amU Faculté des sciences Aix Marseille Université

Année 2025-26

Domaine Sciences et Technologies
Master informatique
**Génie logiciel : TP 2**
**Codes UE : SINA09EL**
**Mars Rover (3/3)**

# 1 Introduction

Le but de ce TP est de continuer le développement du simulateur pour des *rovers* de la NASA explorant Mars du TP précédent.

L'objectif pédagogique du projet est de vous faire travailler sur d'autres outils et bonnes pratiques pour le développement logiciel :

- l'utilisation des tests unitaires pour assurer le bon fonctionnement du code et le cas échéant trouver facilement les erreurs de programmation ;
- La mise en place d'un pipeline très simple lançant le *build* et les *tests* du code à chaque *push* sur le serveur ;
- la gestion des branches au sein de la gestion de version ;
- l'utilisation d'outils de mesure de couverture de code afin d'évaluer la couverture par les tests.

# 2 Consignes pour le rendu

## 2.1 Rendu via `etulab`

Le rendu de votre projet *Mars rover* est à faire pour le **2 octobre 2025**. Il faudra que la branche principale (*main* ou *master*) de votre dépôt *git* sur `etulab` soit à jour pour le 2 octobre 2025 à 23h59 et que l'enseignant suivant soit membre du projet avec un rôle de `reporter` : Arnaud Labourel, identifiant `alaboure`.

Le travail peut être réalisé seul ou en binôme. Dans le cas de binôme, les deux étudiants devront être membres du projet.

## 2.2 Respect de la propriété intellectuelle

Comme pour tout devoir, nous vous demandons de ne pas partager votre programme, complet ou partiel, avec des étudiants n'étant pas membres de votre projet. Tout emprunt que vous effectuez doit être proprement documenté en indiquant quelle partie de votre programme est concernée et de quelle source elle provient (nom d'un autre étudiant, site internet, ...). Les emprunts incluent l'utilisation d'IA génératives telles que ChatGPT qui devront donc aussi être documentés.

## 2.3 Fichier `README.md` du projet

Votre projet devra contenir à sa racine un fichier `README.md` contenant les informations sur ces membres et les emprunts réalisés. Le format du fichier devra être le suivant :

```
# Mars Rover

## Membres du projet
```

```
- NOM Prénom du premier membre
- NOM Prénom du deuxième membre (si applicable)

## Description des emprunts

- Utilisation de ChatGPT pour les classes : `Main.java`, ...
- ...
```

### 2.4 Critères d'évaluation

Vous serez évalué sur :

- **La conception logicielle :** votre projet devra dans la mesure du possible respecter les bonnes pratiques de conception logicielle en programmation orientée objet tels que les principes SOLID. Par exemple, des classes ayant trop de responsabilités vous pénaliseront.
- **La propreté du code :** comme indiqué dans le cours, il est important de programmer proprement. Des répétitions de code trop visibles, des noms mal choisis ou des fonctions ayant beaucoup de lignes de code (plus de dix) vous pénaliseront. Le sujet vous donne les méthodes que vous devez absolument écrire, mais il est tout à fait autoriser d'écrire des méthodes supplémentaires, de créer des constantes, ... pour augmenter la lisibilité du code. On rappelle que vous devez écrire le code en anglais.
- **La correction du code :** on s'attend à ce que votre code soit correct, c'est-à-dire qu'il respecte les spécifications dans le sujet. Comme indiqué dans le sujet, vous devez tester votre code pour vérifier son comportement.
- **Les commit/push effectués :** il vous faudra travailler en continu avec `git` et faire des *push/commit* le plus régulièrement possible. Un projet ayant très peu de *pushs/commits* effectués juste avant la date limite sera considéré comme suspicieux et noté en conséquence. Un minimum accepté pour le projet sera d'au moins **2 pushs sur deux jours différents** et d'au moins **10 commits** au total. Dans le cas d'un projet réalisé en binôme, chacun des deux membres du projet devra réaliser au moins un *push*.

## 3 Test du code

Bien qu'il soit utile de tester son code manuellement (notamment pour les interfaces graphiques), il est souvent plus efficace de le tester automatiquement. Pour cela, le plus simple est d'utiliser JUnit qui permet d'automatiser les tests unitaires.

Les règles de base pour l'écriture des tests unitaires sont les suivantes :

- Mettre le code des tests ainsi que des classes créées pour faire les tests dans le répertoire `src/test/java` de votre projet.
- Écrire au moins une classe de test par classe à tester. Le nom de la classe de test est généralement le nom de la classe testée suivie de `Test`. Par exemple la classe de test d'une classe `Vector` est appelée `VectorTest`.

— Écrire au moins une méthode de test par cas à tester. Le nom de la méthode de test doit indiquer le comportement de l'objet qui est testé. Une manière répandue d'écrire le nom d'une méthode de test est de construire le nom de la méthode de test en mettant `test`, suivi du nom de la méthode, suivi d'un `_` et du comportement testé. Par exemple, un test vérifiant le comportement d'une méthode `withdraw` (retrait) sur un compte en banque à découvert (*overdrawn*) pourra être nommée `testWithdraw_whenOverdrawn`.
— Pour une méthode à tester, il faut tester :
  — **les cas normaux**, utilisation naturelle de la méthode sur une donnée naturelle, par exemple un retrait d'argent d'un montant strictement positif ;
  — **les cas limites**, utilisation de la méthode sur une donnée "étrange", par exemple par exemple un retrait d'argent d'un montant égal à 0 ;
  — **les cas anormaux**, vérification que les erreurs d'utilisation, c'est-à-dire que les cas d'erreurs sont bien pris en compte et gérés, par exemple un retrait d'argent d'un montant strictement négatif.

**Tâche 1 :** Ajoutez des tests à votre projet testant que votre code respecte les spécifications données dans la première planche de TP en suivant les consignes qui suivent.

### 3.1 Tests unitaires avec Junit 5

On vous demande donc de tester à l'aide de tests unitaires toutes les classes que vous avez écrites lors de la première séance de TP. Normalement votre projet est déjà configuré pour le framework de test JUnit 5 grâce au fichier de configuration de *gradle* (Le fichier `build.gradle.kts` à la racine de votre répertoire) qui contient les lignes suivantes (ou des lignes similaires) qui ajoutent *JUnit 5* à votre projet :

```
dependencies {
    testImplementation(platform("org.junit:junit-bom:5.10.0"))
    testImplementation("org.junit.jupiter:junit-jupiter")
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")
}
```

On vous conseille d'utiliser aussi la bibliothèque AssertJ qui ajoute entre autre des assertions facilitant l'écriture des tests. Pour cela, il vous faut rajouter *AssertJ* dans les dépendances de *gradle* en rajoutant la ligne suivante dans les `dependencies` du fichier `build.gradle.kts` :

```
dependencies {
    testImplementation("org.assertj:assertj-core:3.27.3")
}
```

Pour pouvoir utiliser facilement les assertions (méthodes `assertThat`) dans vos classes de test, il vous faut ajouter l'*import* suivant dans chacune vos classes de tests :

```
import static org.assertj.core.api.Assertions.*;
```

Vous trouverez davantage de détails sur les tests unitaires dans le document dédié aux tests

# 4 Intégration continue et déploiement/livraison continu (CI/CD)

CI/CD (*Continuous Integration/Continuous Delivery*) est la combinaison des pratiques d'intégration continue et de livraison/déploiement continu. L'intégration continue consiste à tester et contrôler (via des tests et d'autres outils de mesure de qualité de code) en permanence les modifications incrémentales du code. La livraison continue consiste à rajouter à l'intégration continue la diffusion automatisée du logiciel (le déploiement restant non automatisé dans ce cas). Le déploiement continu ajoute le déploiement automatique dans les environnements de production comme la mise à jour des serveurs avec la nouvelle version du logiciel dans le cas d'une application web.

