TDLOG séance 3 - outils et bonnes pratiques

Xavier Clerc – xavier.clerc@enpc.fr

6 octobre 2025

Table des matières

1 Introduction

À la fois pour démystifier certains aspects de Git et GitHub déjà utilisé, et également pour préparer à leur utilisation dans le cadre des projets, cette séance va présenter les aspects de base de cet outil et de ce service. Nous commencerons par des généralités sur les gestionnaires de versions (section 2) avant de présenter les principales opérations de Git (section 3). Nous présenterons ensuite les éléments essentiels de deux fonctionnalités plus avancées : les branches (section 4) et les dépôts distants (section 5). Enfin, la section 6 présentera un certain nombre de bonnes pratiques liées à l'utilisation de GitHub, notamment les *pull requests* et l'intégration continue.

2 Gestion de versions

L'objectif premier d'un gestionnaire de versions est de garder la trace des évolutions des fichiers d'un projet au fil du temps. En ce sens, l'utilisation d'un gestionnaire de versions n'est pas strictement liée au développement logiciel. Un gestionnaire de versions peut en effet s'avérer très utile dans d'autres contextes, comme la rédaction d'un article. Une limite majeure de la plupart des gestionnaires de versions, cependant, est de ne travailler au meilleur de leurs possibilités que sur des contenus textuels.

Un gestionnaire de versions, parce qu'il garde la trace de toutes les versions, supporte d'autres objectifs, tels que la possibilité de consulter des versions passées, la facilité avec laquelle il est possible de collaborer avec d'autres (puisque le risque de perdre des modifications est sensiblement plus bas) ou de travailler sur des expérimentations (puisqu'il est possible et même aisé de revenir en arrière si l'on n'est pas satisfait).

On peut voir un gestionnaire de versions comme un logiciel relativement complexe qui s'appuie sur deux outils basiques : `diff` et `patch`. Supposons que l'on dispose de deux variantes d'un même contenu, comme Code 2.1 et Code 2.2. Si l'on applique `diff` à ces deux fichiers, on obtiendra Code 2.3.

Code 2.1 – Premier fichier (first.py).

```python
def say(name: str, msg: str) -> None:
    print(f"[{name}] {msg}")

LOGINS : list[str] = []

def start_chat(login: str) -> None:
    say(login, "hello!")
```

Code 2.2 – Second fichier (second.py).

```python
def say(name: str, msg: str) -> None:
    print(f"{name}: {msg}")

def start_chat(login: str) -> None:
    say(login, "hello!")

def end_chat(login: str) -> None:
    say(login, "bye!")
```

Code 2.3 – Différences entre les deux fichiers.

```
@@ -1,7 +1,8 @@
 def say(name: str, msg: str) -> None:
-    print(f"[{name}] {msg}")
+    print(f"{name}: {msg}")

-LOGINS : list[str] = []
```

(suite sur la page suivante)

(suite de la page précédente)

```
-
 def start_chat(login: str) -> None:
     say(login, "hello!")
+
+def end_chat(login: str) -> None:
+    say(login, "bye!")
```

Code 2.3 contient les différences entre les deux fichiers, sous la forme d'un *patch*. Les additions (lignes commençant par un + et figurées en vert) et les délétions (lignes commençant par un - et figurées en rouge) indiquent les modifications à apporter pour "transformer" le premier fichier en le second.

L'outil `patch`, complémentaire de `diff`, permet de calculer "second.py" en appliquant les modifications à "first.py". Il permet aussi de calculer "first.py" si l'on dispose de "second.py", simplement en *inversant* les modifications : les additions deviennent des délétions et les délétions deviennent des additions.

Ce qu'un gestionnaire de versions ajoute aux outils `diff` et `patch` est leur intégration sous la forme d'un ensemble de patches organisés sous la forme d'un historique et avec des métadonnées. Ces métadonnées vont par exemple stocker la date, l'auteur et les raisons d'une modification.

3 Git : opérations de base

3.1 Zones

On présente dans cette section les opérations de bases du gestionnaire de versions Git, très probablement le plus populaire aujourd'hui. Pour bien comprendre la sémantique des opérations, il est utile d'avoir à l'esprit la manière dont les informations sont organisées.

