

# Bibliographie et inspirations

- Scott Chacon : <https://git-scm.com> (graphes et méthodologies)
- Référence en ligne : <http://gitref.org>
- Git Pocket Guide par Richard Silverman
- Man gittutorial (<https://git-scm.com/docs/gittutorial>)
- Cheat Sheets (aide-mémoire) :
  - <https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet>
  - [http://ndpsoftware.com/git-cheatsheet.html#loc=local\\_repo](http://ndpsoftware.com/git-cheatsheet.html#loc=local_repo);
- Internet ...

# Les enjeux du développement

- Le travail en équipe:
  - accès aux développements des autres
  - partage de l'information sur le développement
- L'historisation du travail
  - voir les dernières modifications du projet (sur un certain référentiel)
  - remonter dans le temps et voir les modifications antérieures
  - voir les différences entre les modifications
  - comprendre pourquoi certaines modifications ont été réalisées
    - 💡 Besoin de répondre aux question : qui ? quand ? quoi ? pourquoi ?
- La diffusion du code

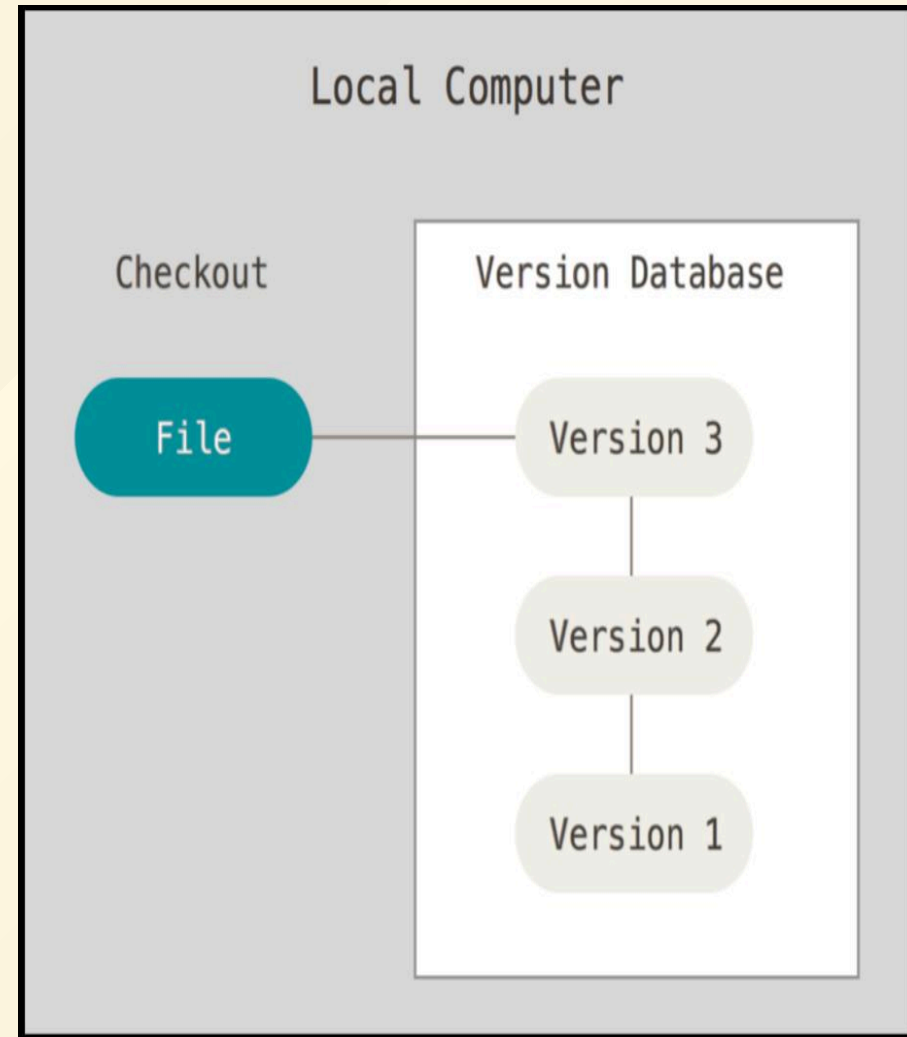
? Comment avez-vous échanger du code jusqu'à maintenant ?

? Comment avez-vous échanger du code jusqu'à maintenant ?

Solution possible: les systèmes de contrôle de version !

