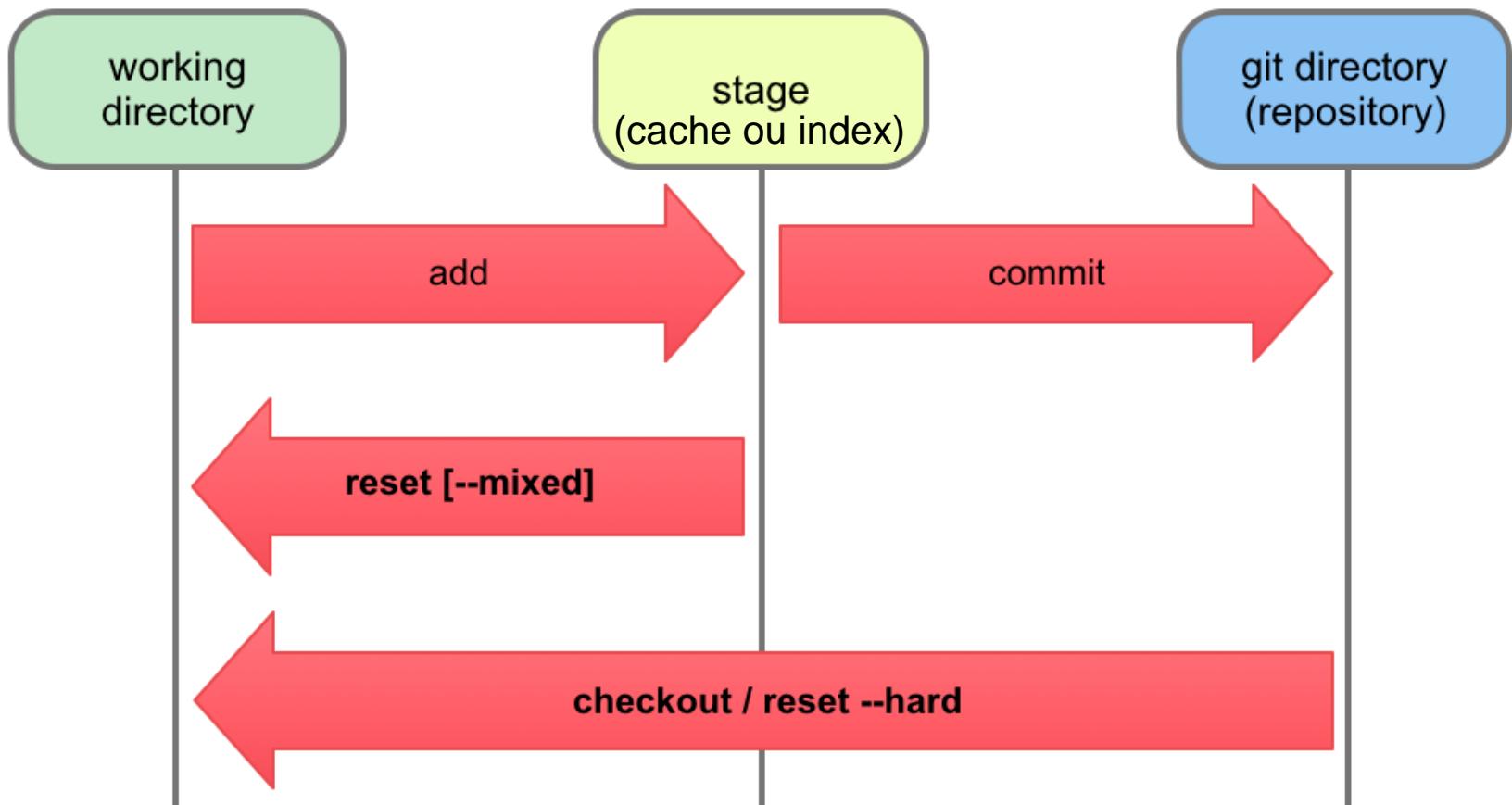


Principaux avantages du décentralisé

- Pas de **criticité d'une seule machine** (single point of failure, SPF)
- Les collaborateurs peuvent travailler **déconnectés** du « serveur ».
- La plupart des opérations sont **plus rapides**, car locales !
- On peut faire du travail « **privé** » : des brouillons, des essais... sans risque de gêner les collaborateurs.
- Rien n'empêche de garder un dépôt de référence, avalisé, sur lequel seuls certains peuvent écrire.

Concepts fondamentaux

3 containers locaux



Super ressource interactive : <http://ndpsoftware.com/git-cheatsheet.html>

3 containers locaux

- **Working directory** : Répertoire de travail. Vu comme un bac à sable. C'est là que les fichiers sont modifiés avant d'être éventuellement envoyés dans le stage/cache avec un `git add`.
- **Stage** : Zone de préparation (aussi nommé « **cache** » ou « **index** »). **Contient** la liste des fichiers qui seront examinée lors du prochain `git commit`.
- **Repository** : Ensemble des commits réalisés. Le HEAD (état courant) pointe vers l'un d'eux. Le HEAD sera le parent du prochain commit. Les commandes `git checkout` (changement de branche) et `git reset` (annulation d'une modif) permettent de déplacer le HEAD.

Exemples

WORKING DIRECTORY -> STAGE : « je prépare un commit »

```
git add myFile.fic
```

STAGE -> WORKING DIRECTORY : « je l'annule »

```
git reset HEAD myFile.fic
```

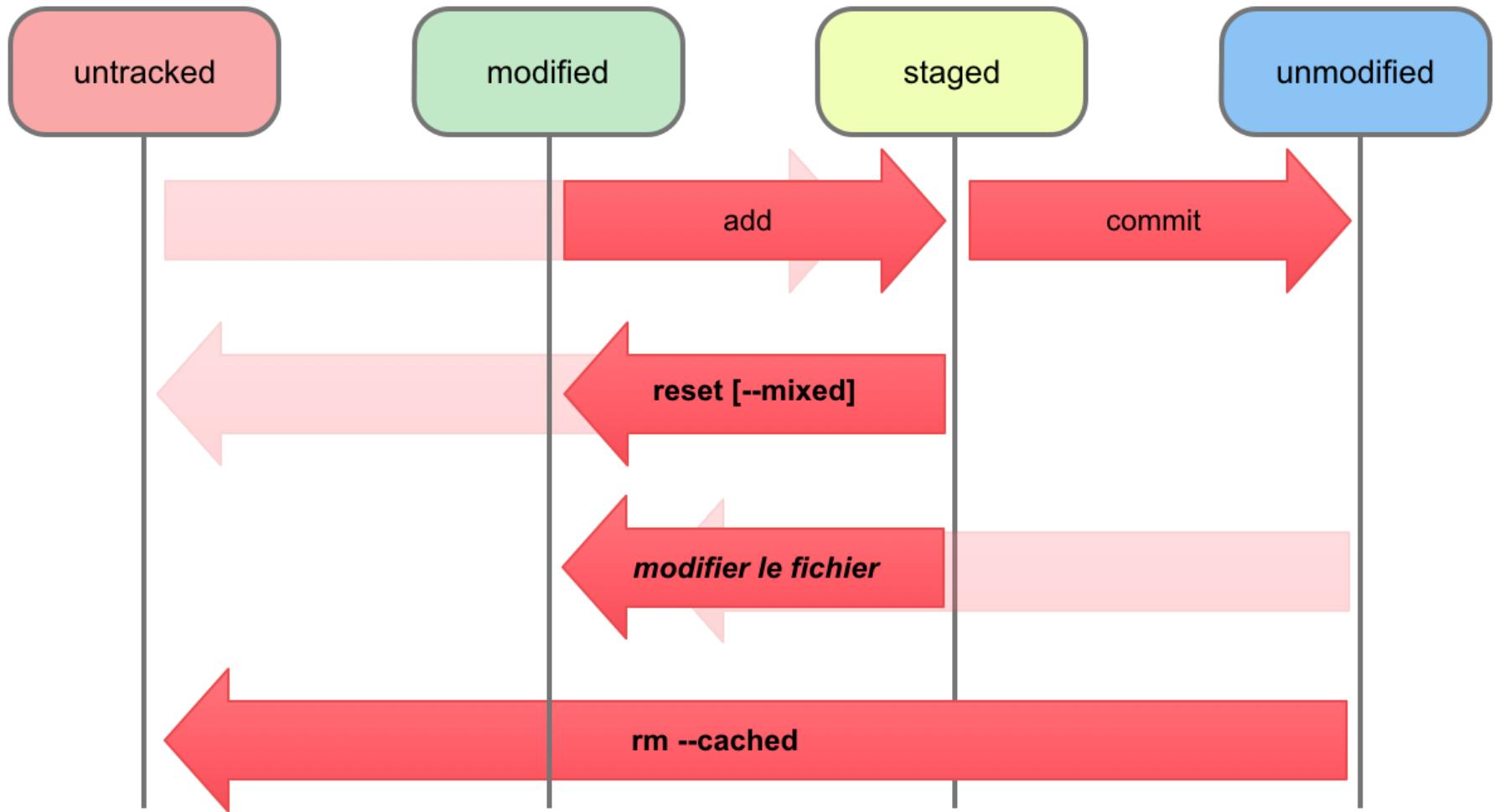
STAGE -> REPOSITORY : « je commit ce que j'ai dans le stage »

```
git commit -m "FIX FIELD-12 ...bla bla"  
(association avec le ticket JIRA FIELD-12)
```

REPOSITORY -> WORKING DIRECTORY : « J'annule des modifs locales »

```
git reset --soft HEAD^  
(c.f. annexe pour signification HEAD^)
```

Les états : le cycle de vie du fichier



Le *stage* / l'index

- `git add` = prendre une photo, stocker dans l'index / le stage
- `git commit` = prendre les photos du stage et en faire un commit (metadonnées + tree + blobs)

On peut donc avoir un même fichier dans deux zones (stage et WD) mais avec un état différent (staged, forcément, mais aussi dirty/changed) : il suffit de le modifier entre le `add` et le `commit`

git help

- git help *commande*
 - Ouverture de la page d'aide dans le browser par défaut

Exemple : git help commit

git-commit(1) Manual Page

NAME

git-commit - Record changes to the repository

SYNOPSIS

```
git commit [-a | --interactive | --patch] [-s] [-v] [-u<mode>] [--amend]
           [--dry-run] [(-c | -C | -fixup | -squash)<commit>]
           [-F <file> | -m <msg>] [--reset-author] [--allow-empty]
           [--allow-empty-message] [--no-verify] [-e] [--author=<author>]
           [--date=<date>] [--cleanup=<mode>] [--no-status]
           [-i | -o] [-S[<keyid>]] [-.] [<file>...]
```

DESCRIPTION

Stores the current contents of the index in a new commit along with a log message from the user describing the changes.

The content to be added can be specified in several ways:

1. by using `git add` to incrementally "add" changes to the index before using the `commit` command (Note: even modified files must be "added");
2. by using `git rm` to remove files from the working tree and the index, again before using the `commit` command;

A photograph of an open cardboard box. The box is made of brown paper and is shown from a top-down perspective, slightly angled. Its four flaps are spread out, revealing the interior. The box appears to be empty.

**Travailler avec son
dépôt localement**

Créer un dépôt

- Soit **git init...** (quand on est les initiateurs du dépôt).
 1. `git init` : créé un répertoire `.git` et son arborescence (en particulier le fichier `.git/config`)
 2. Création d'un fichier `.gitignore` à la racine du répertoire courant (voir [gitignore.io](#) pour la génération automatique du `.gitignore`)
 3. `git add .`
 4. `git commit -m "Initial commit"`
- Pour créer un repository serveur (dans lequel on souhaite pouvoir faire du push). Sur une clef USB [par exemple](#).
 - `git init --bare`

Créer un dépôt

- ...soit **git clone** (quand on récupère un dépôt distant existant).
 - SSH
 - `git clone git@github.com:tdd/adcraft.git`
 - `git clone ssh://git@github.com/tdd/adcraft.git`
 - Protocole GIT (daemon GIT)
 - `git clone git://github.com:rails/rails.git`
 - Http/s
 - `git clone https://git.lolcatz.com/git/yolo.git`

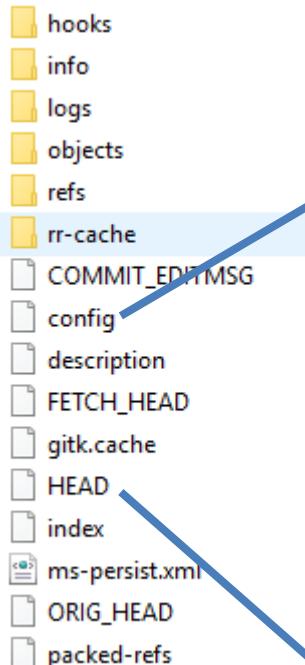
Cloner un dépôt Bitbucket

- Cloner depuis Bitbucket

The screenshot shows the Bitbucket web interface for a repository named 'HARPAGON - SPAD / HARPAGON-SPAD'. On the left, there's a sidebar with various icons. One icon, which is a downward arrow inside a square, is circled in red. The main area shows a list of files: '.vscode', 'apps', 'docs', 'packages', 'test', and '.eslintrc.yml'. To the right, there's a section for cloning the repository. It displays the HTTP URL: <https://svinfulabbk.dassault-avion.fr>. Below the URL, there's a link 'En savoir plus à propos du clonage de dépôts' and a button 'Cloner dans Sourcetree'. To the right of the button is a small graphic of a circuit board.

```
$ git clone https://svinfulabbk.dassault-avion.fr:8447/scm/phm/harpagon-spad.git
```

Le répertoire .git/



```
[core]
repositoryformatversion = 0
filemode = false
bare = false
logallrefupdates = true
symlinks = false
ignorecase = true
[http]
postBuffer = 1524288000
[remote "origin"]
url = http://D106130@svinfulabbk.dassault-avion.fr:7990/scm/fldm/field6.git
fetch = +refs/heads/*:refs/remotes/origin/*
[remote "Iomega"]
url = e:/dev/GIT-REPOSITORY-FIELDTOUCH
fetch = +refs/heads/*:refs/remotes/Iomega/*
```

Liste les repots distants, en particulier origin.
Ici Iomega est un disque USB local.
C'est le **seul** fichier du répertoire que l'on
peut être amené à modifié à la main !

ref: refs/heads/master

Indique la position courante (HEAD)
dans l'arborence des commits

git add

- `git add .` : tout le dossier (< 2.0 : sauf les suppressions !) à partir du répertoire courant.
- `git add -u` : tous les fichiers connus (modifs / suppressions)
- `git add -A` : absolument tout (modifs, supprs, ajouts) indépendamment du répertoire courant (à éviter)
- `git add -f` : force le(s) fichier(s) si le chemin est en temps normal ignoré (cf. `.gitignore`)
- `git add -p fichier...` : seulement certaines modifs du fichier (permet de visualiser le détail de toutes les modifications et de les valider ou non)

git commit

- `git commit -m` : message mono-ligne à la volée
- `git commit -a` : auto-stage tout ce qui est déjà tracké (connu)
Je répète : pas les untracked, alors attention !
- `git commit --amend` : permet de mettre à jour le dernier commit local
(attention ne pas utiliser que le commit a déjà été pushé sur le serveur car modifie le SHA-1)
- `git commit -t template` : pré-charge un gabarit de message
- Vous voulez savoir un peu plus comment tout ça fonctionne ?
 - `git help core-tutorial / git help tutorial-2`
 - [git help repository-layout](#)

git commit avec JIRA

- Le commit doit être associé à un ticket JIRA

git commit -m "SPAD-373 SPAD-375 templates"

The screenshot shows a Jira Software interface for the project 'SPAD'. A red arrow points from the commit message 'SPAD-373 SPAD-375 templates' in the terminal to the ticket numbers 'SPAD-373' and 'SPAD-375' in the Jira header. Another red arrow points from the commit message to the '1 commit' link in the Jira commit history.

Jira Software - Dassault Aviation

SPAD / SPAD-375

Homogénéisation - Création d'un template Vue modale

SPAD-375: 1 unique commit

HARPAGON-SPAD (Bitbucket O)

Author	Commit	Message	Date	Files
ab2f414ea45	SPAD-373 SPAD-375 templates		1 week ago	15 files

Created: 28/Aug/20 5:32 PM
Updated: 1 week ago

Attachments

Drop files to attach, or browse.

1 commit

git status (alias git st)

- Permet de voir immédiatement :
 - Les fichiers non versionnés (*untracked*)
 - Les fichiers modifiés (*changes not staged for commit*)
 - Les modifs validées (*changes to be committed*)
 - Certains cas spéciaux (ex. *both modified* : conflits de fusion...)
- Fournit des indications utiles aux débutants
 - Pour valider une modification
 - Pour retirer une modif validée du *stage*
- Le *prompt* fournit une synthèse (et même davantage)
- *git status -sb* : version concise (un peu plus jolie)

git diff

- `git diff [file]` : compare staged avec repository
- `git diff --staged [file]` : compare WD avec staged (idem `git diff --cached [file]`)
- `git diff --stat/numstat/shortstat/dirstat[=...]` : statistiques sur les modifs en cours
- `git diff -w` : ignore whitespace (idem `--ignore-all-space`)
- `git diff --ignore-blank-lines` : ignore les lignes vides
- `git diff --word-diff/word-diff-regex=...` : comparaison au niveau du mot et pas au niveau de la ligne
- `git diff SHA1-commit1:[file] SHA1-commit2:[file]` : Comparaison de deux versions (dans 2 commits différents)
- Contenu du fichier dans le stage : `git show :0:chemin`
- Configuration (dans `./.git/config`) :
 - `diff.mnemonicPrefix = true`
 - `diff.wordRegex = .`

Exemple : diff

- Ajout et commit d'un premier fichier
 - `git add file.txt`
 - `git commit -m "Premier commit"`
- Modification du fichier
- Comparaison WD <-> STAGE
 - `git diff file.txt`
- `git add file.txt`
- Comparaison WD <-> STAGE
 - `git diff file.txt`
- Comparaison STAGE <-> REPOSITORY
 - `git diff --staged file.txt`
- Comparaison WD <-> REPOSITORY
 - `git diff HEAD file.txt`

```
$ git lg
* 1ce3b33 - (HEAD -> master) Modif dans file2.txt (Stephane Michel 76 minutes ago)
* 1b58269 - Modif dans toto.txt (Stephane Michel 76 minutes ago)
* ded5ea6 - Suppression du fichier file2.txt (Stephane Michel 3 hours ago)
* a57bdbb - new commit (Stephane Michel 3 hours ago)
* dfeb1ac - Ajout de toto.txt (Stephane Michel 3 hours ago)
* 857c44e - Premiere modif (Stephane Michel 3 hours ago)
* a750e4b - Ajout de Test.java (Stephane Michel 3 hours ago)
* 3826cdb - initial commit (Stephane Michel 4 hours ago)
```

- Comparaison d'un fichier entre 2 commits :
 - `git diff dfeb1ac:toto.txt 1ce3b33:toto.txt`

git difftool

- Recourt à un outil tiers pour visualiser le diff, par ex. pour...
 - Un surlignage localisé des lignes
 - Des sémantiques de diff spécifiques (e.g. XML, JSON, SVG, CSS...)
 - Plus de confort visuel
- Listez les outils connus, dont ceux détectés chez vous
 - `git difftool --tool-help`
- Configurez `diff.tool` et `difftool.prompt` (dans `.gitconfig`)
- Quelques outils à explorer
 - [BeyondCompare](#) sur Windows, [Kaleidoscope](#) sur OSX
 - [vimdiff](#) / vimdiff3 en terminal, [Meld](#) ou [P4Merge](#) en multi-plate-forme
- Le diff de VS Code est très bien aussi...

Configuration pour Beyond Compare

- Dans `~/.gitconfig` (U:`:/ .gitconfig` sur poste @)

```
[diff]
tool = bc3
# Use better, descriptive initials (c, i, w) instead of a/b.
mnemonicPrefix = true
# Show renames/moves as such
renames = true
# When using --word-diff, assume --word-diff-regex=.
wordRegex = .
# Display submodule-related information (commit listings)
submodule = log
[difftool "bc3"]
cmd = \"c:/program files/beyond compare 4/bcomp.exe\" \"$LOCAL\" \"$REMOTE\"
```

- `git difftool "monFichier"`

git log (alias git lg ou git l)

- `git log --stat/numstat/shortstat/dirstat[=...]`
 - Stats rapides sur l'impact de l'historique sur le dépôt
 - dirstat en particulier est sympa pour une vue à 10 000 m
- `git log --graph` : Visualise les branches, fusions, etc.
- `git log -p` : Affiche les diffs à la volée
- `git log --author=...` : filtre par auteur
- `git log --grep -E -i ...` : filtre par message de commit (complet)
- `git log -S / -G` : filtre sur les lignes actives du diff (texte / regexp)
- `git log -L` : filtre local à un corps de fonction / fragment de code !
- `git log -{n}` : récupère les n dernières lignes du log (exemple : `git log -10` pour les 10 dernières lignes)
- `git log --all --full-history -- **/monfichier.ext` : permet de retrouver tous l'historique de commit qui concerne un fichier donné (même s'il est supprimé)
- Configuration (dans `/.git/config`) :
 - `log.abbrevCommit = true` (version abrégée du SHA-1 au lieu des 40 caractères)
 - Alias (défini dans `.gitconfig`) : `git lg` et `git l`

```
[alias]
lg = log --graph --date=relative --pretty=format:'%Cred%h%Creset -%C(auto)%d%Creset %s %Cgreen(%an %ad)%Creset'
l = log --graph --date=relative --pretty=format:'%Cred%h%Creset -%C(auto)%d%Creset %s %Cgreen(%an %ad)%Creset' --all -15
```