L'instance Gitlab de l'université d'Aix Marseille connu sous le nom d'etulab est configuré pour le CI/CD avec des *runners* intégré qui permette d'exécuter automatiquement un pipeline contenant des tests et/ou des scripts de livraison/déploiement. Gitlab permet de facilement mettre en place un processus de CI/CD simple.

> **Tâche 2 :** Ajoutez un pipeline compilant et testant votre projet
> à chaque *push* sur le dépôt en suivant les consignes qui suivent.

La principale étape pour mettre en place l'intégration continue est de configurer le pipeline que vous souhaitez exécuter lors des *push* sur le git. Pour cela, il faut passer par un fichier de configuration `.gitlab-ci.yml` à la racine du projet. Le fichier est au format YAML et permet de définir :

- l'image à utiliser pour exécuter les scripts ;
- la structure en *stages* (étapes) des différents *jobs* à exécuter ;
- les différents *jobs* avec le script à exécuter pour chacun et les artéfacts associés, c'est-à-dire les fichiers produits à conserver sur le serveur.

Vous trouverez le fichier de base à rajouter à votre projet au lien suivant : .gitlab-ci.yml.

La première ligne du fichier configure l'image à utiliser comme étant l'image alpine de gradle qui est l'image conseillée pour un CI/CD utilisant *gradle* comme moteur de production.

```
image: gradle:jdk24-alpine
```

Les lignes suivantes indiquent que le pipeline est composé de deux *stages build* et *tests* qui sont des *stages* classiques.

```
stages:
  - build
  - test
```

Les lignes suivantes configurent les options de *gradle* afin que le *Gradle Daemon* soit désactivé, car il est inutile dans un contexte de pipeline CI/CD.

```
variables:
  GRADLE_OPTS: "-Dorg.gradle.daemon=false"
```

Les lignes suivantes définissent un script à lancer avant le pipeline. Il configure la variable système `GRADLE_USER_HOME` afin que *gradle* stocke sa configuration globale dans le répertoire du projet.

```
before_script:
  - export GRADLE_USER_HOME=`pwd`/.gradle
```

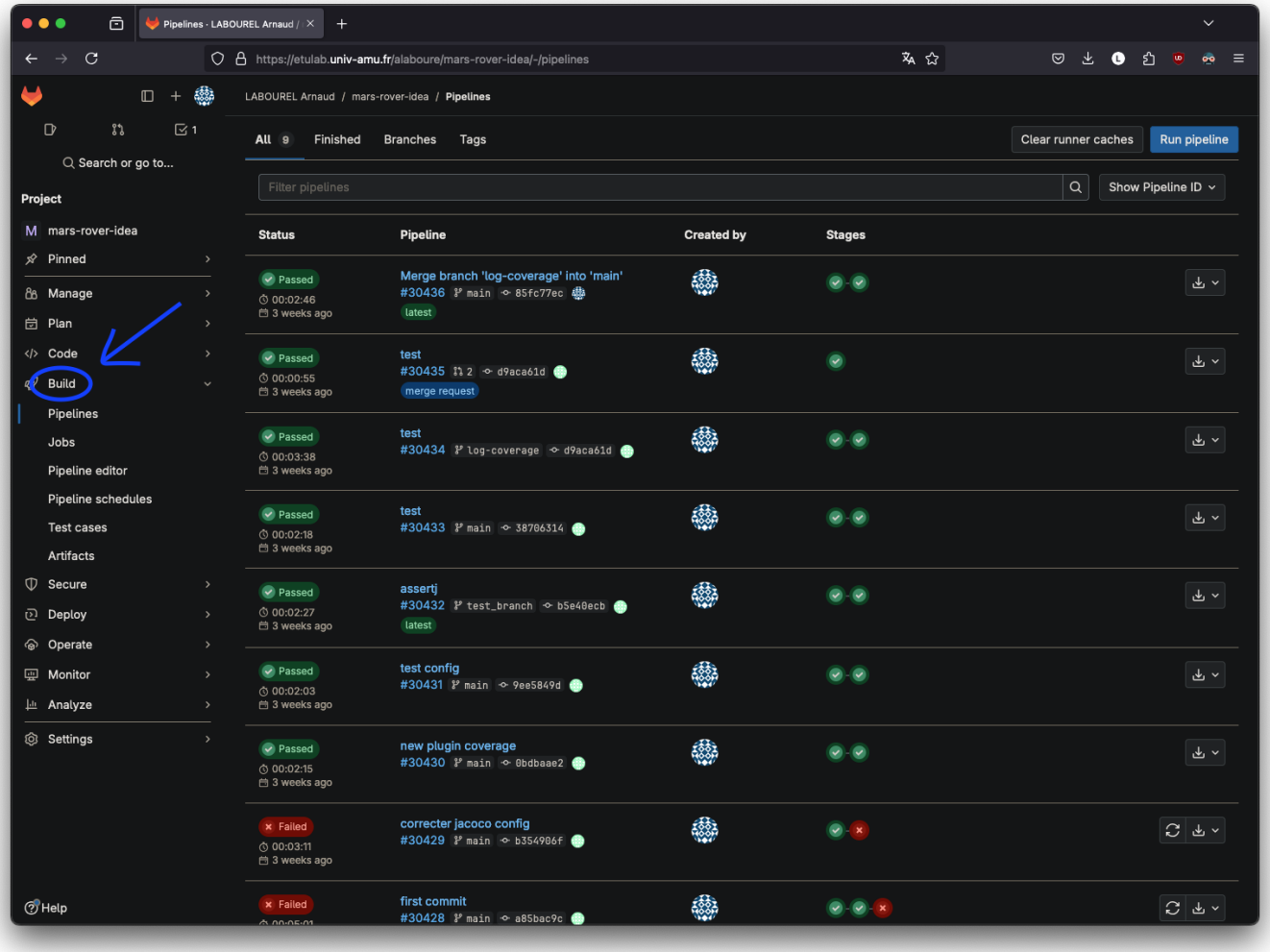
Les lignes suivantes définissent le job *build* dans le stage *build*. Ce *job* se contente d'exécuter la tâche *build* de *gradle* et de définir comme artéfact les fichiers *jar* produits dans le répertoire dédié.

```
build:
  stage: build
  script:
    gradle --build-cache build
  artifacts:
    paths:
      - build/libs/*.jar
    expire_in: 1 week
```

Finalement, les dernières lignes du fichier définissent le job *tests* dans le *stage test*. Ce *job* se contente d'exécuter la tâche *test* de *gradle* et de définir comme rapports les fichiers *xml* produits dans le répertoire dédié.

```
tests:
  stage: test
  script:
    - gradle test
  artifacts:
    when: always
    reports:
      junit: build/test-results/test/**/TEST-*.xml
```

Une fois que vous avez ajouté ce fichier à votre projet et que vous avez poussé la mise à jour sur le serveur, le pipeline devrait s'exécuter. Vous pouvez accéder à toutes les informations de vos pipelines via l'interface web de *gitlab* en cliquant sur *build* dans le menu de votre projet.