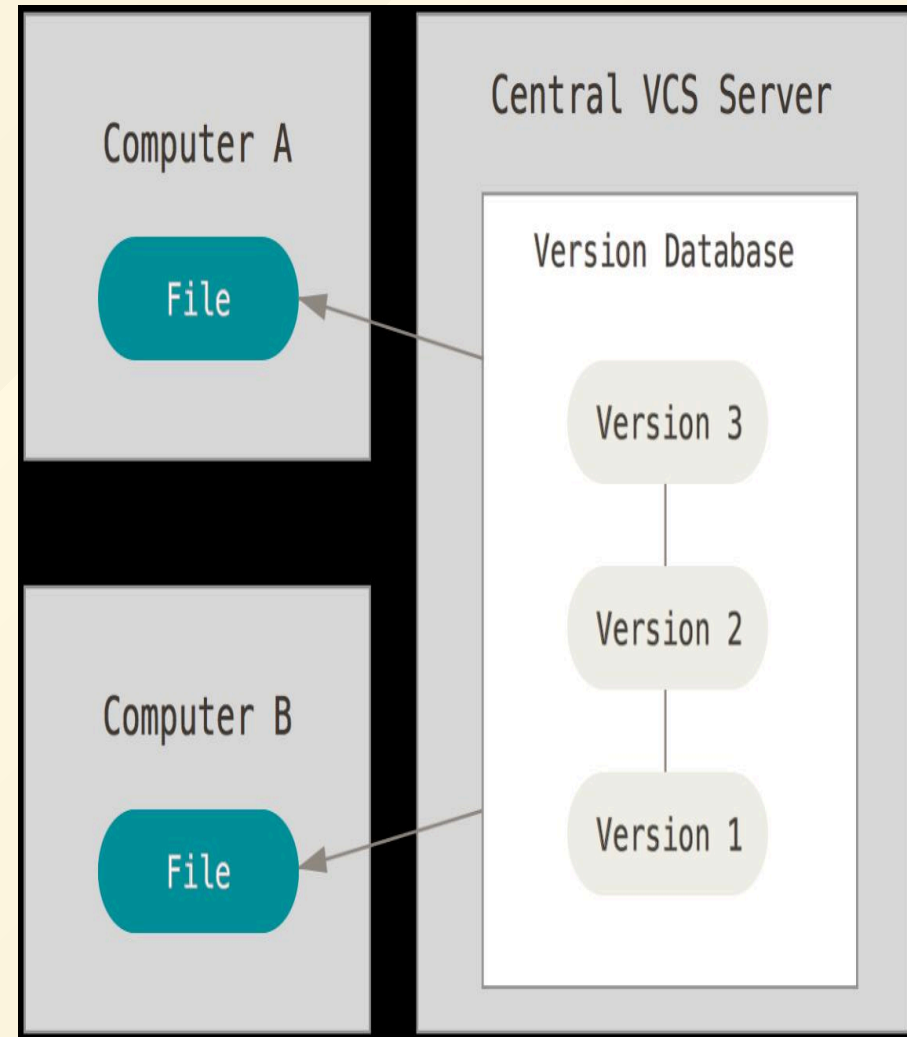
# RCS

- Opère sur des fichiers uniques
- Introduction d'un nouveau fichier d'extension « ,v » contenant des métadonnées
- ☒ Sauvegarde des révisions ne dépend pas d'un dépôt central
- ☐ L'historique des révisions peut être modifié par un utilisateur
- ☐ Un seul utilisateur peut travailler sur un fichier à la fois







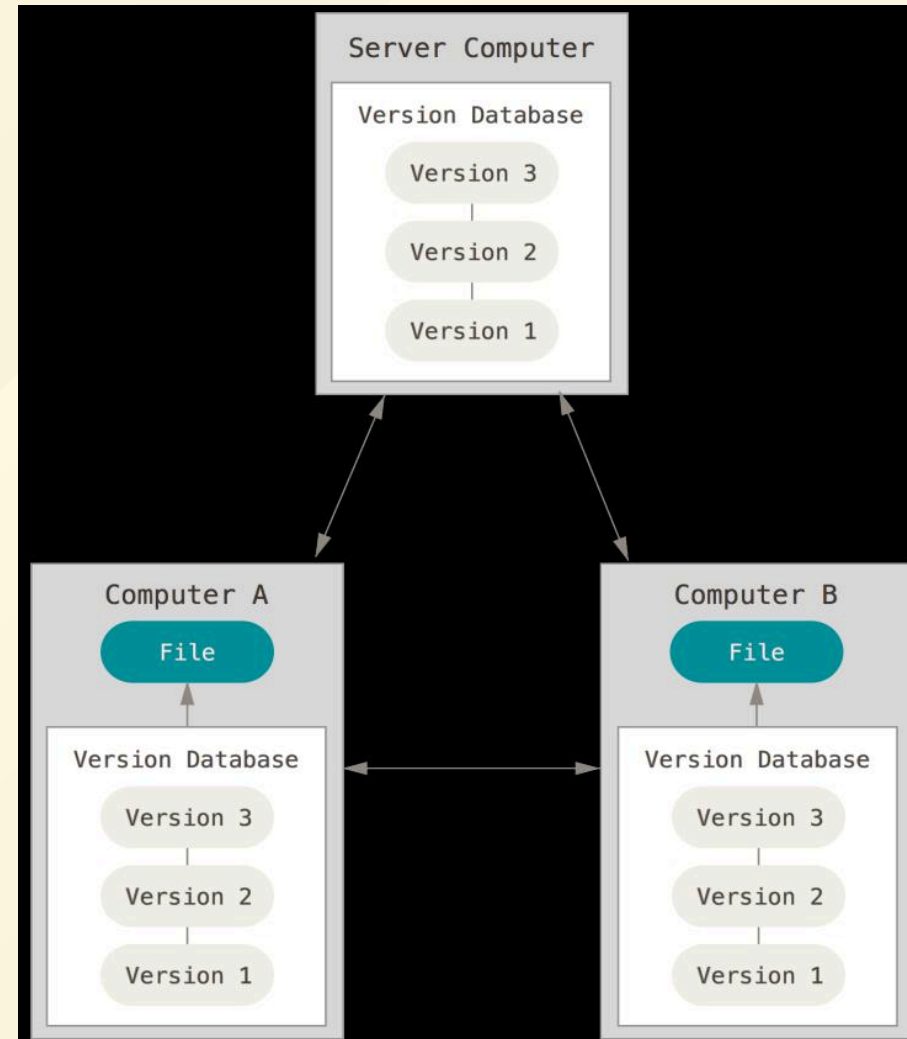
# SVN

- Système centralisé : stockage du code et métadonnées sur un serveur
- ☒ Enregistrement des modifications par *delta differencing*
- ☐ Pas pratique pour le travail *hors ligne* (des enregistrement en un)
- ☐ Surcoût lié au réseau
- ☐ Empêche l'enregistrement de modifications local sans partager



# Git

- Système distribué né pour gérer les sources de Linux
-  Actions extrêmement rapide
  - Nécessite l'accès au disque dur
  - N'utilise le réseau que pour publier / télécharger
-  Permet le développement local
  -  Limite la casse en cas de mauvaise manipulation
-  Any ?



