Développement logiciel	
Contenus :	– Utilisation d'un outil de versionning (git par exemple) – Cycle de vie d'un projet informatique (conception, UML, AGL) – Gestion de projet informatique (méthode agile)

I. Utilisation de git pour le contrôle de versions d'un projet

1. Avantages d'utilisation d'un système de contrôle de versions

Imaginons qu'on travaille en équipe sur un projet de longue échéance comportant plusieurs fichiers de code, il est important de pouvoir faire ces opérations:

- **Suivi des modifications** en enregistrant chaque modification apportée aux fichiers. Cela inclut les ajouts, les suppressions, les modifications et les déplacements de fichiers.
- **Historique des versions** : Chaque modification est enregistrée sous forme de "commit" ou de "révision". Ces commits créent un historique complet de toutes les versions antérieures des fichiers. Chaque commit est accompagné d'un message décrivant les modifications apportées. Cela facilite la compréhension des changements et des raisons derrière eux.
- **Collaboration en équipe** : en facilitant la fusion des modifications des personnes travaillant sur les mêmes fichiers en même temps, en résolvant les conflits éventuels de manière contrôlée, et en transmettant les modifications d'un développeur à l'autre.
- **Définition et gestion de plusieurs flux de développement (branches)** : les branches permettent de travailler sur des versions distinctes du projet. On crée des branches pour développer des fonctionnalités, corriger des bugs, etc., tout en maintenant la version principale stable.
- **Rétablissement et exploration** permettant de revenir en arrière dans l'historique et de récupérer une version précédente des fichiers. Cela facilite la gestion des erreurs et des régressions.
- **Étiquetage (tagging)** : en utilisant des balises permettent de marquer des points spécifiques de l'historique comme des versions stables, des versions de publication, etc.

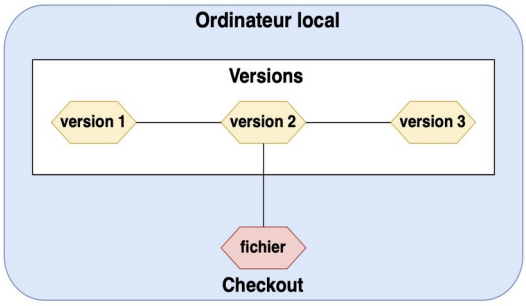
- **Différences et comparaisons** entre les versions, ce qui aide à comprendre les changements effectués et à résoudre les problèmes.

Pour pouvoir effectuer ces tâches de façon efficace et sécuritaire il existe ce que l'on nomme un **système de contrôle de versions** .

2. Stratégies de gestion des sources

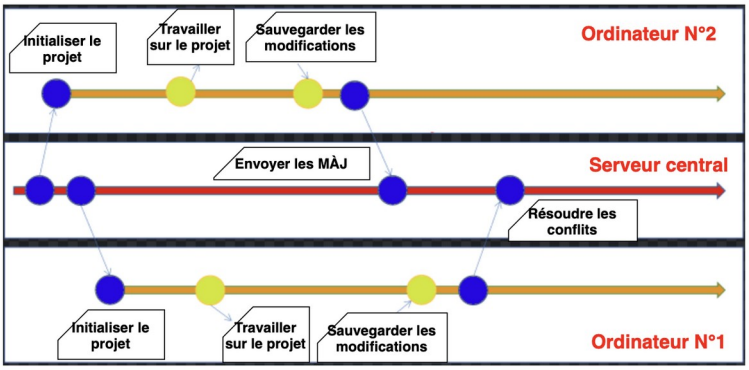
On distingue trois manières pour contrôler les versions :

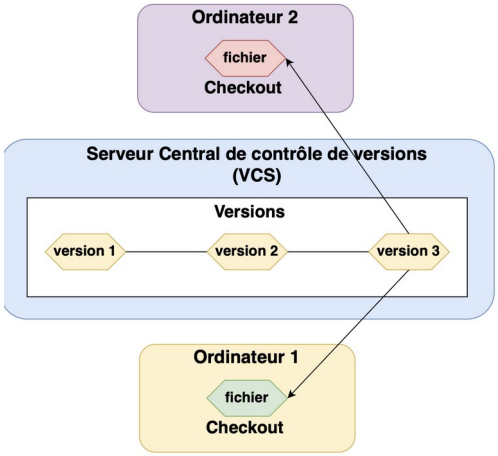
→Stratégie locale : fait référence à la gestion des versions d'un fichier ou de plusieurs fichiers uniquement sur un ordinateur individuel comme le montre la figure. Il n'y a pas de communication avec un serveur ou un dépôt centralisé. Tout le processus de création, de modification, et de suivi des versions des fichiers se fait localement.



L'action de `checkout` permet à l'utilisateur de choisir et d'utiliser une version spécifique parmi celles disponibles localement.

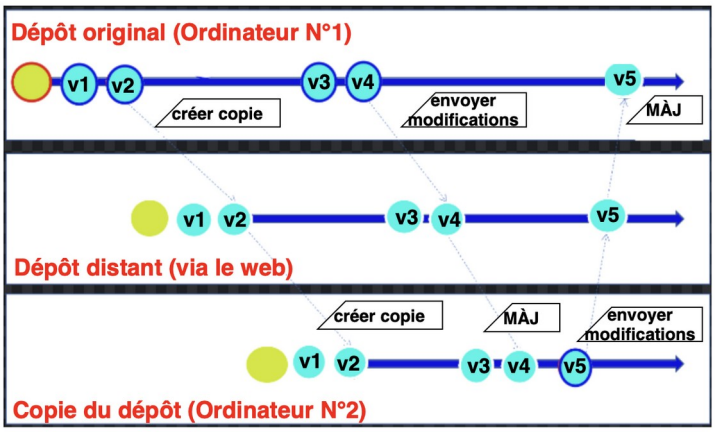
→Stratégie centralisée (serveur-clients) : Tous les historiques des fichiers partagés sont stockés sur un **serveur central**. Les clients, souvent des développeurs, disposent d'une copie locale de ces fichiers sur laquelle ils travaillent. Une fois les modifications locales terminées, le serveur est responsable de combiner ces modifications avec la version centrale, tout en résolvant les éventuels conflits générés par les modifications simultanées d'autres développeurs. Parmi les systèmes qui utilisent ce modèle centralisé, on trouve notamment Subversion (SVN) et CVS.

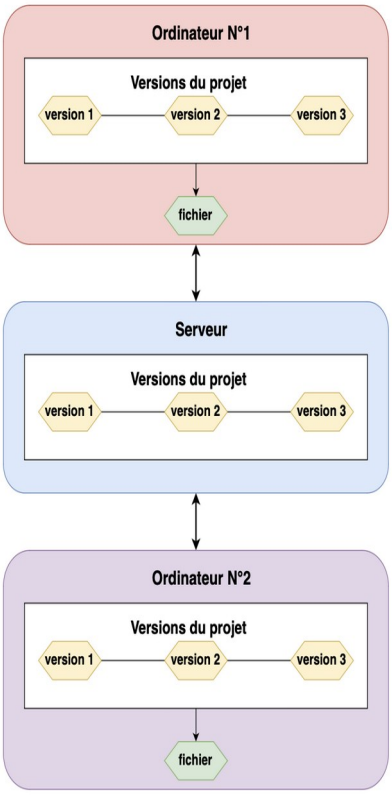




→Stratégie distribuée :

Dans les **systèmes distribués**, à l'inverse des **systèmes centralisés**, chaque développeur conserve une copie complète du projet sur son ordinateur, incluant uniquement la nécessité de transmettre et de synchroniser l'historique des modifications entre les différents postes. Par exemple, si un développeur possède une version du projet avec des modifications récentes, un autre développeur peut copier cette version (incluant les modifications) sur son propre système et commencer à apporter ses contributions. Ils devront ensuite synchroniser régulièrement leurs dépôts pour partager et mettre à jour l'historique des modifications apportées au code. Dans la pratique, comme les interactions directes entre les machines de plusieurs développeurs ne sont pas toujours possibles, il est courant de recourir à un point d'échange central, souvent en ligne, pour faciliter cette synchronisation. **git** est un exemple de système de contrôle de versions qui adopte cette approche, et **gitLab** offre un service de dépôt en ligne facilitant cet échange.





