GIT

INTRODUCTION

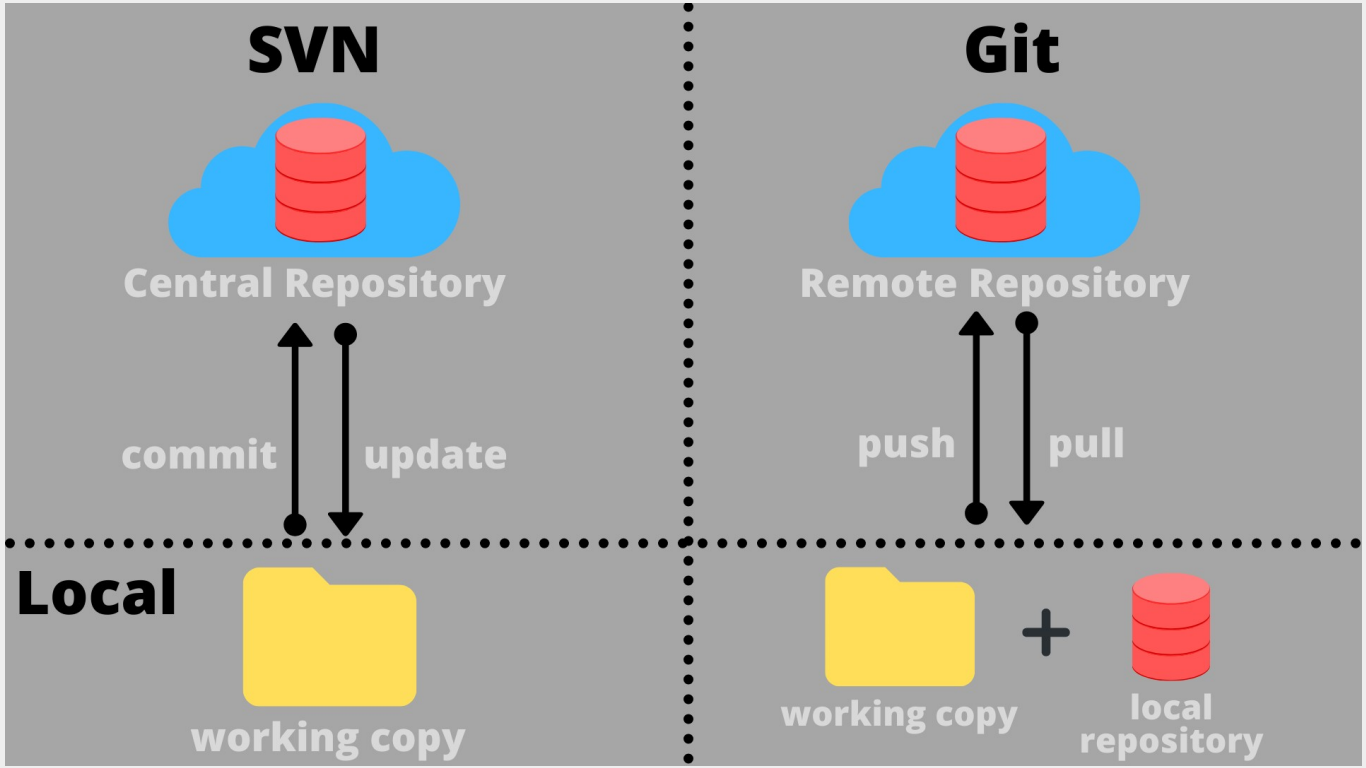
- Gestion de code source
  - Source Code Management (**SCM**) ou Version Control System (**VCS**)
  - gère les modifications d'un ensemble de fichiers donné situés dans un dossier nommé **copie de travail**.
  - Enregistre les **lots de modifications**, comme **version**, dans un dossier appelé **dépôt**.
  - Maintient **l'historique** des versions

- GIT :
  - SCM **décentralisé** : chaque utilisateur possède un dépôt local et un historique
  - Liste de commandes très fournie pour manipuler les versions, revenir en arrière, ...

  - **Doc officielle**
  - **Articles par Atlassian**

INTRODUCTION

- Git vs SVN

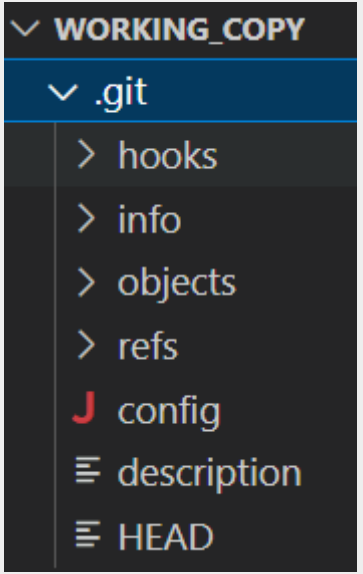


DEPOT LOCAL

- Création d'un dépôt local
  - **Dans le dossier de travail ou copie de travail**

```
cd working_copy

git init
```

création du dépôt, le dossier **.git**



DEPOT LOCAL

- Métadonnées utilisateur
  - GIT nécessite 4 métadonnées pour enregistrer une version :
    - la date de l'enregistrement (automatique)
    - un message précisant l'objet de la version (renseigné au moment de l'enregistrement)
    - le nom et l'email de l'utilisateur de git
  - Ces deux dernières métadonnées sont configurées pour un dépôt ou tous ceux de l'utilisateur

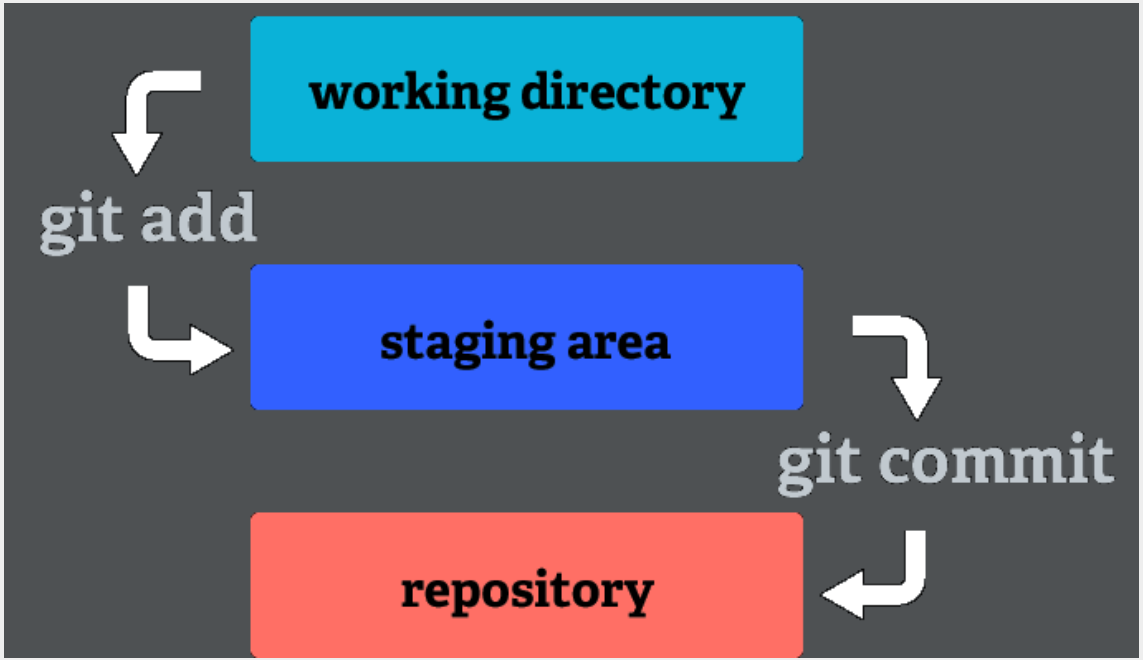
```
git config --local user.name my_name

git config --global user.email myemail@example.fr
```

  - La configuration locale est enregistrée dans working_copy/**.git/config**
  - La configuration globale est enregistrée dans **~/.gitconfig**

    « **~** » désigne le dossier utilisateur

CREATION DE COMMITS

- Enregistrement d'une version, ou « **commit** »



modifications dans la **copie de travail**

ajout des modifs dans l'**index** ou zone de préparation (.git/index)

validation des modifs dans le dépôt

=> création d'un **commit** (version)