# Git

- Gestion des binaires larges (solution `git-lfs`)
- Dans le cas de projet à historique long, le téléchargement peut prendre beaucoup de temps et de place



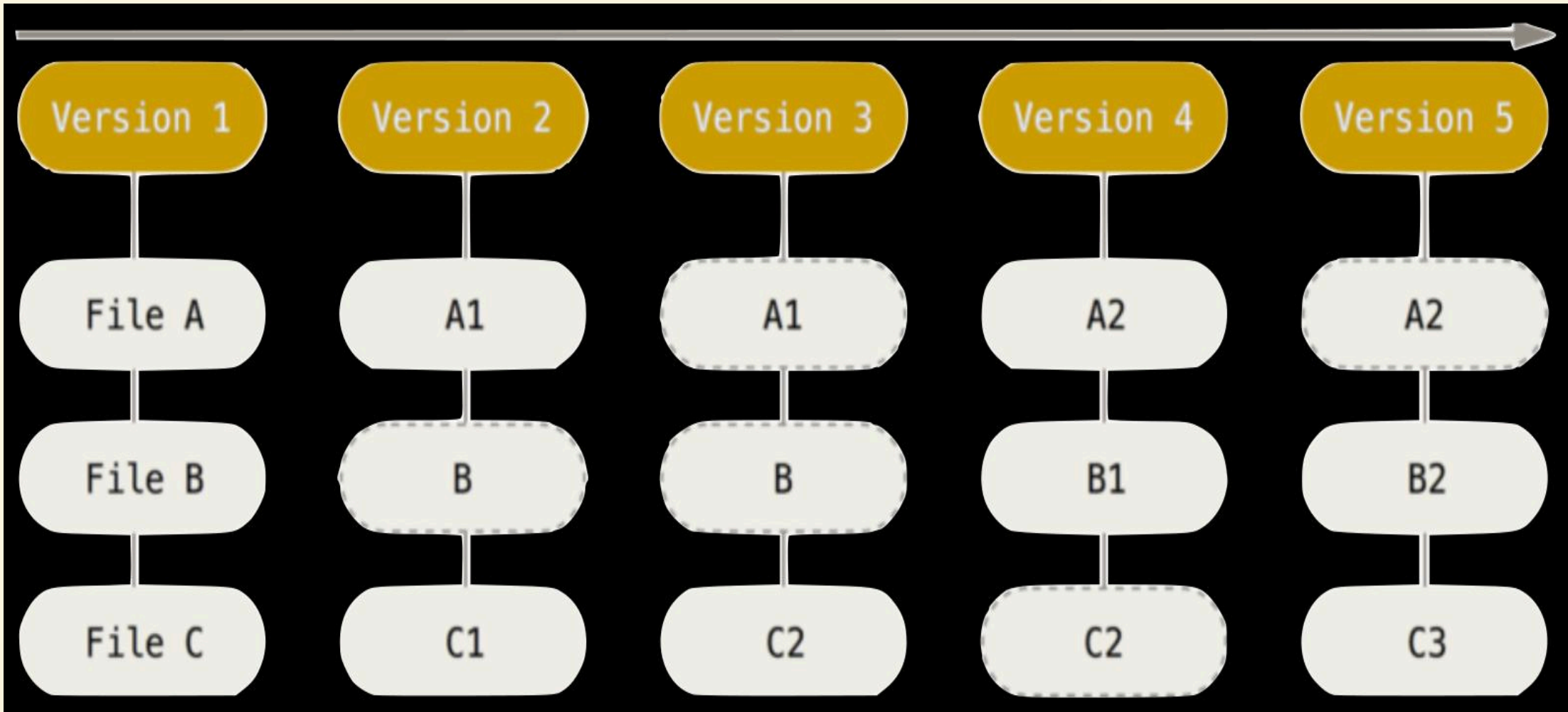
# Installation de Git !