Exemple : log

- Logs de la branche courante
 - git log
 - git lg (alias défini dans .gitconfig)
 - git l (idem git lg mais toutes les branches et limite à 15 commit par défaut)

```
$ git l -50
* 1f268df7 - (HEAD -> master, origin/master, origin/HEAD) home-spa-app et harpagon-spa-app : deploy cross plateforme (Stephane Michel 4 days ago)
* 8fed4186 - SPAD-360 Fix test unitaires apres merge (Stephane Michel 4 days ago)
* 6669c4d3 - Suppression commentaires inutiles et printWidth a 120 caracteres dans .prettierrc (Stephane Michel 5 days ago)
* 4cca6d3c - SPAD-360 Fix apres modification des settings (Stephane Michel 5 days ago)
* adeb220c - ENVIRONNEMENT Config VSCode par defaut (tab a 2 caracteres, etc.) (Stephane Michel 5 days ago)
* 1f0e3524 - SPAD-360 Gestion des cas limites (Stephane Michel 5 days ago)
* 236e5818 - ELECTRON Pas de refresh token si l'utilisateur s'est deconnecte (Stephane Michel 5 days ago)
* 4331ed7b - SPAD-360 Masquage SPA courante pour affichage popup (Stephane Michel 6 days ago)
* 1b6cbb7e - SPAD-360 tests unitaires (Stephane Michel 7 days ago)
* 8e469778 - SPAD-360 Gestion premier appel et gestion des erreurs (Stephane Michel 7 days ago)
* 4a8d05cd - SPAD-360 Settings URL du serveur d'API (Stephane Michel 10 days ago)
* 29564852 - couverture des testes de date-helper-pkg (Jérémie Béhérec 5 days ago)
* 63ff764f - merge populate (Jérémie Béhérec 5 days ago)
| \
* 77ceced2 - SPAD-292 synchronisation des FUA/phases (Jérémie Béhérec 5 days ago)
|   * 417d91ab - (SETTINGS) Suppression commentaires inutiles (Stephane Michel 5 days ago)
|   * 938da755 - SPAD-360 Fix apres modification des settings (Stephane Michel 5 days ago)
|   * bb0a6dc9 - ENVIRONNEMENT Config VSCode par defaut (tab a 2 caracteres, etc.) (Stephane Michel 5 days ago)
|   * 1033238d - SPAD-360 Gestion des cas limites (Stephane Michel 5 days ago)
|   * 337f98d2 - ELECTRON Pas de refresh token si l'utilisateur s'est deconnecte (Stephane Michel 5 days ago)
|   * 61e8d5be - SPAD-360 Masquage SPA courante pour affichage popup (Stephane Michel 6 days ago)
|   * 47a40a18 - SPAD-360 tests unitaires (Stephane Michel 7 days ago)
|   * cfac113c - SPAD-360 Gestion premier appel et gestion des erreurs (Stephane Michel 7 days ago)
|   * 76b38644 - SPAD-360 Settings URL du serveur d'API (Stephane Michel 10 days ago)
| /
* ab2f414e - SPAD-373 SPAD-375 templates (Dousset Sylvain 5 days ago)
* f9eee335 - ecran referentiels #SPAD-369 (JPP 5 days ago)
* 2dd33b8c - SPAD-183: Dialog state et site (Sébastien Philippe 6 days ago)
* b2a32d11 - merge populate (JPP 6 days ago)
* 3f8c4353 - SPAD-373 modifs test coverage (Dousset Sylvain 6 days ago)
* 6ca17a31 - SPAD-373 modifs suite remarques de la réunion (Dousset Sylvain 6 days ago)
|
* 0fa0b055 - SPAD-373 couverture des tests (Dousset Sylvain 7 days ago)
```

git show

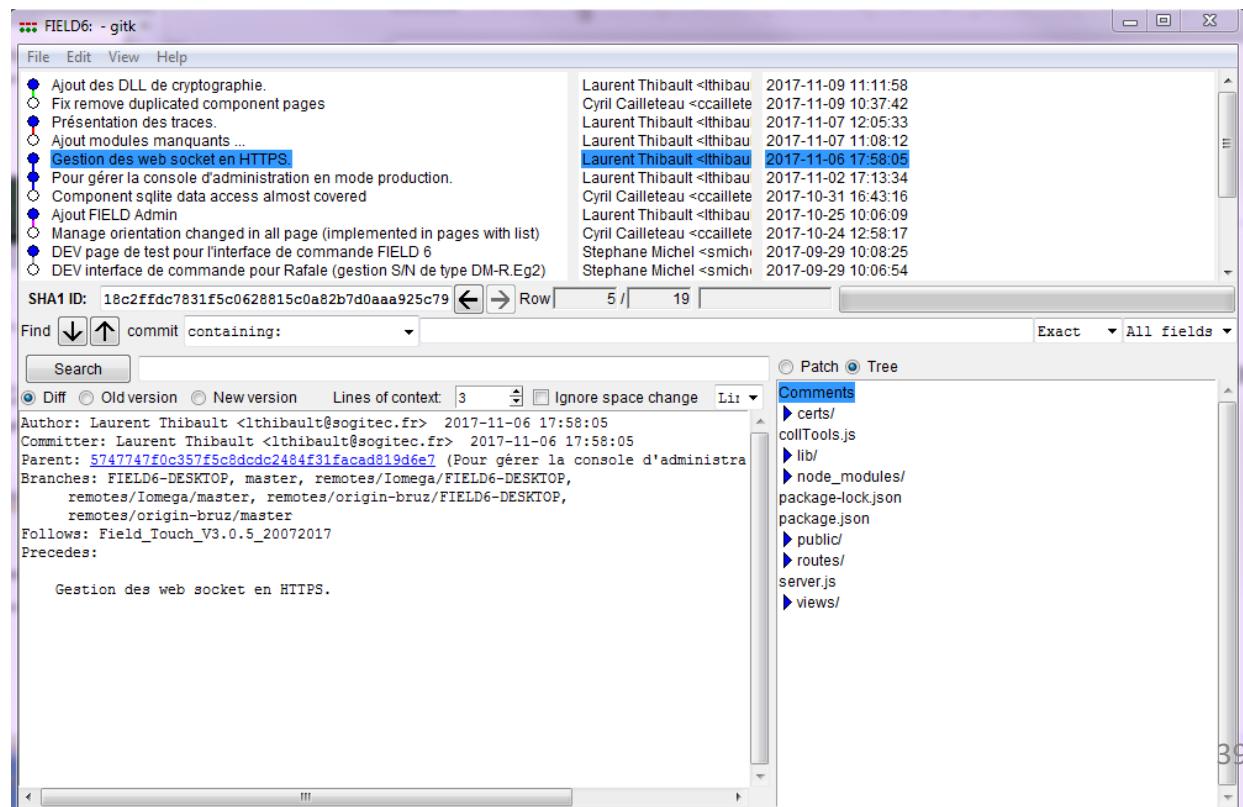
- Affiche une entité Git (*tag, commit, branche, fichier*)
- `git show commit`
 - Message + diff
- `git show fichier`
 - Contenu du fichier (version spécifique du fichier)
 - Fichier dans un commit précis : *commit:chemin*
- `git show tag`
 - Détails du tag + vue de l'entité taggée (en général un commit)

gitk

- Interface graphique inclue avec GIT

gitk [mon_fichier]

- Permet de visualiser l'historique de modification d'un fichier



Le fichier `.gitconfig`

- Fichier de configuration par défaut de GIT
 - Permet de :
 - Définir des alias de commandes comme « git st », « git log » ou « git l »
 - Spécifier les informations utilisateur
 - User.name
 - User.email
 - Paramétriser le difftool ou le mergetool
 - à copier sous :
 - C:\utilisateurs\%mon_user%\ sur un poste « normal »
 - U:\ sur les postes @
 - Voici [ici pour un exemple de fichier .gitconfig](#)

Le fichier .gitconfig

- Info utilisateur :

```
[user]
  name = Mon nom
  email = Mon.Nom@dassault-aviation.com
```

- Alias

```
[alias]
  st = status
  ci = commit
  lg = log --graph --date=relative --pretty=tformat:'%Cred%h%Creset -
%C(auto)%d%Creset %s %Cgreen(%an %ad)%Creset'
  l = log --graph --date=relative --pretty=tformat:'%Cred%h%Creset -
%C(auto)%d%Creset %s %Cgreen(%an %ad)%Creset' --all -15
  oops = commit --amend --no-edit
  dt = difftool
  mt = mergetool
```



Que réalise la commande suivante ?

```
$ git st
```

```
D106130@BUU00002 MINGW64 /l/wRK/harpagon-spad/apps/home-spa-app (master)
$ git st
On branch master
Your branch is ahead of 'SM64/master' by 26 commits.
  (use "git push" to publish your local commits)

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:   README.md

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

« *Liste les fichiers en cours de modification et ceux non encore suivis par GIT* »

Que réalise la commande suivante ?

```
$ git diff README.md
```

```
D106130@BUU00002 MINGW64 /l/WRK/harpagon-spad/apps/home-spa-app (master)
$ git diff README.md
diff --git i/apps/home-spa-app/README.md w/apps/home-spa-app/README.md
index 070afa5f..cb1a2019 100644
--- i/apps/home-spa-app/README.md
+++ w/apps/home-spa-app/README.md
@@ -1,5 +1,6 @@
 # Description

Cette application est l'application principale. On y trouve notamment la page d'accueil présentant l'ensemble des applications clientes du projet.
+ajout d'une ligne

-Le répertoire `/extensions` contient les plugin React et Redux des DEvTools Chromium.
+Le répertoire `/extensions` contient les plugin React et Redux des DEvTools Chromium et patati et patata.
```

« Affiche les modifications sur le fichier README.md dans le Working Directory »

Que réalise la commande suivante ?

```
$ git add .
```

```
D106130@BUU00002 MINGW64 /l/WRK/harpagon-spad/apps/home-spa-app (master)
$ git add .

D106130@BUU00002 MINGW64 /l/WRK/harpagon-spad/apps/home-spa-app (master)
$ git st
On branch master
Your branch is ahead of 'SM64/master' by 26 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README.md
```

« Les fichiers en cours de modification ont été placés dans le « stage », ils sont prêt à être commités. »

Que réalise la commande suivante ?

```
$ git commit -m "HARP-123 bla bla."
```

```
D106130@BUU00002 MINGW64 /l/WRK/harpagon-spad/apps/home-spa-app (master)
$ git commit -m "HARP-123 bla bla."
[master e5e76ccb] HARP-123 bla bla.
 1 file changed, 2 insertions(+), 1 deletion(-)
```

« Les fichiers du stage (ici *README.md*) sont commités dans le repository local (pas encore dans le repo distant). »

Que réalise la commande suivante ?

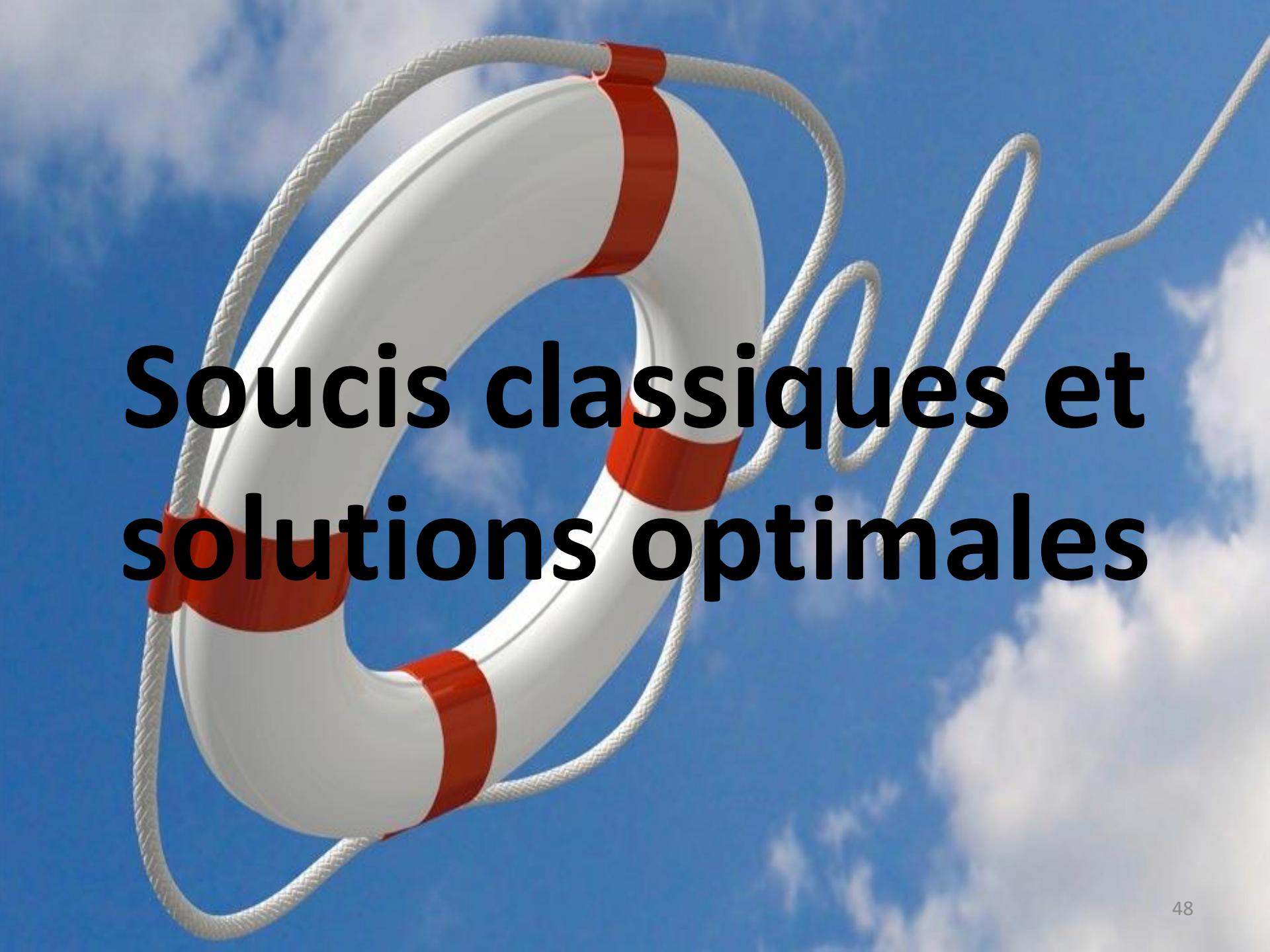
```
$ git l
```

oubien (si l'alias n'est pas défini)

```
$ git log --graph --date=relative --pretty=tformat:'%Cred%h%Creset - %C(auto)%d%Creset %s %Cgreen(%an %ad)%Creset' --all -15
```

```
D106130@BUU00002 MINGW64 /l/WRK/harpagon-spad/apps/home-spa-app (master)
$ git l
* e5e76ccb - (HEAD -> master) HARP-123 bla bla. (Stephane Michel 2 minutes ago)
* 55492fcf - Racine Projet : update package-lock.json (Stephane Michel 3 hours ago)
* 35aefab9 - (origin/master, origin/HEAD) SPAD-183: Mise en place du tri des avions (Sébastien Philippe 23 hours ago)
* 547df0b3 - SPAD-183: enregistrement des avions en bdd aircraft. Data à revoir (Sébastien Philippe 25 hours ago)
* b5723e55 - spad vega group base mp (jean-baptiste.pelletier 28 hours ago)
* 58d8e4d9 - Remove Phase #SPAD-289 (JPP 2 days ago)
```

« Affiche l'état du repo (historique des commits) »

A white and red lifebuoy hangs from a white rope against a backdrop of a clear blue sky with wispy white clouds. The lifebuoy is oriented vertically, with its red bands at the top and bottom. A white rope is wrapped around it twice, once near the top and once near the bottom. The rope extends from the left side of the frame towards the right.

Soucis classiques et solutions optimales

Soucis classiques et solutions optimales

- **J'ai oublié un truc (ou j'ai mis un truc en trop) dans mon commit !**
 - `git commit --amend`
 - En fait un `reset --soft HEAD^` suivi d'un commit classique...
 - Permet aussi de supprimer un fichier d'un commit : dans ce cas remettre le fichier dans le stage dans l'état « avant le commit » (voir exemple suivant)
 - `git commit --amend -m "nouveau message"`
 - Permet de renommer le commit précédent
 - `git commit --amend --no-edit` (alias “`git oops`”)
 - Ré-utilise les méta-données d'auteur et le message du commit d'origine
 - Ni vu ni connu, on croirait que tu avais tout bon du 1er coup !



Exemple : commit --amend 1/3

- On avait préparé notre commit (*stage/index* prêt) sauf qu'au moment du commit nous allons avoir ce réflexe malheureux d'ajouter l'option -a.
 - Du coup nous avons commité un fichier (fic1, qui était en cours de modification) en trop.
 - On souhaite le supprimer du dernier commit !

```
$ git lg
* 03ecb65 - (HEAD -> master) Ajout contenu dans fic2 (Stephane Michel 3 minutes ago)
* 5502a19 - fic2 (Stephane Michel 4 minutes ago)
* 03a1048 - Modif1 (Stephane Michel 6 minutes ago)
* cb60533 - fic1 (Stephane Michel 10 minutes ago)
```

```
$ git show --name-only 03ecb65
commit 03ecb65
Author: Stephane Michel <smichel@sogitec.fr>
Date:   Tue Jan 17 19:33:15 2017 +0100

        Ajout contenu dans fic2

        fic1
        fic2
```

Exemple : commit --amend 2/3

1 en dessous du HEAD voir [ici](#) pour plus de détails

- `git reset HEAD~1 fic1` (ou `git reset 5502a19 fic1`)

```
$ git reset 5502a19 fic1
Unstaged changes after reset:
M      fic1
```

-> le fichier fic1 est remis dans le stage/index dans l'état « avant le commit »

```
$ git diff --staged fic1
diff --git c/fic1 i/fic1
index 2e4d402..f41f4f7 100644
--- c/fic1
+++ i/fic1
@@ -1,3 +1,2 @@
Ajout content 1
-Modif que l'on ne souhaite pas encore commiter
```

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   fic1

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:   fic1
```

Exemple : commit --amend 3/3

- git commit --amend --no-edit
- La modif a bien disparu du dernier commit

AVANT

```
$ git show 03ecb65
commit 03ecb65
Author: Stephane Michel <smichel@sogitec.fr>
Date:   Tue Jan 17 19:33:15 2017 +0100

    Ajout contenu dans fic2

diff --git a/fic1 b/fic1
index f41f4f7..2e4d402 100644
--- a/fic1
+++ b/fic1
@@ -1,2 +1,3 @@
 
    Ajout content 1
+Modif que l'on ne souhaite pas encore committer
diff --git a/fic2 b/fic2
index 8b13789..2dbcce5 100644
--- a/fic2
+++ b/fic2
@@ -1 +1 @@
-
+Ajout contenu dans fic2
```

APRES

```
$ git show 50bc91c
commit 50bc91c
Author: Stephane Michel <smichel@sogitec.fr>
Date:   Tue Jan 17 19:33:15 2017 +0100

    Ajout contenu dans fic2

diff --git a/fic2 b/fic2
index 8b13789..2dbcce5 100644
--- a/fic2
+++ b/fic2
@@ -1 +1 @@
-
+Ajout contenu dans fic2
```

Notez que le SHA-1 du commit a changé !

Soucis classiques et solutions optimales

- **J'ai stagé un truc que j'aurais pas dû !**
 - `git reset`
 - Le couteau suisse de Git
 - Une des commandes hélas les plus méconnues :-(
 - `git rm --cached`
 - Si on est en *root commit*
 - Ou pour déversionner un fichier tout en le gardant dans le WD

git reset

<https://git-scm.com/book/fr/v2/Utilitaires-Git-Reset-démystifié>

Mode	HEAD	Stage	WD	Fichiers précis ?
--soft	✓			non
--mixed	✓	✓		oui
--merge	✓	✓	✓	non
--hard	✓	✓	✓	non

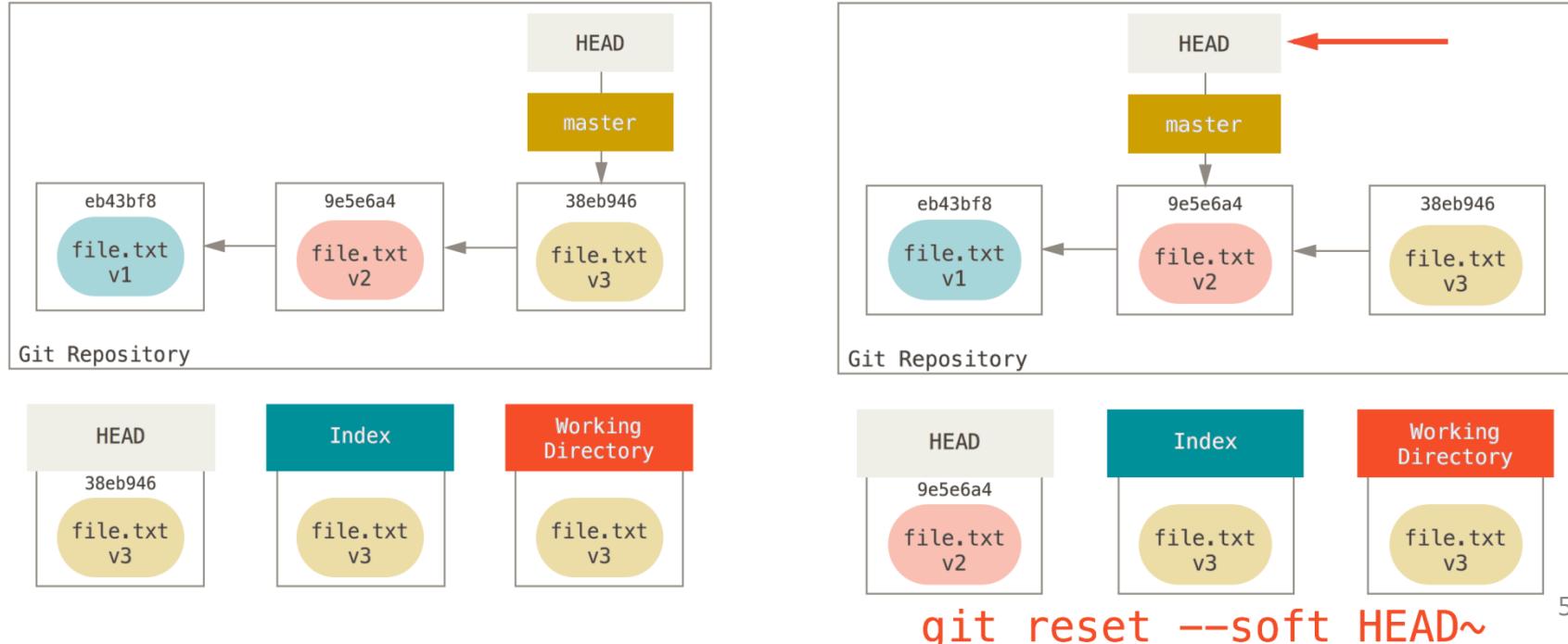


Signifie « modification »

- `git reset --merge` préserve les modifications du WD alors que `git reset --hard` fait **table rase**

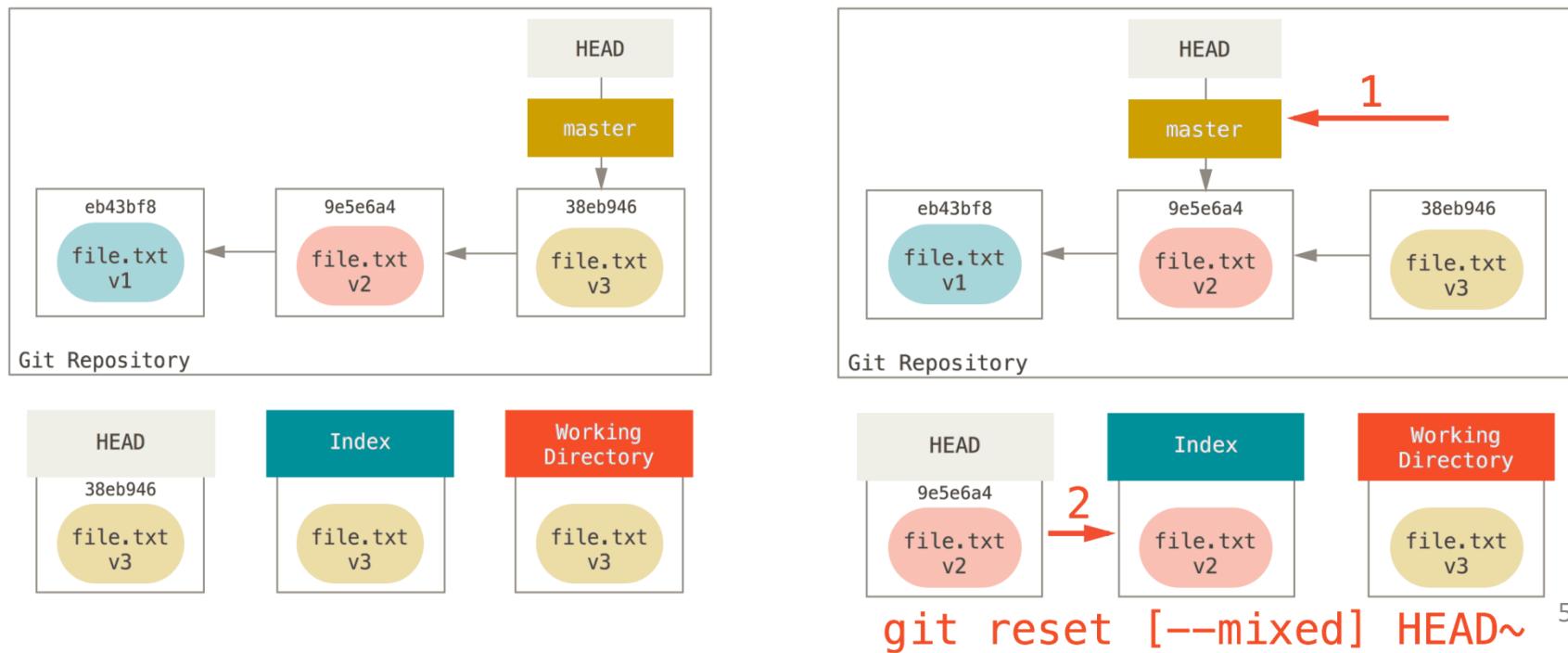
git reset --soft

- Se contente de déplacer ce qui est pointé par HEAD.
- Stage/Index n'est pas modifié.
- Working Directory n'est pas modifié.