3. git vers gitHub

git est un **système de gestion de versions open source** qui est librement accessible et peut être installé localement sur les machines des développeurs. Ce système permet l'exécution de toutes les commandes nécessaires directement depuis le poste de travail du développeur. Il a été développé initialement par la communauté du noyau Linux, notamment par Linus Torvalds en 2005.

gitLab, **gitHub**, et **Bitbucket** sont des plateformes d'hébergement web qui offrent un espace commun de communication via une interface web, facilitant ainsi l'utilisation de

git pour la synchronisation des dépôts entre développeurs. Ces services permettent également de configurer les paramètres d'un projet, tels que l'ajout ou la suppression de membres de l'équipe, ainsi que l'ajustement des paramètres d'accès. Les conditions d'utilisation de ces services varient, notamment en ce qui concerne l'hébergement web pour les projets publics et privés :

- **gitHub** offre un service **gratuit** pour les projets publics, avec des options payantes pour les projets privés.
- **Bitbucket** propose un accès **gratuit** pour tous les projets, y compris les projets privés jusqu'à 5 collaborateurs.
- **gitLab** fournit un accès gratuit tant pour les projets publics que privés.

4. Présentation de git

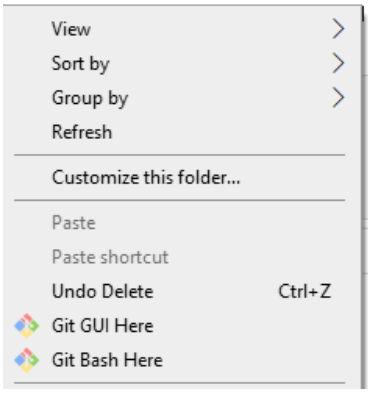
a) Installation de git

Pour installer **git** sur votre machine, commencer par le télécharger depuis le site officiel (https://git-scm.com/download/win). Ensuite, suivre les instructions fournies pour l'installation. Durant le processus d'installation, notamment à la quatrième étape après avoir cliqué trois fois sur 'Suivant', vous aurez l'opportunité de choisir l'éditeur de texte pour **git**, comme Notepad++ ou VS Code.

Pour confirmer que **git** est correctement installé, ouvrir un terminal et saisir la commande `git --version`.

b) Configuration initiale de git

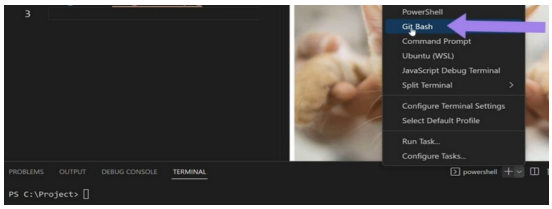
Pour ouvrir rapidement la console **git** dans un répertoire spécifique de votre ordinateur, il faut naviguer jusqu'à ce dossier, faire un clic droit et sélectionner l'option '**git Bash Here**' dans le menu contextuel. Cela ouvrira directement l'interface de commande de git à cet emplacement.



c) Utilisation de git sous Visual Studio Code

À faire :

Pour utiliser **git Bash** dans **Visual Studio Code**, commencer par ouvrir ce dernier et lancer un nouveau terminal. Ensuite, choisir **git Bash** dans le menu déroulant des options de terminal :



Un symbole **$** devrait apparaître dans le terminal, indiquant que **git Bash** est actif. Si on souhaite définir **git Bash** comme le shell par défaut dans **VS Code** sous Windows, il faut

aller dans le **menu Affichage**, sélectionner **Palette de commande**, taper « **sélectionner le profil par défaut** », puis choisir « **git Bash** ».

La commande **`git config`** est utilisée pour configurer les paramètres de **git**. Après avoir ouvert la console, saisir les commandes suivantes pour configurer votre identité, ce qui permettra aux autres collaborateurs de vous reconnaître dans les projets partagés :

```
git config --list
git config --global user.name "Votre Nom Complet"
git config --global user.email "Votre courrier@unicaen.fr"
```

`git config --list` affiche la liste de toutes les configurations git définies sur votre système. Cela inclut les paramètres tels que le nom d'utilisateur, l'adresse e-mail, les alias, les options de formatage et plus encore (une liste de paires clé-valeur pour chaque paramètre configuré) :

```
git config --list
...
user.name= Miryem HRARTI-DRAFATE
user.email=miryem.drafate@unicaen.fr
...
```

Partie 1 : Configuration

→ Afficher la version installée de **git**.

→ Configurer initialement **git** en déterminant votre nom et votre adresse e-mail, puis vérifier que la configuration a été correctement appliquée.

d) Terminologie et commandes essentielles de git

→ **Voici quelques termes à connaître** :

- **Projet** : ensemble du code, modifications, informations des collaborateurs et propriétés d'accès (privé ou public).
- **Dépôt** (***`repository`***) : emplacement physique ou virtuel où les fichiers d'un projet sont stockés avec leur historique de modifications. Ce terme peut s'appliquer à la fois à un dépôt local et à un dépôt distant.
- **Dépôt local** : le dépôt sur lequel on travaille, généralement situé sur notre ordinateur personnel. Il inclut le sous-dossier ***`.git`*** où Git stocke toutes les métadonnées nécessaires au suivi de version telles que l'historique des commits, la configuration, les références de branches, et plus.
- **Dépôt distant** (***`remote`***) : un dépôt externe utilisé pour la collaboration et le partage, souvent hébergé sur une plateforme comme gitHub ou sur un serveur. Ce

dépôt est utilisé pour partager les modifications, collaborer avec d'autres, et sécuriser une copie en ligne des données du projet.

• **Changement officiel** (***commit***) : enregistrement officiel des modifications, identifiable par un identifiant unique hexadécimal. Chaque commit est un nœud dans l'arbre des commits.

• **Branche** (***branch***) : les branches sont des séquences de commits permettant de diverger de la ligne principale de développement. Une branche est marquée par une étiquette pointant vers le dernier commit de la séquence. La branche principale est souvent appelée ***master***, et généralement, seuls les mainteneurs ont le droit de modifier directement cette branche.

→ **Voici quelques principales commandes** : Pour exécuter une commande git, taper **`git`** suivi du nom de la commande et des options nécessaires. . Voici un résumé des commandes principales :

Commande	Description
Créer un dépôt	
`init` *projet*	Créer un dépôt vide
`clone` *versDépôtDistant*	Créer un dépôt en clonant un dépôt distant
`cd` *repertoire_projet* **`init git add –A`**	Créer un dépôt local dans un répertoire local existant (**`-A`** pour **`all`**)
`config --global alias.ci commit` **`config --global alias.cim "commit -m"`** **`config --global alias.st "status -s"`** **`config --global alias.co checkout`** **`config --global alias.br branch`**	Créer des alias
Modifications locales (mise à jour des branches)	
`pull` *nomDépotDistant nomBrancheDistante*	Récupérer les données de la branche d'un dépôt + les dernières modifications et fusionner dans la branche courante;
`add` *fichier1 fichier2 fichier3*	Ajouter les fichiers modifiés ou nouveaux à la zone de préparation de la branche locale courante.
`rm` *nomDesFichiers* **`rm`** *repertoire/ -r*	Effacer les fichiers de la branche locale courante (de la zone d'index et du répertoire du travail). Ce changement doit être confirmé par un commit. Supprimer récursivement les fichiers d'un répertoire.