- Installation : <https://git-scm.com/book/fr/v2/D%C3%A9marrage-rapide-Installation-de-Git>

Une fois installé, configurez le en:

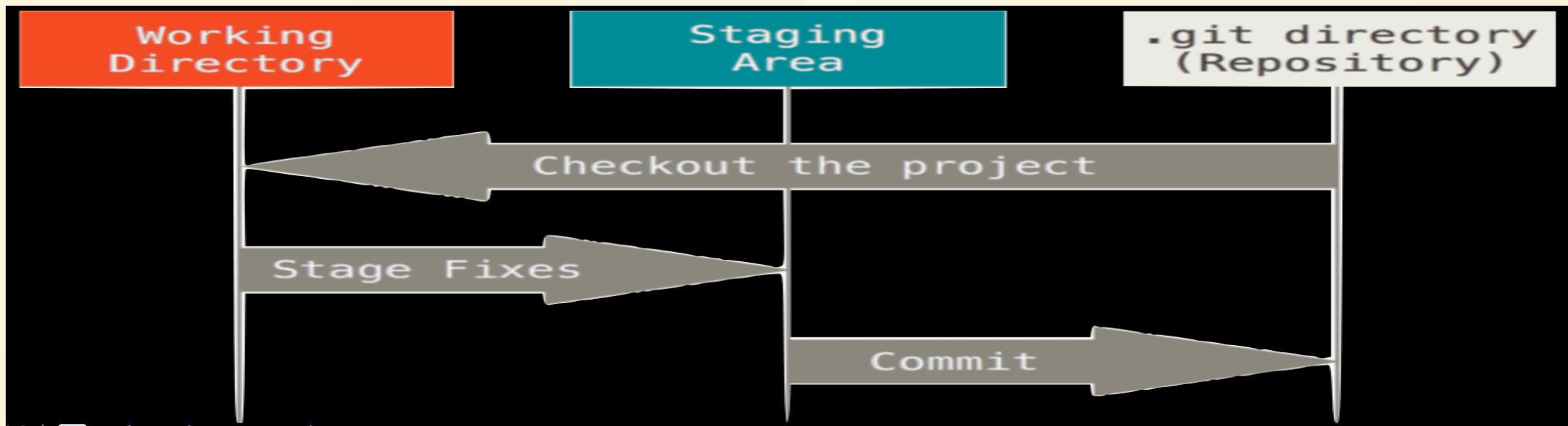
- vous créant une identité (avec vos infos):
  - `git config --global user.name 'Morais Sébastien'`
  - `git config --global user.email 'sebastien.morais@univ-evry.fr'`
- choisissant votre éditeur de texte:
  - `git config core.editor notepad`
    - ☁ Si vous utilisez un IDE récent, l'édition peut être réalisée par le biais de celui-ci, e.g. Visual Studio Code.

# Fondamentaux



# Fondamentaux

- **Index** (*staging area*) : zone de stockage des instantanés (*snapshots*) à inclure dans la version
- **Commit** : enregistrement d'une version regroupant les *snapshots* placés dans l'index



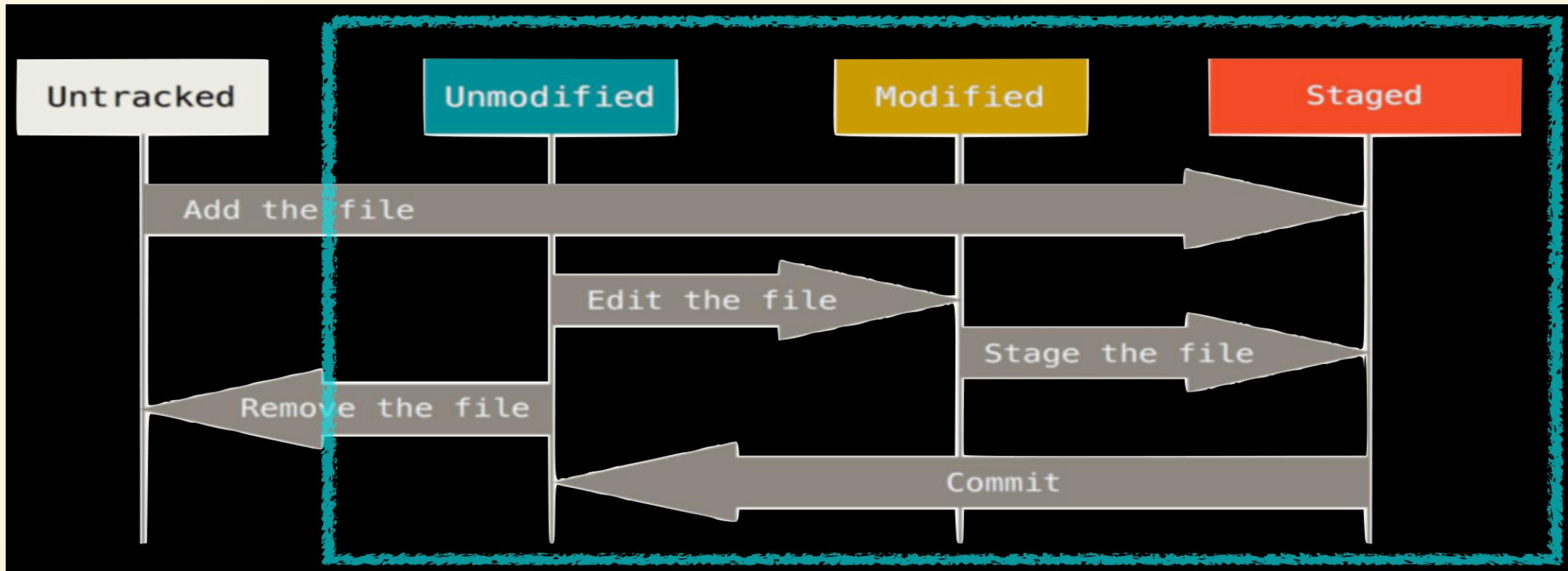
# Premier projet sous git

**Repository** (dépôt) : base de données où les modifications sont stockées et à partir de laquelle les modifications sont publiées