CREATION DE COMMITS

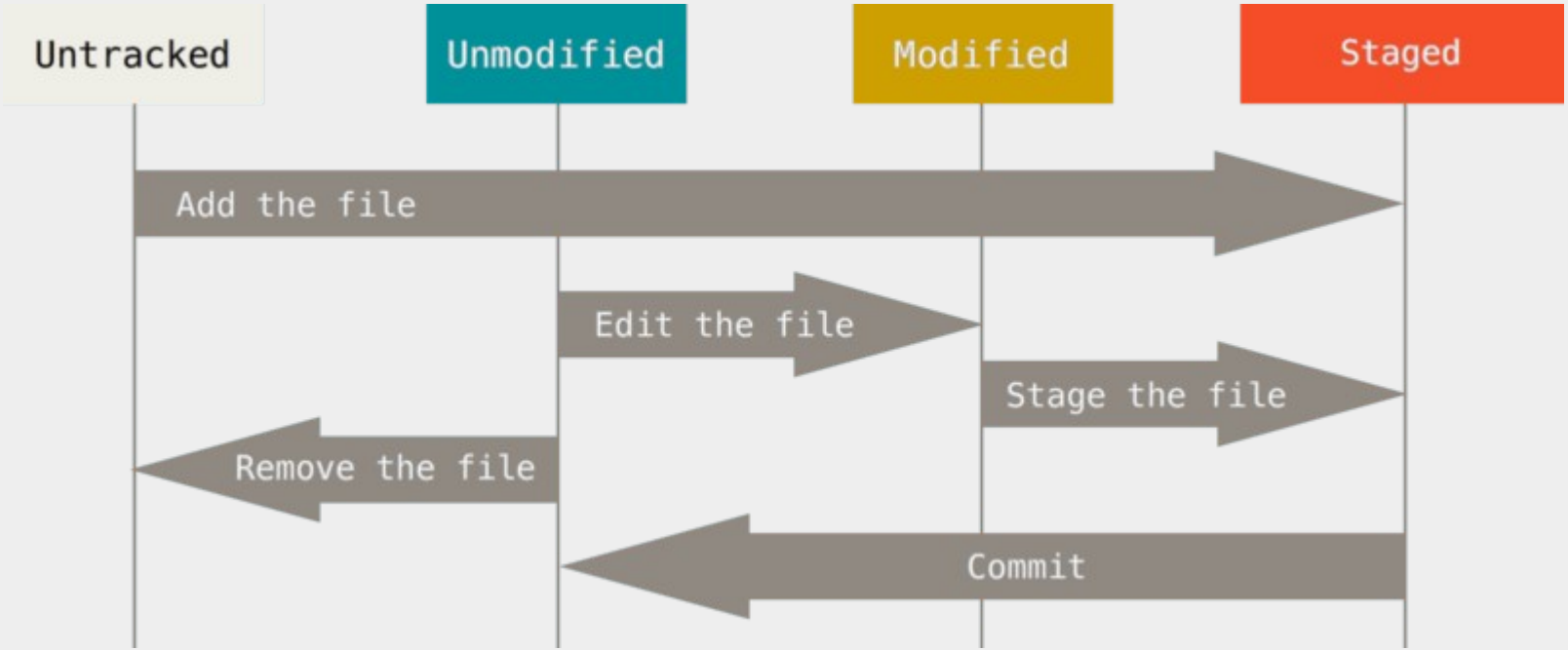
- Statuts des fichiers : **git status**
  - Permet de connaître l'état des fichiers dans la copie de travail par rapport au commit courant
  - Permet de connaître les fichiers présents dans l'index

```
git status

git status -s

git status --short
```

fr	en	Desc.
Non suivi	untracked	inconnu du commit courant
Non modifié	unmodified	Identique dans commit courant
modifié	modified	Différent dans commit courant
ajouté	staged	Ajouté à l'index
À supprimer	to_delete	À supprimer du commit
À renommer	to_rename	À renommer dans le commit

CREATION DE COMMITS

- Cycle des états



CREATION DE COMMITS

- **git add** < paths | . | * | -A >
  - Ajoute des fichiers dans l'index
  - **Obligatoire** pour les fichiers non suivis

- **git add -i** : mode interactif
  - 1 : s : status => git status (uniquement pour les fichiers déjà dans l'index ou « Modifiés »)
  - 2 : u : update => ajout de fichiers à l'état « Modifié »
  - 3 : r : revert => inverser les ajouts
  - 4 : a : add untracked => ajout de fichiers à l'état « non Suivi »
  - 5 : p : patch => ajout des modifs « **hunks** » dans les fichiers une à une (**git add -p**)
  - 6 : d : diff => git diff (cf infra)
  - 7 : q : quit => quitter

CREATION DE COMMITS

git add -p

- Permet d'ajouter à l'index certaines modifications, dites **« hunks »**, plutôt que les fichiers eux-mêmes
- Plusieurs options d'ajouts disponibles
  - **y, n :** oui, non et passer au hunk suivant
  - **a, d :** oui, non pour tous les autres
  - **q :** non et quitter
  - **s :** séparer les modifications du hunks en fonction de la présence de lignes non modifiées
  - **e :** éditer le fichier au moment de l'ajout
  - ...

CREATION DE COMMITS

- **git commit -m "message"**
  - Création d'un commit à partir du contenu de l'index
  - Si l'option m est omise, le terminal ouvre un **éditeur par défaut** pour éditer le message
  - git config --global **core.editor** notepad

- **git commit -a**
  - **Ajout auto** des fichiers à l'état « **Modifié** » et commit

CREATION DE COMMITS

- **git log [-p] [-n]**
  - Affiche les métadonnées de commit avec

    - **la révision <rev>** : **identifiant** unique hexadécimal obtenu par une **fonction de hachage (SHA-1)** sur le contenu + méta

    - la date de création (**--date=relative** pour une durée à partir de l'instant t)

    - le nom et l'email de l'auteur du commit

    - le message

  - L'option p affiche les **diffs** par rapport au commit précédent (cf infra)
  - L'option n affiche les n derniers commits

- **git log --format=" %Cred %h %Cgreen %s %n %ai" :** affichage en fonction d'un modèle (voir alias git)
- **git log --oneline :** commits sur une ligne
- **git log --stat** : commits avec la liste des fichiers modifiés

CREATION DE COMMITS

- Description d'un commit dans le dépôt .git

Les commits sont placés dans le dossier .git/objects

- Les dossiers sont classés en fonctions des deux premiers caractères de la révision

- **git cat-file -p <rev>** permet de d'afficher un commit et son contenu
  - ce contenu comprend les métadonnées, le commit parent, et un objet **tree** représentant l'état du dépôt

- **git ls-tree -r <rev>** permet de connaître le contenu d'un objet tree d'après sa révision
  - un **tree** comprend des **blobs** (représentant l'état des fichiers) et des **subtrees** (représentant des dossiers)
  - le tree liste pour chaque commit la **liste de tous les fichiers** du dépôt dans l'état du commit

- Les révisions sont générées à partir de la fonction **SHA-1** sur les contenus (blob, tree, commits)

CREATION DE COMMITS

Filtrer l'historique

- **git log < rev1 >^..< rev2 > : « .. »** signifie « entre », **« ^ »** signifie « compris »
- **git log --author < regex >** : par utilisateur
- **git log --since < iso date | duration > --until < iso date | duration >** : par date ou durées
- **git log --grep < regex >** : par regex sur le message
- **git grep [options] < regex > < rev >** : recherche les lignes contenant la regex dans les fichiers du commit pointé

  - **git grep < regex > $(git rev-list < branch | --all >)** : idem dans les commits d'une ou toutes les branches

  - options classiques :

  git config --global grep.extendedRexep = true (-E)