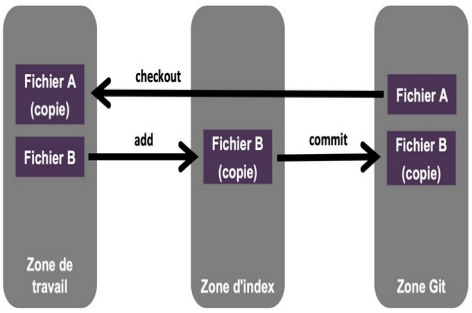
rm *--cached*	permet d'abandonner le suivi de version d'un fichier et de le conserver dans le répertoire de travail
mv *fichier nouveau_nom* **mv** *fichier destination/* **mv** *f_origine f_final* équivalent à : **mv** f_origine f_final **rm** f_origine **add** f_final	Renommer un fichier Déplacer un fichier git ne suit pas explicitement les mouvements des fichiers. git détectera le renommage
commit -m *"Message du commit"* **commit -a** *(ajouter automatiquement les fichiers)* **commit --amend**	Enregistrer sur la branche courante locale les changements placés dans la zone de préparation. Les commentaires sont enregistrés pour décrire le commit. Modifier le dernier commit
tag *nom_tag*	Etiqueter le dernier commit
reset *numeroDeCommit* **checkout** *-- fichier* **reset [--mixed] HEAD** *fichier*	Efface tous les commits à partir du commit correspondant au numéro spécifié, impossible de revenir en arrière. Annuler les modifications réalisées dans un fichier
revert *numeroDeCommit*	Crée un nouveau commit qui renverse les changements du commit correspondant au numéro spécifié. Il est donc possible de conserver l'historique de ces changements.
status	Vérifier s'il y a des fichiers non-ajoutés à la zone de préparation (unstaged) ou des changements non-commis (staged) à la branche courante locale (Afficher l'état des fichiers nouveaux ou modifiés) suivis modifiés
push *nomDépotDistant nomBrancheDistante*	Pousser les commits de la branche locale courante sur la branche distante (Publier les modifications locales d'une branche)

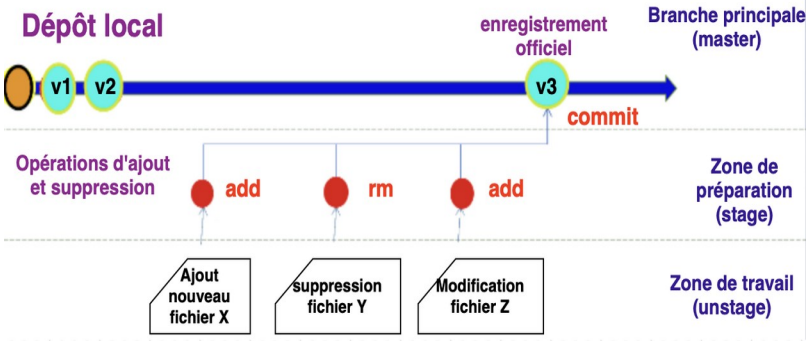
log **log -n** (par exemple **git log -3**) **log --stat** **git log --pretty=oneline**	Pour visualiser l'historique de validations Limiter le nombre de commits Visualiser les différences des commits Personnaliser l'affichage
diff **diff --cached** ou **diff --staged**	Visualiser le contenu modifié non indexé Comparer la zone d'index avec le répertoire git
Gestion des branches	
branch	Affiche la liste des branches
branch *nouvelleBrancheLocale*	Créer une nouvelle branche à partir de la branche courante locale
checkout *autreBrancheLocale*	Basculer sur une autre branche comme étant maintenant la branche active.
branch -d *ancienneBrancheLocale*	Supprimer une branche locale
merge *autreBrancheLocale*	Fusionner les changements d'une autre branche avec la branche courante.

e) Création de dépôt, zones git et états de fichier

Pour créer un dépôt, il faut se positionner dans un répertoire, puis taper la commande : **git init**. Cette dernière initialise le dépôt en générant un répertoire **.git** et crée par défaut la branche "**master**".

Git utilise trois zones principales pour gérer les modifications apportées aux fichiers et pour préparer les commits. Ces zones sont essentielles pour comprendre le flux de travail de Git et la création d'histoires de versions. Voici ces trois zones :





→ **zone ou répertoire de travail (*Working Directory*)** : c'est où on crée, modifie et organise nos fichiers. Le répertoire de travail contient la version actuelle des fichiers, y compris les modifications en cours. On peut effectuer des modifications, ajouter de nouveaux fichiers et supprimer des fichiers dans cette zone.

→ **zone d'index (*staging Area*)** : L'index, également appelé **zone de préparation** (staging area) est comme une zone intermédiaire entre le répertoire de travail et le référentiel Git (zone Git). Lorsqu'on exécute la commande **`git add`**, on déplace les modifications des fichiers du répertoire de travail vers l'index. Cela signifie qu'on prépare ces modifications pour être incluses dans le prochain **`commit`**.

→ **zone Git ou référentiel Git (G*it Repository*)** : est l'endroit où toutes les versions des fichiers sont enregistrées sous forme de commits. Chaque commit représente un instantané complet de l'état des fichiers à un moment donné. Les commits sont stockés dans la base de données Git, et ils contiennent non seulement les modifications, mais aussi les métadonnées comme l'auteur, la date et le message de commit.

Dans Git, les fichiers peuvent passer par plusieurs états qui reflètent leur statut par rapport au processus de gestion des versions. Voici les états courants que les fichiers peuvent avoir dans git :

→ **Untracked (non suivi)** : obtenu lorsque Git ne reconnaît pas le fichier puisqu'il ne fait pas partie de l'historique du dépôt (n'est pas suivi par Git). Cela signifie que le fichier existe dans le répertoire de travail, mais n'a pas encore été ajouté à l'index (staging area). Les nouveaux fichiers créés sont généralement dans cet état.

→ **Unmodified (non modifié)** : obtenu lorsque le fichier n'a pas été modifié depuis son dernier commit. Il existe dans le dépôt Git (le dossier .git), où son dernier état validé est stocké.

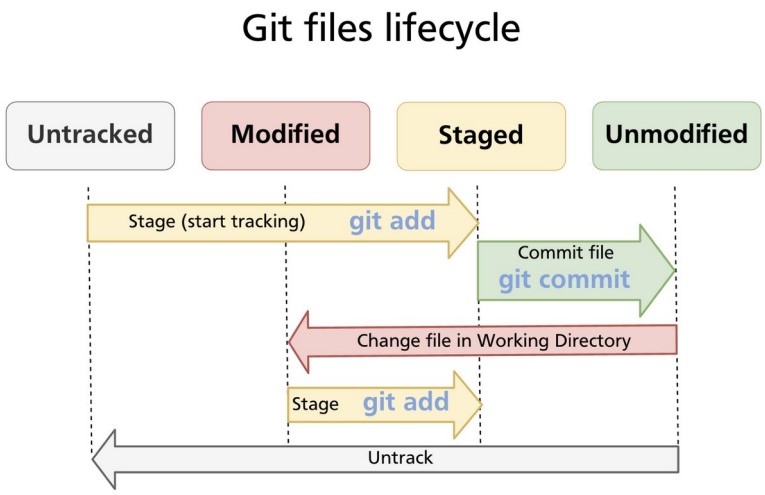
→ **Modified (modifié)** : lorsqu'on apporte des modifications au contenu d'un fichier dans le répertoire de travail après le dernier commit. Git détecte ces modifications et marque le fichier comme modifié. Cependant, ces modifications ne

seront pas incluses dans le prochain commit tant qu'on n'a pas ajouté le fichier à l'index à l'aide de la commande : **`git add`**.