- `git init` : initialise un dépôt Git utilisable en local
- `git init --bare` : initialiser un dépôt Git sur un serveur  
• dépôt "installation de stockage" et pas un environnement de développement
- `git clone` : clone un dépôt déjà existant dans un nouveau répertoire

# Gestion des fichiers

- **Tracked** : au moins un enregistrement associé au fichier existe



# Gestion des fichiers

? Quelle séquence d'actions a permis d'arriver à cet état ?

```
moraiss@moraiss-Latitude-7490:git (master *+)>git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  process.py
```

# Transition d'états

- Ajout d'un fichier dans l'index `git add README`
- Consultation du status des fichiers du dépôt `git status`
- Consultation des modifications depuis le dernier enregistrement `git diff`
- Création d'un enregistrement à partir des modifications dans l'index  
`git commit -m 'Nouveau README'`
- Annulation des modifications depuis le dernier enregistrement  
`git checkout README`

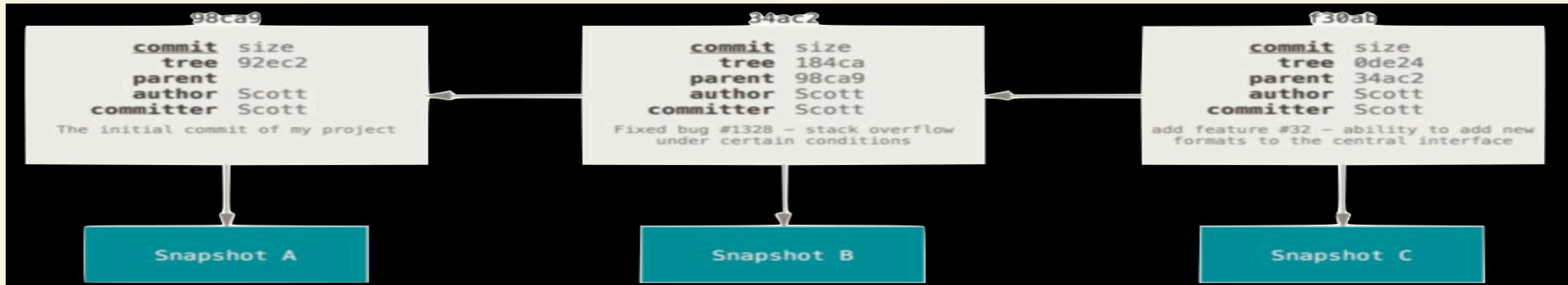
# Transition d'états

- Retrait d'un fichier de l'index `git reset README` / suppression d'un enregistrement `git reset shaDuCommit`
- Annulation d'un enregistrement (ajout du négatif)  
`git revert myshas238e`
- Stockage temporaire des modifications `git stash`



# Organisation du travail

- Chaque enregistrement est lié à son (ses) parent(s)
- Une **branche** est un pointeur vers un enregistrement
- Le pointeur d'une branche avance automatiquement et référence toujours le dernier commit réalisé



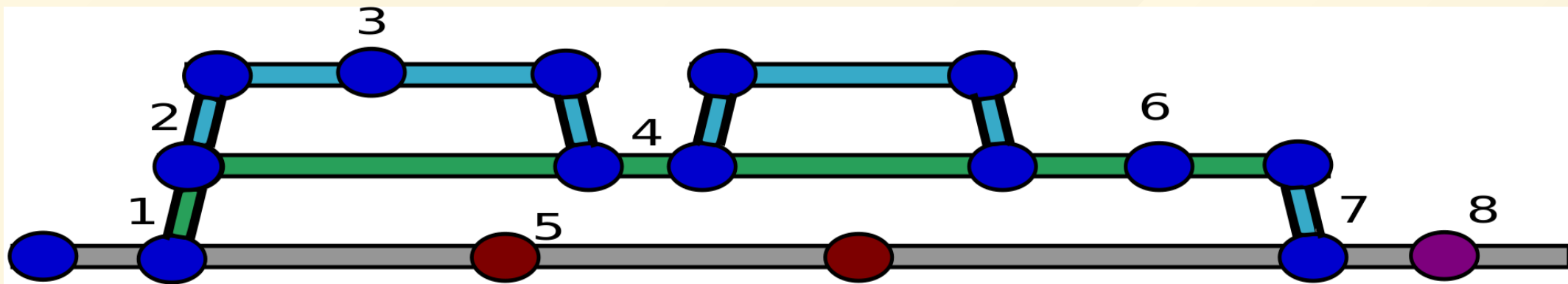
# Organisation du travail

- Visualation de la branche courante `git branch (-vv)`
- Création d'une branche `git branch feature/remove_group`
- Fusion d'une branche `git merge feature/remove_group`
- Suppression d'une branche `git branch -d feature/remove_group`
- Changement de branche :
  - `git checkout autre-branche`
  - `git switch autre-branche` (requiert Git récent)

# Organisation du travail

- Copie d'un enregistrement d'une autre branche `git cherry-pick shaDuCommit (-x)`
  - Notes :
    - référence spéciale : `HEAD` (la tête de la copie de travail)
    - lorsque `HEAD` bouge (checkout), les fichiers sont modifiés
    - en général, pointeur sur le dernier commit de la branche courante.
- 💡 La manière d'utiliser les branches dépend de la gestion de projet !