Le sous-menu *build* contient les articles suivants :

- *Pipelines* qui contient l'historique de tous les pipelines lancés par le projet (normalement un par *push*) ;
- *Jobs* qui contient l'historique de tous les *jobs* lancés par le projet (même contenu que l'article précédent, mais présenté par *job*) ;
- *Pipeline editor* permet de directement reconfiguré le pipeline via l'interface web ;
- *Test cases* qui permet de décrire des scénarios de test (pas utilisé dans ce TP) ;
- *Artefacts* permet de télécharger les artéfacts produits par les *jobs*.

# 5 Utilisation des branches

Une bonne pratique de développement pour des projets utilisant la gestion de versions est de créer des branches pour introduire les modifications du code. Grâce aux branches, les équipes de développement logiciel peuvent apporter des modifications sans affecter la branche principale (*main*). L'historique des *commits* est enregistrée dans une branche créée pour l'ajout de la fonctionnalité, et lorsque le code est prêt, il est fusionné dans la branche *main*. Les branches permettent donc d'organiser le développement et de séparer le travail en cours du

code stable et testé de la branche *main*. Cela permet d'éviter que certains bugs et vulnérabilités ne se glissent dans le code principal et n'affectent les utilisateurs, car il est plus facile de les tester et de les trouver dans une branche séparée.

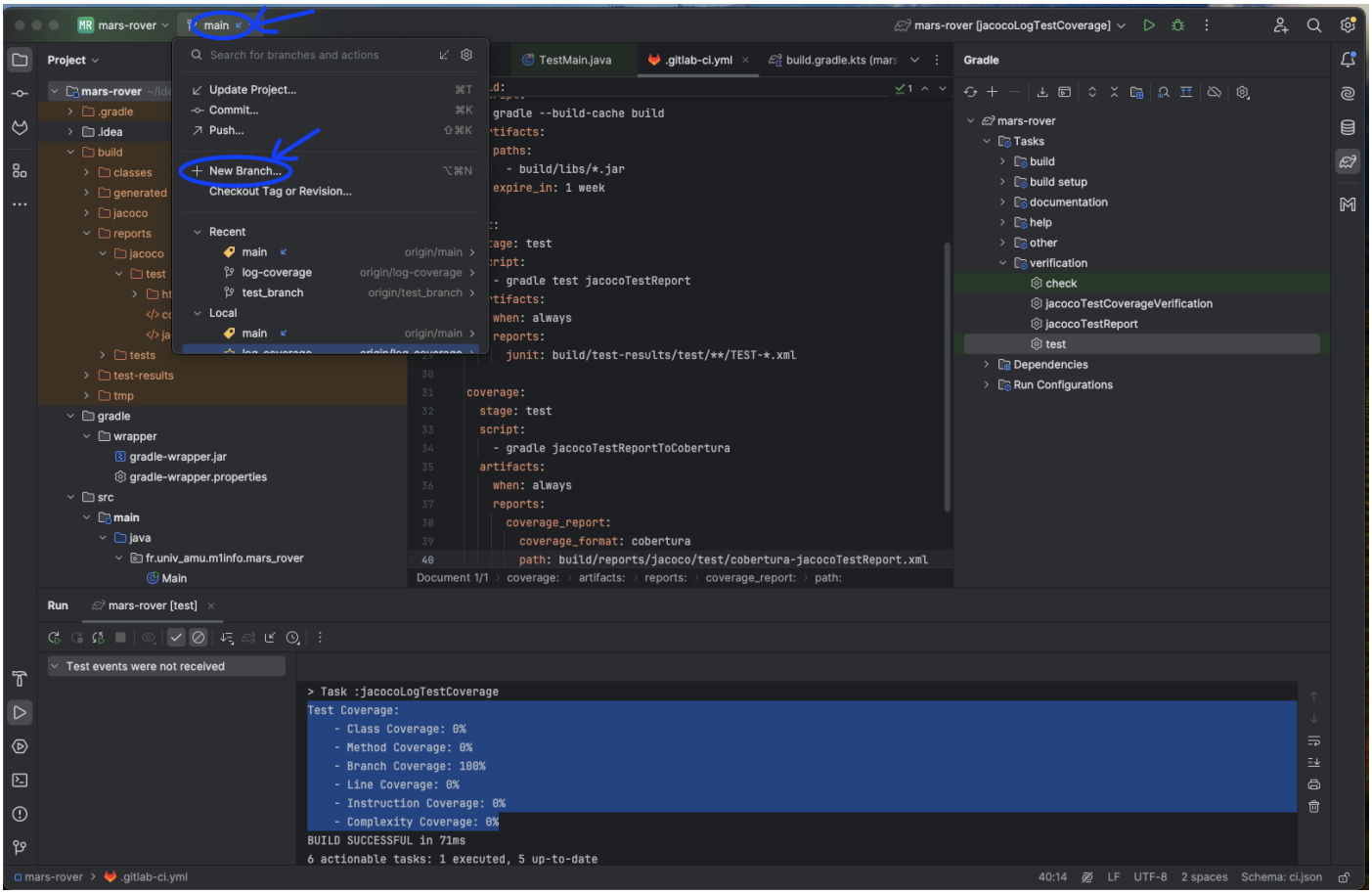
Pour ce TP, vous allez donc créer une branche pour ajouter la couverture par les tests. Une fois le développement fini sur la branche, vous fusionnerez cette branche dans la branche principale. Pour gérer des branches, on vous propose deux manières de procéder soit directement via IntelliJ IDEA ou bien en ligne de commandes.

> **Tâche 3 :** Créez une branche nommée `test-coverage` qui ajoute la couverture par les tests à votre projet. Une fois que la branche est opérationnelle, fusionnez la avec la branche principale.

## 5.1 Les branches avec IntelliJ IDEA

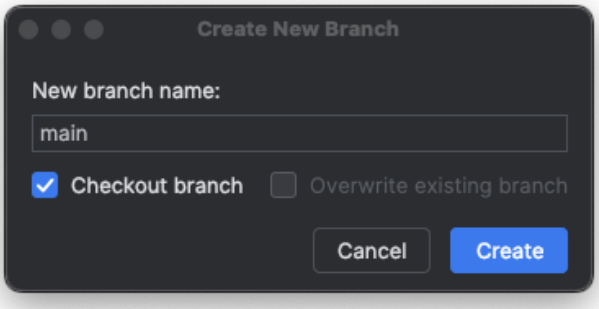
### 5.1.1 Création de branche

Pour créer une nouvelle branche sous *IntelliJ IDEA* il faut passer par le menu gestion de version en haut à gauche de la fenêtre puis cliquer sur `new branch` :




La branche courante est d'ailleurs indiqué dans le nom du menu (*main* dans ce cas).

Une fenêtre s'ouvre et vous devez mettre le nom voulu pour votre branche (nom devant décrire le but de la branche et donc la fonctionnalité visée) :




On vous conseille de laisser cochée la case `Checkout branch` afin de passer directement dans la nouvelle branche créée.