→ **Staged (préparé)** : obtenu lorsqu'on utilise la commande **`git add`** pour déplacer les modifications du fichier de l'état modifié ou non suivi à l'index (staging area). En ajoutant un fichier à l'index, on indique à Git qu'on souhaite inclure ses modifications dans le prochain commit. Ces fichiers sont prêts à être enregistrés dans le référentiel Git via un commit.

Il est à noter que la commande **`git rm --cached <fichier>`** permet de retirer un fichier de la zone d'index de Git tout en le laissant dans le répertoire de travail. Ce fichier était précédemment suivi par Git (donc **Unmodified** si aucune modification n'avait été faite depuis le dernier commit ou **Staged** s'il avait été modifié et ajouté à l'index) et il est rendu **Untracked**, et donc Git ne suivra plus ses modifications dans les commits futurs, bien qu'il reste dans le répertoire de travail.

Voici un schéma qui illustre comment les fichiers passent d'un état à un autre dans Git, en fonction des commandes utilisées :



Pour connaître l'état d'un dépôt, on utilise la commande **`git status`**. Ceci permet d'afficher les fichiers modifiés, non versionnés, etc. On peut utilise l'option **`--short`** ou **`-s`** pour avoir une vue synthétique de l'état du dépôt. Pour chaque élément, 2 informations sont disponibles : l'état dans la zone d'index (à gauche), et l'état dans la zone de travail (à droite).

Partie 2 : Comprendre les zones git

1- Ouvrir un terminal sous VS Code. Créer un nouveau répertoire nommée « **`RapportDeStage`** » à l'aide de la commande **`mkdir`** (**`mkdir nomDossier`**). Se déplacer à l'intérieur de ce répertoire à l'aide de la commande **`cd chemin/vers/rep`**, puis créer

un fichier nommée « **`Chapitre1.txt`** » à l'aide de la commande **`touch`** suivie du nom du fichier. Afficher le contenu de votre répertoire à l'aide de la commande **`ls -la`**. Ouvrir le fichier à l'aide de la commande **`nano Chapitre1.txt`**, puis ajouter ce contenu : « **Nettoyer les données, c'est comme brosser la poussière d'un trésor caché, révélant ainsi les joyaux cachés de l'information.** ». Sauvegarder les modifications en tapant Ctrl+O (ce qui signifie "write Out" en nano), puis en appuyant sur Entrer pour confirmer le nom du fichier. Pour sortir de nano, tapez Ctrl+X. Afficher finalement le contenu de « **`Chapitre1.txt`** » à l'aide de la commande **`cat`** suivie du nom du fichier.

2- Initialiser un dépôt git avec **`git init`**. Cette commande génère un **répertoire caché `.git`** et crée la branche **`master`**. Afficher le contenu du répertoire courant (fichiers, sous répertoires y compris ceux cachés) avec la commande **`ls -la.`** Taper la commande **`start .`** (ou **`open .`** pour Mac) afin ouvrir le dossier actuel dans l'Explorateur Windows.

3- Se déplacer à l'intérieur de **`.git`**, et taper la commande **`cat HEAD`** permettant d'afficher le contenu du fichier **HEAD**, qui indique la branche sur laquelle vous êtes actuellement positionné (**`main`** ou **`master`**).

3-Lancer la commande **`git status`** qui permet de connaitre l'état d'un dépôt. Vous allez trouver la liste des fichiers/répertoires non suivis :

```
(base) hrartimiryem@MacBook-Air-de-HRARTI RapportDeStage % git status
Sur la branche master

Aucun commit

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
        Chapitre1.txt

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents (utilisez
 "git add" pour les suivre)
```

Quel(s) sont les fichiers non suivis ? Il(s) se trouve(nt) dans quelle zone ?

4- **`git status -s`** est équivalent à **`git status --short`**, lancer cette commande. Vous obtenez :

```
?? Chapitre1.txt
```

L'état **`?`** signifie **`Untracked`** (non versionné). La mention **`??`** signifie que le fichier ou répertoire est à l'état **`Untracked`** dans le répertoire de travail.

5- Ajouter le fichier « **`Chapitre1.txt`** » dans la zone d'index, puis afficher l'état du dépôt. Vous obtenez :

```
Modifications qui seront validées :
  (utilisez "git rm --cached <fichier>..." pour désindexer)
        nouveau fichier : Chapitre1.txt
```

ou

A **Chapitre1.txt**

La mention **A** signifie que le fichier a été ajouté dans la zone d'index et prêt à être validé dans le prochain commit.

6- Créer un premier commit avec comme message « **Chapitre N°1 à compléter** ». Afficher l'historique des commits dans un dépôt git avec **git log**. On obtient

```
(base) hrartimiryem@MacBook-Air-de-HRARTI RapportDeStage % git log
commit e8c688c8d837aa140e79c98731b1f7dc47a0811b (HEAD -> master)
Author: miryemHD <miryem.drafate@unicaen.fr>
Date:   Sun Dec 10 18:54:02 2023 +0100

    Chapitre N°1 à compléter
```

Notez la présence de la date, du message, de l'auteur du commit, ainsi qu'un code de hachage unique permettant d'identifier ce commit de manière distincte.

7- Créer deux autres fichiers : « **Chapitre2.txt** » et « **Chapitre3.txt** ». Modifier les fichiers en mettant dans le premier : « **Les données sont le nouvel or noir du 21e siècle, mais au lieu de les extraire du sol, nous les extrayons de l'information**. », et dans le second : « **La visualisation des données, c'est comme si les données prenaient vie pour raconter leur propre histoire, avec des graphiques et des diagrammes comme acteurs principaux**. ». Afficher l'état du dépôt. Vous obtenez :

```
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
        Chapitre2.txt
        Chapitre3.txt
```

Ajouter les deux fichiers 2 et 3 dans la zone d'index via la commande **git add .**

Le point ici spécifie tout le contenu du répertoire courant.

Afficher l'état du dépôt. Vous obtenez :

A **Chapitre2.txt**

A **Chapitre3.txt**

Créer un second commit avec comme message « **Chapitre 2 et 3 à compléter** », puis afficher l'historique des commit. Vous obtenez :

```
commit 5772070b9854ee608e8945e72b88b749da7209f0 (HEAD -> master)
Author: miryemHD <miryem.drafate@unicaen.fr>
Date:   Sun Dec 10 19:21:33 2023 +0100

    Chapitre 2 et 3 à compléter

commit 3f6af784803134391461638f865cfc3608db8a762
Author: miryemHD <miryem.drafate@unicaen.fr>
Date:   Sun Dec 10 19:14:19 2023 +0100

    Chapitre N°1 à compléter
```

Quel est l'état des trois fichiers ?

8- Ouvrir le chapitre N°3, puis modifier son contenu : « **Dans le monde académique, plus le titre d'une thèse est long, plus il faut creuser profondément pour trouver le contenu !** ». Afficher l'état du dépôt. Vous obtenez :

```
M Chapitre3.txt
```

Notez que le fichier est dans un état "**Modified**" (modifié). Cela signifie que des modifications ont été apportées au contenu de ce fichier dans votre répertoire de travail depuis le dernier commit.