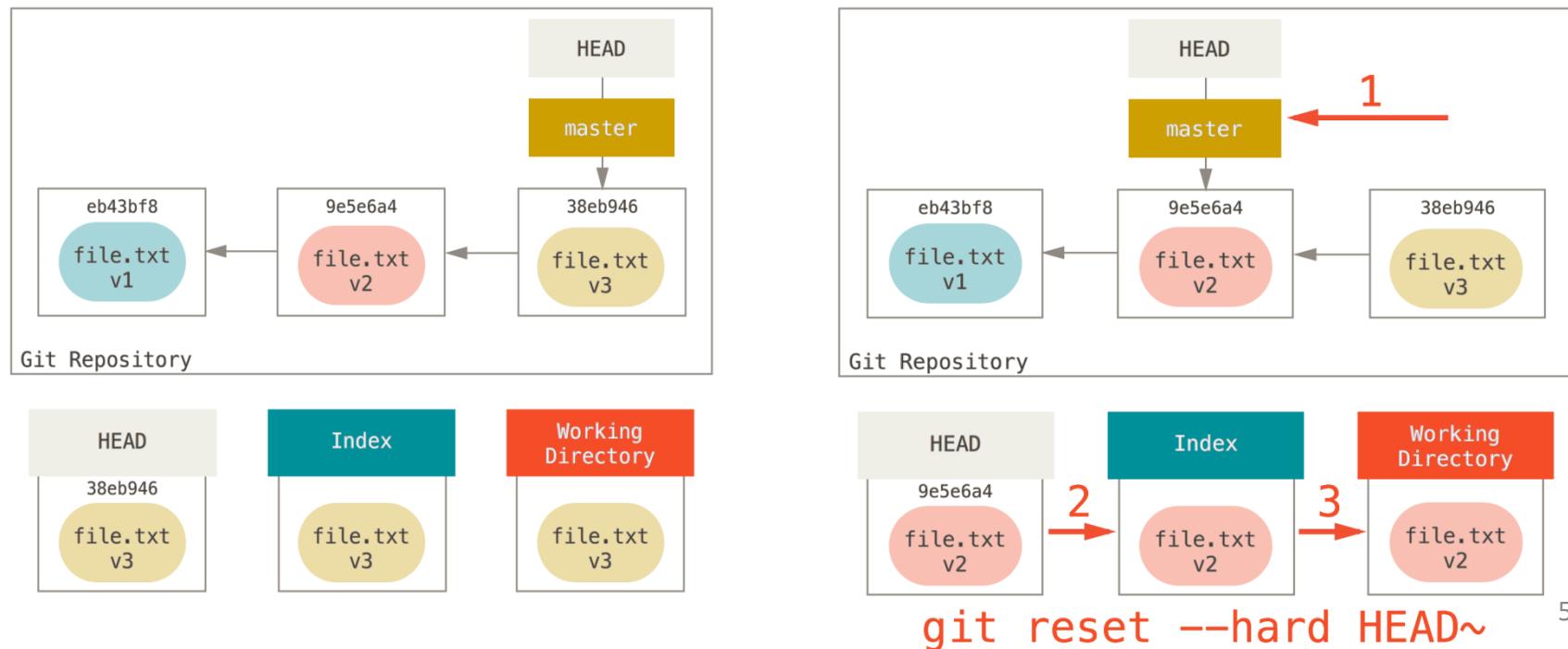
git reset --mixed (*par défaut*)

- Idem --soft + mise à jour du stage/index
- Revient à l'état précédent le dernier commit (permet d'annuler le dernier commit)



git reset --hard

- Idem --mixed + mise à jour Working directory
- **Attention** : perte des éventuelles modifications du Working directory

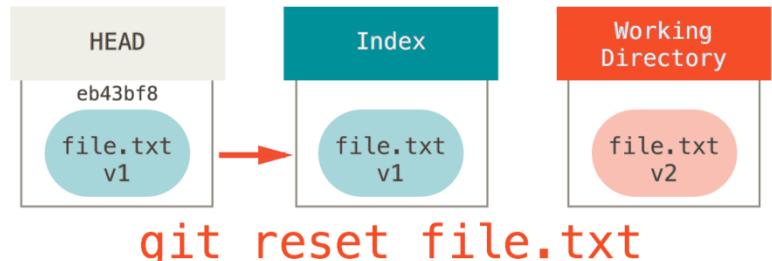
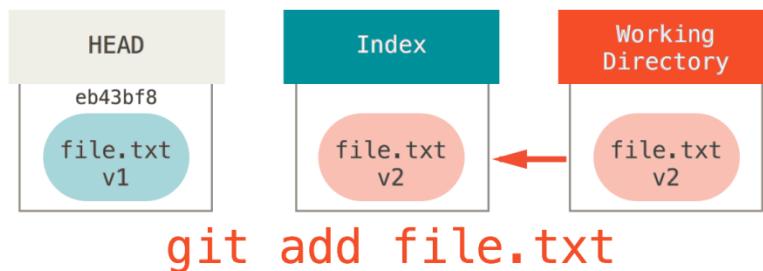
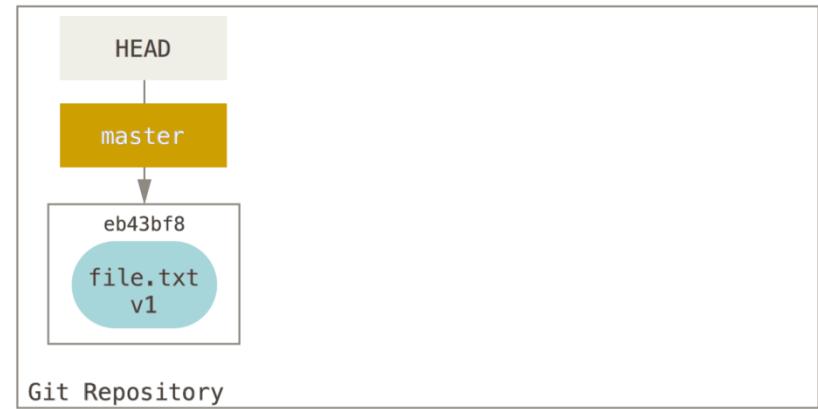
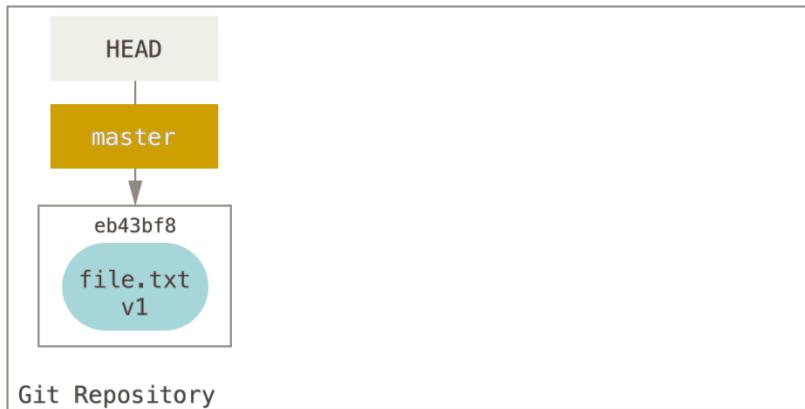


git reset --merge

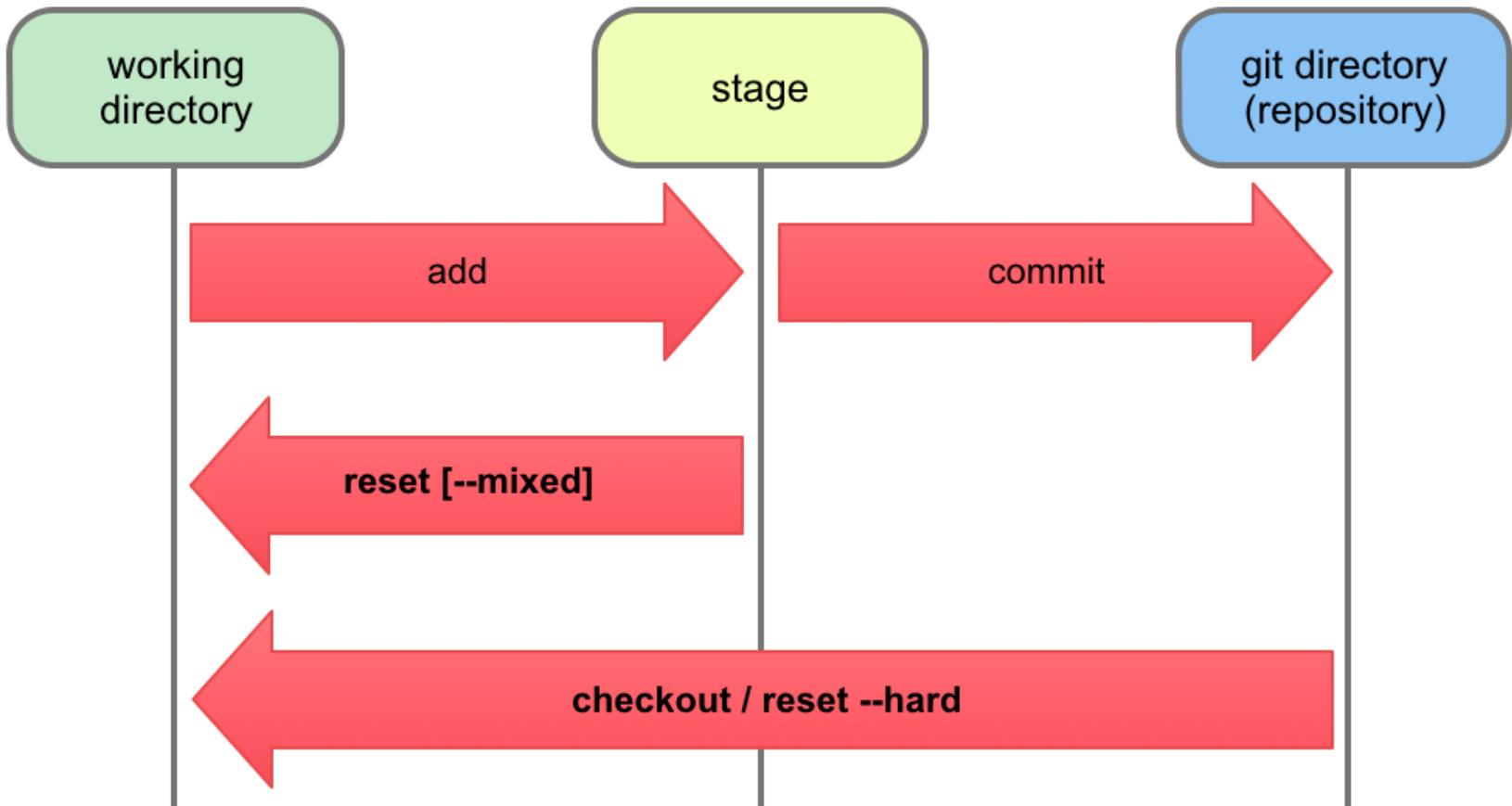
- Idem --mixed + mise à jour Working Directory
- A la différence --hard, il n'écrase pas les fichiers du Working Directory qui ont été modifiés localement (pas de risque de perte de données)

git reset *fichiers*

- Raccourci de `git reset --mixed HEAD fichiers`
- Ne déplace pas le HEAD
- Copie le ou les fichier dans le stage/index
- Ne touche pas au Working Directory
- **Fait l'exact opposé de `git add`**



Rappel : Concepts fondamentaux



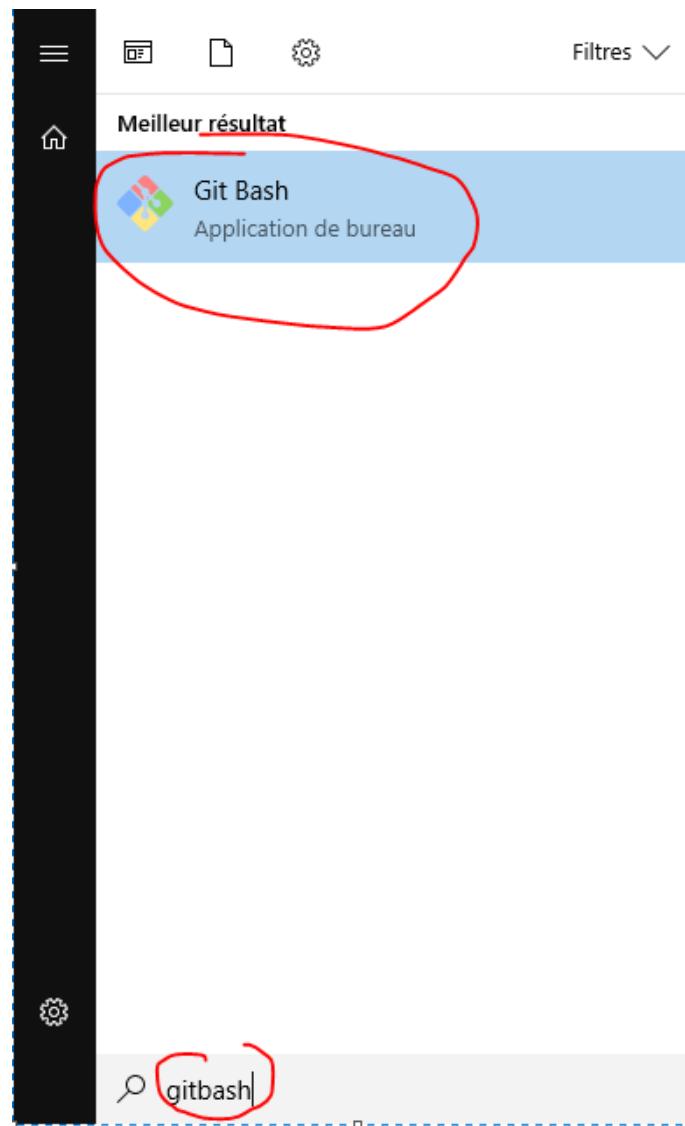
Super ressource interactive : <http://ndpsoftware.com/git-cheatsheet.html>

Practice Time

Practice makes perfect!

Préparation de l'environnement 1/2

- Lancer gitbash



Préparation de l'environnement 2/2

- Créer un répertoire temporaire sous L:/WRK/
 - cd /L/WRK/
 - mkdir practice
 - cd practice
 - git init



The screenshot shows a terminal window titled "MINGW64:/L/WRK/practice". The command history is as follows:

```
D106130@BUU00002 MINGW64 ~
$ cd /L/WRK
D106130@BUU00002 MINGW64 /L/WRK
$ mkdir practice
D106130@BUU00002 MINGW64 /L/WRK
$ cd practice/
D106130@BUU00002 MINGW64 /L/WRK/practice
$ git init
Initialized empty Git repository in L:/WRK/practice/.git/
D106130@BUU00002 MINGW64 /L/WRK/practice (master)
$ |
```

A red circle highlights the text "(master)" at the end of the last line, and a red arrow points from this circle to the text "Branche par défaut" located below the terminal window.

Branche par défaut

Exercice : reset --mixed

- Création d'un fichier reset.txt
- Ajout du fichier dans le stage
 - `git add` reset.txt
 - `git status` (ou `git st` si l'alias est défini)
- Annulation, enlève le fichier du stage
 - `git reset --` reset.txt (git reset --mixed reset.txt)
 - `git status`(ou `git st` si l'alias est défini)

Exercice : reset --soft

- Création et commit d'un fichier reset2.txt
 - git add reset2.txt
 - git commit -m "Ajout reset2.txt"

- git lg

```
$ git lg
* f15efba - (HEAD -> master) Ajout reset2.txt (Stephane Michel 43 seconds ago)
* 51b4826 - Modif dans file2.txt (Stephane Michel 2 hours ago)
* 1b58269 - Modif dans toto.txt (Stephane Michel 2 hours ago)
* ded5ea6 - Suppression du fichier file2.txt (Stephane Michel 3 hours ago)
* a57bdbb - new commit (Stephane Michel 3 hours ago)
* dfeb1ac - (tag: TAG1) Ajout de toto.txt (Stephane Michel 4 hours ago)
* 857c44e - Premiere modif (Stephane Michel 4 hours ago)
* a750e4b - Ajout de Test.java (Stephane Michel 4 hours ago)
* 3826cdb - initial commit (Stephane Michel 5 hours ago)
```

- git reset --soft 51b4826

- git lg

```
$ git lg
* 51b4826 - (HEAD -> master) Modif dans file2.txt (Stephane Michel 2 hours ago)
* 1b58269 - Modif dans toto.txt (Stephane Michel 2 hours ago)
* ded5ea6 - Suppression du fichier file2.txt (Stephane Michel 3 hours ago)
* a57bdbb - new commit (Stephane Michel 3 hours ago)
* dfeb1ac - (tag: TAG1) Ajout de toto.txt (Stephane Michel 4 hours ago)
* 857c44e - Premiere modif (Stephane Michel 4 hours ago)
* a750e4b - Ajout de Test.java (Stephane Michel 4 hours ago)
* 3826cdb - initial commit (Stephane Michel 5 hours ago)
```

- git status

Exercice : reset --hard

- Création et commit d'un fichier reset3.txt
 - git add reset3.txt
 - git commit -m "Ajout reset3.txt"
- git lg

```
$ git lg
* d35be5b - (HEAD -> master) Ajout de reset4.txt (Stephane Michel 8 minutes ago)
* 332bd59 - Commit pour les tests (Stephane Michel 15 minutes ago)
* 51b4826 - Modif dans file2.txt (Stephane Michel 2 hours ago)
* 1b58269 - Modif dans toto.txt (Stephane Michel 2 hours ago)
* ded5ea6 - Suppression du fichier file2.txt (Stephane Michel 4 hours ago)
* a57bdbb - new commit (Stephane Michel 4 hours ago)
* dfeb1ac - (tag: TAG1) Ajout de toto.txt (Stephane Michel 4 hours ago)
* 857c44e - Premiere modif (Stephane Michel 4 hours ago)
* a750e4b - Ajout de Test.java (Stephane Michel 4 hours ago)
* 3826cdb - initial commit (Stephane Michel 5 hours ago)
```

Lire reset3.txt

- Ajout d'une ligne dans reset3.txt
- git reset --hard d35be5b
- Le fichier reset3.txt a été remis dans l'état du commit **d35be5b**. La ligne ajoutée a disparue.

Exercice : reset --merge

- Création et commit d'un fichier reset4.txt
 - git add reset4.txt
 - git commit -m « Ajout reset4.txt »
- git lg

```
$ git lg
* d35be5b - (HEAD -> master) Ajout de reset4.txt (Stephane Michel 8 seconds ago)
* 332bd59 - Commit pour les tests (Stephane Michel 7 minutes ago)
* 51b4826 - Modif dans file2.txt (Stephane Michel 2 hours ago)
* 1b58269 - Modif dans toto.txt (Stephane Michel 2 hours ago)
* ded5ea6 - Suppression du fichier file2.txt (Stephane Michel 4 hours ago)
* a57bdbb - new commit (Stephane Michel 4 hours ago)
* dffeb1ac - (tag: TAG1) Ajout de toto.txt (Stephane Michel 4 hours ago)
* 857c44e - Premiere modif (Stephane Michel 4 hours ago)
* a750e4b - Ajout de Test.java (Stephane Michel 4 hours ago)
* 3826cdb - initial commit (Stephane Michel 5 hours ago)
```

- Ajout d'une ligne dans le fichier reset4.txt
- git reset --hard d35be5b
- Les modifications du fichier reset3.txt ont été conservées. La ligne ajoutée est toujours présente.

Soucis classiques et solutions optimales

- **Et voilà! j'ai paumé mon commit !**
 - Virtuellement impossible
 - **git reflog**
 - Enregistre la séquence complète des positions successives du HEAD (très pratique lorsqu'on gaufre ses resets...)
 - « **Strictement local** » (tout commit y figurant qui n'est dans aucun log public reste sur votre machine)
 - **Quand est-ce purgé ?**
 - Lors d'un GC (manuel ou auto)
 - En obéissant aux réglages `gc.reflogExpire` (60 jours par défaut) et `gc.reflogExpireUnreachable` (90 jours par défaut)

Exemple

```
$ git reflog
d35be5b HEAD@{0}: reset: moving to d35be5b
d35be5b HEAD@{1}: reset: moving to d35be5b
d35be5b HEAD@{2}: reset: moving to d35be5b
d35be5b HEAD@{3}: reset: moving to HEAD@{1}
332bd59 HEAD@{4}: reset: moving to 332bd59
d35be5b HEAD@{5}: commit: Ajout de reset4.txt
332bd59 HEAD@{6}: reset: moving to 332bd59
ee937d6 HEAD@{7}: commit: Ajout reset3.txt
332bd59 HEAD@{8}: commit: Commit pour les tests
51b4826 HEAD@{9}: reset: moving to 51b4826
f15efba HEAD@{10}: commit: Ajout reset2.txt
51b4826 HEAD@{11}: reset: moving to 51b4826
009f844 HEAD@{12}: reset: moving to 009f844
009f844 HEAD@{13}: reset: moving to HEAD
009f844 HEAD@{14}: reset: moving to 009f844
009f844 HEAD@{15}: reset: moving to 009f844
009f844 HEAD@{16}: reset: moving to HEAD
009f844 HEAD@{17}: commit: Ajout fichier reset.txt
51b4826 HEAD@{18}: commit (amend): Modif dans file2.txt
123bbd7 HEAD@{19}: commit (amend): Modif dans file2.txt
1ce3b33 HEAD@{20}: checkout: moving from Test1 to master
3db6b5f HEAD@{21}: commit: Modif de test.java
a750e4b HEAD@{22}: checkout: moving from master to Test1
1ce3b33 HEAD@{23}: commit: Modif dans file2.txt
1b58269 HEAD@{24}: commit: Modif dans toto.txt
ded5ea6 HEAD@{25}: commit: Suppression du fichier file2.txt
a57bdbb HEAD@{26}: commit: new commit
dfeb1ac HEAD@{27}: reset: moving to HEAD
dfeb1ac HEAD@{28}: reset: moving to HEAD^
2091e43 HEAD@{29}: reset: moving to HEAD^
e2cb852 HEAD@{30}: commit: Ajout File2
2091e43 HEAD@{31}: reset: moving to HEAD
2091e43 HEAD@{32}: reset: moving to HEAD
2091e43 HEAD@{33}: revert: Revert "Premiere modif"
dfeb1ac HEAD@{34}: reset: moving to HEAD
dfeb1ac HEAD@{35}: reset: moving to HEAD
dfeb1ac HEAD@{36}: commit: Ajout de toto.txt
857c44e HEAD@{37}: commit: Premiere modif
a750e4b HEAD@{38}: commit: Ajout de Test.java
3826cdb HEAD@{39}: commit (initial): initial commit
```

git reflog

- Permet d'afficher l'historique des positions du HEAD, puis, à l'aide d'un git reset de revenir à des positions précédentes (et donc revenir à des états précédents pour annuler une modification par exemple).
- `@{date-spec}` : commit le plus récent du *reflog* qui est au moins aussi vieux que date-spec.
 - Par défaut le `HEAD`, mais on a un *reflog* par branche locale aussi.
 - `"@{last Thursday 10am}"`, `"@{2 weeks ago}"`...
- `@{n}` : position absolue dans le *reflog*
 - `HEAD@{1}` : position précédente du `HEAD`
 - `feature@{1}` : position précédente de la pointe de branche feature.
- `@{-1}` : branche active précédente
 - Filtre le *reflog* à la recherche des *checkouts* sur pointes de branches.
 - Généralement abrégable en `"-"` (*checkout*, *merge*, *rebase*), tout comme la commande de shell `"cd -"`.