En première approximation, il existe deux zones principales comme le montre Figure 3.1 : la copie de travail et le dépôt. La copie de travail est simplement constituée des fichiers que l'on a l'habitude de manipuler. Le dépôt, sous le répertoire *caché* `.git`, est la partie du système de fichiers que Git utilise pour stocker l'historique et l'ensemble des informations dont il a besoin. L'utilisateur de Git ne modifie jamais directement les fichiers du répertoire `.git`. Il exécute des commandes qui ont pour résultat des modifications du contenu de `.git` et/ou de sa copie de travail.

Pour appréhender au mieux le fonctionnement de Git, il est utile de raffiner l'approximation précédente et de considérer que le dépôt est constitué de deux sous-parties, comme le montre Figure 3.2. Git maintient l'historique des patches déjà évoqué et une zone de "chargement" (*staging area*) ou "index". Cette zone sert à préparer les patches que l'on souhaite enregistrer.

3.2 Création d'un dépôt

Afin de pouvoir travailler avec Git, il faut créer un dépôt. Cela se fait principalement de deux manières : soit en créant un dépôt *ex nihilo*, soit en copiant un dépôt.

Pour créer un dépôt *ex nihilo*, et donc vide, on utilise la commande `git init` qui crée un dépôt dont la racine est le répertoire d'exécution de la commande. Cela crée un répertoire `.git` dont

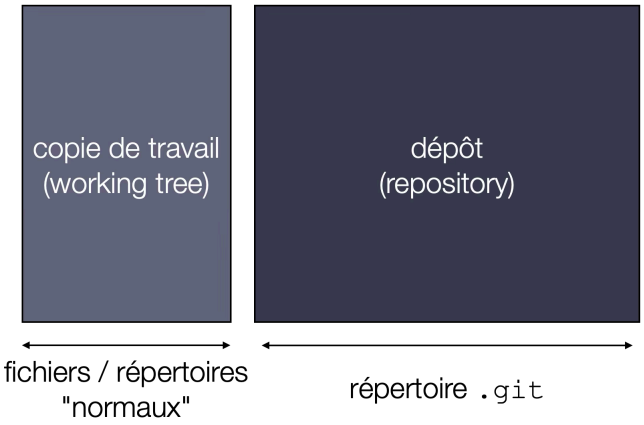

FIGURE 3.1 – Copie de travail et dépôt.

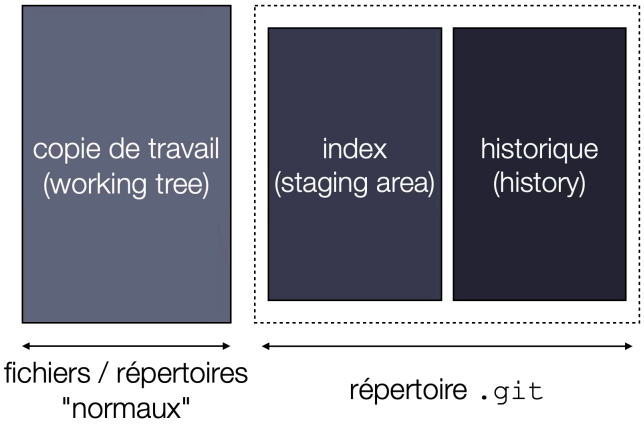

FIGURE 3.2 – Copie de travail et dépôt.

l'index et l'historique sont vides.

Pour créer un dépôt par copie, on utilise la commande `git clone` suivie de la source. Cette source peut être un répertoire local ou l'adresse d'un dépôt Git distant, comme par exemple `git@github.com:TDLOG/2526-TP2-GROUPE4-XC.git`. La commande va créer un répertoire dont le nom est la "partie principale" du dépôt à copier, `2526-TP2-GROUPE4-XC` dans notre exemple. Ce répertoire contient une copie de travail du dépôt copié, et son historique, mais un index vide. Le répertoire `.git` est placé sous le répertoire créé par la copie, `2526-TP2-GROUPE4-XC` dans notre exemple.

3.3 Configuration

S'il est déjà possible d'utiliser certaines commandes immédiatement après la création du dépôt, il n'est pas possible de créer des *commits* avant de s'être identifié (cela est logique dans la mesure où les métadonnées d'un *commit* incluent des informations sur son auteur).

Les deux éléments à renseigner pour s'identifier sont le nom (`user.name`) et l'adresse mail (`user.email`). Pour enregistrer ces éléments, on peut exécuter :

- `git config --add user.name` suivi du nom (entre guillemets s'il contient une espace) ;
- `git config --add user.email` suivi de l'adresse mail.