Chapitre3.txt se trouve actuellement dans quelle zone ?

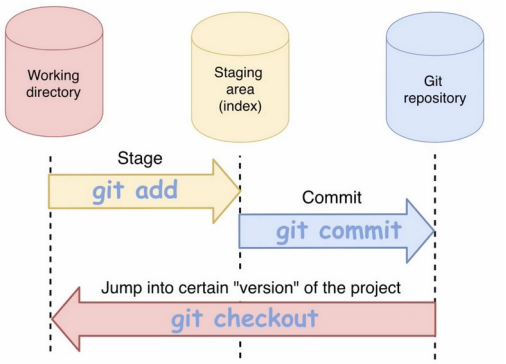
9- Taper la commande **`git diff Chapitre3.txt`**. Vous obtenez :

```
(base) hrartimiryem@MacBook-Air-de-HRARTI RapportDeStage_vF % git diff Chapitre3.txt
diff --git a/Chapitre3.txt b/Chapitre3.txt
index 249725d..02e24a2 100644
--- a/Chapitre3.txt
+++ b/Chapitre3.txt
@@ -1 +1 @@
-"La visualisation des données, c'est comme si les données prenaient vie pour raconter leur prop
re histoire, avec des graphiques et des diagrammes comme acteurs principaux.
\ No newline at end of file
+Dans le monde académique, plus le titre d'une thèse est long, plus il faut creuser profondément
 pour trouver le contenu !
\ No newline at end of file
```

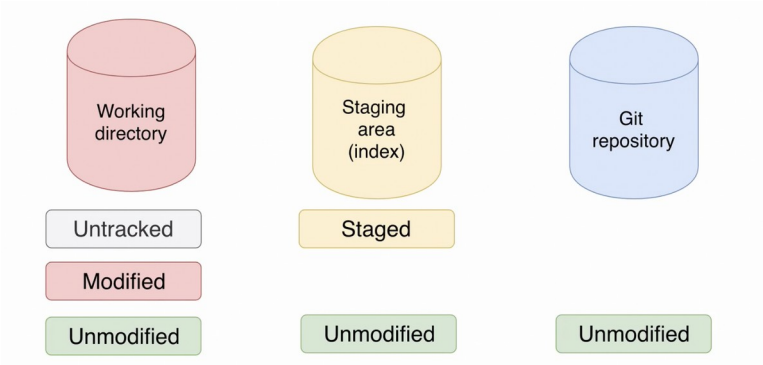
Cette commande est utilisée pour afficher les différences de contenu entre deux états du référentiel git (entre le répertoire de travail et la zone d'index, ou entre la zone d'index et le dernier commit si suivie de **`--staged`**). Elle est couramment utilisée pour afficher les modifications apportées aux fichiers avant de les ajouter à la zone d'index (staging area) ou avant de faire un commit.

10- On souhaite annuler les modifications apportées au **`Chapitre3.txt`** et restituer sa version originale. Pour ce faire, taper la commande **`git checkout Chapitre3.txt`**. Vérifier le contenu de ce fichier, et afficher de nouveau l'historique des changements effectués avec : **`git log -- Chapitre3.txt`** (Il faut trouver la même date du dernier commit, ce qui veut dire que le fichier est revenu à l'état dans lequel il était lors de ce commit).

Notez que **`git checkout nom_fichier`** annule les modifications réalisées dans un fichier. Pour les versions les plus récentes de Git, on utilise **`git restore nom_fichier.`**



La figure suivante la localisation d'un fichier dans les différentes zones selon son état :



Partie 3 : Gestion des branches

Git fonctionne sur un système de branches. La branche principale (par défaut) du projet s'appelle **`main`** (anciennement **`master`**) sur laquelle, on aura toutes les modifications. Cependant, il est déconseillé de réaliser les modifications directement sur la branche principale, mais de les insérer dans autres branches et après les tests, il suffit de les intégrer sur la branche **`main`**.

Dans un dépôt Git, il est possible de trouver plusieurs branches, chacune étant une séquence de commits permettant de diversifier le développement du projet. Les références à ces branches sont stockées dans le dossier **`.git/refs/heads`**. Lorsqu'une branche est active et qu'on effectue des commits, le pointeur de cette branche (**`HEAD`**) se déplace automatiquement pour pointer vers le dernier commit réalisé (**`HEAD`** suit toujours le dernier commit sur la branche active). Ainsi, la branche active enregistre et suit les nouveaux commits au fur et à mesure de leur création.

1- Placez-vous dans le dossier **`.git/refs/heads`**, puis tapez **`ls -l`** pour lister de ses fichiers. Chaque fichier représente une branche du dépôt, et le contenu du fichier contient le SHA-1 (*Secure Hash Algorithm*) du dernier commit sur cette branche. Tapez **`cat main`** pour afficher ce code.

Voici un ensemble de commandes permettant de gérer les branches :

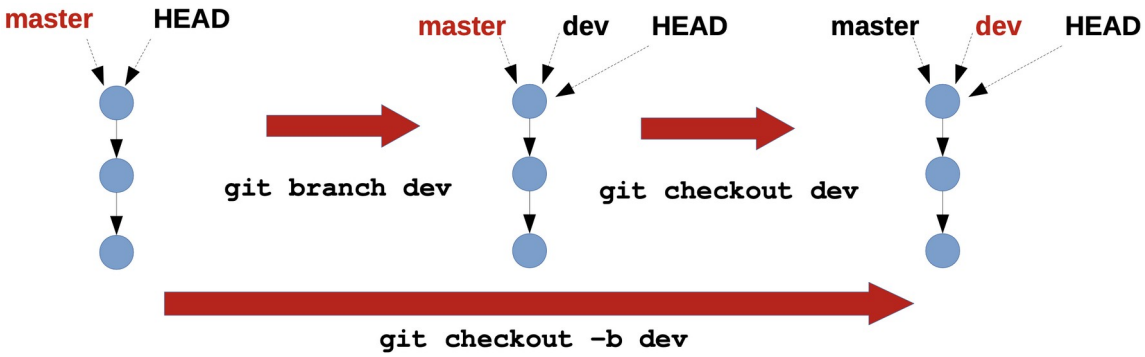
-**`git branch :`** pour connaître la liste des branches d'un projet donné

- **`git branch nomBranche :`** pour créer une nouvelle branche

-***`git branch dev 3c136a3 :`*** pour créer une branche à partir d'un commit particulier.

-**`git checkout nomBranche`**: pour positionner le répertoire de travail sur une branche spécifique.

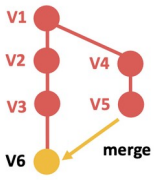
-**`git checkout -b nomBranche`** : pour créer la branche **`nomBranche`** et se positionne sur elle.



-**`git branch -d`** ***`nom_branche`*** : pour supprimer localement une branche

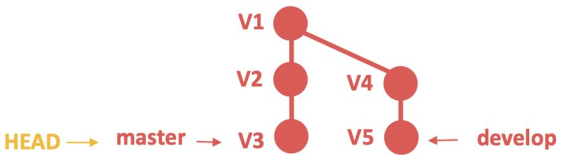
-**`git diff`** ***`nom_branche1...nom_branche2`*** : pour afficher les différences entre deux branches.

-**`git merge`** ***`nom_branche`*** : pour fusionner une branche dans la branche courante.