### 5.1.2 Utilisation des branches

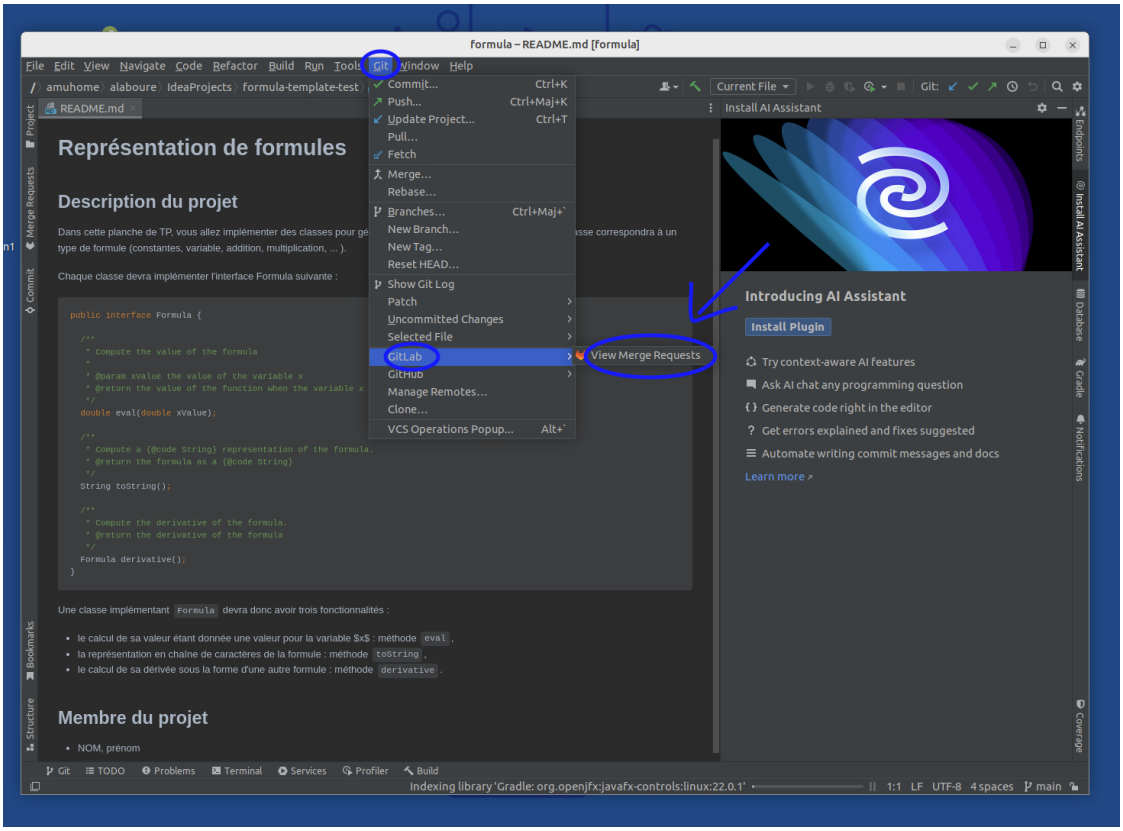
À l'aide du même menu, vous avez aussi accès aux différentes branches de votre projet. Pour chaque branche, vous pouvez

- passer sur la branche avec l'option `checkout` (changer la branche courante pour la branche sélectionnée);
- créer une nouvelle branche à partir de la branche avec l'option `new branch from ...` ;
- changer le nom de la branche avec l'option `rename` ;
- comparer la branche avec la branche courante avec l'option `compare with ...`.

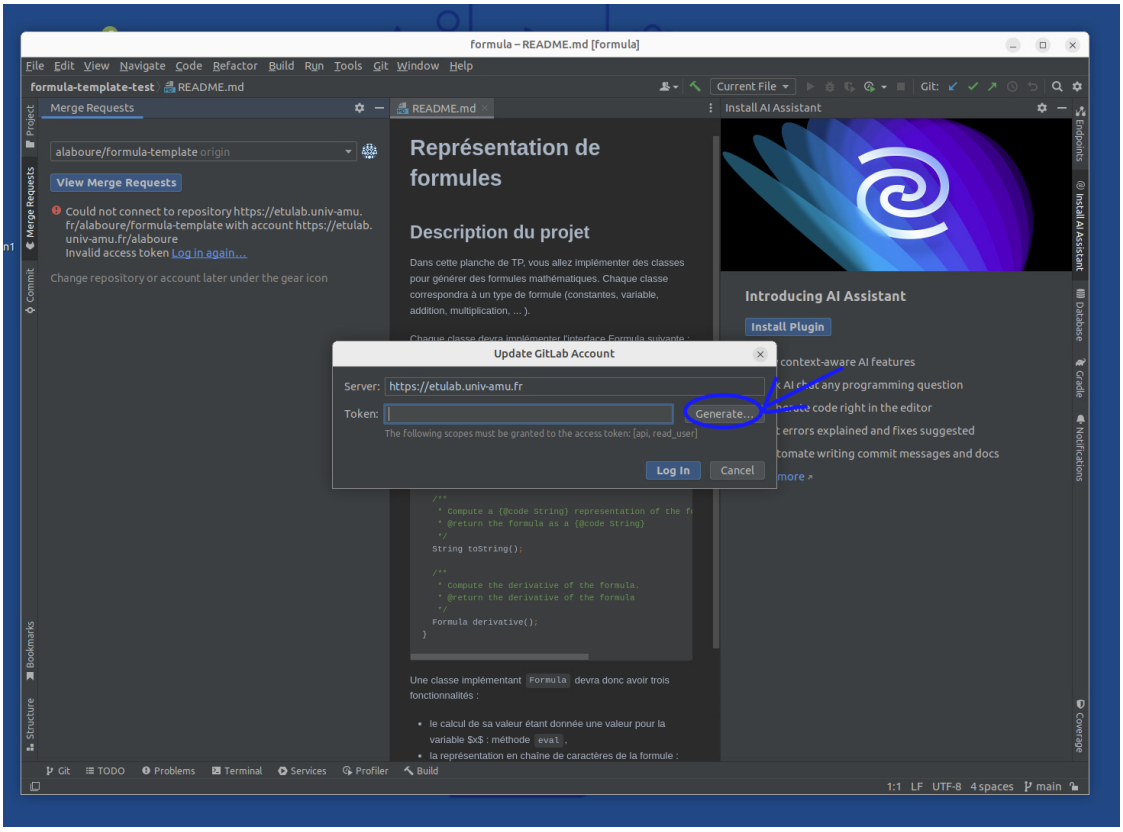
Lorsque vous êtes sur une branche, tous les *commits* et *pushs* impactent celle-ci.