# Organisation du travail



● Commit standard

● HotFix

● Release

■ Branche Main

■ Branche de sprint ou de release

■ Branche spécifique au développeur

1 : création d'une nouvelle branche feature à partir de la main

2 : création d'une nouvelle branche pour un développeur (regroupement de tâches)

3 : commit du développeur lié aux tâches qui lui sont affectées

4 : merge avec la branche principale de la feature (de préférence journalier)

5 : HotFix en cours de développement (seulement sur la main)

6 : fix des merges ou travail sur les hotfixs apparus durant le cycle

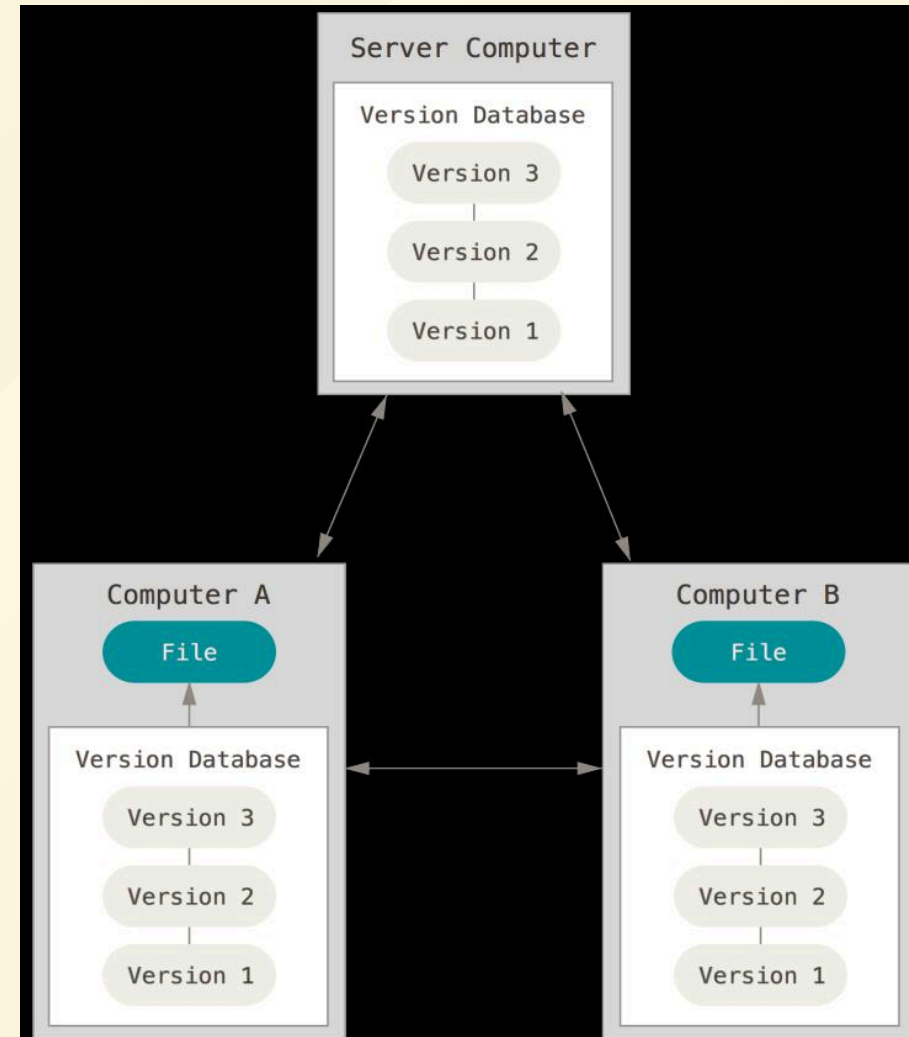
7 : pull request et merge avec la branche master