Les commandes précédentes n'enregistrent les informations que pour le dépôt courant. Si on souhaite les enregistrer plus largement, afin qu'elles s'appliquent par défaut à tous les dépôts sur la machine, on peut ajouter `--global` aux commandes ci-dessus.

On peut vérifier que les éléments ont été bien enregistrés en exécutant `git config --list` qui liste la configuration qui s'applique au dépôt.

3.4 Principales commandes

git status

La commande `git status` compare la copie de travail au dépôt et indique à l'utilisateur quels sont les fichiers de la copie de travail qui ont été modifiés, ajoutés, et supprimés depuis le dernier *commit*.

La commande `git status` ne modifie ni la copie de travail ni le dépôt.

git add

La commande `git add`, suivie d'un ou plusieurs noms de fichiers, copie le "diff" des ces fichiers entre la copie de travail et le dépôt vers l'index. La commande `add` sert donc à préparer le prochain *commit*, en décidant ce qui doit en faire partie, par modification de l'index.

La commande `git add` ne modifie pas la copie de travail, mais modifie l'index.

git commit

La commande `git commit -m "..."` sert à enregistrer un *commit* dans le dépôt. Elle déplace le diff dans l'index vers l'historique en créant un *commit* auquel elle attache des métadonnées comme l'identité de l'auteur et le message passé (`"..."` dans la commande ci-dessus). Dans les

projets au long cours, les messages de *commits* sont importants, car il permettent à l'utilisateur de se faire une idée des changements introduits par un *commit* dans avoir à lire le "diff" associé.

La commande `git commit` ne modifie pas la copie de travail, elle enregistre le "diff" de l'index dans l'historique, et vide l'index afin qu'il soit vide en préparation du prochain *commit*.

git diff

La commande `git diff` compare la copie de travail au dépôt et affiche les différences entre l'état enregistré par le dernier *commit* et la copie de travail. Le "diff" affiché correspond ainsi aux additions et délétions pour passer du dernier *commit* à la copie de travail.

La commande `git diff --staged` montre le "diff" dans l'index. Il s'agit donc du "diff" du *commit* qui sera enregistré si on exécute `git commit`.

La commande `git diff id1..id2` montre le "diff" entre les deux *commits* dont les identifiants sont passés (ici `id1` et `id2`). Il s'agit du "diff" qui permet de passer de l'état du dépôt tel qu'enregistré par le *commit* `id1` à l'état du dépôt tel qu'enregistré par le *commit* `id2`. L'identifiant spécial `HEAD` correspond au dernier *commit* enregistré.

La commande `git diff` ne modifie ni la copie de travail ni le dépôt.

git restore

La commande `git restore`, suivie d'un ou plusieurs noms de fichiers, remplace les fichiers de la copie de travail par leurs versions dans le dépôt. Il s'agit donc d'annuler des modifications de la copie de travail qui n'ont pas été enregistrées.

La commande `git restore` modifie la copie de travail, mais laisse le dépôt inchangé.

git log

La commande `git log` permet de consulter l'historique des *commits* enregistrés dans le dépôt. Les *commits* sont présentés en ordre antichronologique et par défaut sont affichés :

- l'identifiant du *commit* ;
- l'auteur du *commit* ;
- la date du *commit* ;
- le message associé au *commit*.

La commande `git log` ne modifie ni la copie de travail ni le dépôt.

4 Git : branches

L'apport fondamental des branches dans un gestionnaire de versions consiste à permettre de passer d'un historique linéaire à un historique arborescent. Il ne s'agit pas simplement d'augmenter l'expressivité de l'outil, mais d'un changement crucial en pratique, dans la mesure où les branches facilitent le travail en équipe et l'expérimentation.

En effet, il va être possible de travailler sur des variantes du code en parallèle avant de décider si certaines de ces variantes doivent être fusionnées dans le code du projet. Comme l'illustre Figure 4.1, on peut brancher (*fork*) à partir de n'importe quel *commit* de l'historique. Sur la figure, après le *commit* ayant le message "String conversion", on voit que l'historique continue sur la

branche principale avec "Tweak string representation" et sur une nouvelle branche nommée "third-dimension" avec "Z component".