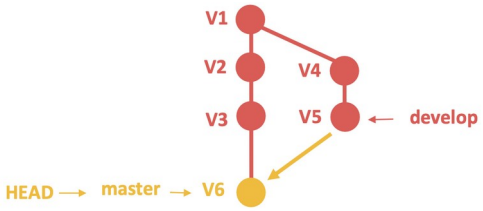


→ Explication de fusion de branches :

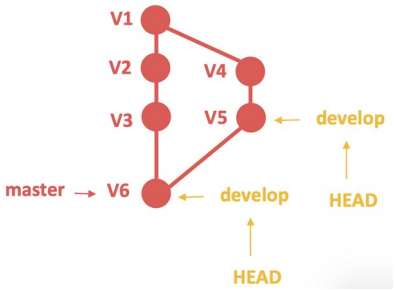
Soit l'arbre ci-contre ou la branche "***`master`***" référence la version "V3". La branche "**`develop`**" référence la version "V5". Pour se positionner sur la branche à fusionner (sauf si c'est déjà le cas) : ***`git checkout master`***



Pour fusionner la branche "**`master`**" avec "**`develop`**" on utilise : **`git merge develop`** Sauf cas exceptionnel, Git va créer un "commit" de fusion. Dans le cas d'un conflit, il faut résoudre le conflit puis créer un "commit" de fusion. La branche "**`develop`**" n'est pas impactée par l'opération.



Et pour mettre à jour la branche "**develop**", il faut se positionner sur la branche "**develop**" avec : **git checkout develop**, puis fusionner la branche "**develop**" avec la branche "**master**" en utilisant : **git merge master**.



2- Se positionner à nouveau dans le répertoire « **RapportDeStage** », puis afficher l'historique des commits. Créer une nouvelle branche nommée « **citation** », puis afficher la liste des branches (on aura **citation** et ***master** (ou **main**), où * indique que **master** ou **main** est la branche active actuellement).

3- Se positionner sur la branche « **citation** » et afficher la liste des branches (vous obtenez ***citation**, **master** ou **main**). À ce stade, tous les fichiers qui étaient disponibles et dans l'état où ils étaient au moment du second commit sur **master** ou **main** sont accessibles et identiques sur la branche citation.

4- Changer le contenu des trois chapitres. Ajouter ces fichiers à la zone d'index, puis créer un commit avec comme message : « **Modification chapitres avec des citations** ».

5- Afficher les historiques des changements et noter qu'il y un commit sur la branche « **citation** » et des commits sur la branche « **master** ou **main**».

6-Revenir à la branche principale **main**. Noter que les trois fichiers ne sont pas modifiés. Créer un fichier nommé « **Chapitre4.txt** ». Ajouter le contenu à la zone d'index, puis faire un commit avec comme message « **Ajout de Chapitre 4** ». Afficher l'historique des commits et vérifier que vous avez 4 commits.

7- Pour fusionner deux branches dans Git, voici les étapes à suivre : commencez par créer une nouvelle branche à partir de la branche principale. Sur cette nouvelle branche, effectuez les modifications nécessaires et validez-les par un commit. Ensuite, revenez sur la branche principale, qui est la branche destinataire pour la sélectionner. Enfin, fusionnez la branche de fonctionnalité avec la branche principale actuelle en exécutant **git merge feature-branch**. Fusionner les commits de la branche **citation** dans **master** ou **main.** Vous obtenez :

Vous pouvez taper un message de fusion en appuyant sur la touche A. Taper ESC, puis **`:q !`** pour quitter l'éditeur de texte. Notez les modifications dans les fichiers.

Pour voir les différences entre deux états du projet (par exemple, avant et après la fusion), on utilise **`git diff`**. Pour visualiser les modifications entre la branche **`main`** avant la fusion et après, taper : **`git diff HEAD^ HEAD`**, avec **`HEAD^`** fait référence au commit juste avant le dernier (c'est-à-dire l'état de main avant la fusion), et **`HEAD`** est le dernier état après la fusion.

→ Gestion de conflit de fusion :

Un conflit de fusion (merge conflict) dans Git se produit généralement lorsque deux branches ont apporté des modifications différentes au même segment de contenu dans un fichier, et que Git ne peut pas résoudre automatiquement ces modifications. Cela se produit souvent quand les mêmes lignes d'un fichier sont modifiées de manière différente sur deux branches distinctes que vous essayez ensuite de fusionner. Les conflits de fusion sont plus susceptibles de se produire dans les situations suivantes :

→ **Modifications Concurrentes** : Deux branches modifient la même partie d'un fichier. Par exemple, si on change une ligne dans **`Chapitre1.txt`** sur la branche **`main`** et qu'un collaborateur ou une autre branche, comme **`citation`**, modifie également cette même ligne.

→ **Suppression et Modification Concurrente** : un conflit peut aussi survenir si une branche supprime un fichier tandis qu'une autre branche effectue des modifications sur ce même fichier.

→ **Renommages Concurrents** : Si un fichier est renommé différemment dans deux branches distinctes, cela peut également créer un conflit lors de la fusion.

Pour résoudre un conflit de fusion, il faut suivre ces étapes : Ouvrir les fichiers en conflit, rechercher les marqueurs de conflit (**`<<<<<<<`**, **`=======`**, **`>>>>>>>`**) et

examiner les différences. Décider du contenu à conserver, modifier ou fusionner. Ensuite, éditer le fichier, supprimer les marqueurs de conflit et modifier le fichier pour incorporer la version finale du contenu. Finalement ajouter le fichier résolu à l'index pour marquer le conflit comme résolu, et terminer la fusion avec un commit.

8- Créer et sélectionner une nouvelle branche nommée **`revision-finale`**.

9- Changer la première ligne du contenu du **`Chapitre4.txt`**. Par exemple : « Pour discuter mes résultats-**Cette ligne a été modifiée dans la branche revision-finale** ». Modifier également la première ligne du **`Chapitre2.txt`**. Ajouter les modifications à l'index et faire un commit.

Sélectionner la banche **`main`** (ou **`master`**), puis modifier les mêmes lignes des chapitres 2 et 4. Ajouter les modifications à l'index et faire un commit.

Fusionner les modifications de la branche **`revision-finale`** dans la branche **`main`** (ou **`master`**). Que remarquez-vous ?

10- Pour résoudre le conflit, accéder au fichier qui contient le conflit, choisir le contenu à conserver en supprimant les marqueurs de conflit ainsi que les portions de texte qu'on ne veut pas, puis choisissez le contenu à sauvegarder et supprimer le reste. Ajouter le fichier à l'index et faire un commit pour valider les modifications.

Pour approfondir vos connaissances de git, on vous recommande de compléter les défis proposés ici : https://learngitbranching.js.org/

Partie 4 : Utilisation de gitHub (Dépôt distant)

Les dépôts distants comme GitHub ou Bitbucket servent de points centraux pour échanger des mises à jour entre collaborateurs. Les commandes Git pour interagir avec ces dépôts incluent **`git push`**, **`git fetch`**, et **`git pull`**. Les commandes **`git add`**, **`git commit`**, et **`git checkout`** sont utilisées pour gérer le dépôt local (voir figure ci-dessous) :

→ **`git add`** : Prépare les modifications (ajouts, modifications, suppressions de fichiers) pour le commit en les plaçant dans la zone d'index.