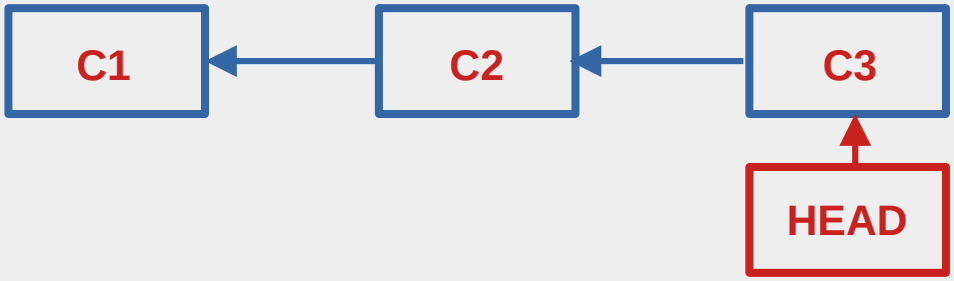
  git config –global grep.lineNumber = true (-n)

CREATION DE COMMITS

- Le fichier .gitignore
  - Fichier contenant les chemins de fichiers **invisibles pour git**
  - Fonctionnalités des chemins :
    - support partiel des expression régulières : classes **[ ]** ex : [cC]hat
    - ***** : n'importe quelle suite de caractères
    - ****** : n'importe quelle suite de sous dossiers
    - **/** en premier caractère : recherche depuis la racine du dossier
    - **!** en premier caractère : exception à une règle précédente
- Nettoyer le dépôt
  - **git clean [-n | -e | -x | -f]** : **supprime** les fichiers non suivis **« Untracked / U »**
    - après **confirmation (sauf -f)** et en préservant les fichiers git-ignorés
  - **-n** simule les suppressions « dry run »
  - **-e** < regex > permet d'exclure des regex de la suppression
  - **-x** inclut les fichiers git-ignorés (compatible avec -e)

CREATION DE COMMITS

- Le pointeur HEAD
  - **Fichier du dépôt qui référence le commit courant**

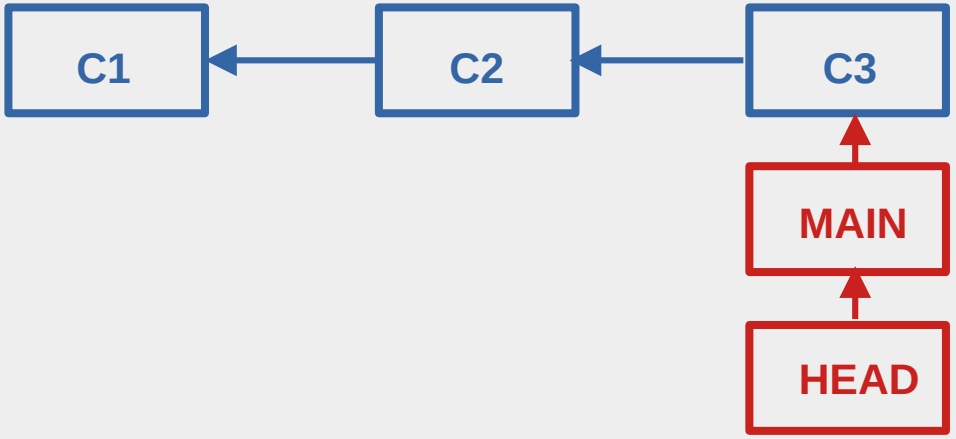


  - On utilise le HEAD dans les commandes git à la place de la révision

    - git show HEAD : affiche le commit courant
    - git diff C1 : modification entre la copie de travail et C1
    - git diff **HEAD~2** : idem en calculant un **nb de commit avant le HEAD**

CREATION DE COMMITS

- Le pointeur de branche
  - Les commits s'enchaînent sur une **branche** nommée « master » ou « main » par défaut
  - Un pointeur de branche dans le dossier **.git/refs** contient la révision du commit courant
  - En réalité, le pointeur HEAD contient le **chemin du pointeur de branche** courante



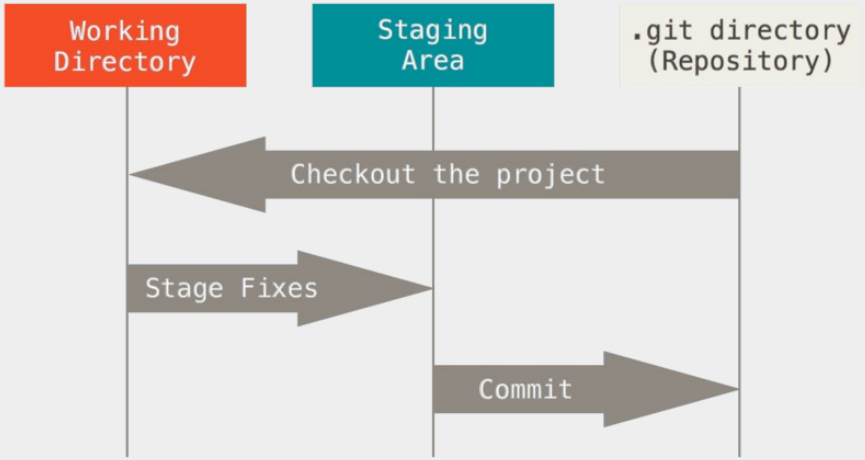
COMMANDES D'INVERSION

- Les commandes pour inverser une action dans git
  - Inverser des modifications dans la copie de travail :
    - **git checkout <rev | HEAD> -- <paths>**
  - Supprimer ou renommer des fichiers dans git
    - **git rm [-r] [---cached] <paths>**
    - **git mv <old_path> <new_path>**
  - Inverser des mises à l'index
    - **git reset -- <paths>**
  - Inverser des commits
    - **git reset [--soft | --mixed | --hard] <rev | HEAD~n>**
    - **git revert <rev | HEAD~n>**

COMMANDES D'INVERSION

git checkout

- git checkout est une commande complexe, dont **l'effet varie selon le type d'argument**
  - checkout sur des fichiers
  - checkout sur des commits
  - checkout sur des branches
- Cela dit, un checkout a toujours les effets suivants :
  - **déplacement du pointeur HEAD** (sauf pour les fichiers)
  - **écrasement de la copie de travail** à partir d'un commit du dépôt



COMMANDES D'INVERSION

- Inversion de modifications
  - git checkout **HEAD -- <paths>** permet d'annuler les modifs en cours dans la copie de travail
    - écrasement du / des fichiers avec la version du commit courant du dépôt
    - on peut ôter HEAD car c'est l'argument « rev » par défaut => **git checkout -- <paths>**
    - **les modifications sont perdues !!!**
  - git checkout **<rev | HEAD~n> -- <paths>**
    - permet d'écraser le/les fichiers avec la version d'un commit particulier
    - attention, les fichiers écrasés sont placés dans l'index !!!

COMMANDES D'INVERSION

- Supprimer un fichier du commit courant
  - **git rm [-r] [--cached] <paths>**

    - l'option **r** permet de supprimer les dossiers et leur contenu
  - Une fois la commande lancée

    - le ou les fichiers sont supprimés de la copie de travail …

    - … sauf si l'option **--cached** est présente (ex : oubli d'ajout au .gitignore)

    - un **ordre de suppression** est placé dans l'index.

    - La suppression sera effective au **prochain commit**

  - Le fichier n'est **pas complètement supprimé** du dépôt

    - git rm permet de ne plus ajouter les « **blobs** » des fichiers supprimés aux « **trees** » des prochains commits

    - ces blobs sont toujours contenus dans les trees de commits précédents

    - on peut donc **restaurer** ces fichiers dans la copie de travail avec des commanes d'inversion (checkout, reset, revert)

COMMANDES D'INVERSION

- Renommer / déplacer un fichier dans le dépôt

  - **git mv <old_path> <new_path>**

  - Une fois la commande lancée :

    - le fichier est renommé / déplacé dans la copie de travail
    - un ordre de renommage / déplacement est placé dans l'index.
    - L'opération sera **effective dans le dépôt au prochain commit**