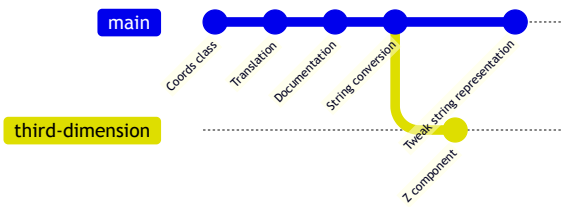

FIGURE 4.1 – Historique non linéaire (deux branches).

La commande `git branch` permet de consulter les branches existantes. Dans notre exemple on obtiendrait :

```
* main
  third-dimension
```

où l'étoile (*) indique la branche courante. En effet, si le gestionnaire de versions est capable de manipuler plusieurs branches, les fichiers de la copie de travail correspondent à un instant donné à l'état d'une branche donnée. Pour passer d'une branche à l'autre on peut utiliser `git checkout name` où "name" est le nom de la branche. La commande `git branch new-name` permet de créer une nouvelle branche avec comme point de départ le point de l'historique sur lequel nous nous trouvons et "new-name" comme nom de cette nouvelle branche.

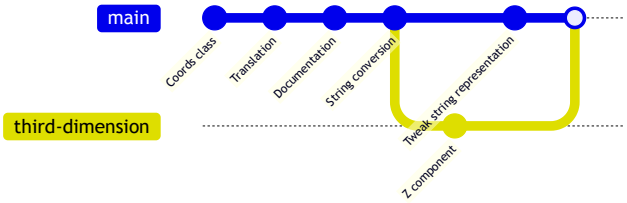

FIGURE 4.2 – Fusion (merge) d'une branche dans une autre.

Lorsque l'on travaille avec plusieurs branches, deux opérations sont particulièrement utiles : la fusion (*merge*) et le changement de base (*rebase*). L'opération de fusion (Figure 4.2) consiste à appliquer les changements intervenus sur une branche à la branche d'origine. On se place sur la branche auxiliaire et on exécute `git merge --into-name main-branch-name`.

L'opération de changement de base (Figure 4.3) consiste à modifier une branche pour que son point de branchement soit la tête (*i.e.* le dernier *commit*) de la branche d'origine. On se place sur la branche auxiliaire et on exécute `git rebase main-branch-name`.

On utilise typiquement la fusion lorsque l'on est satisfait du travail effectué sur la branche auxiliaire et que l'on souhaite l'incorporer à la branche d'origine ; cela marque donc la fin du travail sur la branche auxiliaire. On utilise le changement de base pour s'assurer que la branche auxiliaire dispose des changements apportés en parallèle à la branche principale, donc typiquement lorsque le travail sur la branche auxiliaire continue.

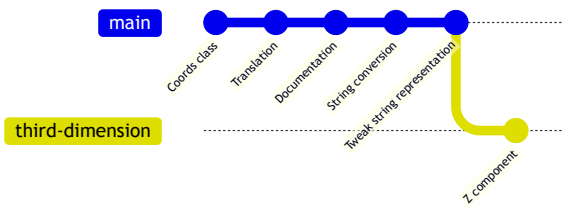

FIGURE 4.3 – Changement de base (rebase) d'une branche par rapport à une autre.

Code 4.1 – Exemple de conflits.

```
1  class Coords:
2
3    ...
4
5    def translate(self, dx: int, dy: int, dz: int) -> None:
6      self._x += dx
7      self._y += dy
8      self._z += dz
9
10   def __str__(self)->str:
11 <<<<<<< HEAD
12     return f"<{self._x}, {self._y}>"
13 =======
14     return f"({self._x}, {self._y}, {self._z})"
15 >>>>>>> b80e92a (Z component.)
```

Bien entendu, de telles opérations ne sont pas toujours directement possibles, car il est possible que des changements incompatibles aient été apportés sur les deux branches en question. Un gestionnaire de versions garde la trace, on l'a vu, des modifications effectuées par les *commits* ligne par ligne. Il est donc capable de détecter des modifications concurrentes sur les mêmes lignes. Il n'est en revanche pas capable de savoir laquelle des deux versions doit être conservée (ou si une troisième version doit être créée sur la base des deux autres).

Si le gestionnaire de versions détecte de tels *conflits*, il modifie les fichiers concernés en plaçant des "marqueurs" comme le montre Code 4.1. Les lignes 11 à 15 sont à interpréter comme suit :

- ligne 11 : le marqueur <<<<<<< indique le début du conflit ;
- ligne 12 : contenu du fichier sur la branche principale (`HEAD` sur la ligne précédente) ;
- ligne 13 : le marqueur ======= indique la séparation entre les deux versions en conflit ;
- ligne 14 : contenu du fichier après le commit `b80e92a` sur la branche auxiliaire (*cf.* ligne suivante) ;
- ligne 15 : le marqueur >>>>>>> indique la fin du conflit.