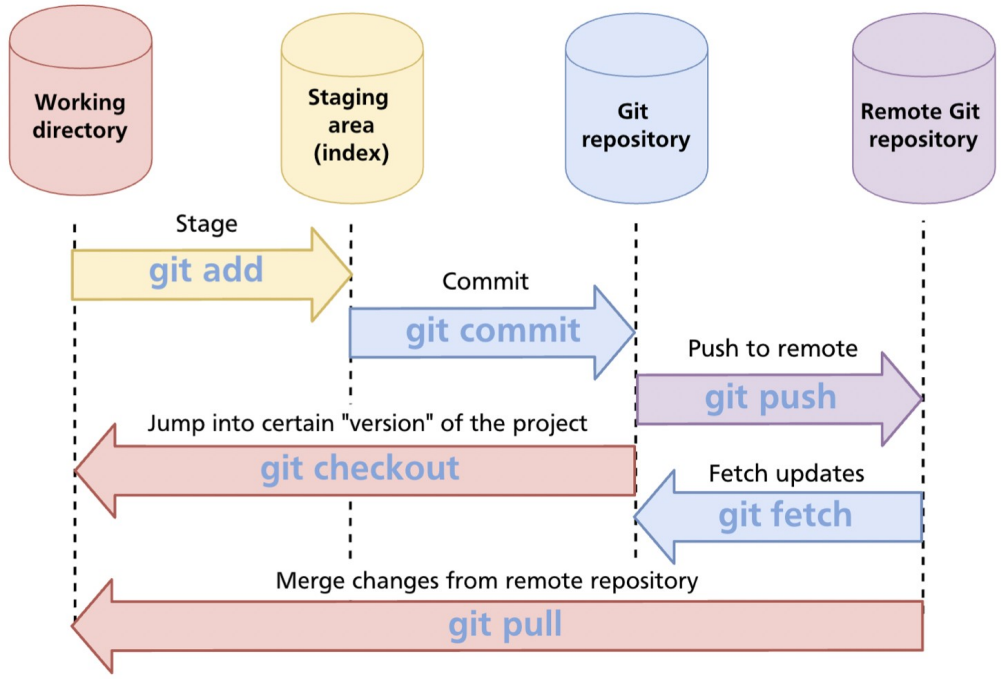
→ **`git commit`** : Enregistre les modifications de la zone d'index dans le dépôt local.

→ **`git checkout`** : Permet de changer de branche ou de revenir à un commit spécifique.

Pour les interactions distantes :

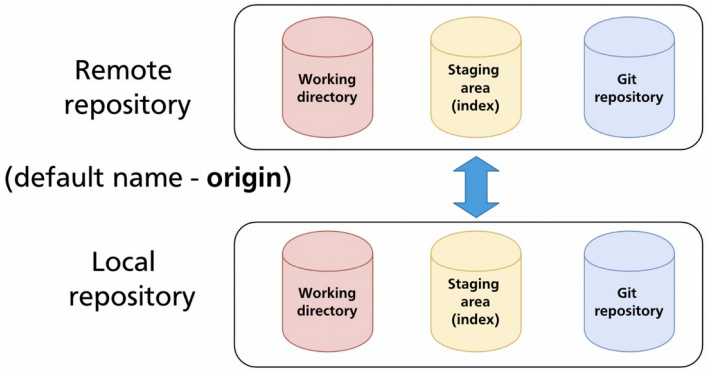
→ **`git push`** : Après avoir commité localement, les modifications sont poussées vers le dépôt distant. Ce processus suppose que les changements sont déjà validés localement et ne briseront rien dans le projet.

→ **`git fetch`** : Met à jour le dépôt local avec les informations du dépôt distant sans changer le répertoire de travail local.
→ **`git pull`** : Combine **`git fetch`** suivi d'un **`merge`** dans le répertoire de travail, actualisant la zone d'index et le répertoire avec les dernières modifications du dépôt distant.
Avec **`git merge`** Immédiatement après le **`fetch`**, git tente de fusionner les modifications téléchargées dans la branche locale.



La question qui se pose maintenant est comment le dépôt Git local (*local repository*) est-il connecté au serveur ou dépôt distant (*remote repository*) ?

Par défaut, lorsqu'on clone un dépôt distant sur un dépôt local, Git crée automatiquement une liaison entre les deux, et il définit un dépôt distant par défaut pour le dépôt local. Le nom par défaut de ce dépôt distant est **`origin`**. On peut choisir n'importe quel nom, mais "**`origin`**" est une convention standard utilisée pour désigner la branche par défaut ou la branche principale utilisée pour toutes modifications. Ainsi, il est possible de connecter un dépôt local à plusieurs dépôts distants. Dans ce cas, chaque dépôt distant aura un nom différent. Lorsqu'on effectue des



opérations comme **`push`**, **`pull`** ou **`fetch`**, on choisit en fait avec quel dépôt distant on souhaite interagir.

1- Créer un compte sous GitHub, puis créer un nouveau dépôt **public** avec « **`New repository`** ». Nommer le dépôt « **`RapportDeStage`** » et saisir une brève description puis appuyer sur le bouton : « **`create repository`** ». Ce dépôt est repéré par une adresse (URL) : **`https://github.com/NomUtilisateur/RapportDeStage.git`**

Vous pouvez pousser les commits effectués ci-dessus en ligne de commande. Pour ce faire, il faut créer un dépôt distant (*remote repository*) avec :

`git remote add <nom> <url_de_votre_dépôt_distant>`

2- Liez votre dépôt local à celui distant, attribuez-lui le nom **`origin`** en ajoutant une nouvelle référence. Utilisez la commande **`git remote`** pour afficher les noms des dépôts distants configurés pour votre dépôt local, et la commande **`git remote -v`** pour afficher la liste des dépôts distants avec les URLs associées.

3- Taper ensuite la commande : **`git push -u origin main`**

Cette commande est utilisée pour envoyer les modifications de la branche **`main`** du dépôt local vers la branche correspondante dans le dépôt distant, qui est ici identifié par l'alias **`origin`**. L'option **`-u`** joue un rôle important dans cette opération. Cette option est une abréviation de **`--set-upstream`**. Lorsqu'on l'utilise avec **`git push`**, elle définit la branche distante spécifiée (**`main`** sur **`origin`** dans ce cas) comme la branche ***upstream*** (en amont) par défaut pour la branche locale actuelle (**`main`**). Configurer une branche en amont signifie que Git associe la branche locale à une branche spécifique sur le serveur distant. Une fois que la branche en amont est définie, on peut ensuite utiliser **`git pull`** ou simplement **`git push`** sans spécifier explicitement le nom de la branche ou du dépôt distant pour ces commandes futures. Git comprendra qu'on fait référence à la branche **`main`** sur **`origin`**. Cela simplifie les commandes futures, car Git sait déjà quelle branche distante est liée à votre branche locale.

Si erreur, taper **`git branch`** pour avoir une idée sur sur les branches qui existent dans le dépôt local. Si vous trouvez la branche **`master`**, remplacer **`main`** par **`master`**.

Vérifier que tous les fichiers de votre dépôt local sont hébergés sur votre dépôt distant.

Cliquer sur **`Insights`** puis **`Network`**, vous trouvez votre branche principale qui contient les commits.

Cliquer sur **`Code/N Commits`**, avec **`N`** est le nombre de commits pour visualiser l'ensemble des changements effectués.

4- Qu'affiche la commande **`git branch -a`** ?

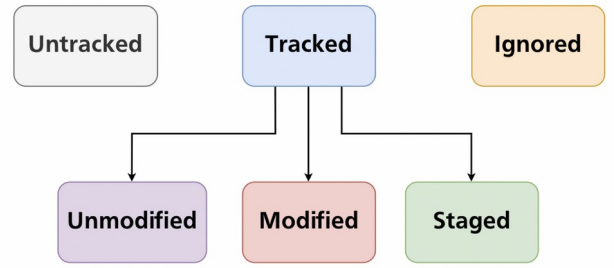
Parmi les résultats de la commande précédente, on affiche **`remotes/origin/main`** qui est une référence à la branche distante. Le préfixe **`remotes/origin/`** indique que cette branche est une branche distante sur le dépôt distant nommé **`origin`**. **`main`** est le nom de la branche sur ce dépôt distant.