### 5.1.3 Merge request

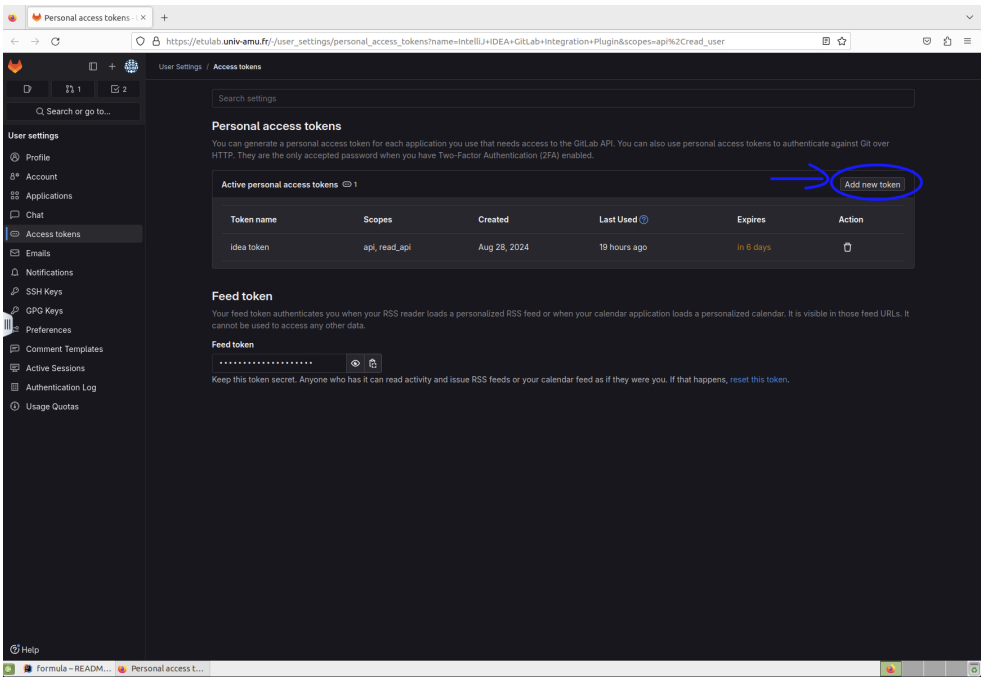
Une fois qu'une fonctionnalité est finalisée au sein d'une branche, la prochaine étape est la fusion à la branche principale *main*. Pour cela, il faut, sur *gitlab*, passer par une demande de fusion appelée *merge request*. Vous pouvez utiliser *IntelliJ IDEA* pour créer une telle demande. Pour cela il faut choisir, *Git* dans le menu *IntelliJ IDEA* puis *Gitlab* et finalement *View merge request*.




La première fois que vous faites cette opération, on vous demande de créer un jeton d'accès afin de pouvoir vous connecter à l'API *Gitlab*. Le bouton *generate* devrait automatiquement vous ouvrir la page web.

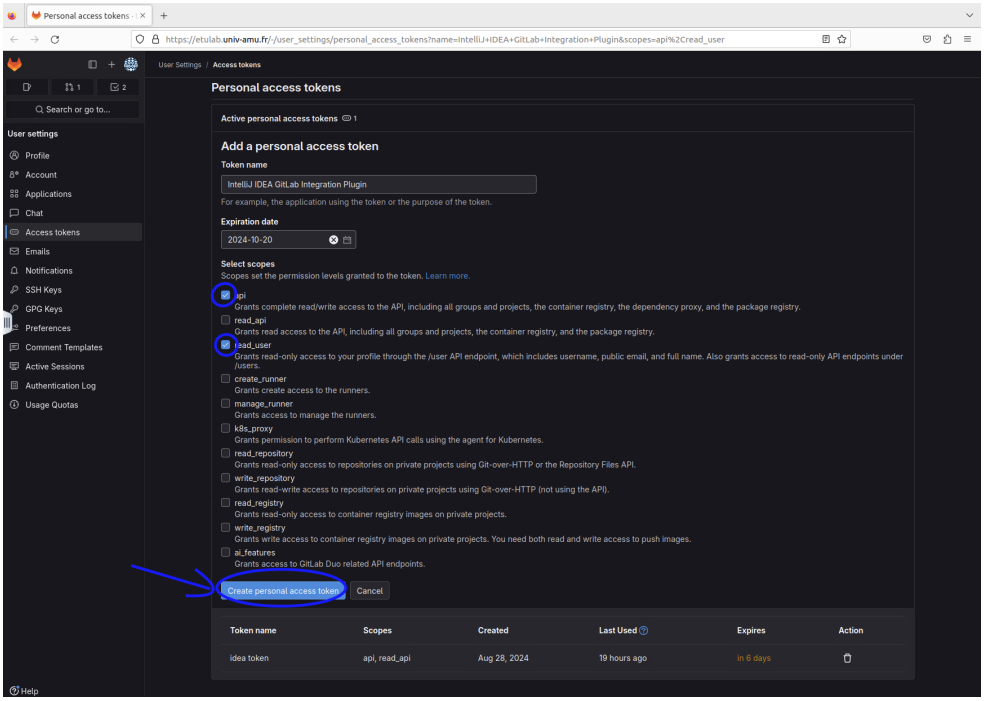



Une fois que vous avez cliquez sur le bouton *generate*, vous devriez avoir l'affichage suivant (aussi accessible via *Access token* dans votre profil *etulab*). Il vous faut maintenant cliquez sur le bouton *add new token*.

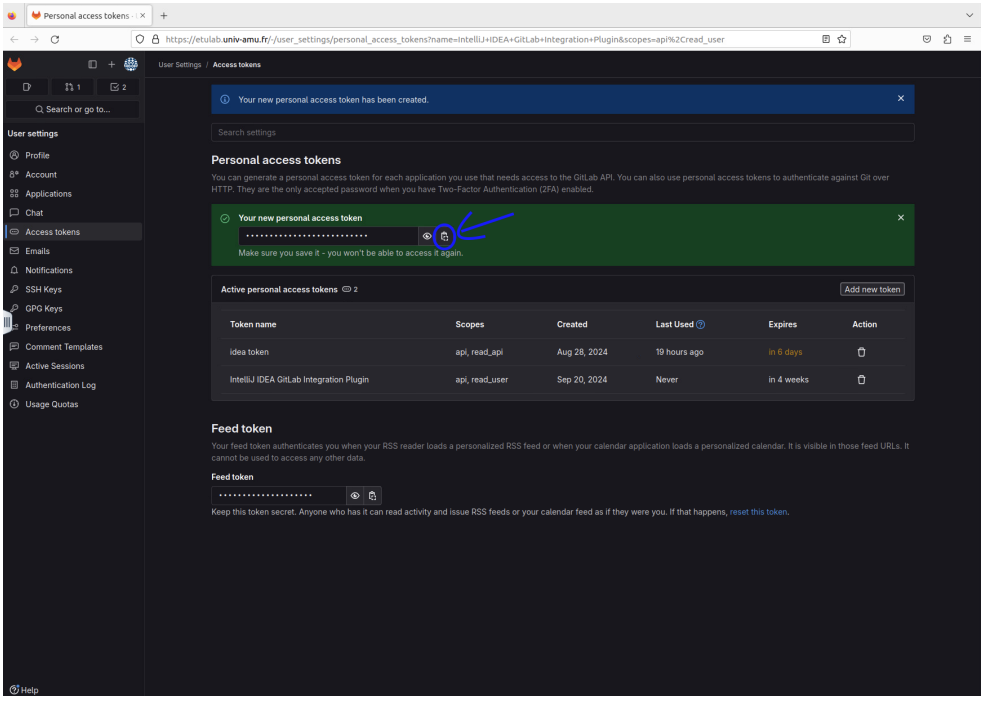



Une fois sur cette page, vous devez créer un token avec le nom de votre choix avec les options *api* et *read_user*

cochées (normalement le lien *generate* vous pré-coche ces choix). Cliquez sur le bouton *create personal access token* pour le créer.