COMMANDES D'INVERSION

- Désindexer un fichier

  - **git reset [--mixed] -- < paths > :** retire un fichier de l'index

    - « --mixed » est l'option par défaut de git reset : on peut l'occulter

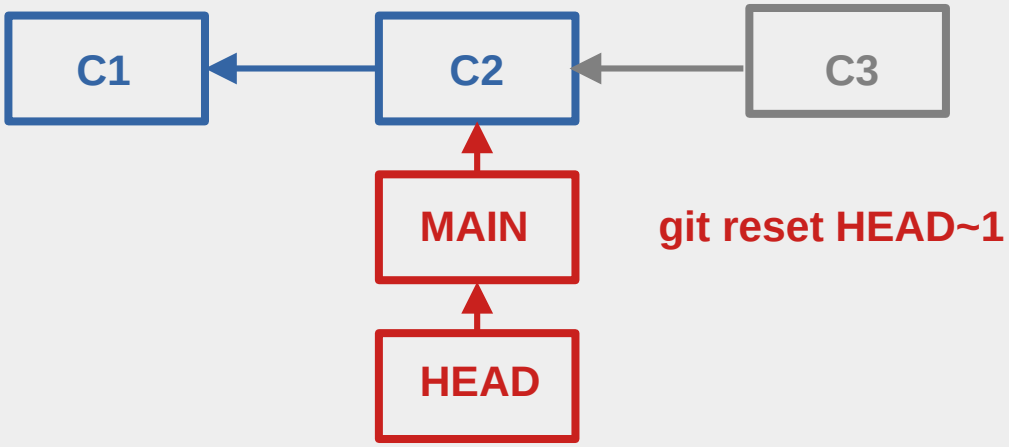
COMMANDES D'INVERSION

- Reset : Supprimer un / des commits de l'historique
  - **git reset [--soft | --mixed | --hard] <rev | HEAD~n>**
  - **Déplace le pointeur HEAD** sur le commit en paramètre
  - **Supprime les commits** situés devant le nouveau HEAD de l'historique
  - **Conserve ou écrase** la copie de travail et l'index selon l'option choisie

Option	Dépôt	Index	Copie de travail
--soft	HEAD déplacé	Conservé	Conservée
--mixed (defaut)	//	vidé	Conservée
--hard	//	//	Ecrasée

COMMANDES D'INVERSION

- Inverser un reset



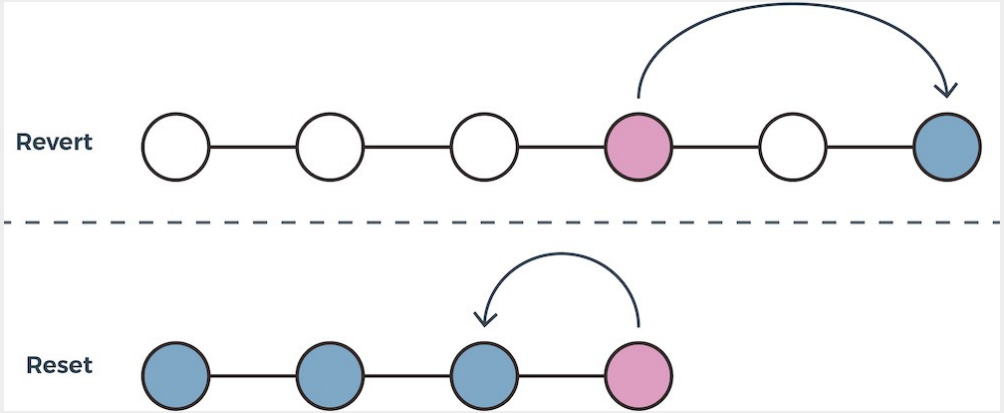
- C3 a disparu de l'historique mais pas du dépôt.
- On peut le retrouver grâce à la commande **git reflog** qui enregistre les déplacements de HEAD
- **git reset HEAD@{ n }** replace HEAD sur le commit correspondant au nième précédent déplacement
- permet donc **d'annuler un reset**

COMMANDES D'INVERSION

- Revert : inverser un commit

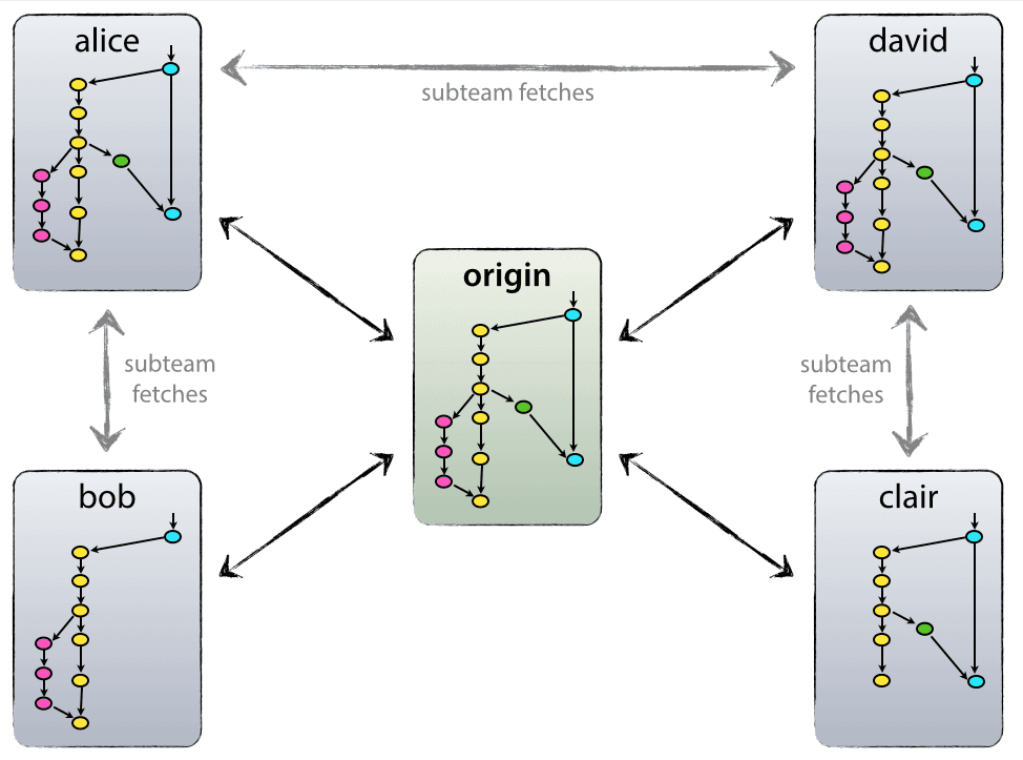
  - **git revert < rev > ou HEAD~n**

    - crée unnouveau commit qui inverse les modifs du commit en paramètre
    - ouvre un éditeur par défaut pour le message ou**--no-edit** pour laisser le message par défaut
    - **--no-commit** pour inverser les modifs sans créer le commit



DEPOT DISTANT

- Communication entre dépôt
  - Les dépôts git peuvent communiquer deux à deux car chacun possède un historique propre
  - On crée souvent un dépôt central de référence, que l'on nomme par convention **origin**



DEPOT DISTANT

- Sécurisation des échanges par clés SSH privée/publique
  - La commande **ssh-keygen [-t < prot > -f <path> -N <passphrase>]** génère une paire de clés permettant d'authentifier une connexion entre dépôts
  - **< prot >** est le protocole **RSA** par défaut (gère la taille de la clé)
  - On peut rajouter une **passphrase** pour protéger la connexion, qui sera demandée systématiquement sur les git pull et git push
  - La **clé publique** est envoyée sur le **serveur** de connexion
  - La **clé privée** reste du côté du **client** de connexion, on la place dans le dossier **~/.ssh**
  - Des commandes git pilotent le **client ssh** local : on doit configurer ce client pour utiliser

    la clé privée liée au serveur de connexion distant

```
# dans le fichier ~/.ssh/config
Host <distant.server.url>
 IdentityFile "/c/users/<username>/.ssh/<priv_key>"
 UserKnownHostsFile /dev/null
 StrictHostKeyChecking no
```

DEPOT DISTANT

- **git remote add < repo_name > < repo_url >**
  - Ajout dans un dépôt local de l'url d'un dépôt distant
    - représenté par un mot clé arbitraire : « **origin** » par convention pour le dépôt distant principal
  - Les commandes **git remote [add | remove | rename ...]** administrent les cnx aux dépôts distants

- Échanges de commits
  - Un dépôt peut demander ou « tirer » des nouveaux commits depuis un dépôt distant : **FETCH**
  - Un dépôt peut pousser des nouveaux commits sur un dépôt distant : **PUSH**

- **git clone [https://URL ou user@host/path/to/repo.git]**

  Création par clonage d'un dépôt local depuis un dépôt distant via **HTTPS ou SSH**