5- Qu'affiche la commande **`git pull`** ?

Sur votre dépôt distant, ajouter un 5e chapitre à la branche citation puis effectuer à nouveau un **`git pull`**, Que remarquez-vous ? Afficher l'historique des modifications effectuées.

Partie 5 : Ignorer des fichiers avec .gitignore

Le fichier **`.gitignore`** est utilisé pour dire à Git quels fichiers et dossiers ignorer, ce qui signifie que les modifications apportées à ces éléments ne seront pas suivies par Git. Ce fichier contient des règles définissant les éléments à ignorer et est généralement placé à la racine du dépôt. Il est crucial de commettre le fichier .gitignore lui-même dans le dépôt afin qu'il soit pris en compte. Habituellement, ce fichier est créé au début du projet pour gérer efficacement les fichiers dès le démarrage.
Ignorer des fichiers introduit un état supplémentaire dans la gestion des fichiers par Git, comme illustré dans le diagramme associé.



1- À votre répertoire ajouter deux fichiers : «**`secrets.txt`** » et « **`.gitignore`** ». Dans « **`secrets.txt`** », sauvegarder deux mots de passe.

Dans « **`.gitignore`** », on ajoute les fichiers devant être ignorés lorsqu'on livre le projet à Git. Ajouter ces nouveaux fichiers à la zone d'index, puis afficher l'état du dépôt.

Vous remarquez que le fichier contenant les mots de passe a été ajouté avec peut être des fichiers générés par le système d'exploitation (**`.DS_store`** dans le cas de MacOs). Pour les supprimer de la zone d'index, taper la commande :

`git rm --cached -r .` (supprimer récursivement **`-r`** tous les fichiers qui se trouvent dans la zone d'index)

Afficher l'état du dépôt.

Ouvrir le fichier « **.gitignore** », puis taper sur chaque ligne les noms des fichiers à ignorer :

```
.DS_store
secrets.txt
# Pour un commentaire et ignore tous les fichiers log
*.log
```

Voici des exemples de motifs couramment utilisés dans un fichier « **.gitignore** » :

- Les lignes vides ou commençant par # sont ignorées.
- ***.ext** : Ignore tous les fichiers avec l'extension **.ext** (**.class**, **.log**, etc).
- **cible/** : ignore le dossier **cible** et tout son contenu.
- **/conf** : ignore le dossier **conf** situé à la racine du dépôt
- **!test.class** : suivre le fichier **test.class** malgré la règle précédente (***.ext** où **ext** est **class**)
- **src/*.txt** : ignorer les fichiers **.txt** du répertoire **src.**

Ajouter tous les fichiers à la zone d'index à nouveau et afficher l'état du dépôt. Que remarquez-vous ?

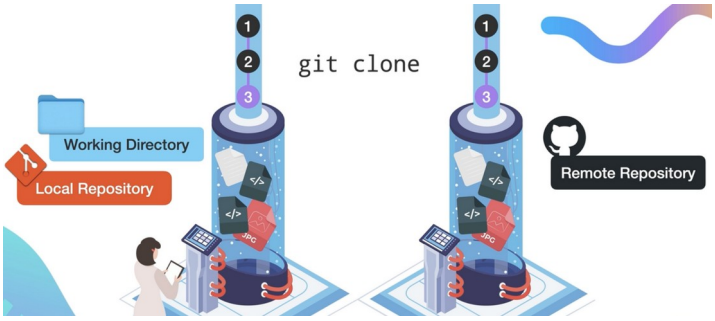
Faire un commit avec comme message : « **Ignorer OS fichiers et MDP** ». Afficher l'historique des commits.

Envoyer les modifications à votre dépôt distant. Que remarquez-vous ?

https://github.com/github/gitignore contient une collection de modèles utiles de fichiers **.gitignore**. Ces modèles sont conçus pour aider les développeurs à configurer facilement les fichiers **.gitignore** pour différents types de projets et environnements de programmation.

Partie 6 : Cloner un dépôt distant

Cloner un dépôt distant, dans le contexte de la gestion de versions avec des systèmes comme git, signifie créer une copie locale d'un dépôt de code qui est stocké sur un serveur distant. On utilise pour cela la commande **git clone url**, où **url** représente l'url du dépôt distant :



Ceci permettra d'exploiter le code de quelqu'un d'autre, d'étendre ses fonctionnalités et l'adapter selon vos besoins.

1- **Wordle** est un jeu populaire en ligne de devinettes de mots. Son but est de deviner un mot mystère en six essais ou moins. Le mot mystère est généralement un mot de cinq lettres. Il a été acheté par le New York Times tellement il est devenu viral.
L'url du dépôt distant de ce jeu est : **https://github.com/ritik48/Wordle-Game.git**
Cloner cet espace distant. Vérifier l'existence du répertoire « **Wordle-game** » dans votre espace de travail.
Il s'agit d'un jeu développé en 100 % avec Python, ouvrir le répertoire de ce jeu, créer un nouveau environnement virtuel et l'activer ensuite, puis installer **requirements.txt** avec **pip install -r requirements.txt**. Lancer le jeu. Amusez-vous.

Partie 7 : Collaborer à plusieurs sur un projet

Supposons que vous avez un projet de gestion de compte en Python où différentes opérations comme la création de compte, l'affichage des détails, le dépôt, le retrait, et l'obtention du solde doivent être implémentées. Chaque étudiant est responsable de développer une fonction spécifique. Le projet est hébergé sur GitHub pour faciliter la collaboration via ll'URL suivante :

https://github.com/MiryemH/gestion_compte_bancaire.git

Voici les étapes à suivre pour collaborer avec Git et GitHub :

→**Étape 1** : votre enseignante (ou le responsable du projet) ajoute tous les collaborateurs via **Settings/Manage Access/Add People**.

→**Étape 2** : Chaque collaborateur clone le dépôt sur son ordinateur local

→**Étape 3** : Chaque collaborateur crée une branche pour la fonctionnalité qu'il développera : **git checkout -b fonctionnalite_nom**

→**Étape 4** : Chaque collaborateur développe sa partie assignée en local, teste son code, puis commit ses changements avec

```
git add .
git commit -m "Ajout de la fonctionnalité X"
```

→**Étape 5** : Après avoir commité leurs changements locaux, les collaboratues poussent leurs branches sur GitHub avec : **`git push origin fonctionnalite_nom`**

→**Étape 6** : Une **`Pull Request`** (PR) permet à un contributeur de notifier les membres d'un projet sur des modifications qu'il a poussées vers un dépôt. La **`pull request`** demande aux autres développeurs du projet de revoir le code avant de le fusionner (**`merge`**) dans la branche principale ou toute autre branche cible. Chaque collaborateur crée une **`Pull Request`** pour fusionner sa branche avec la branche principale (**`main`**).

→**Étape 7** : Le responsable enseignant ou un étudiant, révise les **`Pull Requests`**, discute des modifications si nécessaire, puis les fusionne.

→**Étape 8** : Après la fusion, chaque collaborateur met à jour sa branche principale locale avec **`git checkout main`** et **`git pull origin main`**

→**Étape 9** : Après la fusion, les branches de fonctionnalités peuvent être supprimées pour garder le dépôt propre avec **`git branch -d fonctionnalite_nom`** et **`git push origin --delete fonctionnalite_nom`**