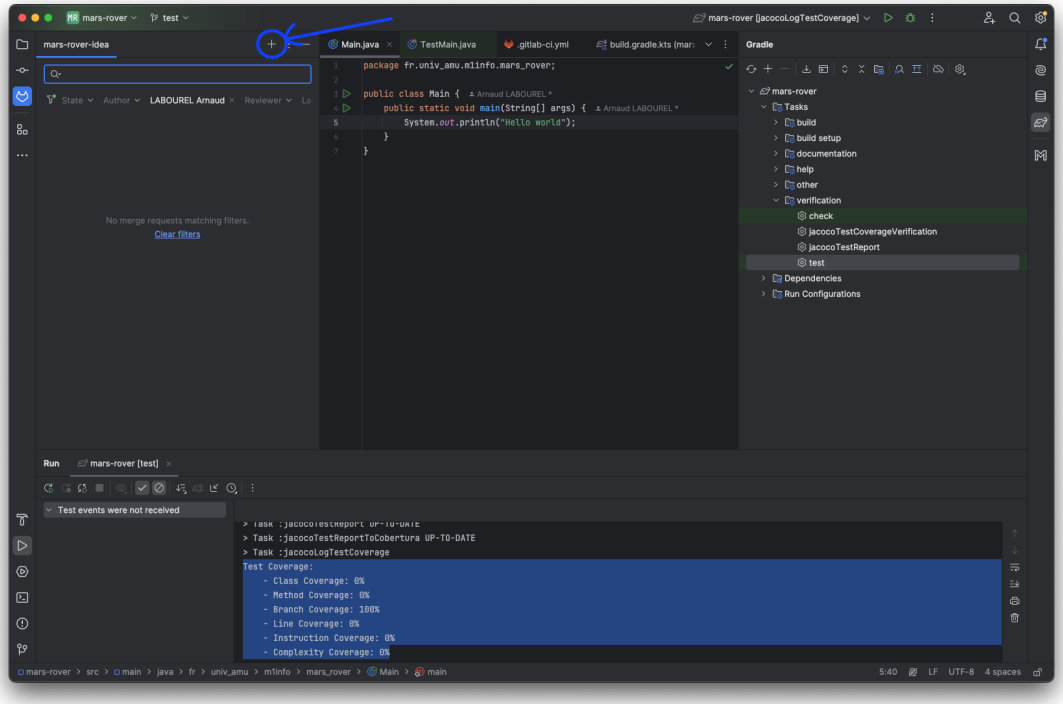



Une fois le *token* créé, il vous suffit de le copier en cliquant sur le bouton dédié.



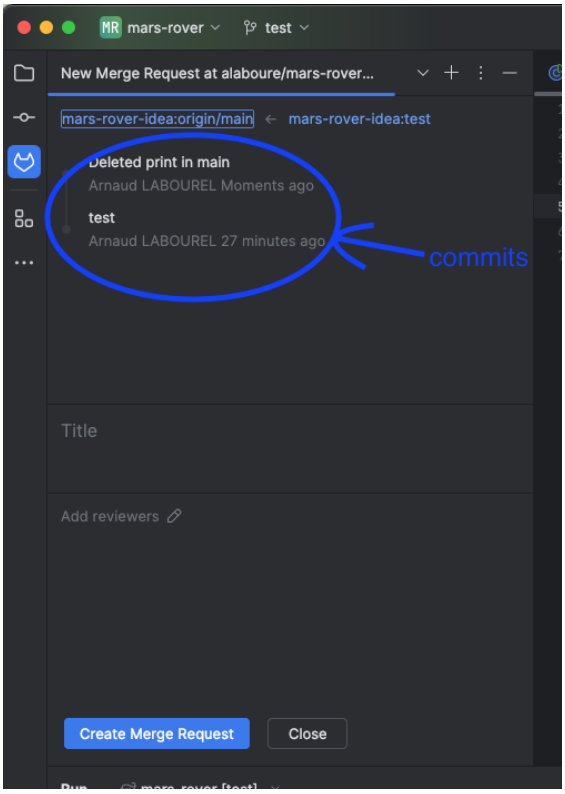

Finalement, il vous suffit de coller ce que vous avez copié dans le champ *Token* de la fenêtre d'*IntelliJ IDEA* pour pouvoir vous connecter avec le bouton *Log in*.

Une fois connecté un onglet dédié aux *merge requests* devrait s'ouvrir. Pour créer un *merge requests*, il vous suffit de cliquer sur le bouton + de l'onglet :