  Le dépôt cloné peut communiquer avec le dépôt distant

DEPOT DISTANT

- **git fetch < repo_name > < branch_name >**
  - Création / maj en local d'une branche en lecture seule **repo_name/branch_name**
  - Tire les nouveaux commits de la branche distante sur cette branche de **suivi / tracking / upstream**
  - Permet **d'observer l'activité distante** sans interférer avec le travail local
  - **git fetch --all** permet de mettre à jour toutes les branches de suivi à partir des dépôts distants

- **git checkout <rev | HEAD~n>**
  - Déplace le pointeur HEAD **directement sur un commit** sans pointer de branche : **mode détaché**
  - On peut alors **observer** le commit dans la copie de travail
  - On peut **expérimenter** en créant des commits **en dehors des branches**
  - Si l'expérimentation est concluante, on peut créer une branche à partir du commit
  - On retourne sur la branche courante avec git checkout <branch_name>

DEPOT DISTANT

- **git pull <repo_name> <branch_name>**
  - **Tire** les commits distants,
  - Puis **fusionne** la branche de suivi avec la branche locale de même nom
  - **git pull = git fetch + git merge** (cf infra)

- **git push <repo_name> <branch_name>**
  - Pousse les nouveaux commits sur le dépôt et la branche distante
  - **git push -u <repo_name> HEAD** permet de configurer un dépôt et une branche distante par défaut pour la branche locale, i.e utiliser **git push et git pull sans arguments**

DEPOT DISTANT

- Synchronisation des historiques
  - Deux dépôts peuvent échanger des nouveaux commits **si leurs historiques sont cohérents**
  - Il faut donc faire attention aux commandes qui **réécrivent l'historique**
  - **Ex : ne jamais faire de reset sur des commits déjà poussés !!!**
  - **git push [-f | --force]** permet d'écraser l'historique distant, à utiliser avec précaution !
  - **git pull --allow-unrelated-histories** autorise la fusion de commits provenant de projets (et donc d'historiques) différent. Acceptable en début de projet
  - Conseil : penser à effectuer un git pull avant un git push, pour rapatrier des commits distants avant de pousser le siens => « **synchroniser** »

DEPOT DISTANT

- Conflits de fusion
  - Lors d'un git pull (en fait git merge), git compare les historiques des deux branches à fusionner
  - Si un commit distant contient une version différentes de lignes présentes dans un commit local non poussé, alors git ne peut pas choisir entre ces deux versions => il y a **échec de la fusion, conflit**

```
<<<<<<< HEAD (local)
1. introduction à l'agilité
  * historique
  * méthodes agiles
2. rappels sur le lean IT
3. La Culture Devops
=======
1. rappels sur le lean IT
2. introduction à l'agilité
  * historique
  * méthodes agiles
3. DevOps
>>>>>>> 752b181947cac474513e8935ba78c1bd78c094d2 (distant)
```

DEPOT DISTANT

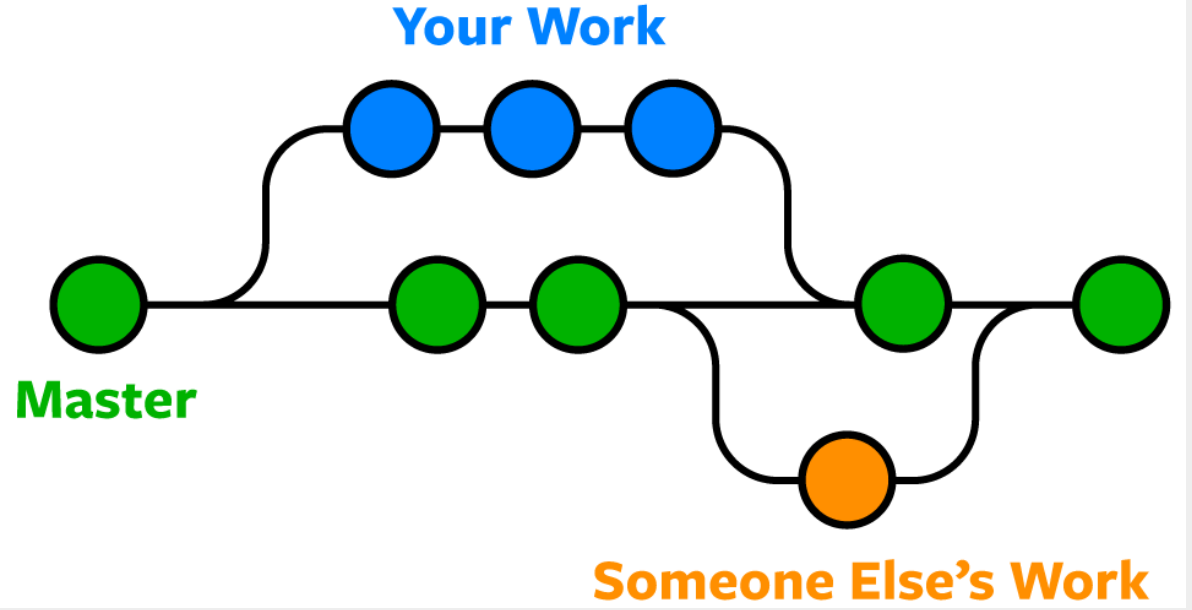
- Résolution de conflits
  - Les responsables des différentes versions doivent décider de la forme correcte du code

Une fois le code corrigé : **git add < files > && git commit -m "fixed ..."**

- ajouter le fichier en conflit et **créer un commit** pour marquer le conflit comme résolu
- un historique est recréé arbitrairement avec le **commit de fusion** en avant
- on pousse le commit de fusion qui sera accepté normalement par les autres dépôts

BRANCHES

- Principe
  - On peut créer autant de branches que désiré dans un dépôt
  - Les branches sont utilisées pour isoler le travail
    - sur une fonctionnalité : **« feature »** branch
    - sur une correction : **« fix ou hotfix »** branch
    - sur un code candidat au déploiement : **« release »** branch