8 : édition d'une release

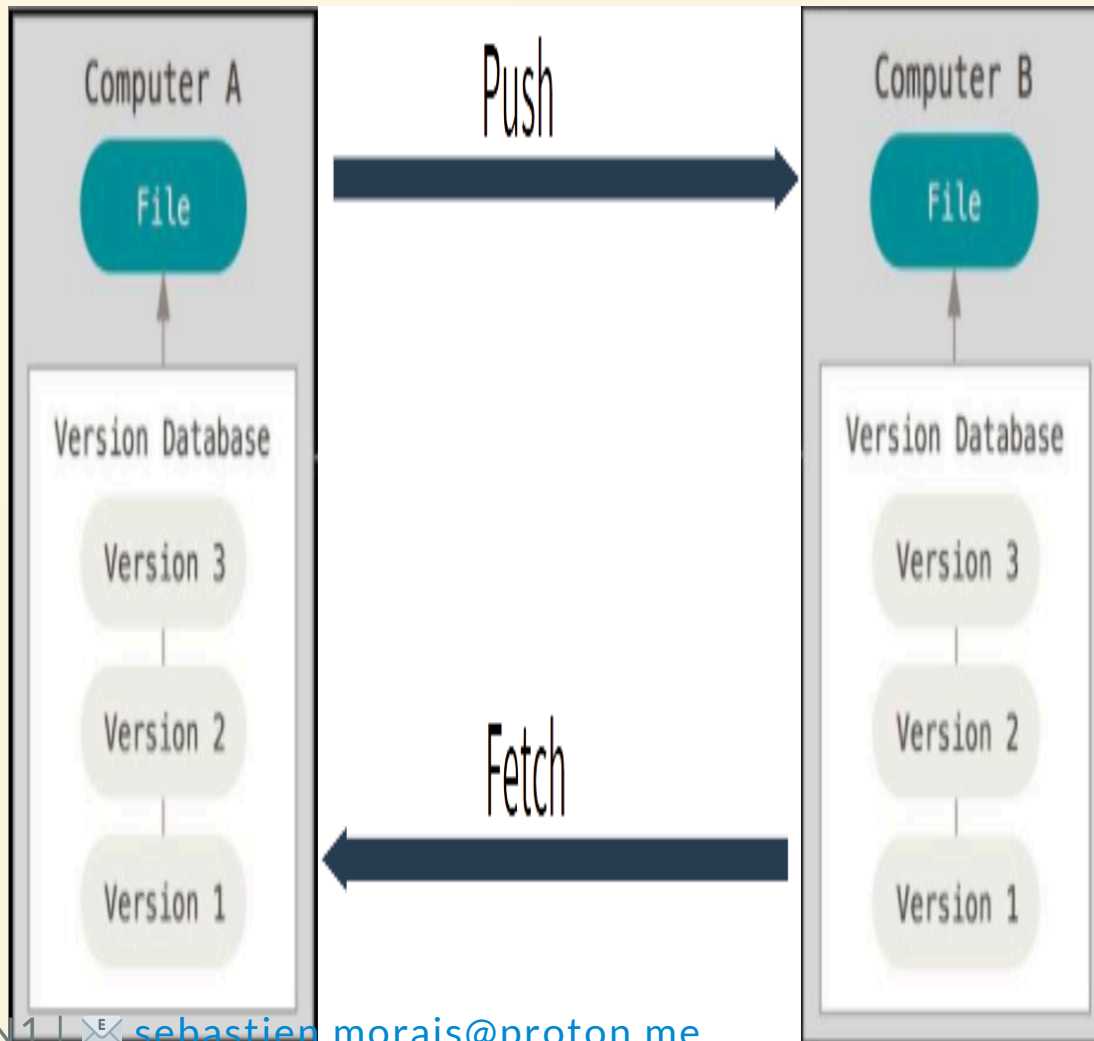
# Collaboration

- Concept de *remotes*
- Pointeurs sur des dépôts distants

```
moraiss@moraiss-Latitude-7490:Git (td_docker *=>)>git remote -v  
bare    /home/moraiss/Documents/Cours/stupid_bare (fetch)  
bare    /home/moraiss/Documents/Cours/stupid_bare (push)  
origin  git@gitlab.com:SMorais/technolog.git (fetch)  
origin  git@gitlab.com:SMorais/technolog.git (push)
```




# Collaboration

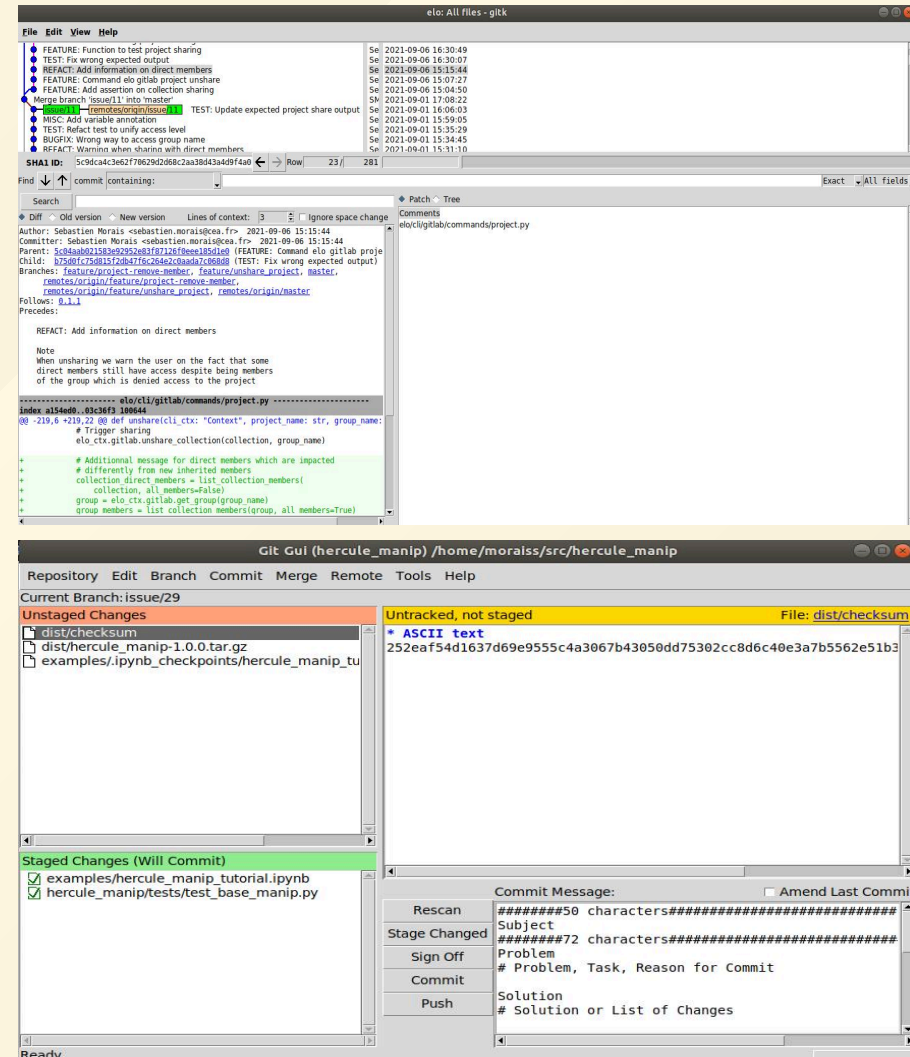


- Récupération des modifications du remote `git fetch origin`
- Publication du travail local sur le dépôt distant  
`git push origin master`
- Récupération des modifications du remote et mise à jour de sa branche  
`git pull origin master`
- Mise à jour de sa branche avec une branche distante  
`git merge origin/master`