Il est important de noter que Git applique un critère purement textuel, sur la foi des numéros des lignes modifiées. Cela signifie que si les branches modifient des parties différentes du fichier, cela n'engendrera pas de conflits. Cela signifie aussi que si les branches effectuent la même modification des mêmes lignes, cela n'engendra pas de conflits non plus. Ces deux cas sont possiblement problématiques du point de vue du développement. Dans le premier cas, on peut par exemple ajouter un paramètre à une fonction, ce qui "cassera" le programme aux points

d'appel sans qu'une forme de conflit ne soit détectée. Dans le second cas, on peut sur les deux branches par exemple incrémenter (`+ 1`) une valeur donnée pour deux raisons différentes, ce qui devrait se combiner en un delta plus important (`+ 2`) mais ne sera pas non plus détecté comme conflit.

Pour résoudre des conflits, le développeur doit éditer les fichiers concernés, choisir le code qu'il souhaite conserver, enlever les marqueurs de conflits, et utiliser `git add` et `git commit` pour enregistrer la résolution du conflit. Lors d'un *rebase*, Git déplace la base *commit* par *commit*. Cela signifie que des conflits peuvent se produire à chaque déplacement. Une fois la résolution pour un *commit* enregistrée, on peut continuer les déplacements en utilisant `git rebase --continue`.

5 Git : dépôts distants

Nous avons jusqu'ici décrit les opérations disponibles lorsque l'on utilise Git *localement*, c'est-à-dire sur une machine donnée. Git est cependant le plus souvent utilisé pour permettre à des membres d'une équipe de collaborer, ce qui implique de faire communiquer une copie locale du dépôt avec des copies distantes, afin d'échanger des *commits*.

Git a été conçu comme un gestionnaire de versions *distribué* ou *décentralisé*. Cela signifie qu'il peut exister de multiples copies du dépôt, qui peuvent chacune être utilisées localement et échanger des *commits* avec les autres copies, comme le montre Figure 5.1.

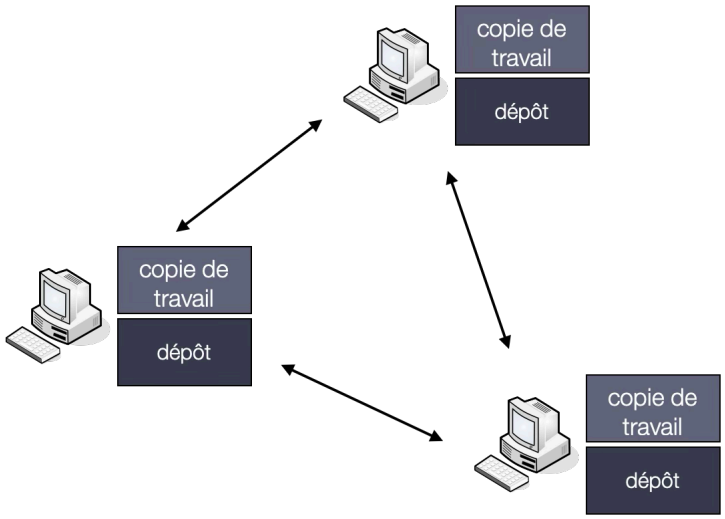

FIGURE 5.1 – Dépôts distants : théorie.

En pratique, cependant, il est utile, voire crucial, d'organiser le développement autour d'une copie "principale" ou "centrale" qui "fait foi". En effet, il est utile de pouvoir s'accorder facilement sur la version que l'on s'apprête à publier par exemple. Pour cette raison, bien que Git soit techniquement *décentralisé*, on organise la plupart des projets autour d'une copie centrale du dépôt comme le montre Figure 5.2. La figure mentionne le site `github.com`, qui est très populaire, mais des alternatives existent comme `gitlab.com` ou `bitbucket.org`.