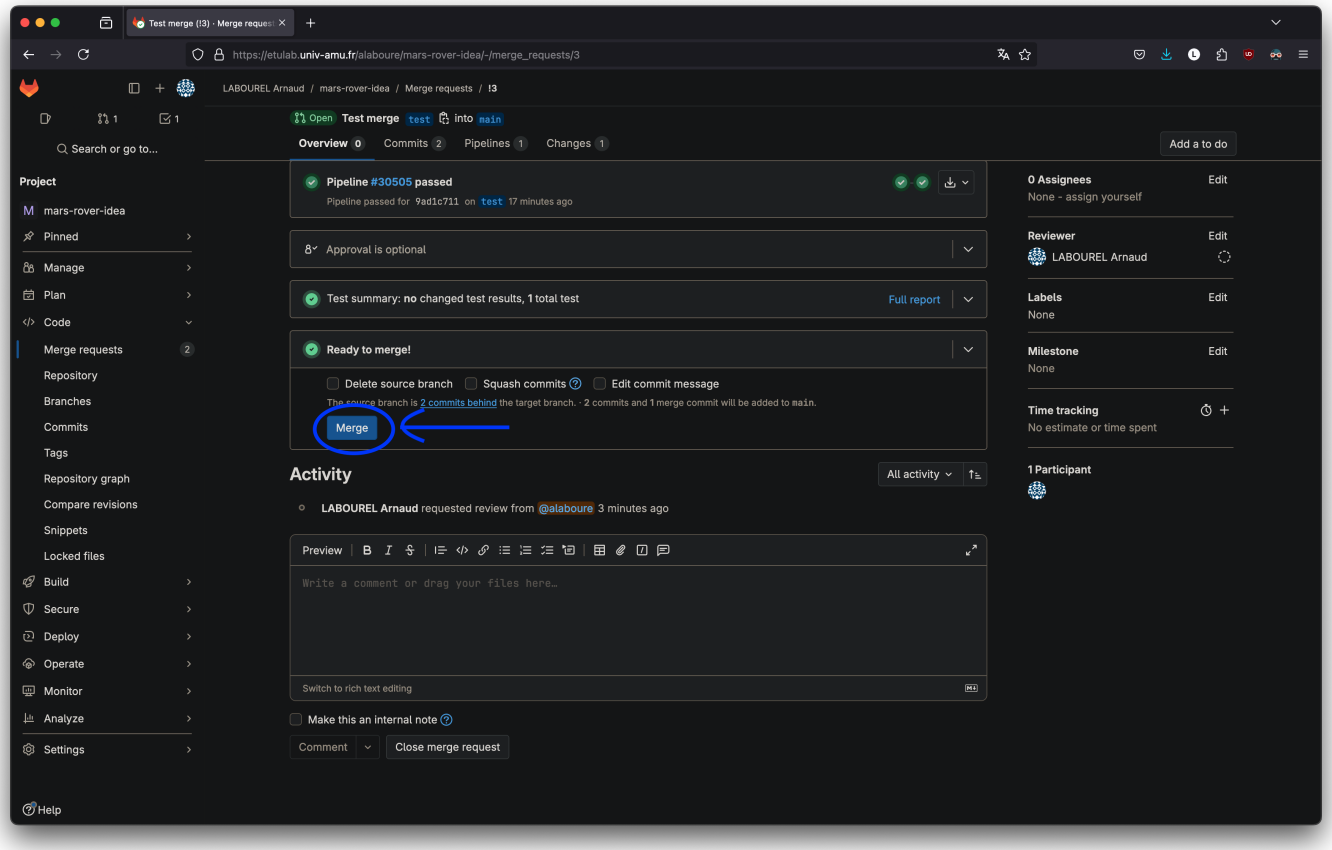



L'onglet devrait changer et contient les éléments suivants (de haut en bas) :

— ligne *branche1* ← *branche2* qui indique que l'opération de *merge* fusionne la *branche2* dans la *branche1*. Normalement, de base cela devrait être votre branche courante qui est fusionnée dans la branche *merge*. Vous pouvez changer les branches concernées en cliquant sur leurs noms ;
— les *commits* de la branche à fusionner ;
— le titre de la demande de fusion que vous pouvez compléter ;
— la liste des *reviewers* qui vont devoir valider la fusion (vous pouvez normalement ajouter n'importe quel membre du projet ayant les droits suffisants).




Une fois la demande envoyée, vous pouvez aller sur l'interface *etulab* puis choisir *merge requests* dans votre menu utilisateur ou bien *code* → *merge requests* dans le menu du projet. Vous devriez voir votre *merge request*. Normalement, vous pouvez la valider (si tout vous semble correct) en cliquant dessus puis en cliquant sur *merge*.




## 5.2 Les branches en ligne de commandes

### 5.2.1 Création de branche

Pour la création de branches en ligne de commande, on vous conseille de lire le *Git book* et en particulier la partie consacrée aux branches

Pour créer une branche, il suffit d'appeler la commande `branch` de *git* avec le nom voulu pour la branche.

```
git branch new_branch
```

Cela commande crée un nouveau pointeur vers le *commit* courant. Notez que vous restez dans la branche dans laquelle vous étiez.

### 5.2.2 Utilisation des branches

Vous pouvez changer de branche avec la commande `checkout` de `git` suivie du nom de la branche à laquelle vous souhaitez passer. Par exemple, la commande suivante permet de passer à la branche `new_branch` :

```
git checkout new_branch
```

Les commits que vous effectuez impactent la branche courante.

Si vous souhaitez *push* votre branche sur le serveur il vous faut spécifier la première fois la branche correspondante sur le serveur. Par exemple, si vous souhaitez avoir sur le serveur une branche `new_branch` correspondant au contenu de la branche courante, vous pouvez utiliser la commande suivante :

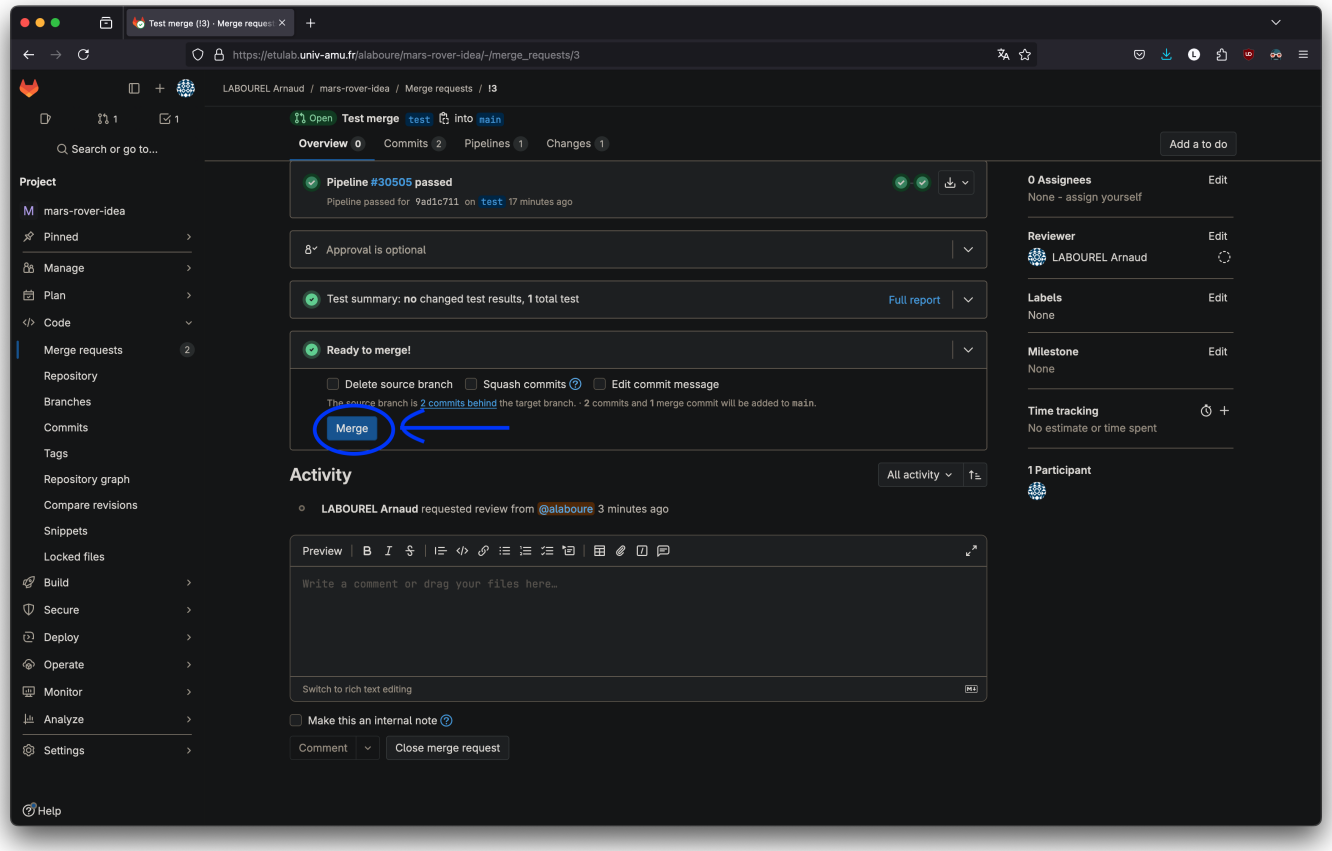
```
git push --set-upstream origin new_branch
```

### 5.2.3 Merge request

Une fois qu'une fonctionnalité est finalisée au sein d'une branche, la prochaine étape est la fusion à la branche principale *main*. Pour cela, il faut, sur *gitlab*, passer par une demande de fusion appelée *merge request*. Vous pouvez faire cette demande lors d'un `push` en spécifiant l'option `merge_request.create`. Vous pouvez choisir la branche cible avec l'option `merge_request.target`. Par exemple, la commande suivante réalise un `push` de la branche courante demandant la création d'un *merge request* vers la branche `main`.

```
git push -o merge_request.create -o merge_request.target=main
```

Une fois la demande envoyée, vous pouvez aller sur l'interface *etulab* puis choisir *merge requests* dans votre menu utilisateur ou bien *code* → *merge requests* dans le menu du projet. Vous devriez voir votre *merge request*. Normalement, vous pouvez la valider (si tout vous semble correct) en cliquant dessus puis en cliquant sur *merge*.