# Quelques outils

- Diff : meld, kompare, tkdiff
- Merge : kdiff3, p4merge, diffMerge
- Log : gitk, gitg, tig
- Edition : git-gui, gitkraken

 <https://git-scm.com/download/gui>



# Gérer un conflit

- L'action de *merge* peut poser problème
  - ☁ Modification de la même portion d'un code sur deux branches différentes
- Git édite le fichier pour y formater les extraits en conflit

```
moraiss@moraiss-Latitude-7490:git_merge (master)>git merge feature/toto
Auto-merging my_app.py
CONFLICT (content): Merge conflict in my_app.py
Recorded preimage for 'my_app.py'
Automatic merge failed; fix conflicts and then commit the result.
```

Branche master

```
def foo(value):
    assert value
    return True
```



```
def foo(value):
<<<<<<< HEAD
    assert value
    return True
=====
    if value:
        return True
    else:
        return False
>>>>>> feature/toto
```



Branche feature/toto

```
def foo(value):
    if value:
        return True
    else:
        return False
```



# Nous n'avons presque rien vu


- `git archive` : génération d'archives à partir de n'importe quelle référence
- `git bisect` : recherche d'un commit fautif de manière automatisée
- `git format-patch/am` : import/export de commits sous forme de texte (diff)
- `git grep` : plus fort et plus puissant que grep sur dépôt Git

Et plein d'autres ...

# Plateformes collaborative

Les forges ou source hubs ne sont pas nécessaire pour héberger votre projet MAIS elles ont plusieurs avantages :

- conçu pour accueillir les utilisateurs les plus novices
- renforce la visibilité de votre projet
- facilite la contribution de tiers
- outils de collaborations : gestionnaire de bogue, wiki, CI, PR, permissions, ...
- retire la « complexité » de la gestion de l'hébergement

 Github, Gitlab, Sourcehut, SourceForge, ...

Pour la suite, créer un compte sur **Github** (tour du proprio)

# Apprendre à utiliser les branches

Réalisez les exercices sur LearnGitBranching

 Comprendre Git à travers la visualisation

<https://learngitbranching.js.org/>

# Living code : contribution à un projet

- Github: voir issues, code, PR, revue de code, avantage / inconvénient d'un forge
- Git commit template
- PR / Issue template
- Workflow: fork, branch de développement, code, push, PR
- Squelette de projet du morpion avec une classe `Morpion`:
  - attributs: `plateau`, `joueur`
  - méthode: `afficher`, `jouer_coup`, `verifie_victoire`, `plateau_plein`, `changer_joueur`, `jouer`

# Living code : contribution à un projet

1. Dans une PR écrire les docstrings (ne pas merge)
2. Dans une seconde PR écrire l'implémentation d'une méthode
3. Dans une troisième branche écrire l'implémentation d'une autre méthode et PR dans #2
4. Répéter 3.
5. Valider #1 et regarder #2.
6. Réécrire l'historique de #2 pour ne pas avoir de conflit et garder un seul commit. Puis re décomposer le commit en plusieurs commits.

# Git attributes

L'utilisation de **différents OS** (Windows, Linux, MacOS) ou **éditeurs de code** (Visual Studio Code, Sublime Text, CLion) peuvent mener à des conflits.

Exemple du saut de ligne (EOL):

- sur une machine Windows : par défaut Carriage Return Line Feed (CRLF)
- sur une machine Linux/MacOS : Line Feed (LF)

**Problématique** : outil de formattage avec une propriété de fin de ligne définie

**Solution** : gitattributes pour utiliser Git et modifier le comportement par défaut de certaines opérations sur les fichiers et répertoires.

Exemple : <https://github.com/ansys/pyaedt/blob/main/.gitattributes>

# Ignorer des révisions Git

Contexte : utilisation de `git blame` pour récupérer des informations sur le dernier changement ayant impacté une ligne de code.

Problème : L'utilisation de formatteur automatique...

Exemple en direct avec `git blame --ignore-revs-file .git-blame-ignore-vers` (twinbuilder L129) en CLI.

Intégré dans Visual Studio Code avec [Gitlens v13.4](#) et dans [Github](#) !

# Precommit

Idée générale : **analyser le code pour imposer des conventions avant commit !**

**Exemple de solution :** <https://pre-commit.com/>

Applications possibles :

- Convention pour les messages de commits, e.g. <https://www.conventionalcommits.org/fr/v1.0.0/>
- Formatter du code automatiquement, e.g. <https://pypi.org/project/black/>
- Valider le code au regard d'une norme, e.g. <https://pypi.org/project/flake8/>
- Corriger les fautes d'orthographe courantes dans les fichiers texte, e.g. <https://github.com/codespell-project/codespell>



# Extra (découverte)

Pour info, je suis tombé sur cela récemment :

<https://github.com/figify/gh-metrics>

Description rapide : une CLI pour calculer des métriques relatives aux PR / issues d'un projet Github.