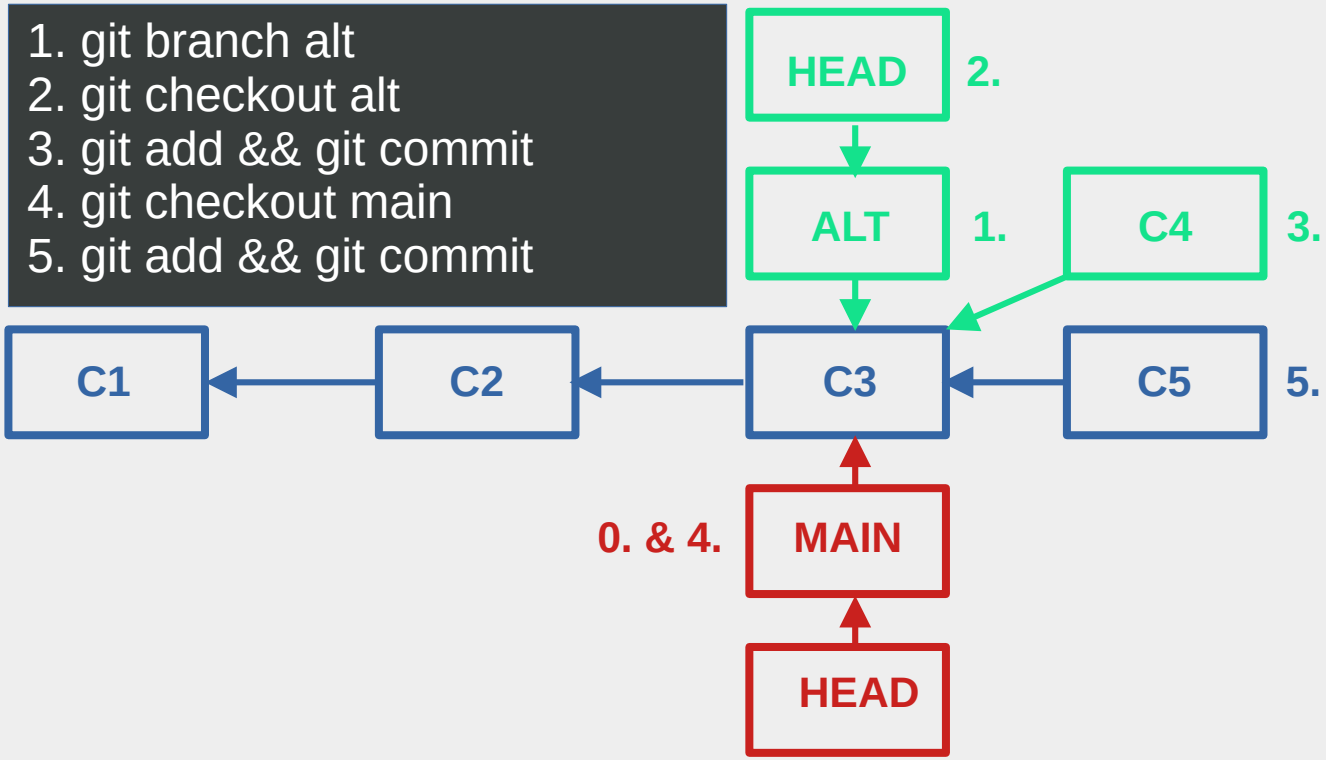
BRANCHES

- git branch
  - **git branch <branch_name> [<rev | HEAD~n>] :** création d'un pointeur de branche sur un commit
  - **git branch -v :** voir les branches et la branche courante
    - « -rv » : voir les branches de suivi
    - « -av » : voir toutes les branches
  - **git branch -d <branch_name> :** suppression d'une branche
    - Il faut se trouver sur une autre branche
    - « -D » pour supprimer une branche qui contient des commits non fusionnés
  - **git branch -m <new_name> :** renommer la branche courante

- git checkout
  - **git checkout <branch_name> :** basculer sur une autre branche
  - **git checkout -b <branch_name> :** création et basculement en une commande

BRANCHES

- Création de branches
  - Une branche est crée à partir d'un commit et d'une branche de **base**
  - La nouvelle branche **partage l'historique** de sa branche de base jusqu'à son commit de base



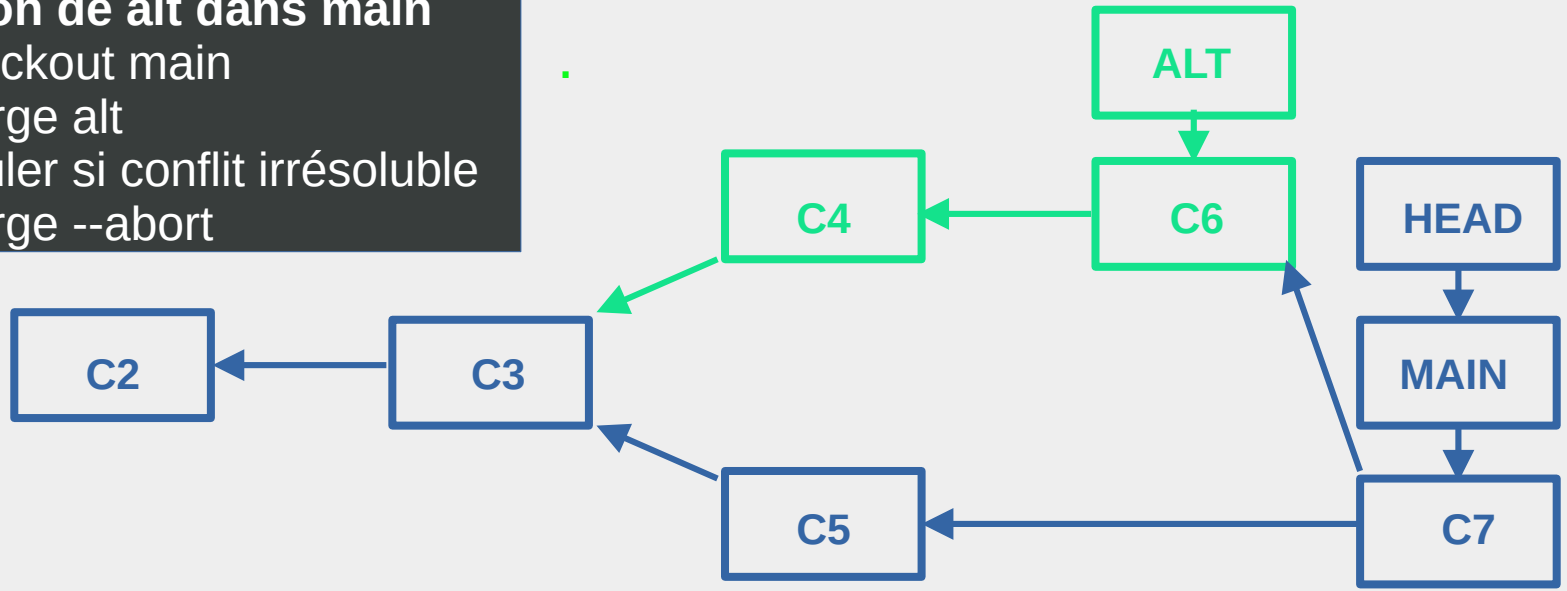
BRANCHES

- Remiser temporairement du code
  - On doit parfois changer de branche **de façon intempestive**, liée à un évènement extérieur
  - **Or, git checkout** va écraser la copie de travail qui peut contenir des modifs non commitées
  - Pour éviter de déplacer indûment ou commiter du WIP, on peut sauvegarder ces modifs via **git stash**
  - Les modifs sont enregistrées dans une **pile indexée** (LIFO : Last In, First Out)
    - On peut donner un nom aux modifs dans le stash : **git stash push -m name**
    - On retrouve un stash d'après son index dans la pile **"stash@{index}"** ou son nom **stash^{/name}**
    - il faut ajouter l'option **« -u »** pour remiser les fichiers **« non Suivis »**
- Opérations
  - git stash list : affiche la pile de modifs
  - **git stash pop [<stash_name>]:** dépile les modifs et les rejoue sur HEAD
  - **git stash apply <stash_name> :** applique les modifs sans les enlever de la pile
  - **git stash drop <stash_name> :** élimine les modifs
  - **Git stash clear :** vide le stash

BRANCHES

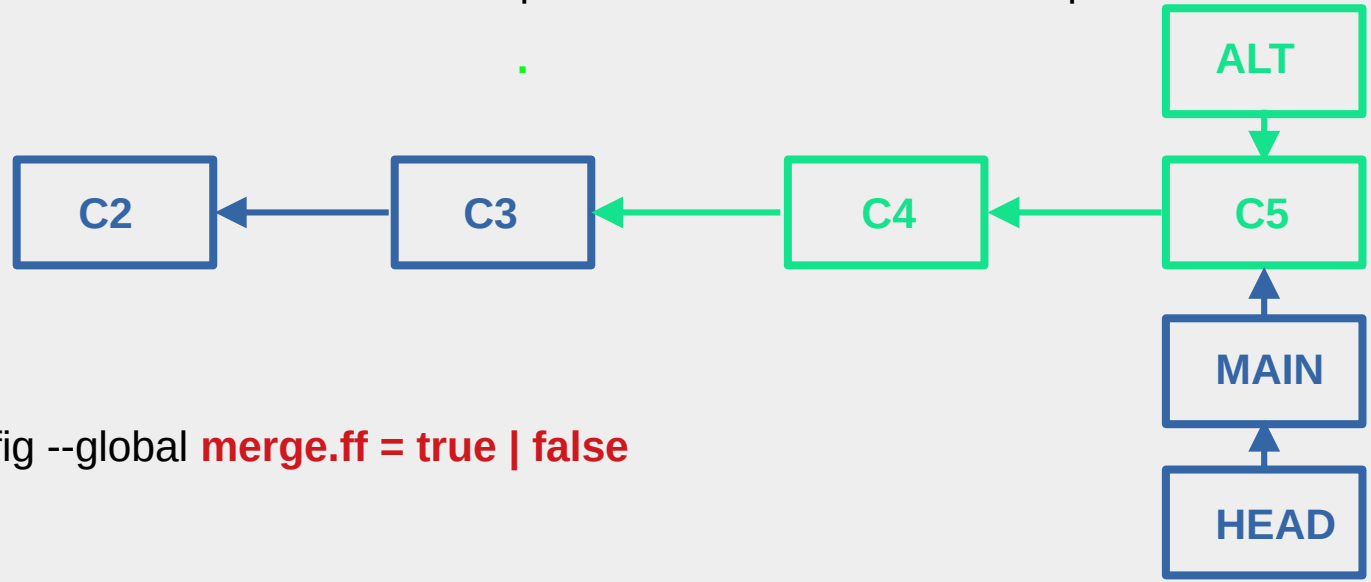
- git merge : fusion de branches
  - Une fusion crée un commit sur la branche courante, dit **commit de fusion « merge commit »**
  - Il contient la somme des modifs de la branche à fusionner et des commits de la branche courante créés après la branche à fusionner, avec de **possibles conflits à gérer**

```
# fusion de alt dans main
git checkout main
git merge alt
# annuler si conflit irrésoluble
git merge --abort
```