Soucis classiques et solutions optimales

- **J'ai fait un reset foireux !**
 - Pas de panique, le *reflog* te donne les positions précédentes...
 - Aurais-tu fait à tort un `git reset --hard HEAD~3` ?
 - `git reset --hard HEAD@{1}` te remet sur les rails.

Exemple : reflog 1/2

- Création et commit d'un fichier reflog1.txt

```
$ git lg
* fdbd64b - (HEAD -> master) Ajout du fichier reflog1.txt (Stephane Michel 2 seconds ago)
* d35be5b - Ajout de reset4.txt (Stephane Michel 17 minutes ago)
* 332bd59 - Commit pour les tests (Stephane Michel 24 minutes ago)
* 51b4826 - Modif dans file2.txt (Stephane Michel 2 hours ago)
* 1b58269 - Modif dans toto.txt (Stephane Michel 2 hours ago)
* ded5ea6 - Suppression du fichier file2.txt (Stephane Michel 4 hours ago)
* a57bdbb - new commit (Stephane Michel 4 hours ago)
* dfeb1ac - (tag: TAG1) Ajout de toto.txt (Stephane Michel 4 hours ago)
* 857c44e - Premiere modif (Stephane Michel 4 hours ago)
* a750e4b - Ajout de Test.java (Stephane Michel 4 hours ago)
* 3826cdb - initial commit (Stephane Michel 5 hours ago)
```

- Annulation du dernier commit

- git reset --hard d35be5b

```
$ git lg
* d35be5b - (HEAD -> master) Ajout de reset4.txt (Stephane Michel 19 minutes ago)
* 332bd59 - Commit pour les tests (Stephane Michel 26 minutes ago)
* 51b4826 - Modif dans file2.txt (Stephane Michel 2 hours ago)
* 1b58269 - Modif dans toto.txt (Stephane Michel 2 hours ago)
* ded5ea6 - Suppression du fichier file2.txt (Stephane Michel 4 hours ago)
* a57bdbb - new commit (Stephane Michel 4 hours ago)
* dfeb1ac - (tag: TAG1) Ajout de toto.txt (Stephane Michel 4 hours ago)
* 857c44e - Premiere modif (Stephane Michel 4 hours ago)
* a750e4b - Ajout de Test.java (Stephane Michel 4 hours ago)
* 3826cdb - initial commit (Stephane Michel 5 hours ago)
```

- Et voilà ! J'ai paumé mon fichier reflog1.txt

Exemple : reflog 2/2

- git reflog

```
$ git reflog
d35be5b HEAD@{0}: reset: moving to d35be5b
fdbd64b HEAD@{1}: commit: Ajout du fichier reflog1.txt
d35be5b HEAD@{2}: reset: moving to d35be5b
d35be5b HEAD@{3}: reset: moving to d35be5b
d35be5b HEAD@{4}: reset: moving to d35be5b
d35be5b HEAD@{5}: reset: moving to d35be5b
```

- git reset --hard HEAD@{1}

- git lg

```
$ git lg
* fdbd64b - (HEAD -> master) Ajout du fichier reflog1.txt (Stephane Michel 5 minutes ago)
* d35be5b - Ajout de reset4.txt (Stephane Michel 22 minutes ago)
* 332bd59 - Commit pour les tests (Stephane Michel 29 minutes ago)
* 51b4826 - Modif dans file2.txt (Stephane Michel 2 hours ago)
* 1b58269 - Modif dans toto.txt (Stephane Michel 2 hours ago)
* ded5ea6 - Suppression du fichier file2.txt (Stephane Michel 4 hours ago)
```

- Mon commit avec le fichier reflog1.txt est revenu... ouf !

Soucis classiques et solutions optimales

- **J'ai commité un truc en trop** (autre méthode) !
 - Si ça vient d'arriver, il suffit de revenir en arrière avec un reset et de refaire le bon commit
 - Fais d'abord un `git rm --cached` puis un `git commit --amend...`
 - ...ou manuellement avec un `git reset --soft`

Soucis classiques et solutions optimales

- **J'ai commisé un truc en trop... y'a un bail !**
 - Tout dépend du niveau de confidentialité...
 - Fichier confidentiel qui ne doit en fait figurer nulle part dans l'historique : il va falloir réécrire l'histoire
 - L'outil qui torche en perfs, notamment sur de gros dépôts/historiques : [The BFG Repo Cleaner](#).
 - Sinon une modif du commit fautif et un rebase par-dessus...
 - Dans tous les cas, on flingue les SHA-1, il faudra faire quelques forçages sur les pushes, fetches/pulls, etc.
 - Fichier non sensible, il faut juste le virer...
 - Juste un commit normal après un `rm --cached` et un `.gitignore`.
 - Suppression de tous les fichiers ignorés par le `.gitignore` (permet de remettre tout d'équerre en fonction du `.gitignore`)
 - `git rm --cached `git ls-files -i -X .gitignore``
(attention au quote qui sont des « antiquote »)

git rev-list

- Permet d'afficher l'historique de commit d'une branche ou d'un fichier donné
 - `git rev-list HEAD` : Liste des commit de la branche courante
 - `git rev-list HEAD <file_path>` : Liste des commit de la branche courante qui concernent le fichier donné en paramètre.
 - `git rev-list -n 1 HEAD <file_path>` : permet de retrouver le dernier commit de la branche courante qui concernent le fichier donné en paramètre.

Soucis classiques et solutions optimales

- **J'ai supprimé un fichier à tord il y a un bail...**
 - Retrouver l'historique de commit du fichier :
 - `git rev-list -n 1 HEAD -- <file_path>`
 - Récupérer la version souhaitée :
 - `git checkout <delete-commit-SHA1>^ -- <file_path>`

Soucis classiques et solutions optimales

- **J'ai oublié de configurer mon nom...**
 - Tu *pourrais* le définir une bonne fois pour toutes :
 - `git config --global user.name "mon nom"`
 - `git config --global user.email "monEmail@dassault-aviation.com"`
 - Puis ajuster ton dernier commit :
 - `git commit --amend --reset-author`
 - Mais si ce n'est *pas ta machine*, tu devrais laisser l'identité tranquille et en utiliser une ponctuelle :
 - `git commit --author= 'John Smith <john@example.com>'`
 - `git commit --author=john`

git clean

- Permet de faire du nettoyage
 - git clean à la rescousse
 - Supprime les fichiers non trackés/non versionnés du WD
 - -d : suppression des répertoires non trackés
 - -f / --force : nécessaire pour forcer la suppression (sauf si `clean.requireForce = true`)
 - -n / --dry-run : liste les items à supprimer sans exécuter le purge
 - -e / --exclude : pattern d'exclusion pour éviter de supprimer certains fichiers/répertoires
 - -X : supprimer uniquement les fichiers ciblés par les patterns du `.gitignore`

Exemple : `clean`

- Création d'un fichier `fic.txt`
- Force la remise en état du working directory
 - `git clean -f`
- Le fichier `fic.txt` a été supprimé (non récupérable!)

An aerial photograph of a large railway interchange. Numerous tracks of varying gauges converge and diverge, creating a complex web of lines across the landscape. Shipping containers are stacked in organized rows along the tracks, indicating a hub for international trade. The surrounding area appears to be a mix of industrial land and some greenery.

Branches, stash et checkout

git stash

- **Permet de mettre temporairement de côté des modifications !**
 - `git stash` : créé un stash avec nom par défaut
 - `git stash save "mon_stash"` : message explicite
 - `git stash list` : liste des stash
 - `git stash show [<stash>]`: donne le détail d'un stash
 - `git stash save -u` : inclut les untracked...
 - `git stash pop` : tente de restaurer le stage en tant que tel à partir du dernier stash réalisé
 - `git stash pop --index` : tente de restaurer le stage en tant que tel
 - `git stash branch` : créé une nouvelle branche à partir d'un stash
 - `git stash drop [<stash>]`: supprime un stash
- **Également utile pour contourner une prudence parfois conservatrice**
 - Refus de merge en raison de fichiers modifiés localement (staged ou dirty) concernés par la fusion, alors qu'on sait pertinemment qu'on n'aura pas de conflit
 - Dans ce cas on souhaite mettre de côté ses modifs, récupérer le code distant (sans faire de merge) puis appliquer à nouveau son stash pour retrouver les modifications qui étaient en cours
 - `git stash save "Mon stash"` (ou `git stash`)
 - `git merge` ou `git pull`
 - `git stash pop --index` (ou `stash apply --index`) (ou `git stash pop` si on veut le dernier)

Practice Time

Practice makes perfect!

Exemple : git stash 1/5

- J'ai commencé une modif mais je dois vite me remettre dans le dernier état du repo pour corriger un truc en urgence !
- Dans l'exemple j'ai commencé à travailler sur fic2

Ajout contenu dans fic2
Ajout d'une super nouvelle fonction

```
D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git l
* 2ff2d78 - (HEAD -> master) Ajout contenu dans fic2 (Stephane Michel 3 years, 7 months ago)
* 5502a19 - fic2 (Stephane Michel 3 years, 7 months ago)
* 03a1048 - Modif1 (Stephane Michel 3 years, 7 months ago)
* cb60533 - fic1 (Stephane Michel 3 years, 7 months ago)

D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git st
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:   fic2

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

Exemple : git stash 2/5

- Avec un `git stash` je mets de côté toutes mes modifications par rapport au Working Directory.

```
D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git stash
warning: LF will be replaced by CRLF in fic2.
The file will have its original Line endings in your working directory
Saved working directory and index state WIP on master: 2ff2d78 Ajout contenu dans fic2
```

- Le stash est visible dans le log :

```
D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git l
* 0552038 - (refs/stash) WIP on master: 2ff2d78 Ajout contenu dans fic2 (Stephane Michel 31 seconds ago)
|\_
| * 454ec15 - index on master: 2ff2d78 Ajout contenu dans fic2 (Stephane Michel 31 seconds ago)
|/
* 2ff2d78 - (HEAD -> master) Ajout contenu dans fic2 (Stephane Michel 3 years, 7 months ago)
* 5502a19 - fic2 (Stephane Michel 3 years, 7 months ago)
* 03a1048 - Modif1 (Stephane Michel 3 years, 7 months ago)
* cb60533 - fic1 (Stephane Michel 3 years, 7 months ago)
```

Exemple : git stash 3/5

- Ensuite je peux faire ma correction urgente (fic2 ne contient plus les modifications que j'avais commencé à faire).

```
D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ cat fic2
Ajout contenu dans fic2
Correction urgente

D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git st
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:   fic2

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

Exemple : git stash 4/5

- Ensuite je « commit » ma correction urgente :

```
D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git add fic2

D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git commit -m "Correction urgente"
[master 76c23c5] Correction urgente
 1 file changed, 1 insertion(+)

D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git l
* 76c23c5 - (HEAD -> master) Correction urgente (Stephane Michel 10 seconds ago)
| * 0552038 - (refs/stash) WIP on master: 2ff2d78 Ajout contenu dans fic2 (Stephane Michel 7 minutes ago)
| |
|/
| *
| * 454ec15 - index on master: 2ff2d78 Ajout contenu dans fic2 (Stephane Michel 7 minutes ago)
|/
* 2ff2d78 - Ajout contenu dans fic2 (Stephane Michel 3 years, 7 months ago)
* 5502a19 - fic2 (Stephane Michel 3 years, 7 months ago)
* 03a1048 - Modif1 (Stephane Michel 3 years, 7 months ago)
* cb60533 - fic1 (Stephane Michel 3 years, 7 months ago)
```

Exemple : git stash 5/5

- Et je peux reprendre mes modifications où je les avais laissées avec la commande `git stash pop` :

```
D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git stash pop
Auto-merging fic2
CONFLICT (content): Merge conflict in fic2
Recorded preimage for 'fic2'
```

- Comme fic2 a été modifié par ma correction urgente, j'ai un conflit à résoudre :

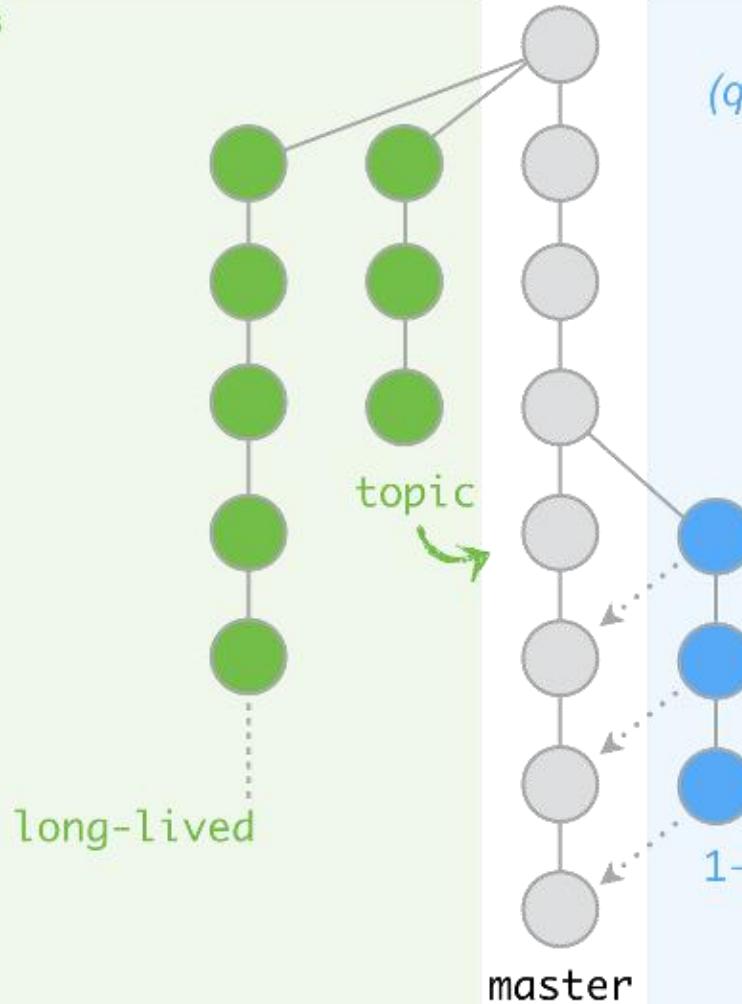
```
Ajout contenu dans fic2
<<<<< Updated upstream
Correction urgente
|||||| merged common ancestors
=====
Ajout d'une super nouvelle fonction
>>>>> Stashed changes
```

Branches et fusions

- Définitions
 - De simples « étiquettes »
 - C'est un historique parallèle aux autres, qu'on peut faire évoluer, donc, sans gêner les autres
 - Grande liberté de manœuvre
 - Nécessitera à terme une réconciliation entre branches : on parle alors de **fusion**
 - Git a le **cheap branching** : c'est instantané et ne coûte pratiquement rien en mémoire/disque, alors **ne surtout pas se priver de faire des branches !**
 - Ta prochaine tâche peut prendre plus de 5 minutes ? **Fais une branche !**

Branches et fusions

Branches thématiques
(topic / feature)



Branches de release
(que des fixes, jamais remergées mais cherry-picked au fil de l'eau)

v1.2.0
v1.2.1
1-2-stable

git branch

- Liste des branches
 - locales par défaut
 - distantes (-r)
 - les deux (-a)
 - plus d'infos (-v, -vv)
 - filtrage par globs (--list)
- Créer une branche locale
 - `git branch ta-branche` [base, HEAD par défaut]
- Supprimer une branche locale
 - `git branch -d ta-branche`
 - si non fusionnée
 - `git branch -d -f ta-branche`
 - ou `git branch -D ta-branche`
- *Supprimer une branche distance*
 - `git push <remote-origin> --delete <branch-name>`

git checkout 1/2

- Techniquement, sert à **refléter un commit dans le WD**
- Si on lui passe un **nom de branche**, la rend **active**
`git checkout maBranche`
`git checkout -` : Revient sur la branche précédente
- Pas que des noms de branche, ceci dit
 - N'importe quel commit possible... mais donne une tête détachée !
`git checkout 2f67531bc`
- Pas que des commits entiers, d'ailleurs
 - **Vous pouvez passer des chemins de fichiers précis, et juste récupérer ces fichiers, sans changer la branche active ou même le HEAD.**
`git checkout src/README.md`

git checkout 2/2

- Raccourci classique de création-et-activation de branche
 - `git checkout -b nouvelle-branche` (équivaut à `git branch nouvelle-branche` puis `git checkout nouvelle-branche`).
- Des modifications dans votre WD s'interposent ?
 - `git checkout -m` ou `git checkout --merge`
- Lors des conflits (merge, rebase, cherry-pick), possibilité de conserver la version locale (`--ours`) ou la version distante (`--theirs`) sans passer par `git mergetool`.
 - `git checkout --theirs mon_fichier`
 - `git checkout --ours mon_fichier`

Practice Time

Practice makes perfect!

Exemple : branch et checkout 1/2

- Création d'une nouvelle branche à la position courante
 - git branch RELEASE1
 - git lg

```
$ git lg
* fdbd64b - (HEAD -> master, RELEASE1) Ajout du fichier reflog1.txt (Stephane Michel 21 minutes ago)
* d35be5b - Ajout de reset4.txt (Stephane Michel 38 minutes ago)
* 332bd59 - Commit pour les tests (Stephane Michel 45 minutes ago)
* 5114026 - Modification fichier.txt (Stephane Michel 2 hours ago)
```

- Positionnement sur la branche RELEASE1
 - git checkout RELEASE1
 - git lg

```
$ git lg
* fdbd64b - (HEAD -> RELEASE1, master) Ajout du fichier reflog1.txt (Stephane Michel 22 minutes ago)
* d35be5b - Ajout de reset4.txt (Stephane Michel 39 minutes ago)
* 332bd59 - Commit pour les tests (Stephane Michel 46 minutes ago)
* 5114026 - Modification fichier.txt (Stephane Michel 2 hours ago)
```

- Commit d'un nouveau fichier : newfile.txt
 - git lg

```
$ git lg
* b7bd447 - (HEAD -> RELEASE1) Ajout fichier newFile.txt (Stephane Michel 5 seconds ago)
* fdbd64b - (master) Ajout du fichier reflog1.txt (Stephane Michel 24 minutes ago)
* d35be5b - Ajout de reset4.txt (Stephane Michel 41 minutes ago)
* 332bd59 - Commit pour les tests (Stephane Michel 48 minutes ago)
```

Exemple : branch et checkout 2/2

- Retour dans la branche master
 - git checkout master

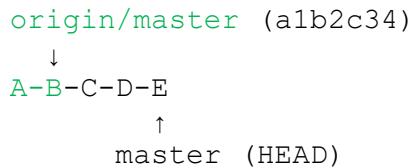
```
$ git lg
* fdbd64b - (HEAD -> master) Ajout du fichier reflog1.txt (Stephane Michel 26 minutes ago)
* d35be5b - Ajout de reset4.txt (Stephane Michel 43 minutes ago)
* 332bd59 - Commit pour les tests (Stephane Michel 50 minutes ago)
```

- Le fichier newfile.txt n'est plus présent
- Retour dans la branche RELEASE1
 - git checkout RELEASE1
 - Le fichier newfile.txt est bien toujours là

Soucis classiques et solutions optimales

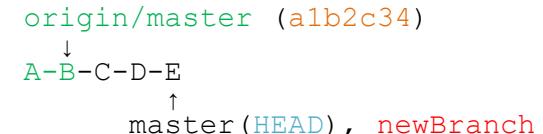
- **J'ai oublié de créer une branche**

Je me suis synchronisé avec le dépôt distant puis j'ai fait des dev (commits) sur master alors qu'il aurait fallu que je les fasse dans une branche distincte (pour pouvoir les pousser sans gêner tous le monde).