Comme le montre Figure 5.3, les deux opérations essentielles lorsque l'on utilise des dépôts distants sont `pull` et `push`. On interprète ces commandes comme relatives à la copie locale du

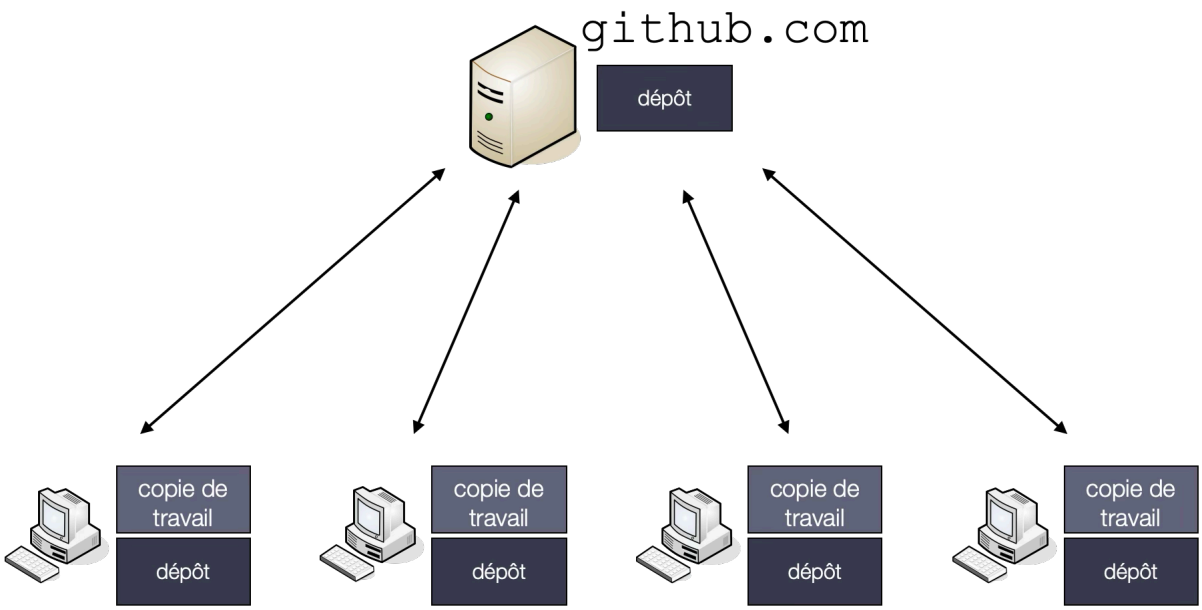

FIGURE 5.2 – Dépôts distants : pratique.

dépôt, de telle sorte que `pull` signifie "copier depuis le dépôt distant" et `push` signifie "copier vers le dépôt distant.

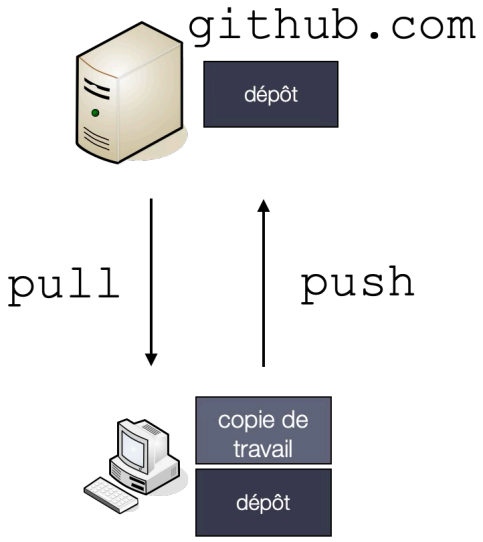

FIGURE 5.3 – Dépôts distants : opérations.

Si l'on travaille avec plusieurs dépôts distants, on peut faire figurer le nom du dépôt qui nous intéresse après `pull` ou `push`. On peut consulter la liste des dépôts distants connus en exécutant `git remote -v`. Si l'on a obtenu la copie locale du dépôt par une opération `clone`, le dépôt cloné est nommé "origin" et on obtient un affichage semblable à :

```
origin git@github.com:TDLOG/2526-TP2-GROUPE4-XC.git (fetch)
origin git@github.com:TDLOG/2526-TP2-GROUPE4-XC.git (push)
```

On peut ajouter de nouveaux dépôts distants en utilisant la commande `git remote add name`

`address` où "name" est le nom sous lequel on souhaite désigner (localement) le dépôt distant et "address" son URL.