BRANCHES

- fusion « fast-forward », « ff »
  - Par défaut, si aucun commit n'a été créé sur la branche de réception, le git merge ajoute directement les commits de la branche à fusionner devant ceux de la branche de réception **sans commit de fusion**
  - Ce comportement peut être désactivé avec l'option **git merge --no-ff alt**
  - Intérêt du ff : simplifier l'historique des merges et des pulls (**git pull [--ff]**)
  - Intérêt du no-ff : conserver l'historique et **revert une branche** à partir du commit de fusion



git config --global **merge.ff = true | false**

BRANCHES DISTANTES

- Inverser une fusion de branche
  - Si un commit de fusion existe (**--no-ff** si besoin)
  - on peut effectuer un **revert** pour revenir sur le code pré-fusion
  - Un commit de fusion ayant **2 commit parents**, utiliser l'option **m « mainline » de revert**
  - **git revert -m 1 <rev | HEAD> (1 désigne la ligne recevant la fusion)**

- Faire disparaître un commit de fusion
  - Si le commit de fusion n'a pas encore été poussé (**réécriture de l'historique!**)
  - On peut reset un commit de fusion avec **git reset --merge < rev | HEAD >**

BRANCHES DISTANTES

- git push
  - git push <repo_name> <branch_name> **: création ou mise à jour**
  - git push **-d | --delete** <repo_name> <branch_name> **:** suppression

- git checkout **--track** <repo_name>/<branch_name> :
  - Après un git fetch, sur une **branche de suivi**
  - Création d'une branche locale avec basculement et push / pull par défaut (cf git push -u)

- git branch **-u | --upstream** <repo_name>/<branch_name> :
  - idem sans basculement

TAGS

- Tag léger
  - Un tag est une **étiquette** associée à un commit (ou HEAD par défaut) **en dehors de toute branche**
  - On l'utilise pour créer une **release** désignant une **version** – Ex : « v1.0 »
  - **git tag <tag_name> < rev | HEAD >** : création du tag

- Tag annoté
  - Un tag léger se contente de pointer sur un commit
  - Le tag annoté enregistre les métadonnées du tag en plus du commit associé
    - auteur et date de création du tag
    - message spécifique au tag
  - **git tag -a <tag_name> < rev | HEAD > -m "msg"**
  - **git show <tag_name> :** affichage des méta du tag en + des données du commits

TAGS

- Opérations sur les tags
  - **git tag -d < tag_name > :** suppression locale
  - **git push [-d] < repo_name > < tag_name > :** gestion des tags distants (envoi, suppression)
  - **git tag -af <tag_name> :** déplacement d'un tag existant sur un nouveau commit
  - **git archive --prefix=v1.0/ -o v1.0.tar.gz v1.0 :** export du tag v1.0 dans l'archive v1.0.tar.gz

REECRITURES D'HISTORIQUE

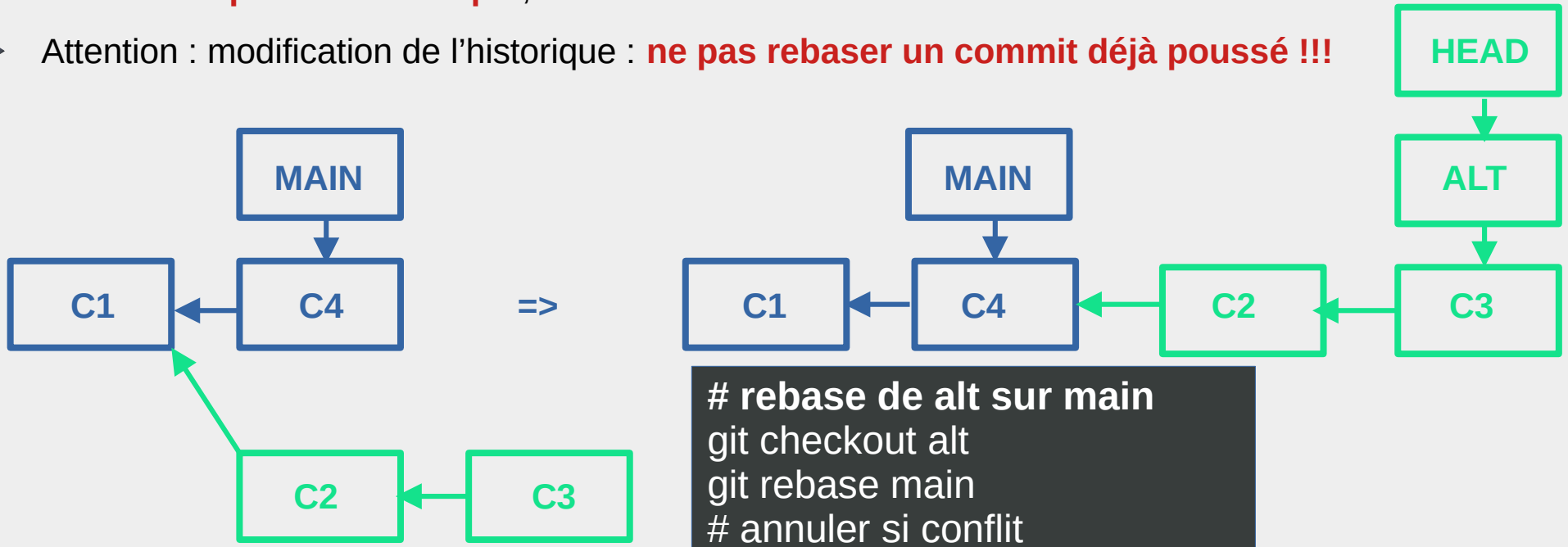
- Les commandes de réécritures sont possible lorsqu'elles concernent
  - Des commit **non encore poussés** sur d'autres dépôts
  - Des commit poussés sur des branches dont on a l'**exclusivité (feature, fix)**

    - on peut alors utiliser le git push -f pout mettre à jour l'historique distant

- git commit --amend
  - Ajout du contenu de l'index au dernier commit, plutôt que création d'un nouveau commit
  - Permet de rattraper un commit incomplet ou incohérent

REECRITURES D'HISTORIQUE

- git rebase
  - Le **rebasage** entre branches est une alternative à la fusion
  - Il consiste à **avancer le commit de base** d'une branche sur un commit (le plus récent par défaut) d'une autre branche
  - Intérêt : **simplifier l'historique**, éviter les commits de fusions
  - Attention : modification de l'historique : **ne pas rebaser un commit déjà poussé !!!**




```
# rebase de alt sur main
git checkout alt
git rebase main
# annuler si conflit
git rebase --abort
```