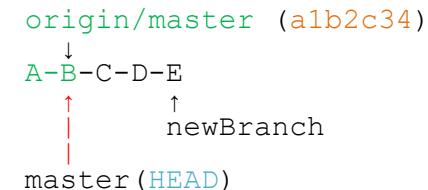
- Crédit d'une nouvelle branche :

- `git branch newBranch`



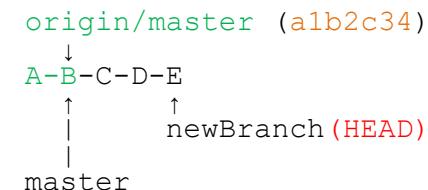
- Déplacer le master sur l'ancêtre souhaité :

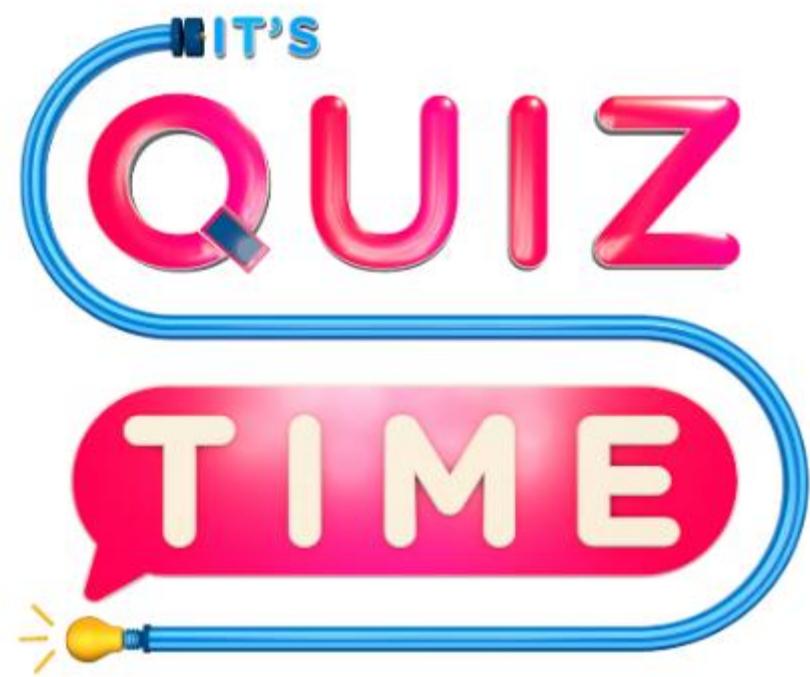
- `git reset --hard a1b2c34`



- Repositionnement sur la nouvelle branche :

- `git checkout newBranch`





Que réalise la commande suivante ?

```
$ git checkout fic2
```

```
D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ cat fic2
Ajout contenu dans fic2
Correction urgente

D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ vi fic2

D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ cat fic2
Ajout contenu dans fic2
Correction urgente
Nouvel ajout

D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git st
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:   fic2

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

D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git checkout fic2

D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ cat fic2
Ajout contenu dans fic2
Correction urgente
```

« Annule les modifications locales de fic2 et le remet dans l'état du repository (HEAD) » 99

Que réalise la commande suivante ?

```
$ git branch maBranche
```

```
D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git l
* 76c23c5 - (HEAD -> master) Correction urgente (Stephane Michel 41 minutes ago)
* 2ff2d78 - Ajout contenu dans fic2 (Stephane Michel 3 years, 7 months ago)
* 5502a19 - fic2 (Stephane Michel 3 years, 7 months ago)
* 03a1048 - Modif1 (Stephane Michel 3 years, 7 months ago)
* cb60533 - fic1 (Stephane Michel 3 years, 7 months ago)

D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git branch maBranche

D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git l
* 76c23c5 - (HEAD -> master, maBranche) Correction urgente (Stephane Michel 41 minutes ago)
* 2ff2d78 - Ajout contenu dans fic2 (Stephane Michel 3 years, 7 months ago)
* 5502a19 - fic2 (Stephane Michel 3 years, 7 months ago)
* 03a1048 - Modif1 (Stephane Michel 3 years, 7 months ago)
* cb60533 - fic1 (Stephane Michel 3 years, 7 months ago)
```

« Crée une nouvelle branche (mais ne se positionne pas dessus) »

Que réalise la commande suivante ?

```
$ git checkout maBranche
```

```
D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git l
* 76c23c5 - (HEAD -> master, maBranche) Correction urgente (Stephane Michel 41 minutes ago)
* 2FF2d78 - Ajout contenu dans fic2 (Stephane Michel 3 years, 7 months ago)
* 5502a19 - fic2 (Stephane Michel 3 years, 7 months ago)
* 03a1048 - Modif1 (Stephane Michel 3 years, 7 months ago)
* cb60533 - fic1 (Stephane Michel 3 years, 7 months ago)

D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git checkout maBranche
Switched to branch 'maBranche'

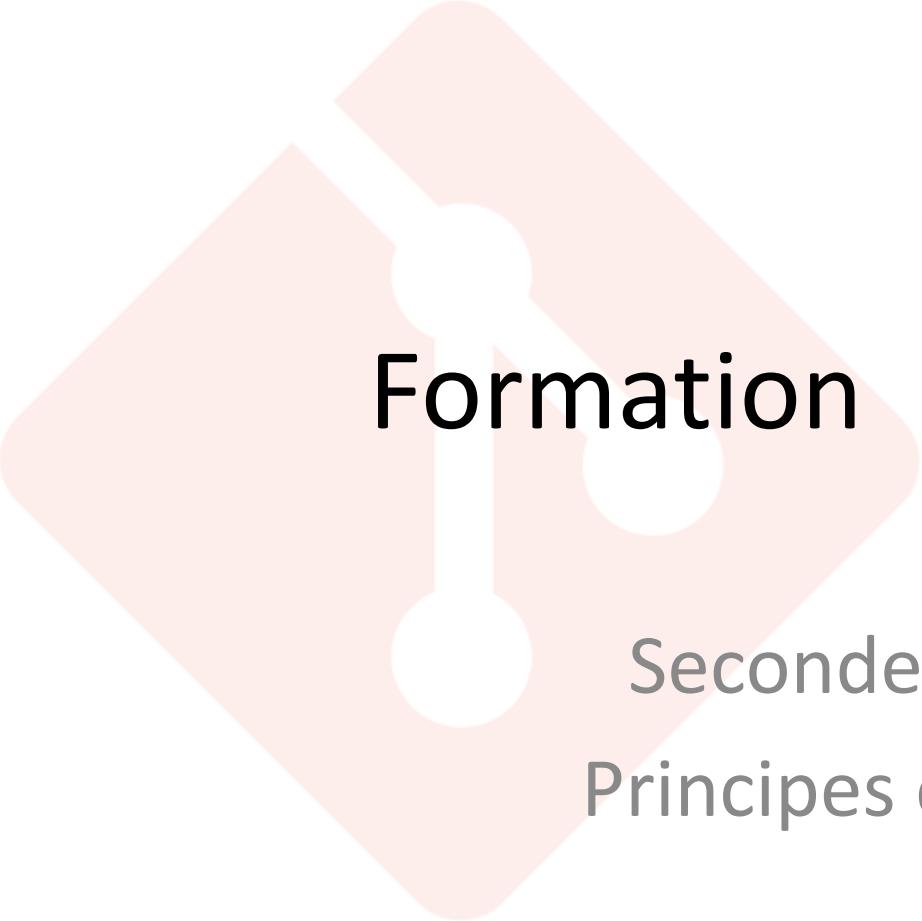
D106130@BUU00002 MINGW64 /C/temp/amend (maBranche)
$ git l
* 76c23c5 - (HEAD -> maBranche, master) Correction urgente (Stephane Michel 43 minutes ago)
* 2FF2d78 - Ajout contenu dans fic2 (Stephane Michel 3 years, 7 months ago)
* 5502a19 - fic2 (Stephane Michel 3 years, 7 months ago)
* 03a1048 - Modif1 (Stephane Michel 3 years, 7 months ago)
* cb60533 - fic1 (Stephane Michel 3 years, 7 months ago)
```

« Ce positionne dans la branche maBranche»



Fin première partie

PAUSE DEJEUNER



Formation



git

Seconde partie

Principes étendus

Plan : seconde partie

- Merges et rebases
 - Commit de merge/fusion, Fast forward, Tête détachée, pourquoi et comment soigner son historique, etc.
- Cherry-picking
- Dépôts distants
 - Push, pull, fetch, comment annuler un push
- Soucis classiques et solutions optimales sur dépôt distant
 - Mon git pull me dit « You have unstaged changes » !
 - Mon git push me dit « [rejected]...
 - Je voudrai créer une copie de mon repos sur un média USB local
- (Rerere : optimiser les merges)** Si on a le temps
- (Chasse aux bugs accélérées :Le bisecting)*
- Les Workflows
 - Centralized workflow, Feature Branch workflow, Git Flow workflow, Forking workflow

An aerial photograph showing a complex network of railway tracks forming a grid pattern. Numerous shipping containers, in various colors like white, blue, and yellow, are stacked along the tracks. The perspective is from above, looking down the length of the tracks.

Merges, et rebases

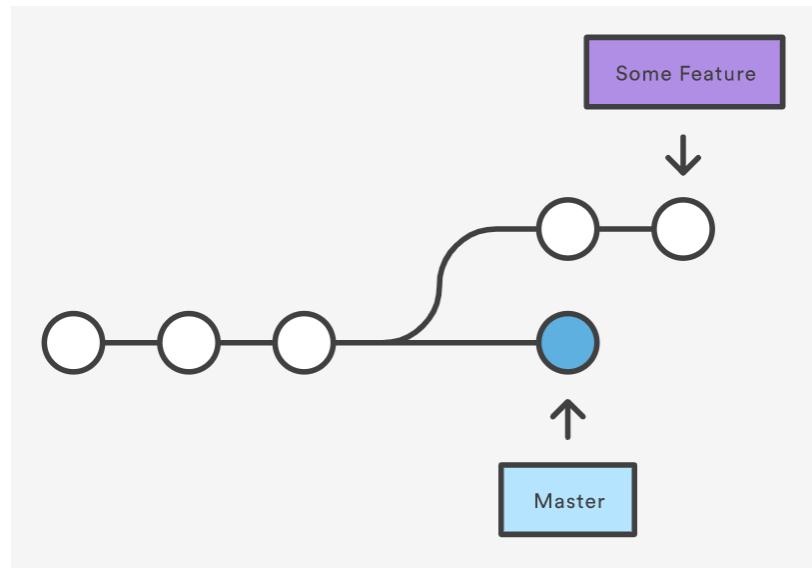
git merge

- Amène l'historique d'une branche tierce **dans la branche courante**
- Si les deux avaient divergé, crée un commit de fusion/merge (2 parents)
 - Message de base simple et plutôt utile
 - `git merge --log` le rend encore plus utile
 - On peut bien sûr définir son propre message (`-m`, `--edit`)
- De nombreux scénarios avancés existent
 - Pas de commit finalisé (vérification manuelle d'abord)
 - *Squash commits*
 - Stratégies de fusion
 - Et plein d'autres trucs qui sortent du cadre de cette formation, mais n'hésitez pas à explorer `git help merge` !

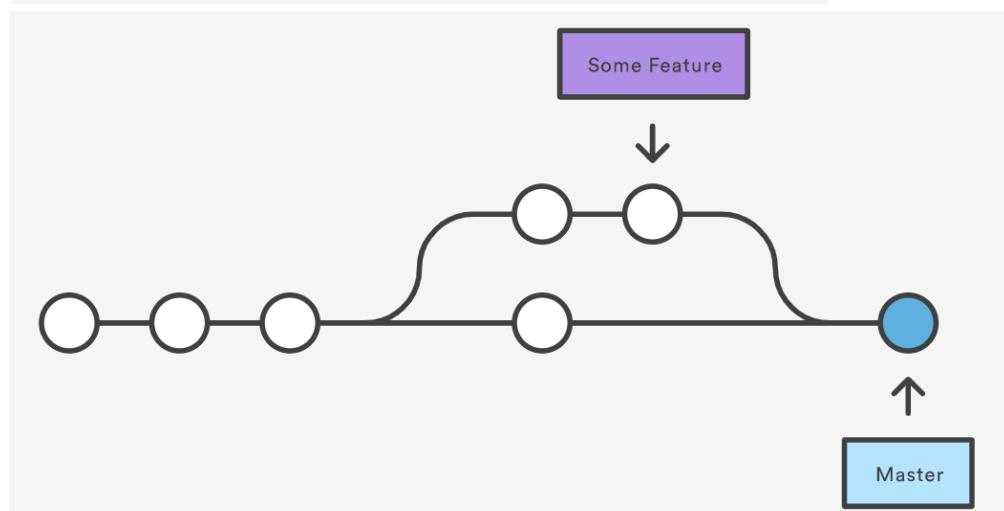
Voir <https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging>
et <https://www.atlassian.com/git/tutorials/using-branches/git-merge>
pour plus de détails

Exemple : commit de fusion/merge

Avant



Après merge (avec
commit de merge)

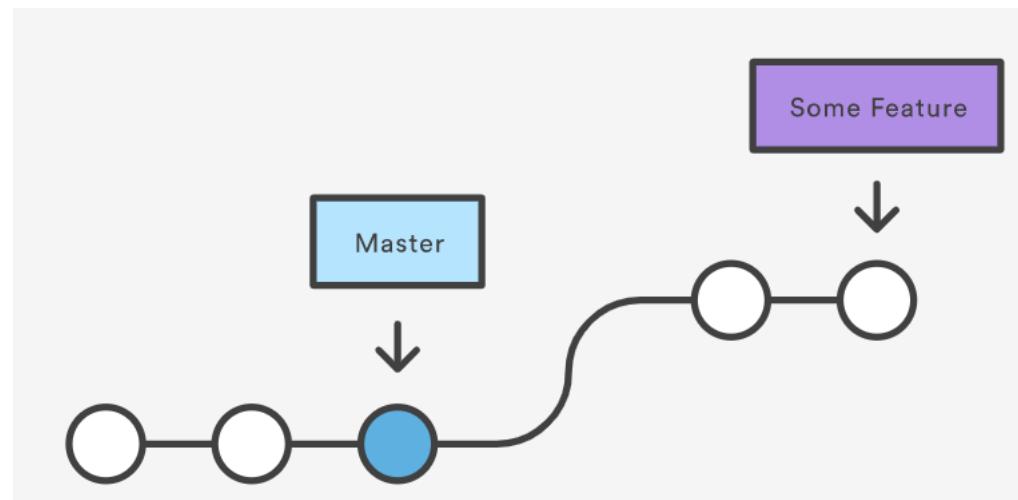


git merge : le fast-forward

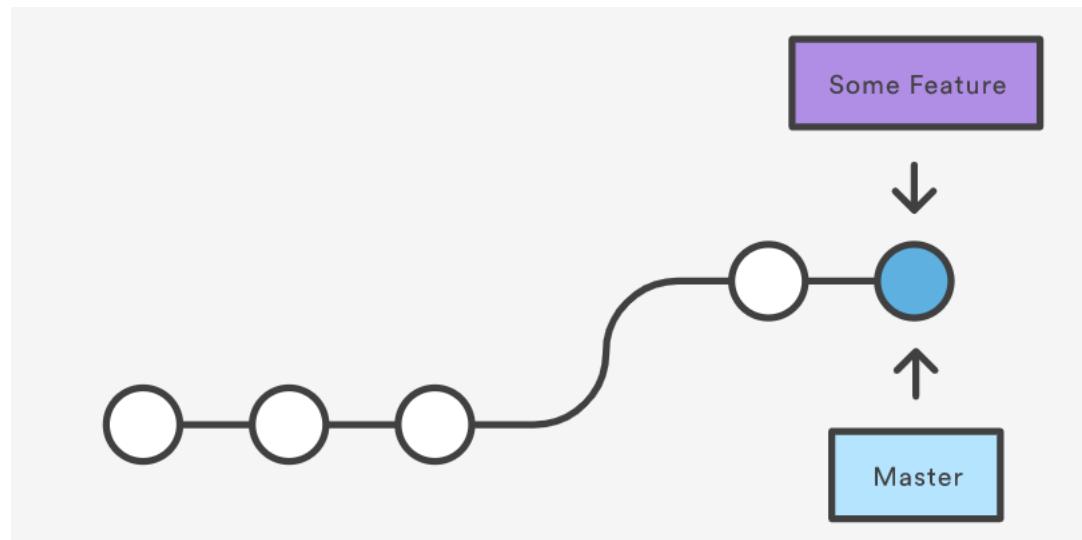
- **Les « fusions » en fast-forward**
 - Le fast-forward (ff) est un **mécanisme qui « aplanit » l'historique** (et le rend du coup plus lisible).
De cette manière on ne conserve pas 2 parents et on économise un commit dédié à la fusion dans notre historique.
 - Techniquement ça revient à déplacer l'étiquette de la branche vers laquelle on souhaite fusionner.
 - Lors des fusions (merge), git utilise le fast-forward **s'il n'y a pas de conflit, sinon il fait un commit de merge.**
 - Si l'on souhaite forcer la création d'un commit de merge, on utilise le mode --no-ff

Exemple fast-forward

Avant



Après fast-forward



git merge

- Nécessite un WD « in good order »
 - En gros, **sans fichier modifié / staged** qui soit concerné par la fusion
- Idéalement, la branche a des commits atomiques (facilite les heuristiques de fusion, donc réduit les faux positifs)
- Quelques options utiles :
 - squash (prépare la fusion mais ne fait pas le commit)
 - log / -m / --edit
 - no-ff (évite le fast-forward, opposée de -ff)
 - s *stratégie* -X *option-de-stratégie* (*ours*, *theirs*...)

Exemple : --no-ff

Avant

```
D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git l
* 5626b42 - (maBranche) Modif fic3 dans MaBranche (Stephane Michel 38 minutes ago)
* 76c23c5 - (HEAD -> master) Correction urgente (Stephane Michel 16 hours ago)
* 2ff2d78 - Ajout contenu dans fic2 (Stephane Michel 3 years, 7 months ago)
* 5502a19 - fic2 (Stephane Michel 3 years, 7 months ago)
* 03a1048 - Modif1 (Stephane Michel 3 years, 7 months ago)
* cb60533 - fic1 (Stephane Michel 3 years, 7 months ago)
```

Après merge (on a
forcé un commit de
merge)

```
D106130@BUU00002 MINGW64 /C/temp/amend (master)
$ git merge --no-ff maBranche
Merge made by the 'recursive' strategy.
  fic3 | 1 +
  1 file changed, 1 insertion(+)
  create mode 100644 fic3
```

```
$ git l
*   eadd0a1 - (HEAD -> master) Merge branch 'maBranche' (Stephane Michel 7 seconds ago)
|\ 
| * 5626b42 - (maBranche) Modif fic3 dans MaBranche (Stephane Michel 38 minutes ago)
|/
* 76c23c5 - Correction urgente (Stephane Michel 16 hours ago)
* 2ff2d78 - Ajout contenu dans fic2 (Stephane Michel 3 years, 7 months ago)
* 5502a19 - fic2 (Stephane Michel 3 years, 7 months ago)
* 03a1048 - Modif1 (Stephane Michel 3 years, 7 months ago)
* cb60533 - fic1 (Stephane Michel 3 years, 7 months ago)
```

git merge : merge manuel

- Si git n'arrive pas à merger tout seul, il donne la main (`git status` donne la liste des fichiers à merger)
- Le contenu des fichiers
 - Marqueurs de conflits : `>>>` / `<<<` / `==` et `|||` (mode *diff3*)
- `git merge` / `git mergetool`
- Lors des conflits (merge, rebase, cherry-pick), possibilité de conserver la version locale (`--ours`) ou la version distante (`--theirs`) sans passer par `git mergetool`.
 - `git checkout --theirs mon_fichier`
 - `git checkout --ours mon_fichier`
- `git log --merge -p fichier` : Liste les « commits uniques » sur ce fichier, de part et d'autre de la divergence, avec leurs diffs individuels
- `git show :1:... / :2:... / :3:...` : Affiche la version intégrale du fichier conflictuel dans l'ancêtre commun (la merge base en jargon Git), ou sur chaque pointe de branche (plus de contexte !)

git merge : merge manuel

- Édition manuelle dans l'éditeur
 - Parfois la seule façon (pas d'outil de fusion disponible sur la machine)
- À l'aide d'outils externes dédiés : **git mergetool**
 - Configurez `merge.tool` (voyons ça avec p4merge!), `mergetool.*`
 - Résolution automatique de nombreux cas « faciles » fréquents.
 - Prise en charge auto d'un nombre grandissant d'outils pour tous OS et EDIs
 - On peut toujours configurer ses outils maison, si besoin
- N'oubliez pas de ***stager* le fichier pour le marquer résolu !**

Practice Time

Practice makes perfect!

Exemple : merge

1/3

- Environnement :

```
$ git lg --all
* 58ac65d - (HEAD -> RELEASE1) Fichier spA@cifique release1 (Stephane Michel 5 seconds ago)
* 3237c15 - Ajout fichier fic2.txt dans RELEASE1 (Stephane Michel 6 minutes ago)
| * 1c178f8 - (master) Ajout fichier spA@cifique master (Stephane Michel 61 seconds ago)
| * 62e0316 - Ajout fic2.txt dans master (Stephane Michel 5 minutes ago)
|/
* 4ccd152 - Ajout fic1.txt (Stephane Michel 9 minutes ago)
* 9be974e - initial (Stephane Michel 10 minutes ago)
```

- 2 branches avec chacune un fichier identique (fic1.txt) un fichier de même nom mais de contenu différent (fic2.txt), et chacune un fichier spécifique :

- Arbo master:

```
smichel@P4800-MICHEL MINGW64 /d/formations/git/TP-MERGE (master)
$ ll
total 3
-rw-r--r-- 1 smichel 1055235 54 Jan 16 15:28 fic1.txt
-rw-r--r-- 1 smichel 1055235 12 Jan 16 15:37 fic2.txt
-rw-r--r-- 1 smichel 1055235 27 Jan 16 15:37 specific-master.txt
```

- Arbo RELEASE1 :

```
smichel@P4800-MICHEL MINGW64 /d/formations/git/TP-MERGE (RELEASE1)
$ ll
total 3
-rw-r--r-- 1 smichel 1055235 54 Jan 16 15:28 fic1.txt
-rw-r--r-- 1 smichel 1055235 13 Jan 16 15:36 fic2.txt
-rw-r--r-- 1 smichel 1055235 28 Jan 16 15:36 specific-release.txt
```

- On souhaite « merger » la branche RELEASE1 dans la branche master

Exemple : merge

2/3

- Positionnement sur la branche **réceptrice** du merge (master ici) :
 - git checkout master
- Merge sans commit du résultat
 - git merge --squash RELEASE1

```
$ git merge --squash RELEASE1
Auto-merging fic2.txt
CONFLICT (add/add): Merge conflict in fic2.txt
Squash commit -- not updating HEAD
Recorded preimage for 'fic2.txt'
Automatic merge failed; fix conflicts and then commit the result.
```

- Le fichier specific-release.txt a été copié
- Le fichier fic2.txt est en conflit (à résoudre)

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   specific-release.txt

Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
    both added:      fic2.txt
```

Exemple : merge

3/3

- Merge du fichier fic2.txt

```
$ cat fic2.txt
<<<<< HEAD
1
2
3
4
|||||| merged common ancestors
=====
1

4
5
>>>>> RELEASE1
```

- Merge manuel du fichier (les IDE proposent des outils, mais souvent un bête éditeur de texte suffit)
- Ajout du fichier et commit du merge :
 - git add fic2.txt
 - git commit -m "Merge de RELEASE1"
- git lg --all

```
$ git commit -m "Merge de RELEASE1"
Recorded resolution for 'fic2.txt'.
[master d7bd7a0] Merge de RELEASE1
```