Enfin, il est important de noter que l'utilisation de plusieurs copies du dépôt, comme l'utilisation locale de plusieurs branches, peut conduire à des conflits. En pratique, il ne sera pas possible de "pousser" les changements tant que l'on n'aura pas "récupéré" les changements, et la récupération échouera en cas de conflit. Dans ce cas, il suffit d'utiliser `git pull --rebase` et Git appliquera les marqueurs de conflits déjà évoqués à la copie locale. Il faudra alors résoudre les conflits et effectuer un *commit* de la résolution comme nous l'avons vu plus haut.

6 Bonnes pratiques

6.1 Pull request

Une pratique commune associée aux gestionnaires de versions est celle des *pull requests*. Dans le cadre de différents modules, elles ont pu être utilisées pour la remise de travaux pratiques, mais il s'agit en quelque sorte d'un détournement de leur usage principal.

L'idée fondamentale d'une *pull request* est de proposer une modification du code déjà présent dans le dépôt. Il faut entendre le terme "pull request" comme "request to pull code from another branch", c'est-à-dire comme la demande d'appliquer (*i. e. merge*) à une branche le code d'une autre branche.

Sur ce principe simple, on peut construire des processus plus élaborés. Par exemple, une *pull request* sur GitHub peut être aisément discutée : la page de la *pull request* permet de consulter le code proposé et de poster des commentaires, soit généraux soit attachés à des lignes particulières du code proposé. Cela permet d'effectuer une revue de code (*code review*) de la proposition.

L'équipe de développement s'entend sur le périmètre exact d'une *pull request*, mais en général il s'agit non seulement de s'assurer que le code est correct, mais également qu'il est de bonne qualité Ce dernier terme peut prendre différentes significations, mais il s'agit a minima de s'assurer que le code est facilement compréhensible et donc aisé à maintenir. Il peut également s'agir de verifier qu'un certain nombre de règles sont appliquées, là encore afin de faciliter la maintenance. Le fait de passer par des *pull requests* permet aussi aux développeurs d'être facilement au courant des différentes modifications apportées au code.

6.2 Organisation des branches

Sur la base des branches et des *pull requests*, une équipe peut mettre en place différentes formes d'organisation, qui correspondent à la manière dont les contributions des uns et des autres vont être fusionnées et déployées au fil du temps.

Une organisation simple, recommandée pour les projets TDLOG, est de considérer que la branche principale est celle qui contient le code du projet, et d'ouvrir une *pull request* pour chaque fonctionnalité ou correction de bug.

La figure Figure 6.1 illustre une telle organisation. On y voit que chaque branche part de la branche principale, est développée au fil du temps et est éventuellement fusionnée dans la branche principale. Bien entendu, il peut arriver qu'un développement soit infructueux ou rejeté. Sur la figure, c'est le cas de la branche nommée "feature-3", qui n'est tout simplement pas fusionnée.

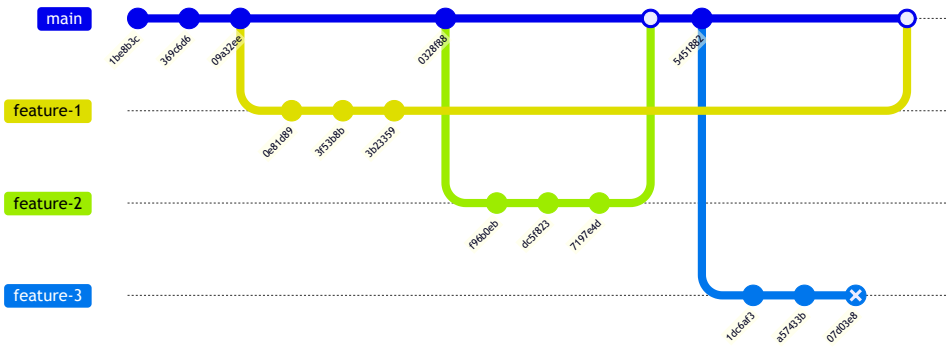

FIGURE 6.1 – Organisation des branches d'un projet.

Il existe des organisations plus complexes, par exemple, pour les projets qui doivent publier ou travailler sur plusieurs versions du projet. Par exemple, on peut imaginer qu'une équipe travaille simultanément sur des versions *1.x*, qui apporteront des fonctionnalités mineures et corrigeront des bugs, et sur la version *2.0* qui apportera des fonctionnalités majeures.