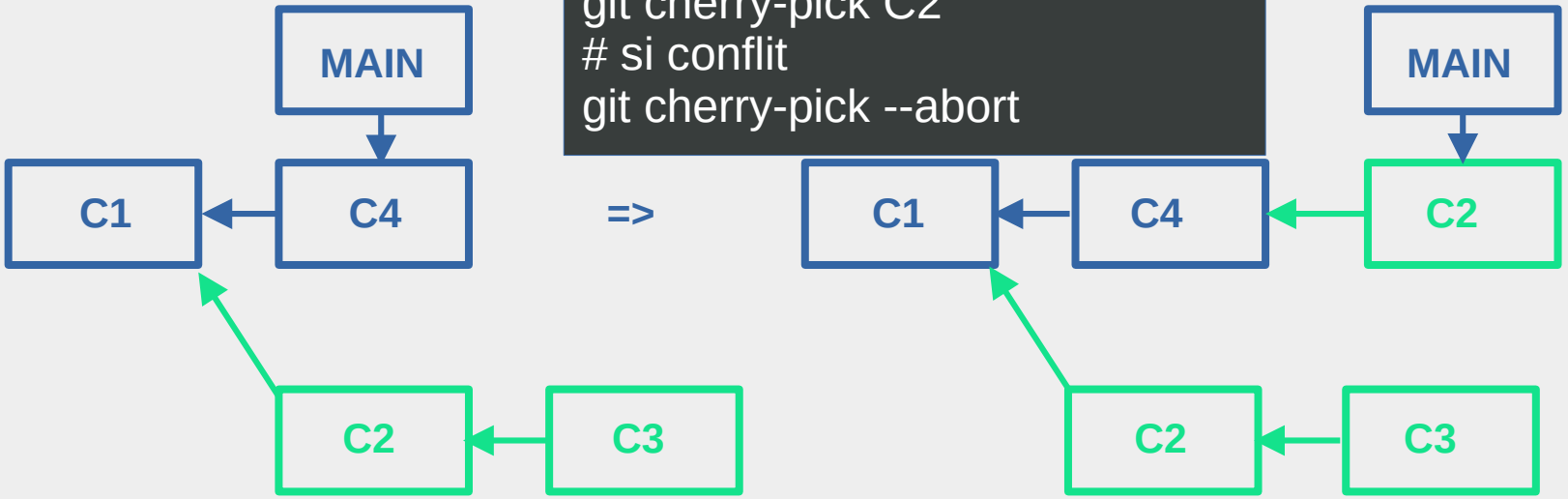
REECRITURES D'HISTORIQUE

- git rebase : conflits
  - Le **rebasage** met à jour les commits déplacés avec les nouveaux commits de base
    - des conflits peuvent donc advenir
  - Quand un conflit survient, le rebasage s'intterrompt et permet trois développement
  - La résolution : modifications + **git add + git rebase --continue** pour passer à l'étape suivante
  - L'annulation du rebase : **git rebase –abort**
  - Le contournement : si plusieurs étapes reproduisent les mêmes conflits sur les mêmes fichiers, **git rebase --skip** pour déplacer la résolution sur un prochain commit

REECRITURES D'HISTORIQUE

- git cherry-pick
  - Le **cherry-pick** consiste à appliquer un commit particulier d'une branche sur une autre
  - Intérêt : récupérer un **commit d'une branche abandonnée**, ou **patcher une branche de release** à partir d'un commit de correction sur une branche principale

```
# cherry-pick de C2 sur main
git checkout main
git cherry-pick C2
# si conflit
git cherry-pick --abort
```



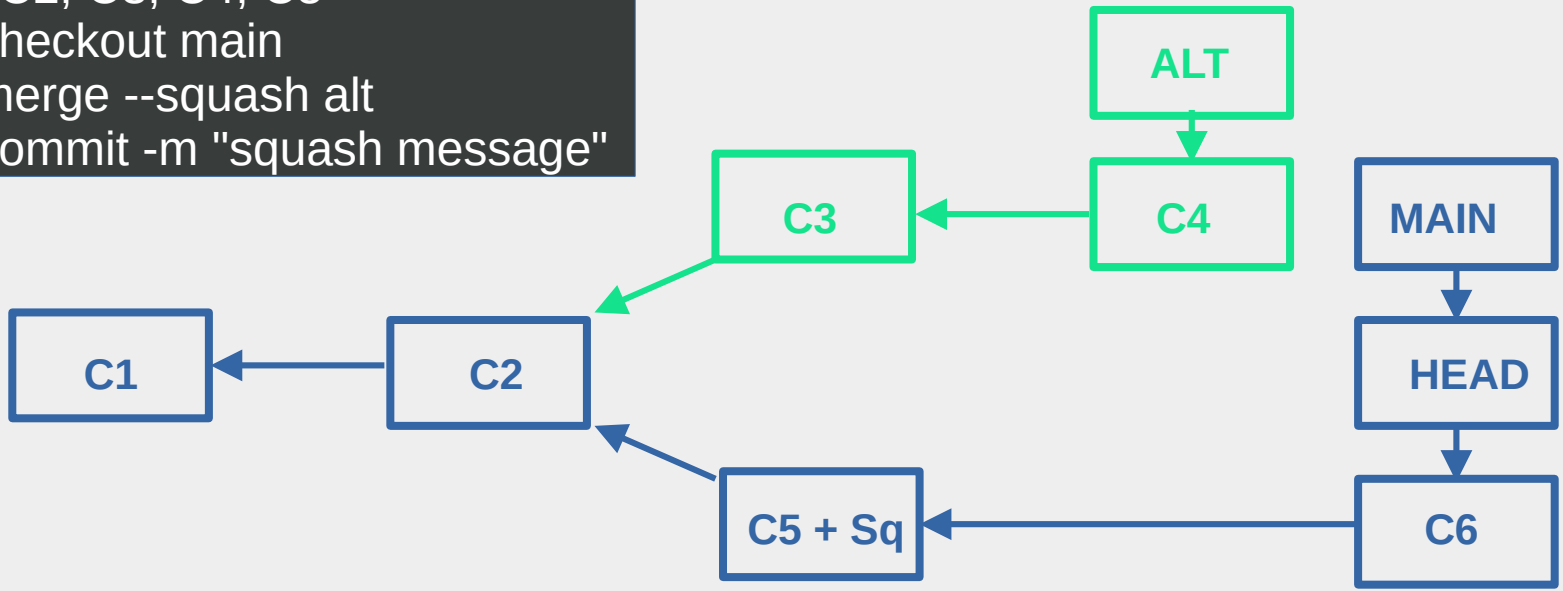
REECRITURES D'HISTORIQUE

- git rebase -i : « git commit --amend automatisé »
  - Cette commande exécute un rebase **commit par commit** de façon interactive en proposant plusieurs choix de **réécriture** pour chaque commit, dans l'éditeur par défaut
  - On l'utilise surtout pour réécrire l'historique sur la branche courante
    - **git rebase -i < hash >^ : « ^ »** signifie que le commit renseigné est inclus dans la liste
  - Actions possibles pour un commit :
    - **p : pick** : sélectionner le commit tel quel
    - **r : reword** : réécrire le message de commit dans l'éditeur par défaut
    - **e : edit** : modifier complètement le commit, ferme l'éditeur et stoppe le rebase le temps des modifs
    - **s : squash** : fusionner le commit avec le précédent et fusionner les messages
    - **f : fixup** : idem mais suppression du message du commit
    - **drop** ou éliminer une ligne : élimine un commit
  - En cas d'édition (e) :
    - faire les modifs (ajout, modif, suppression, renommage) suivi de **git add && git commit --amend**
    - reprendre l'exécution du rebase avec **git rebase --continue**

SQUASH

- git merge --squash : « fusion » de commits
  - Transfert des modifs de la branche à fusionner sur le commit courant de la branche de réception
  - Reste à créer le commit validant ces modifs et éventuellement supprimer la branche à fusionner

```
# squash de alt dans main
0. C1, C2, C3, C4, C5
1. git checkout main
2. git merge --squash alt
3. git commit -m "squash message"
```



ANNEXES

Alias de commandes réels

- Accélérer les écritures de commandes grâce à la section alias de la config

  - git config --global **alias.[my_alias] [git_cmd]**

- Alias usuels :

  - st => status
  - ci => commit
  - co => checkout
  - br => branch
  - ll => "log --oneline"
  - graph => "log --oneline --all –graph"

- Utilisation

  - git st, git ci, git ll -5, ...

ANNEXES

- Alias via le shell ou git-bash
  - Créer des alias via le fichier ~/.bashrc
    - **alias gst="git status"**
    - ...
  - Ajouter le chargement du fichier .bashrc dans le fichier ~/.bash_profile
  - … lui même chargé automatiquement à l'ouverture d'un shell ou git-bash
    - **test -f ~/.bashrc && source ~/.bashrc**

ANNEXES

- Ajout d’une clé privée avec passphrase dans le ssh-agent
  - dans ~/.bashrc

    - **eval `ssh-agent -s`**

    - **ssh-add** ~/.ssh/<privkey>

  - Au lancement d’un terminal (y compris git bash sous windows)

    - la passhphrase est demandée une fois pour toute la session du terminal