```
$ git lg --all
*   d7bd7a0 - (HEAD -> master) Merge de RELEASE1 (Stephane Michel 29 seconds ago)
|\ 
| * 58ac65d - (RELEASE1) Fichier spÃ©cifique release1 (Stephane Michel 21 minutes ago)
| * 3237c15 - Ajout fichier fic2.txt dans RELEASE1 (Stephane Michel 27 minutes ago)
| * 1c178f8 - Ajout fichier spÃ©cifique master (Stephane Michel 22 minutes ago)
| * 62e0316 - Ajout fic2.txt dans master (Stephane Michel 26 minutes ago)
|/
* 4cccd152 - Ajout fic1.txt (Stephane Michel 30 minutes ago)
* 9be974e - initial (Stephane Michel 31 minutes ago)
```

« Tête détachée » ?!

- Dans Git, le HEAD peut avoir l'un des deux états suivants :
 - **Branche active** : il référence une pointe de branche par son nom
 - .git/HEAD est de la forme ref: refs/heads/xxxxx
 - **Tête détachée** : tous les autres cas
 - .git/HEAD contient juste le SHA1 du commit référencé
 - Ça peut survenir après...
 - Un checkout sur autre chose qu'un nom de branche locale
 - Un rebase sur autre chose qu'un nom de branche locale
 - Que rebase vous donne la main (edit, conflit...)
 - Un submodule update (tête détachée dans le dossier du submodule)
 - Divers autres cas à la marge

Je suis en tête détachée, je fais quoi ?

- Évitez de committer directement : les commits résultants seraient orphelins (ils n'appartiendraient à aucune branche) -> Risque de les perdre
Faites une branche plutôt, idéalement dès le départ.
 - `git checkout -b ma-super-branche`
- Vous pouvez bien sûr revenir sur une branche active :
 - `git checkout la-branche-voulue`
- Lors d'un rebase, vous pouvez aussi décider de lâcher l'affaire
 - `git rebase --abort`

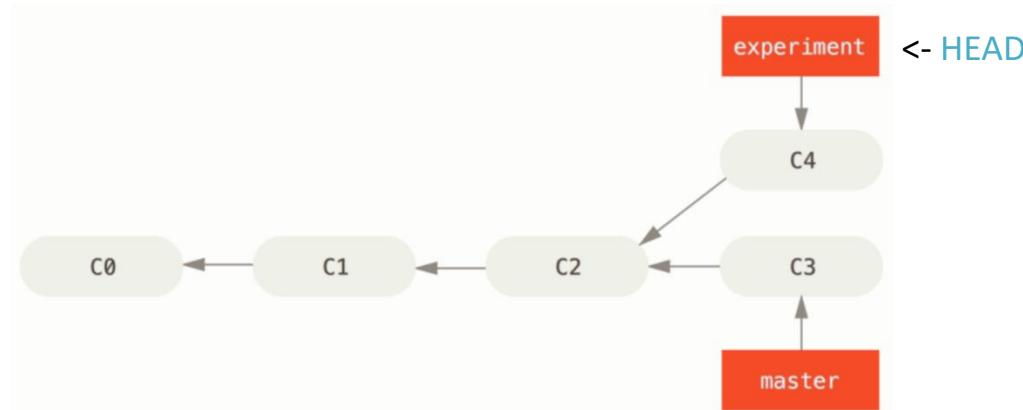
git rebase

- `git rebase nouvelle-base [branche-qui-rebase]`
 - Par défaut, rebase le `HEAD` sur la nouvelle base
- **Rejoue chaque commit** sur la nouvelle base, ce qui revient à faire une série de `cherry-picks`.
 - Donc risques de conflits, comme d'habitude
 - S'il y en a, rebase vous donne la main pour les résoudre avant de continuer
- Quand rebase vous donne la main et que vous avez terminé de résoudre les conflits:
 - `git rebase --continue` : Continue le rebase (passe aux cherry-picks des commits suivants)
 - `git rebase --abort` : Abandonne le rebase. Remet dans l'état avant le rebase.
 - `git rebase --skip` : Abandonne le commit courant et passe au suivant.
- Si on fait un rebase sur sa propre branche cela permet avec l'option `--interactive (-i)` de « propriétier » son historique de commit (fusion de commit, suppression de commit, modification de message de commit, etc.)
 - `git rebase -i SHA1` : où SHA1 est le commit (exclu) à partir duquel on souhaite « propriétier » les choses.

Voir détail <https://git-scm.com/book/en/v2/Git-Branching-Rebasing>

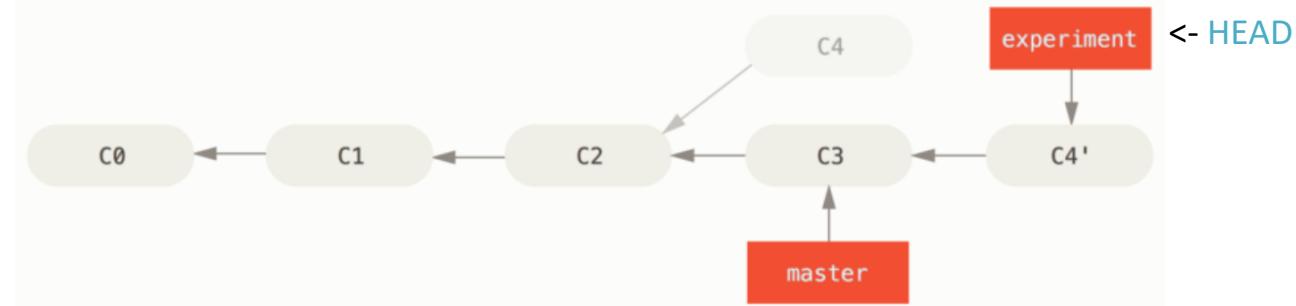
Illustration : git rebase

Avant, 2 branches



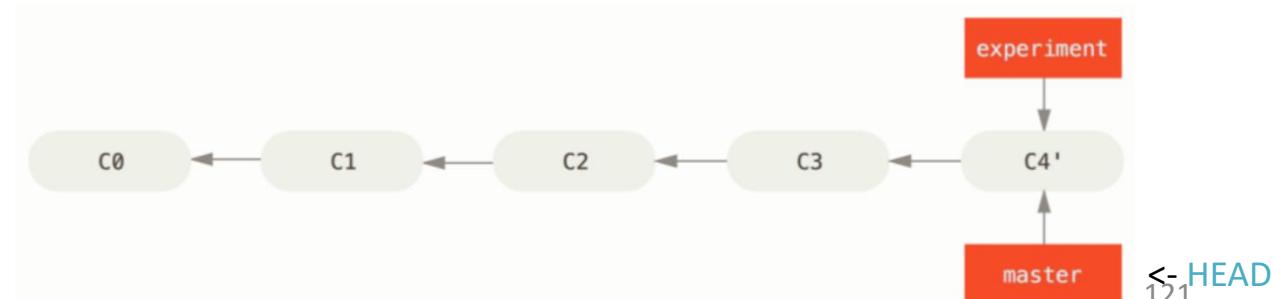
git rebase master

Rebase, rejoue
(cherry-pick) du C4



git checkout master
git rebase experiment

Fast-forwarded master
to experiment



Practice Time

Practice makes perfect!



TP-MERGE.zip

Exemple : rebase

1/3

- Environnement (idem merge) :
 - 2 branches avec chacune un fichier identique (fic1.txt) un fichier de même nom mais de contenu différent (fic2.txt), et chacune un fichier spécifique :

- Arbo master:

```
smichel@P4800-MICHEL MINGW64 /d/formations/git/TP-MERGE (master)
$ ll
total 3
-rw-r--r-- 1 smichel 1055235 54 Jan 16 15:28 fic1.txt
-rw-r--r-- 1 smichel 1055235 12 Jan 16 15:37 fic2.txt
-rw-r--r-- 1 smichel 1055235 27 Jan 16 15:37 specific-master.txt
```

- Arbo RELEASE1 :

```
smichel@P4800-MICHEL MINGW64 /d/formations/git/TP-MERGE (RELEASE1)
$ ll
total 3
-rw-r--r-- 1 smichel 1055235 54 Jan 16 15:28 fic1.txt
-rw-r--r-- 1 smichel 1055235 13 Jan 16 15:36 fic2.txt
-rw-r--r-- 1 smichel 1055235 28 Jan 16 15:36 specific-release.txt
```

- On souhaite « rebaser » la branche RELEASE1 dans la branche master

Exemple : rebase

2/3

- git checkout master
- git rebase RELEASE1

```
$ git rebase RELEASE1
First, rewinding head to replay your work on top of it...
Applying: Ajout fic2.txt dans master
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging fic2.txt
CONFLICT (add/add): Merge conflict in fic2.txt
Recorded preimage for 'fic2.txt'
error: Failed to merge in the changes.
Patch failed at 0001 Ajout fic2.txt dans master
The copy of the patch that failed is found in: .git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
```

```
$ git status
rebase in progress; onto 58ac65d
You are currently rebasing branch 'master' on '58ac65d'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)

    both added:      fic2.txt

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

- Résolution du conflit de fic2.txt
- git add fic2.txt
- git rebase --continue

```
$ git rebase --continue
Applying: Ajout fic2.txt dans master
Recorded resolution for 'fic2.txt'.
Applying: Ajout fichier spA@cifique master
```

Exemple : rebase

3/3

- Une fois le rebase terminé, la branche RELEASE1 a été « aplatie »
 - Rappel AVANT rebase

```
$ git lg --all
* 58ac65d - (HEAD -> RELEASE1) Fichier spÃ©cifique release1 (Stephane Michel 5 seconds ago)
* 3237c15 - Ajout fichier fic2.txt dans RELEASE1 (Stephane Michel 6 minutes ago)
| * 1c178f8 - (master) Ajout fichier spÃ©cifique master (Stephane Michel 61 seconds ago)
| * 62e0316 - Ajout fic2.txt dans master (Stephane Michel 5 minutes ago)
|
* 4ccd152 - Ajout fic1.txt (Stephane Michel 9 minutes ago)
* 9be974e - initial (Stephane Michel 10 minutes ago)
```

- Après rebase

```
$ git lg --all
* af1888e - (HEAD -> master) Ajout fichier spÃ©cifique master (Stephane Michel 36 minutes ago)
* d23db12 - Ajout fic2.txt dans master (Stephane Michel 40 minutes ago)
* 58ac65d - (RELEASE1) Fichier spÃ©cifique release1 (Stephane Michel 35 minutes ago)
* 3237c15 - Ajout fichier fic2.txt dans RELEASE1 (Stephane Michel 41 minutes ago)
* 4ccd152 - Ajout fic1.txt (Stephane Michel 44 minutes ago)
* 9be974e - initial (Stephane Michel 45 minutes ago)
```

Remarque 1 : les commit **1c178f8** et **62e0316** existent toujours mais ne sont plus rattachées à une branche

Remarque 2 : C'est une très mauvaise idée de rebaser la master sur une branche locale... (il vaut mieux faire le contraire)

Pourquoi soigner son historique

- Faciliter la lecture du log
 - Plus concis, sans commits superflus / réciproques
 - Mieux séquencé
 - Mieux rédigé
 - Sujets mieux regroupés
- Faciliter la récupération de thèmes
 - *Cherry-picking*
 - *Merge / rebase* d'un intervalle continu de commits
 - Récupération d'une branche sans dépendance historique superflue
 - Réduit les risques de conflits hors-sujet
- Avoir l'air de super-héros
 - « *Elle réussit toujours du premier coup, et dans l'ordre en plus !* »

Plus d'information [dans notre article sur git rebase](#)

Comment soigner son historique

- Je veux fusionner des commits, en supprimer et en renommer.
- `git rebase [-i | --interactive] SHA1_DE-DEPART`
 - Ouvre un éditeur qui permet, pour chaque commit de choisir ce que l'on souhaite en faire :
 - **pick** : conserve le commit tel quel
 - **reword** : conserve le commit mais permet de redéfinir le message.
 - **edit**
 - **squash** : merge le commit avec le commit précédent et permet de choisir le message du commit résultat
 - **fixup** : comme squash mais ignore le message du commit (on prend du coup celui du commit précédent)
 - **drop** : Annule le commit (les modifications du commit ne sont pas conservées)
 - **exec** : permet de lancer des commandes externes...

Practice Time

Practice makes perfect!



Exemple : rebase -i

1/3

- J'ai un historique trop verbeux

```
$ git lg
* bce0afd - (HEAD -> master) Modif fic3 (Stephane Michel 4 minutes ago)
* f0a6b38 - fic3 (Stephane Michel 5 minutes ago)
* cfb2ddf - modif fic2 (Stephane Michel 5 minutes ago)
* 0ba3384 - fic2 (Stephane Michel 6 minutes ago)
* d3ac4b6 - fic (Stephane Michel 6 minutes ago)
```

- Je souhaite fusionner les lignes « fic3 » et « Modif fic3 » en « fic3 » ainsi que les lignes « fic2 » et « modif fic2 » en « Modif fic2 » pour avoir au final :

```
$ git lg
* aefdea9 - (HEAD -> master) fic3 (Stephane Michel 31 minutes ago)
* 4d558e9 - Modif fic2 (Stephane Michel 33 minutes ago)
* d3ac4b6 - fic (Stephane Michel 33 minutes ago)
```

- Mais sans perdre de modifications dans les fichiers bien évidemment !



Exemple : rebase -i

2/3

- git rebase -i d3ac4b6

```
pick 0ba3384 fic2
pick cfb2ddf modif fic2
pick f0a6b38 fic3
pick bce0afd Modif fic3

# Rebase d3ac4b6..bce0afd onto d3ac4b6 (4 commands)
#
# 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
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
```



rebase.zip

Exemple : rebase -i

3/3

```
pick 0ba3384 fic2
pick cfb2ddf modif fic2
pick f0a6b38 fic3
pick bce0afd Modif fic3
```



```
pick 0ba3384 fic2
squash cfb2ddf modif fic2
pick f0a6b38 fic3
fixup bce0afd Modif fic3
```



```
# This is a combination of 2 commits.
# The first commit's message is:

fic2

# This is the commit message #2:

modif fic2
```



```
# This is a combination of 2 commits.
# The first commit's message is:

#fic2

# This is the commit message #2:

Modif fic2
```



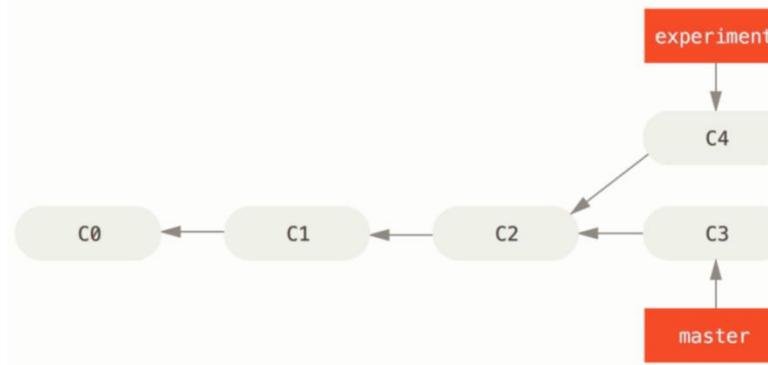
```
$ git lg
* aefdea9 - (HEAD -> master) fic3 (Stephane Michel 31 minutes ago)
* 4d558e9 - Modif fic2 (Stephane Michel 33 minutes ago)
* d3ac4b6 - fic (Stephane Michel 33 minutes ago)
```

Merge vs. Rebase

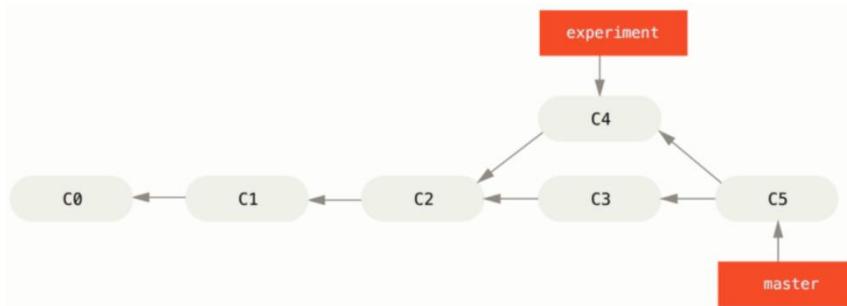
- Le rebase est là pour modifier tout ou partie de votre graphe d'historique (en général, tout ou partie d'une branche) pour le faire partir d'un autre commit de base (« re-base »).
- Là où un merge fusionne vers le point courant, rebase rebasse par-dessus un autre point : inversion de perspective.
- **Deux scénarios principaux :**
 - Soigner / restructurer l'historique (on verra tout ça plus loin)
 - Éviter les merge commits en leur préférant la consolidation des commits individuels dans une même branche
- [L'article de fond qui va bien](#)

Illustration : git rebase vs git merge

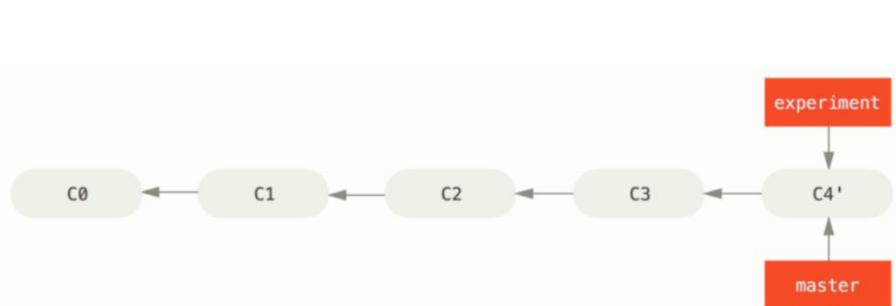
Avant, 2 branches



Après merge (sans fast-forward)



Après rebase





Cherry-picking

git cherry-pick

- Parce qu'on n'est pas obligé de se farcir un historique pour récupérer juste un commit
 - Typiquement un *bugfix* autonome, orthogonal
- `git cherry-pick [-x] un-commit`
 - Conceptuellement, un rebase miniature (limité à un commit)
 - L'option -x permet d'ajouter au texte de commit des informations complémentaires permettant de retracer le commit initial (« cherry picked from commit... »)
- Comment sélectionner des commits cibles (liste les commits que je n'ai pas encore sur ma branche) ?
 - `git cherry -v` / `git log --cherry`
 - Dans le même univers :
 - `git branch --contains`
 - `git branch --merged` / `--no-merged`
 - Fréquent pour porter un fix local sur l'*upstream* (ex. d'un fork vers le dépôt d'origine). Si sur GitHub / GitLab / BitBucket / Stash, préférez une *pull request* :-)

Illustration : Cherry picking

Avant cherry-pick

```
* 66abd2b - (HEAD -> maBranche) Modif fic5 dans maBranche (Stephane Michel 2 seconds ago)
| *
| * 7967097 - (master) Modif fic4 (Stephane Michel 55 seconds ago)
|/
|* 5626b42 - Modif fic3 dans MaBranche (Stephane Michel 85 minutes ago)
|* 76c23c5 - Correction urgente (Stephane Michel 17 hours ago)
|* 2ff2d78 - Ajout contenu dans fic2 (Stephane Michel 3 years, 7 months ago)
|* 5502a19 - fic2 (Stephane Michel 3 years, 7 months ago)
|* 03a1048 - Modif1 (Stephane Michel 3 years, 7 months ago)
|* cb60533 - fic1 (Stephane Michel 3 years, 7 months ago)
```

Récupération du commit **7967097** dans la branche « **maBranche** »

```
$ git cherry-pick 7967097
[maBranche c7ea0fc] Modif fic4
Date: Fri Aug 21 10:12:35 2020 +0200
1 file changed, 1 insertion(+)
create mode 100644 fic4
```

Après cherry-pick, les modif du commit **7967097** ont été reportées dans la branche **maBranche** sous le commit **c7ea0fc** (**notez que le SHA-1 est différent du commit d'origine**)

```
* c7ea0fc - (HEAD -> maBranche) Modif fic4 (Stephane Michel 2 minutes ago)
* 66abd2b - Modif fic5 dans maBranche (Stephane Michel 73 seconds ago)
| *
| * 7967097 - (master) Modif fic4 (Stephane Michel 2 minutes ago)
|/
|* 5626b42 - Modif fic3 dans MaBranche (Stephane Michel 87 minutes ago)
|* 76c23c5 - Correction urgente (Stephane Michel 17 hours ago)
|* 2ff2d78 - Ajout contenu dans fic2 (Stephane Michel 3 years, 7 months ago)
|* 5502a19 - fic2 (Stephane Michel 3 years, 7 months ago)
|* 03a1048 - Modif1 (Stephane Michel 3 years, 7 months ago)
|* cb60533 - fic1 (Stephane Michel 3 years, 7 months ago)
```

Practice Time

Practice makes perfect!



TP-CHEERRYPICK.zip

Exemple : cherry-pick 1/2

- Environnement : une branche master, une branche RELEASE1.

Le fichier Class1.java est corrigé dans la branche RELEASE1 à plusieurs reprises pour plusieurs FTA et on souhaite récupérer la dernière correction dans master (c'est-à-dire la FTA 456231 et pas la FTA 456221).

```
public final class AESUtil
{
    protected static String startSession(final File traceFile) throws Exception
    {
        String sessionId = String.valueOf(new Date().hashCode());
        final Random r = new SecureRandom();

        // Correction FTA 456231 : c'est 64, pas 32
        byte[] salt = new byte[64];
        r.nextBytes(salt);

        // N'importe quoi, c'est 1024 !
        SecretKeySpec secret = initKey(salt, 256, 1024);

        Cipher cipher = initCipher(secret);
        AESUtilSession session = new AESUtil.AESUtilSession(cipher, salt, traceFile);
        addSession(sessionId, session);
        return sessionId;
    }
}
```

```
$ git lg --all
* aa01072 - (HEAD -> RELEASE1) Correction anomalie 456231 : 64 au lieu de 32 (Stephane Michel 12 seconds ago)
* 358f83a - Correction anomalie 456221 (Stephane Michel 3 minutes ago)
* c5315b3 - (master) Class1.java version initiale (Stephane Michel 6 minutes ago)
* efa5768 - Commit initial (Stephane Michel 8 minutes ago)
```

Exemple : cherry-pick 2/2

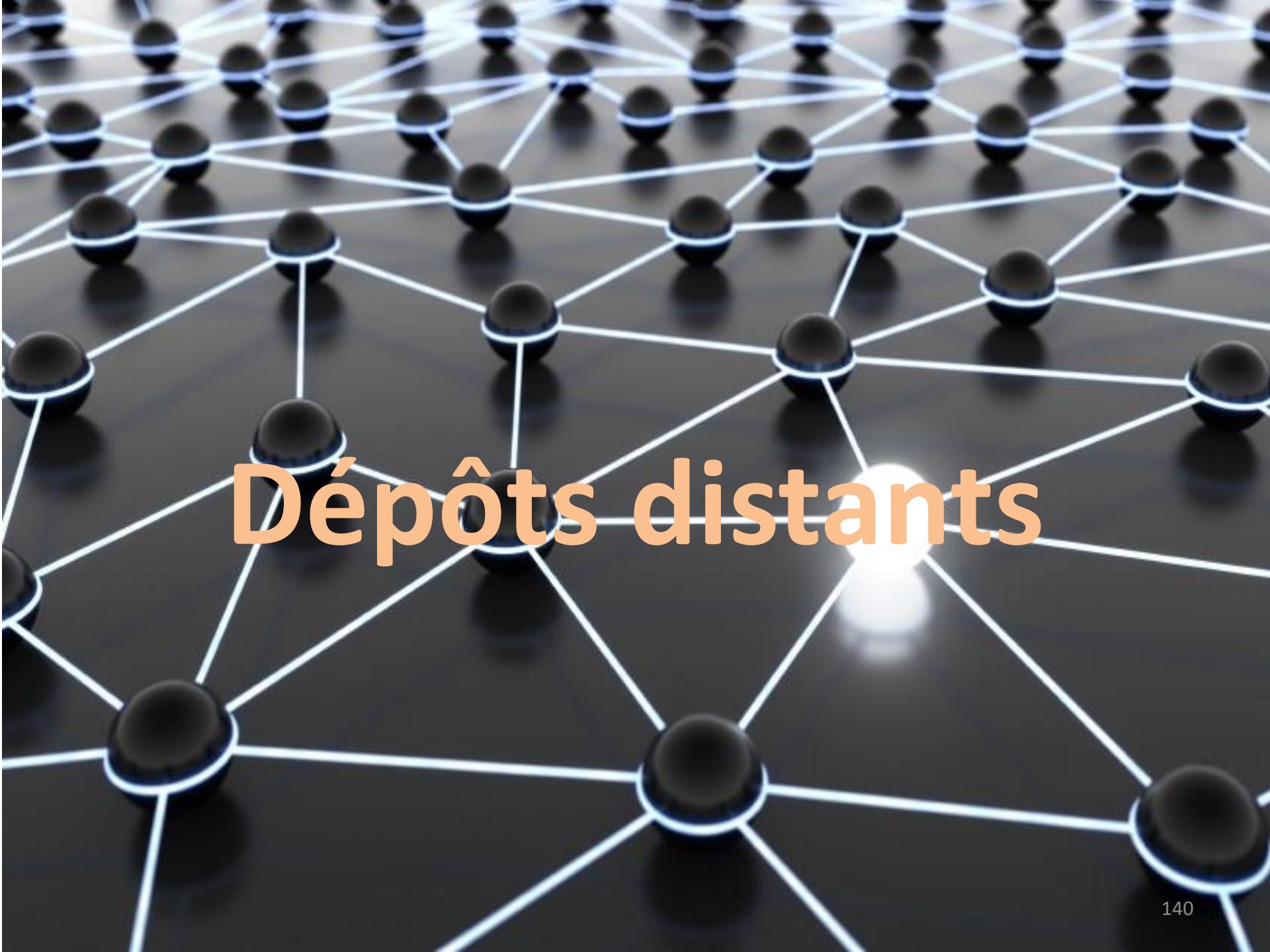
- Positionnement sur la branche master
 - git checkout master
- Cherry-pick
 - git cherry-pick -x aa01072

```
$ git cherry-pick -x aa01072
[master c480a14] Correction anomalie 456231 : 64 au lieu de 32
  Date: Mon Jan 16 16:29:49 2017 +0100
  1 file changed, 3 insertions(+), 1 deletion(-)
```

```
public final class AESUtil
{
    protected static String startSession(final File traceFile) throws Exception
    {
        String sessionId = String.valueOf(new Date().hashCode());
        final Random r = new SecureRandom();

        // Correction FTA 456231 : c'est 64, pas 32
        byte[] salt = new byte[64];
        r.nextBytes(salt);
        SecretKeySpec secret = initKey(salt, 256, 2048);
        Cipher cipher = initCipher(secret);
        AESUtilSession session = new AESUtil.AESUtilSession(cipher, salt, traceFile);
        addSession(sessionId, session);
        return sessionId;
    }
}
```

```
$ git lg --all
* c480a14 - (HEAD -> master) Correction anomalie 456231 : 64 au lieu de 32 (Stephane Michel 7 minutes ago)
| * aa01072 - (RELEASE1) Correction anomalie 456231 : 64 au lieu de 32 (Stephane Michel 7 minutes ago)
| * 358f83a - Correction anomalie 456221 (Stephane Michel 9 minutes ago)
|
* c5315b3 - Class1.java version initiale (Stephane Michel 12 minutes ago)
* efa5768 - Commit initial (Stephane Michel 15 minutes ago)
```