Dans ce cas, il est vraisemblable que, contrairement à l'organisation ci-dessus, nous aurons plusieurs branches au long cours. Au lieu de n'avoir que la branche "main", nous aurons probablement une branche "v1" et une branche "v2". Des *pull requests* pourront être ouvertes contre n'importe quelle branche au long cours, ce qui permettra de travailler simultanément sur les deux versions du logiciel. Bien entendu, certaines modifications doivent parfois être appliquées à plusieurs branches. Par exemple, la correction d'un bug peut devoir être appliquée à plusieurs versions. Pour faciliter cela, un gestionnaire de versions comme Git permet d'appliquer un *commit* quelconque à une branche (commande *cherry-pick*).

6.3 Intégration continue

L'intégration continue est une pratique qui consiste à exécuter automatiquement un certain nombre d'actions lorsqu'un événement donné se produit sur le dépôt. Par exemple, on peut décider d'exécuter la suite de tests pour validation à chaque *commit*. Cela permet, par exemple, de s'assurer que tous les tests "passent" avant de fusionner une *pull request*.

Le code Code 6.1 montre un exemple d'intégration continue. Lorsque l'on utilise GitHub, on peut stocker dans des fichiers nommés `.github/workflows/xyz.yml` des *workflows* GitHub Actions. L'exemple Code 6.1 montre le *workflow* qui était défini dans les dépôts des séances de travaux pratiques.

Code 6.1 – Exemple d'intégration continue (GitHub Actions).

```
1 name: Exercises
2
3 on: [pull_request, push]
4
5 jobs:
6   checks:
7     runs-on: ubuntu-latest
8     defaults:
```

(suite sur la page suivante)

(suite de la page précédente)

```
9       run:
10        shell: bash -l {0}
11    steps:
12    - uses: actions/checkout@v5
13      with:
14        ref: {{ github.event.pull_request.head.sha || github.ref }}
15    - uses: conda-incubator/setup-miniconda@v3
16      with:
17        activate-environment: env_exercises
18        environment-file: environment.yml
19        python-version: "3.13"
20        miniconda-version: "latest"
21        auto-update-conda: true
22        auto-activate-base: false
23    - name: List packages
24      if: always()
25      run: |
26        conda list
27    - name: Format with black
28      if: always()
29      run: |
30        black -check -diff .
31    - name: Lint with flake8
32      if: always()
33      run: |
34        flake8 .
```

Sans trop entrer dans les détails, on peut noter les éléments essentiels suivants :

— ligne 3 : événements qui doivent déclencher l'exécution du *workflow* ;
— ligne 6 : nom du *workflow* (ici `checks`) ;
— lignes 12 à 14 : obtention d'une copie du dépôt ;
— lignes 15 à 22 : installation d'une version de Python ;
— lignes 27 à 30 : exécution de l'outil `black` (qui vérifie que le code est bien formaté) ;
— lignes 21 à 34 : exécution de l'outil `flake8` (qui recherche des erreurs dans le code).

Les résultats des différentes étapes sont, bien entendu, consultables depuis la page de la *pull request* (s'il s'agit d'un événement déclenchant l'intégration continue, *cf.* ligne 3). Cela permet donc aux développeurs effectuant la revue de code de s'appuyer sur les diagnostics produits par les *workflows*.

Outre des outils "statiques" (*i. e.* ne nécessitant pas d'exécuter le code du projet) comme `black` (*formatter*) ou `flake8` (*linter*), il est habituel d'utiliser également des outils dynamiques (*i. e.* qui se fondent sur l'exécution du code). Le plus commun de ce type d'outils est l'exécution de tests.

Exécuter les tests à chaque *commit* permet de s'assurer qu'une *pull request* n'introduit pas de régression, c'est-à-dire qu'elle ne "casse" pas une fonctionnalité existante. En effet, lorsqu'un projet atteint une taille conséquente, il n'est pas possible d'effectuer manuellement tous les tests et il est important de pouvoir se reposer sur leur exécution automatique.

L'utilisation conjointe de tests et d'un gestionnaire de versions peut grandement faciliter le débugage d'un programme. Supposons que l'on constate un *bug* sur une version donnée du logiciel. Si l'on peut écrire un test automatique qui permet de dire si une version donnée est affectée par le bug, on peut procéder par dichotomie (*bisection*) afin de trouver le *commit* ayant introduit le *bug*. Cela signifie que l'on cherchera l'erreur introduite non pas sur l'ensemble du code, mais essentiellement sur le "diff" associé au *commit*. Bien entendu, cela doit amener à préférer des *commits* et *pull requests* de tailles raisonnables.