Dépôts distants

Dépôts distants

- Les vieilles habitudes ont la vie dure : souvent un seul dépôt distant (celui de référence)
 - Très vrai en entreprise
 - Beaucoup moins dans l'open-source
- On peut parfaitement avoir plusieurs *remotes*, certains R/O, d'autres R/W, chacun avec ses propres commits, branches, tags...
 - ex. F/LOSS (*Free / Libre and Open Source Software*): un par contributeur noyau
 - ex. modèle à lieutenants : un par équipe interne sans accès R/W au *remote* canonique, pour qu'elles puissent quand même collaborer en interne.
- Bon chez DA, avec Bitbucket, on n'a qu'un seul dépôt distant...

git remote

- ***remote*** = dépôt distant
- git remote gère ces définitions pour chaque dépôt local
 - git remote : à vide, liste les dépôts distants existants
 - git remote add *nom url* : ajoute une référence locale *nom* pour le dépôt distant à l'adresse *url*
git remote add origin git@github.com:tdd/attitude.git
 - git remote rename *ancien-nom* *nouveau-nom* : renomme (en local) le dépôt désigné
 - git remote prune *nom* : vire les refs locales obsolètes (branches distantes déjà supprimées)

Protocoles de dépôts distants

- `user@ == ssh://user@` — **git over SSH** (port 22)
 - Protocole Git R/W (ou R/O) sécurisé par SSH. Le plus fréquent en collab'.
- `git://` — protocole Git (port 9418)
 - Généralement anonyme et R/O. Le plus fréquent en consultation pure.
- `http(s)://` ou `ftp(s)://` — moins cool (ports 80/443, 20/21)
 - Parfait si vous êtes derrière un pare-feu pénible ou un PALC, ou dans un environnement sans SSH, type 100% Windows en Active Directory.
- `filesystem` — pratique pour les tests à l'arrache
 - Un simple chemin de dossier suffit !

git push

- Envoie au remote les commits locaux.
- Pas mal de syntaxes, dont :
 - `git push -u origin maBranche` (1er envoi + *tracking*)
 - `git push origin maBranche` (branche maBranche de *origin*)
 - `git push origin` (branches trackées de *origin*)
 - `git push` (dépend de *push.default*)
- `--tag` récupère en plus les tags
- Vous n'êtes **pas obligés** d'utiliser le même nom distant...
 - `git push -u origin master:christophe-master`
- Également utilisé pour **supprimer des branches distantes**
 - `git push --delete origin branch-to-delete`

git push : annuler un push

- Git push permet également d'annuler le dernier commit envoyé sur le serveur
 - Si on n'a pas récupéré en local ledit commit :
 - `git push origin +MON_SHA1^:MA_BRANCHE`
 - MON_SHA1 : sha1 que l'on souhaite supprimer
 - MA_BRANCHE : branche dans laquelle on souhaite supprimer le commit
 - Si on a en local le commit :
 - `git reset HEAD^ --hard`
 - `git push origin -f`

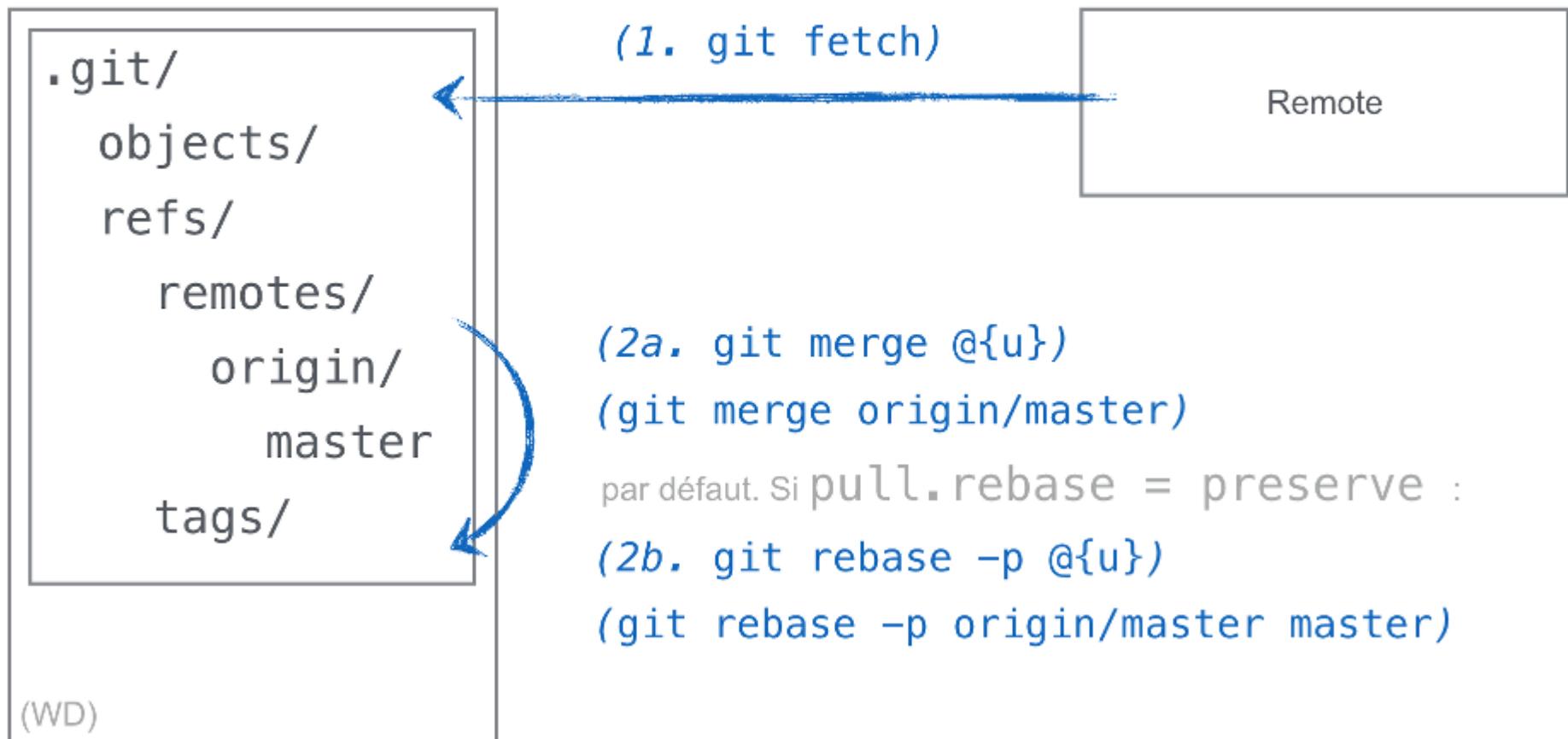
Attention gros pb potentiel si quelqu'un a récupéré le commit non souhaité entre temps (à n'utiliser que dans les secondes qui suivent le commit)

git pull

- Techniquement, `fetch` + `merge` (sur l'*upstream* courant)
 - `git pull origin master` (explicite)
 - `git pull origin` (le *remote* complet)
 - `git pull` (le *remote* de l'*upstream* courant ; *origin* par défaut)
- `--tags` récupère en plus les tags
- Pour faire plutôt un `rebase`, on utilisera `git pull --rebase`
- Puisqu'on peut **découper** `fetch` et `merge/rebase`, on a tendance à le faire quand on a très peu de temps connecté devant soi...
 - `git fetch`
 - déconnexion (ex. embarquement dans un avion)
 - `git merge` ou `git rebase` (au calme, hors-ligne...)

git pull, une danse à deux temps

(master u=) \$ **git pull**



(2a. git merge @{u})
(git merge origin/master)
par défaut. Si `pull.rebase = preserve` :
(2b. git rebase -p @{u})
(git rebase -p origin/master master)

git fetch

- Récupère **toutes les nouveautés** du *remote* dans le cache local
- Ne touche **pas** aux branches locales.
- Plus besoin de connectivité au remote ensuite !
 - `git fetch origin master` (explicite)
 - `git fetch origin` (le *remote* complet)
 - `git fetch` (le *remote* de l'*upstream* courant ; *origin* par défaut)
 - `git fetch --all` (tous les *remotes*)
 - `git fetch --prune` (voir aussi la configuration `fetch.prune`)

git tag

- Utilisés en général pour indiquer un moment précis dans l'historique, le plus souvent un **numéro de version** sur une branche de *release*.
- Un tag n'est pas censé bouger, pour des raisons de QA / validation / certification principalement.
En revanche, une branche de version (release/prod) évolue au fil des bugfixes, hotfixes et security fixes.
 - `git tag [-f] nom-du-tag`
 - `git tag -l [motif]`
 - `git tag -a -m [message]`
- Option de config (`./.git/config`) :
traite les noms de tags comme des versions
`tag.sort = version:refname`

Exemple

```
$ git lg
* 1ce3b33 - (HEAD -> master) Modif dans file2.txt (Stephane Michel 76 minutes ago)
* 1b58269 - Modif dans toto.txt (Stephane Michel 76 minutes ago)
* ded5ea6 - Suppression du fichier file2.txt (Stephane Michel 3 hours ago)
* a57bdbb - new commit (Stephane Michel 3 hours ago)
* dffeb1ac - Ajout de toto.txt (Stephane Michel 3 hours ago)
* 857c44e - Premiere modif (Stephane Michel 3 hours ago)
* a750e4b - Ajout de Test.java (Stephane Michel 3 hours ago)
* 3826cdb - initial commit (Stephane Michel 4 hours ago)
```

- **git tag TAG1 dffeb1ac**

```
$ git lg
* 1ce3b33 - (HEAD -> master) Modif dans file2.txt (Stephane Michel 86 minutes ago)
* 1b58269 - Modif dans toto.txt (Stephane Michel 86 minutes ago)
* ded5ea6 - Suppression du fichier file2.txt (Stephane Michel 3 hours ago)
* a57bdbb - new commit (Stephane Michel 3 hours ago)
* dffeb1ac - (tag: TAG1) Ajout de toto.txt (Stephane Michel 3 hours ago)
* 857c44e - Premiere modif (Stephane Michel 3 hours ago)
* a750e4b - Ajout de Test.java (Stephane Michel 4 hours ago)
* 3826cdb - initial commit (Stephane Michel 4 hours ago)
```

- **git show TAG1**

Quelques ressources en plus...

- Les tutos et glossaires Git intégrés
 - `git help tutorial`
 - `git help glossary`
 - `git help cli`
- Doc en français :
 - <https://git-scm.com/fr/v2/>
- Une tonne de ressources au top
 - <http://git-scm.com/documentation>
- Git en bref
 - <http://gitref.org/>
 - <http://git-scm.com/docs/everyday> (“Everyday Git”)
 - <http://gitimmersion.com/>
- Re-pratiquer de zéro, différemment
 - [Explain Git With D3](#) and [Learn Git Branching](#)

A white and red lifebuoy hangs from a white rope against a backdrop of a clear blue sky with wispy white clouds. The lifebuoy is oriented vertically, with its red bands at the top and bottom. A white rope is wrapped around its middle and extends downwards and to the left.

Soucis classiques et solutions optimales sur dépôt distant

Soucis classiques et solutions optimales sur dépôt distant

- Mon **git pull** me dit « You have unstaged changes » !

```
$ git pull origin master
error: cannot pull with rebase: You have unstaged changes.
error: please commit or stash them.
```

- Des modifications sont en cours dans le WD !
- Trois solutions :
 1. On souhaite annuler les modifications locales :
 - `git st` : pour connaitre les fichiers qui posent problème
 - `git checkout mon_fichier` : pour annuler la modification sur le fichier
 2. On souhaite commiter les modifications car on a terminé un travail :
 - `git add .` : Pour placer les fichier du WD vers le stage
 - `git commit -m "bla bla"` : Pour commiter le stage
 3. On souhaite mettre de côté les modifications car elles ne sont pas terminées :
 - `git stash`
 - Ensuite (après le git pull) on retrouvera ses modifacations avec un `git stash pop`

Soucis classiques et solutions optimales sur dépôt distant

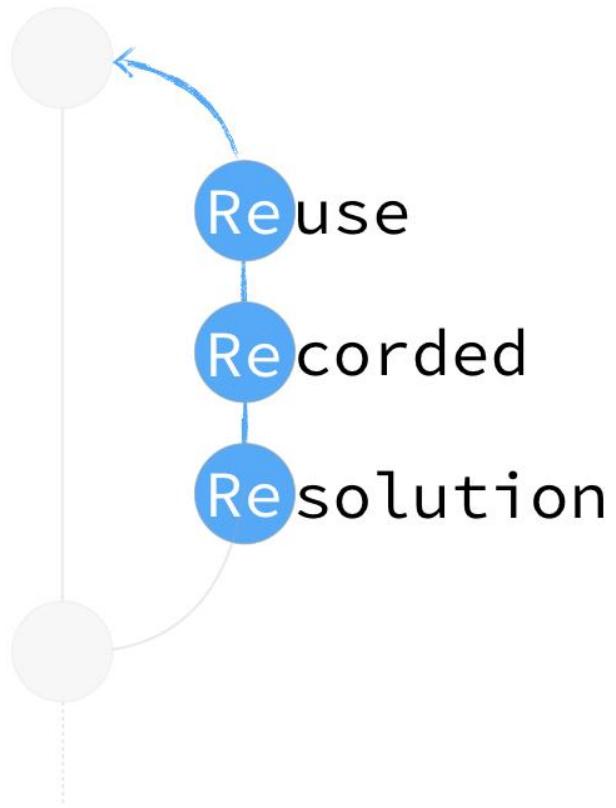
- Mon **git push** me dit « [rejected] ...la branche distante contient du travail que vous n'avez pas en local» !

```
[08:57] sogitec@MACPRO8:GIT-LOCAL-FIELD6-DASSAULT (master $) $ git push SM64 master
To /Volumes/SM64/repos/GIT-LOCAL-FIELD6
! [rejected]          master -> master (fetch first)
error: impossible de pousser des références vers '/Volumes/SM64/repos/GIT-LOCAL-FIELD6'
astuce: Les mises à jour ont été rejetées car la branche distante contient du travail que
astuce: vous n'avez pas en local. Ceci est généralement causé par un autre dépôt poussé
astuce: vers la même référence. Vous pourriez intégrer d'abord les changements distants
astuce: (par exemple 'git pull ...') avant de pousser à nouveau.
astuce: Voir la 'Note à propos des avances rapides' dans 'git push --help' pour plus d'information.
```

- Il y a des commits sur le serveur qui ne sont pas présents en local. Il faut d'abord les récupérer avec de pusher ses commits.
 - **git pull origin master** : pour récupérer les derniers commits

Soucis classiques et solutions optimales sur dépôt distant

- Je voudrai créer une copie de mon repos sur un média USB local (pour faire des transferts sur une autre machine par exemple)
 - Créer un répertoire sur le device :
Exemple : `D:/MON_REPERTOIRE`
 - Dans ce répertoire faire : `git init --bare` pour initier le repo GIT de type « bare »
 - Se positionner à la racine du repo à copier (sur la machine locale)
 - `git remote add MON_USB_DEVICE D:/MON_REPERTOIRE` : Déclaration du répôt distant
 - `git push MON_USB_DEVICE master` : On Recopie le repos sur le device



ReReRe

- REuse REcorded REsolution
- Flag global (fichier [.gitconfig](#)) : `rerere.enabled`
- Fichier `.git/rr-cache`
- Enregistre les « empreintes » des **résolutions manuelles** de conflits, afin de les rejouer si l’« empreinte réapparaît ».
- **Réutilisé automatiquement** lors d’une fusion (*merge* ou *rebase*, pas de jaloux) : Si la résolution a déjà été faite une fois, il saura refaire la même chose automatiquement.
- Ne fonctionne que localement
- Permet notamment des workflows sympathiques de « *control merges* » annulables mais profitables

Activer et utiliser ReReRe

- Options de config (.gitconfig) fournies désactivées :
`rerere.enabled` : active *rerere*
`rerere.autoupdate` : auto-*stage* les résolutions de conflits
- Nettoyage des empreintes de résolutions avec :
`git rerere clear` : la totale
`git rerere forget chemin` : chirurgical
`git rerere gc` : pour les références obsolètes



**Chasse aux bugs accélérée :
Le bisecting**

le bisecting

- **Le principe**
 - **Recherche dichotomique** entre le dernier point que vous identifiez comme OK et le point KO (généralement le HEAD le master)
 - On délimite l'historique à tester, et on y va à coup de « plus petit plus grand », en quelque sorte
 - Ne marche que si le bug n'a pas toujours été là !
 - Dichotomique, donc en **$O(\ln 2(n))$** au lieu de $O(n)$...
 - C'était il y a 30 commits : 5 tests tout au plus
 - C'était il y a 1 000 commits : 10 tests tout au plus
 - C'était il y a 1 000 000 commits : 20 tests tout au plus

git bisect

- `git bisect start` : lance le bisecting
- `git bisect bad` : indique que le bug est présent
- `git bisect good` : indique que le bug n'est pas présent

(pour chaque checkout auto, on teste,
puis `git bisect bad/good/skip`)
- `git bisect reset` : Annule le bisecting



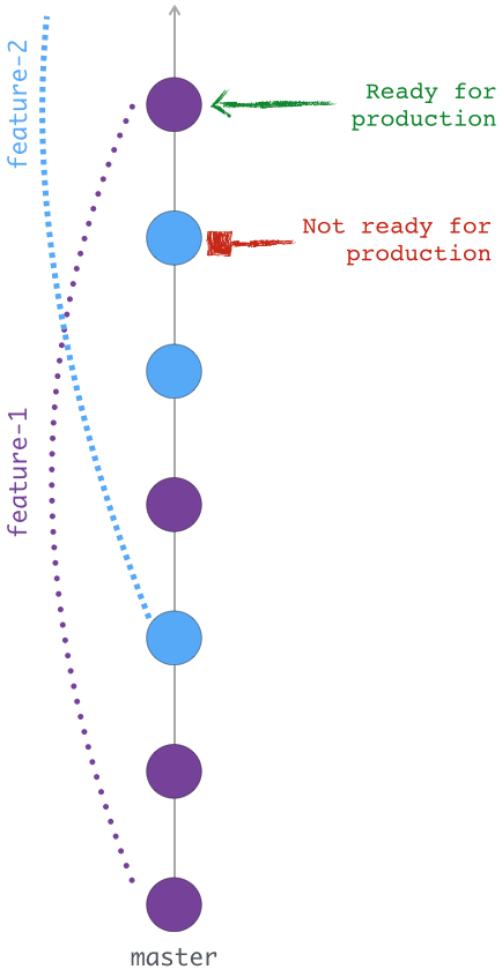
Les workflows

- Centralized workflow
 - À la SVN, mais avec les avantages de Git (copie locale, branches...)
- Feature Branch workflow
 - Branches systématiques de features
 - Pull requests
- Git Flow workflow
 - Strict et complexe
 - Multi-branches : master, develop, release, features, fixes
- Forking workflow
 - Copie distante/remote du dépôt initial pour chaque développeur/contributeur
 - Utilisé dans l'open-source (Github & co)

Centralized workflow

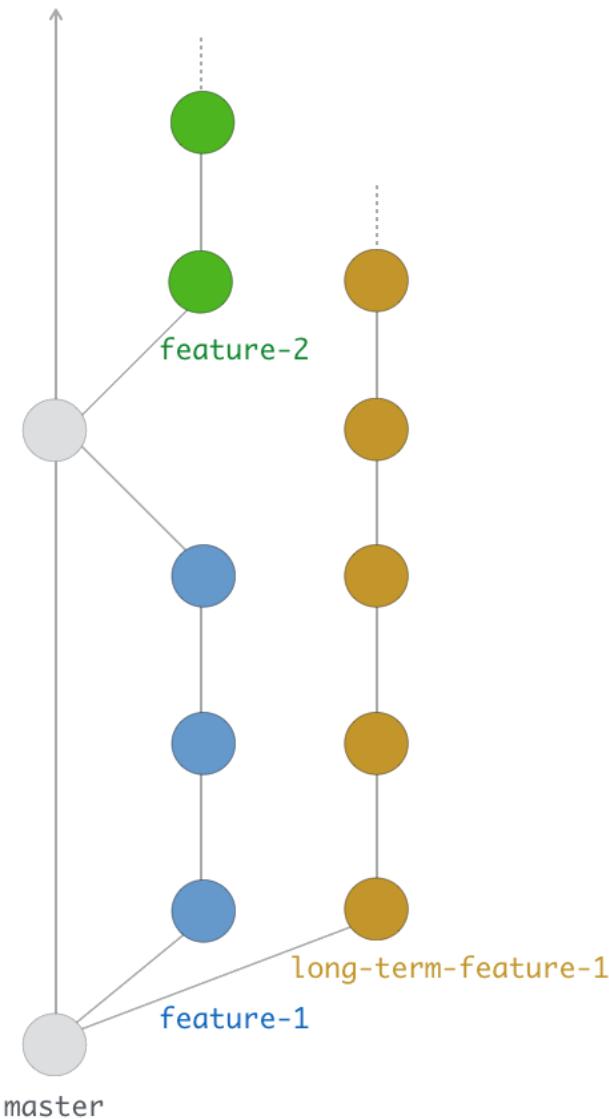
- Un dépôt central pour tous
- Branche unique `master`
- « SVN like » : *add, commit, pull, push*
- Se limite au strict minimum des avantages de Git :
 - copie locale
 - branches et fusions
 - résolution de conflits
 - rebasing (lors du pull)

Centralized workflow



- Difficulté d'avoir une version stable...
- et donc d'appliquer des correctifs urgents ;
- Pas d'isolation des lots de développement ;
- Historique difficile à analyser ;
- Arbitrage des conflits plus fréquent ;
- ...

Feature Branch workflow



- Un dépôt central
- Autant de branches qu'il y a de « features »
 - isole les lots de développement
 - limite les perturbations liées à des développements indépendants
- Encourage l'utilisation des ***merge-requests / pull-requests***
- Favorise la mise en place du **Semantic Versioning**

Git Flow workflow

- Le plus complexe de tous
- Notions utiles pour du gros projet type solution logicielle
- Imparfait
 - les merges vers de multiples destinations limitent certaines fonctions de Git (*bisect*, syntaxes de révision...)
- Variation simplifiée
 - utilisation du cherry-picking plutôt que du double merge
 - branche stable taguée : `master` (au lieu de `develop`)
- [Git flow par Vincent Driessen](#)

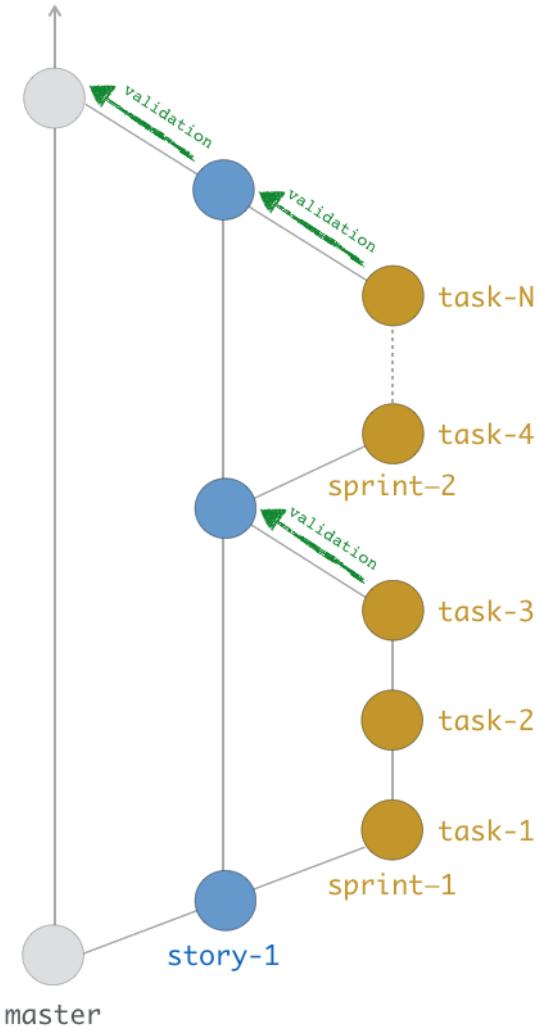
Forking workflow

- Fondamentalement différent des autres : isolation de l'origine
- Principalement utilisé dans l'open-source
- Encouragé par des solutions d'hébergement Git ([GitHub](#), [Bitbucket](#), [GitLab](#))
- Impose l'emploi des *merge-requests / pull-requests*
- Offre une gestion plus fine des droits / de la sécurité
 - un dépôt privé (local) et public (remote) par développeur ou groupe de développeurs
 - un dépôt « officiel » (origine du *fork*) avec droits plus restreints

Et pourquoi pas le vôtre ?

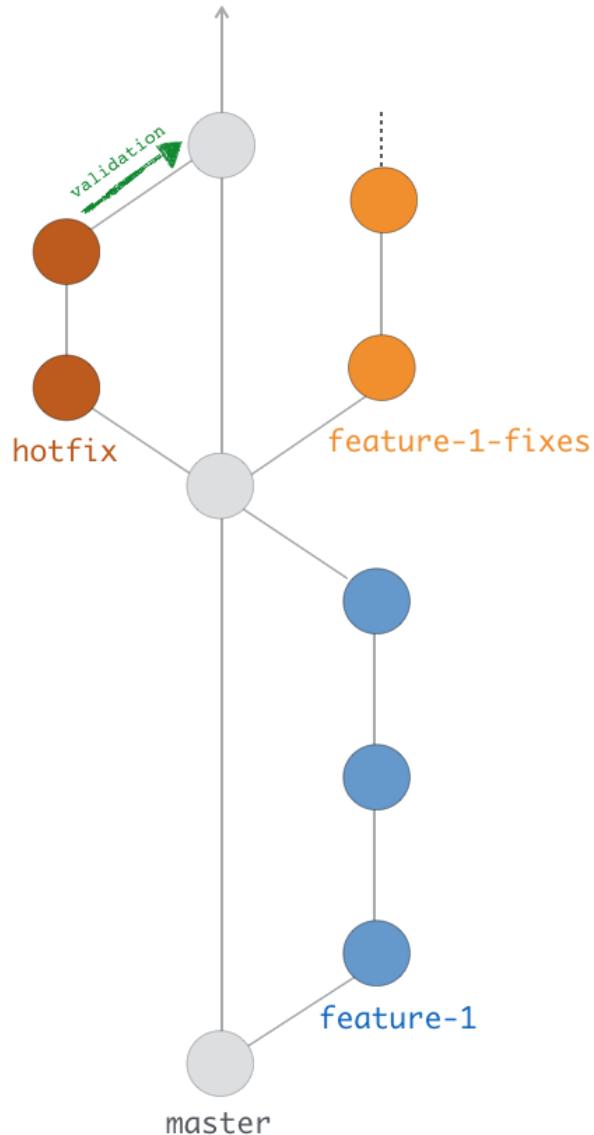
- Un workflow adapté à vos contraintes
 - nombre de collaborateurs
 - architecture du projet
 - méthodologie en place (ex: Agile, Scrum, cycle en V...)
- La simplicité pour maître mot
- Une mise en place progressive
 - formation des collaborateurs
 - définition des étapes / objectifs intermédiaires : modèle de branches, *merge-requests*...

Rester au plus proche de votre méthodologie projet



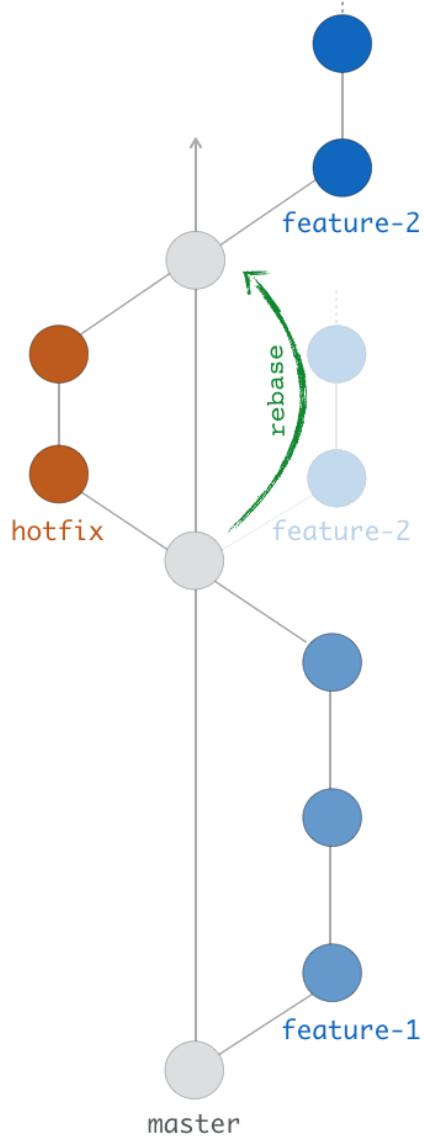
- Exemple avec la méthodologie Agile Scrum
- Convention
 - Une branche par *Story* ;
 - Une branche par *Sprint* ;
 - Un commit par *Task*.
- Retranscrit fidèlement la méthodologie ;
- Encourage un découpage fin du projet ;
- Facilite les validations intermédiaires.

Gestion des bugs



- Workflow « à deux voies » (maintenance ≠ gestion de projet classique)
- Gestion des correctifs sur des branches dédiées :
 - Processus de validation systématique (a minima l'Intégration Continue) ;
 - Meilleure lisibilité ;
 - Facilite le report par cherry-picking.
- Traitement simultané de plusieurs correctifs ;
- Historique plus clair.

Retranscription des correctifs



Retranscription de l'ensemble des correctifs de la branche initiale = mise à jour « par-dessus » la branche initiale.

Attention à la mise à jour en profondeur (par défaut seulement la branche désignée, pas ses descendantes).

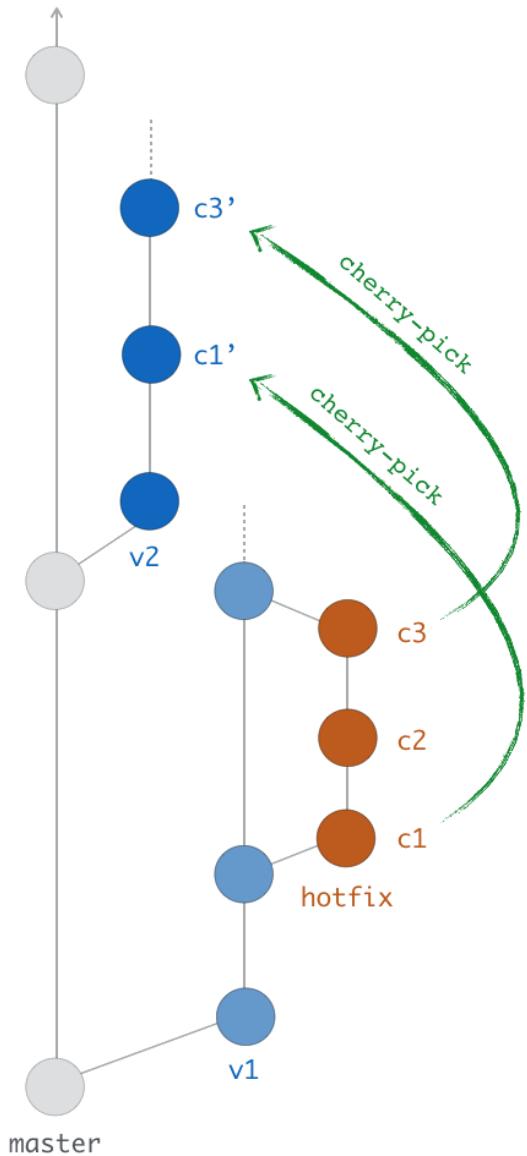
Pour conserver les branches mergées :

- `git rebase --preserve-merges master feature-2`

Pour chaque sous-branche sinon :

- `git rebase feature-2 feature-2-sub-1`
- `git rebase feature-2 feature-2-sub-2 ...`
- `git rebase feature-2 <sub-branch-N>`

Récupérer certains correctifs



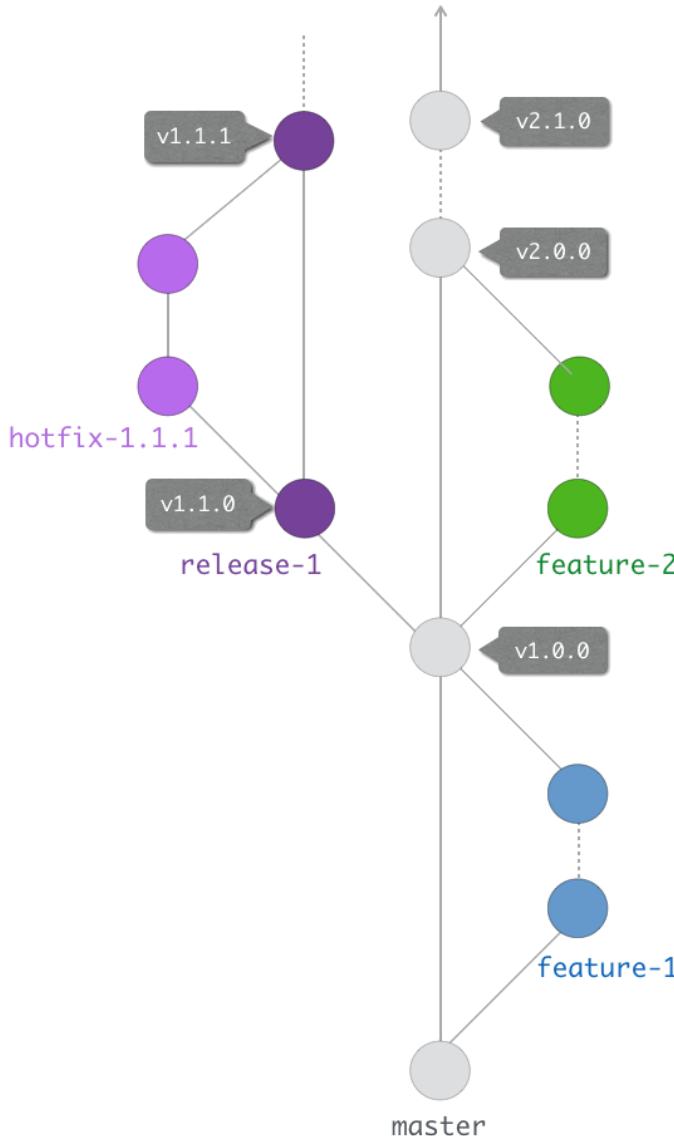
Parfois seulement certains correctifs sont utiles :

- Quand la branche depuis laquelle nous rapatrions n'est pas la branche d'origine ;
- Quand le *rebase* n'est pas opportun (ex : pour conserver l'historique de branche).

Nous irons alors « piocher » les commits désirés avec

- `git cherry-pick -x c1`
- `git cherry-pick -x c3`
- `git cherry-pick -x <sha1-du-commit>`

Identifier les versions



Pour des raisons de **maintenance** et/ou de **suivi de projet**.

L'identification des versions logicielles vous permettra :

- De déployer des **références nommées** ;
- De communiquer les évolutions d'une version à une autre (**release notes**) ;
- De définir le **point de départ de branche** de maintenance pour ces versions.

Annexes



Installation environnement

- Git sous Windows :
 - <https://git-scm.com/download/win>
 - <\\sct-d-97222.st-cloud.dassault-avion.dev\Shared\Git-2.21.0-64-bit.exe>
(si pas d'accès internet)
- git config http.postBuffer 524288000
- .gitconfig à copier sous :
 - C:\utilisateurs\%mon_user%\ sur un poste « normal »
 - U:\ sur les postes @
- Configuration précisée dans :
 - <https://installations.delicious-insights.com/software/git.html>
 - <https://confluence.exterieur.dassault-aviation.com:8445/pages/viewpage.action?pageId=532054713> pour SPAD

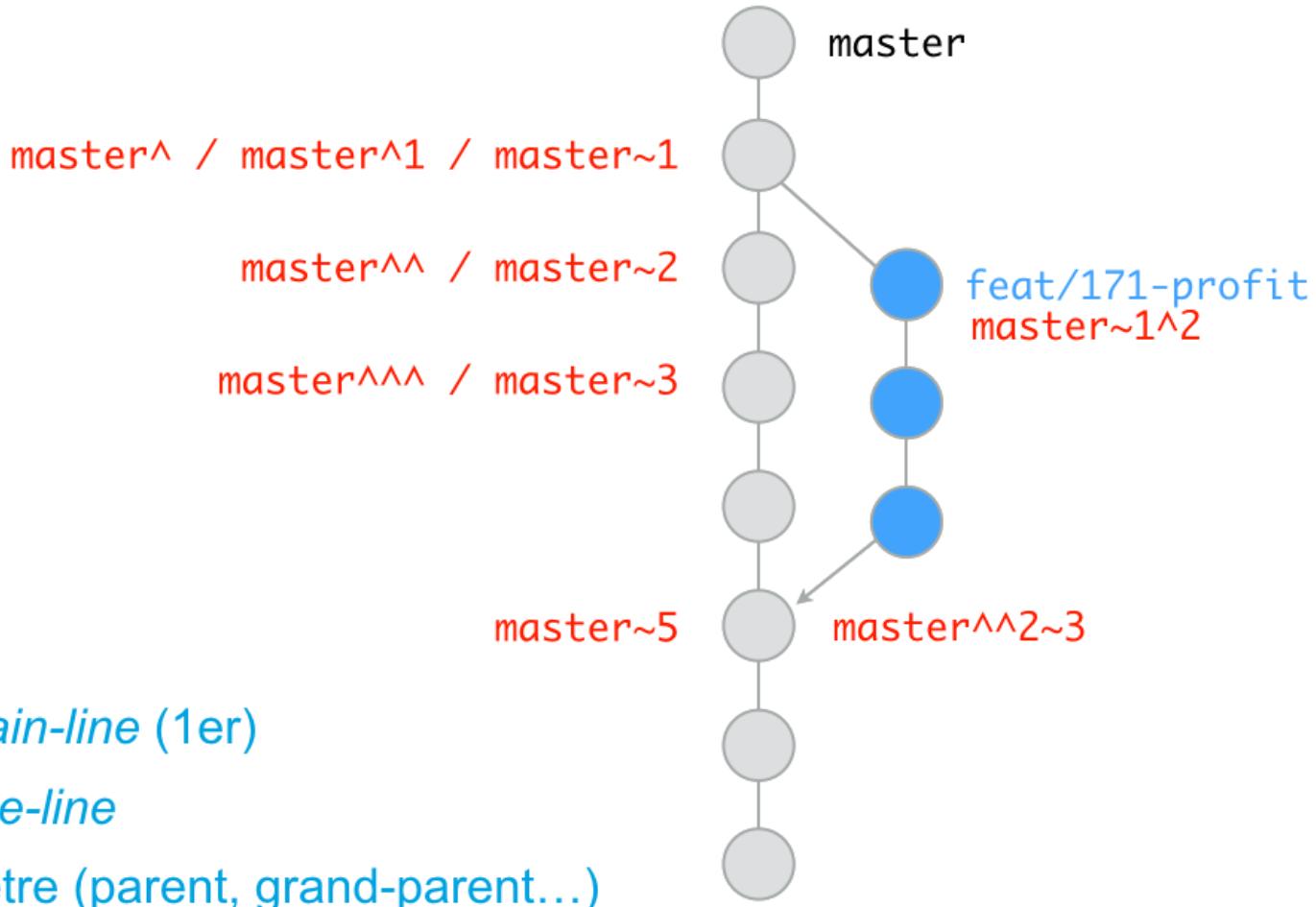


Memento : icônes du prompt

Symbole Signification

#	Root commit : pas encore de HEAD. N'arrive qu'une fois, entre le git init et le premier git commit.
%	Il y a des fichiers untracked (dans le WD mais ni dans le stage ni dans le HEAD).
+	Il y a du <i>stage</i> .
*	Il y a des fichiers modifiés (versionnés mais dont le WD diffère du stage, ou à défaut du HEAD).
\$	Il y a du <i>stash</i> (des états locaux mis en attente pour plus tard).
u=/+/-...	La branche locale courante est synchro / en avance / en retard par rapport à son <i>upstream</i> .

Memento : syntaxe de révisions 1/2



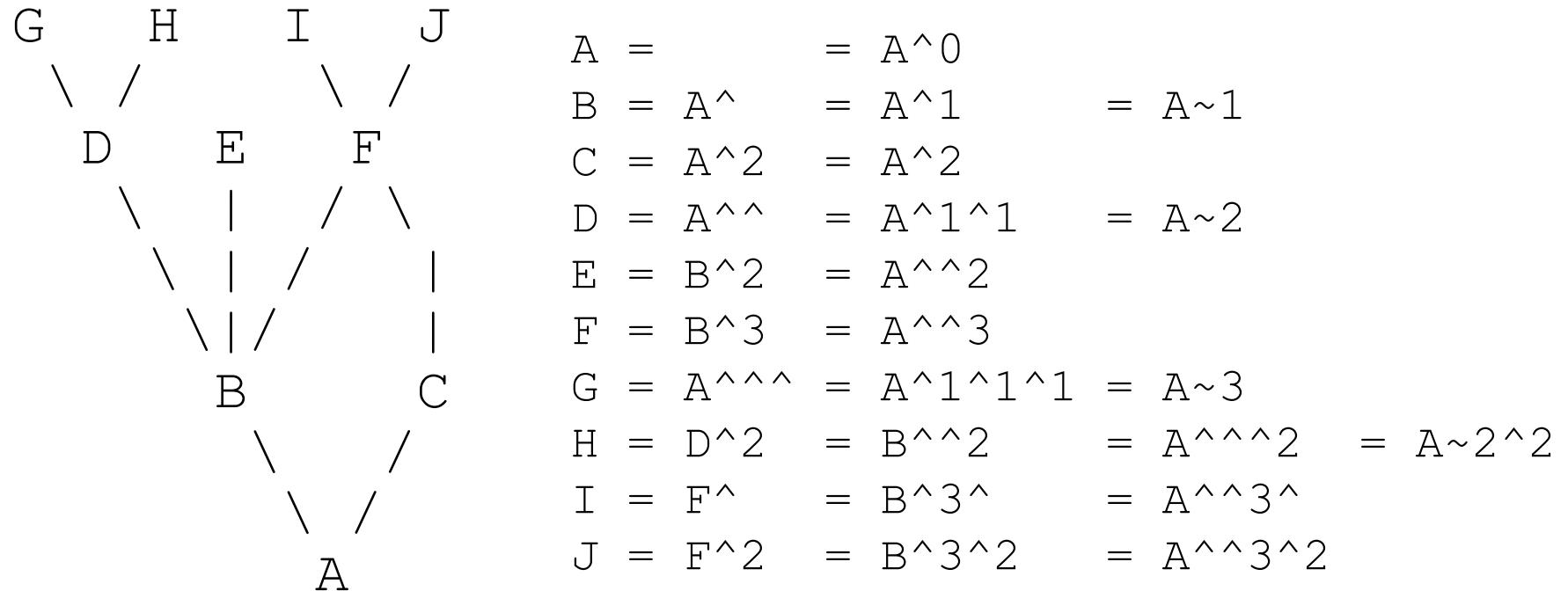
^, ^1 : parent *main-line* (1er)

^2 : parent *merge-line*

~n : *n*-ième ancêtre (parent, grand-parent...)

Memento : syntaxe de révisions 2/2

Here is an illustration, by Jon Loeliger. Both commit nodes B and C are parents of commit node A. Parent commits are ordered left-to-right.



Memento : Révisions symboliques

HEAD	En général la pointe de la branche courante, mais peut être détachée ; en somme, le commit de référence pour les comparaisons de statut, de diff...
ORIG_HEAD	Position du HEAD avant le dernier merge ou <i>rebase</i> . Pour un merge, équivaut normalement à HEAD@{1}
MERGE_HEAD, REBASE_HEAD	Commit en cours de fusion ou de <i>rebase</i>
FETCH_HEAD	Dernier commit récupéré par <i>fetch</i> (le plus souvent, la pointe de la branche distante trackée)

THANK

YOU