# 6 Couverture de code par les tests

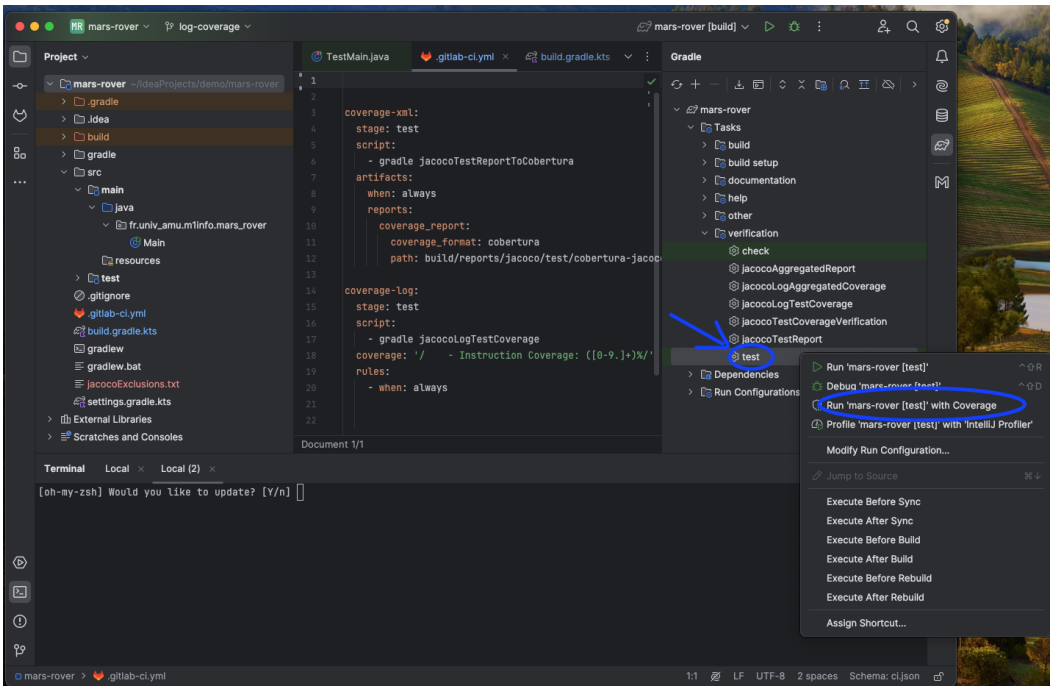
## 6.1 Description

La couverture de code consiste à mesurer la part du code source ayant été exécuté via différentes métriques comme le pourcentage de méthodes appelées ou le pourcentage d'instructions exécutées. Utilisée sur des tâches de tests, elle permet de mesurer la part du code source ayant été testée. Un programme avec une haute couverture de code a davantage de code exécuté durant les tests ce qui laisse à penser qu'il a moins de chance de contenir de bugs logiciels non détectés, comparativement à un programme ayant une faible couverture de code. Néanmoins, une couverture totale du code ne garanti pas que celui-ci soit dépourvu de bugs.

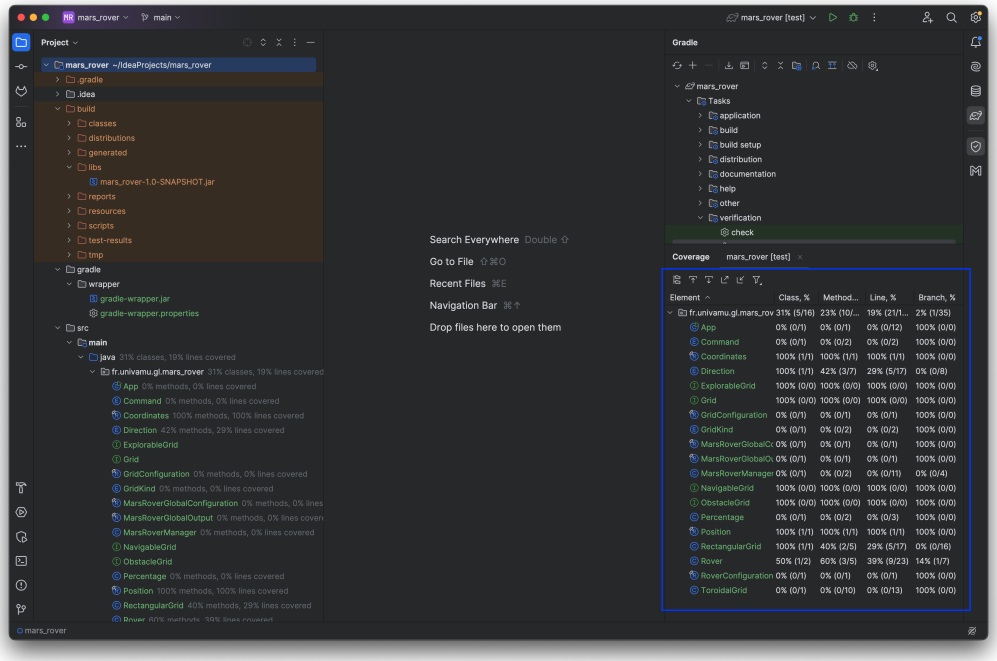
Pour ce TP, on vous demande de mesurer la couverture de votre code par les tests que vous avez écrits au TP précédent. Pour cela, on vous propose deux outils : JaCoCo

## 6.2 Couverture de code via les outils d'IntelliJ IDEA

Pour mesurer la couverture de code via IntelliJ, il existe différents moyens d'accès lié aux différentes façons d'exécuter le code. Pour les tâches *gradle*, il suffit d'aller dans l'onglet *gradle*, de faire un clic droit sur la tâche et de choisir *Run with coverage*.




Une fois que vous avez lancé la couverture, un onglet de décrivant les données de couverture devraient s'ouvrir en bas à gauche de la fenêtre d'*IntelliJ IDEA*. Il donne pour chaque fichier *Java*, le pourcentage de classes, méthodes, lignes et branches couvertes par vos tests.




## 6.3 Couverture de code via JaCoCo

Une autre manière de mesurer la couverture de code est d'utiliser JaCoCo. Une manière simple pour utiliser JaCoCo avec votre projet est de rajouter deux plugins dédiés dans la configuration *gradle* :

— le plugin JaCoCo qui est le plugin de base pour l'utilisation de *JaCoCo* via *gradle* ;
— le plugin JaCoCo log qui permet de produire facilement un log des

Pour rajouter ces deux plugins à votre projet, il vous faut ajouter les lignes suivantes dans le fichier `build.gradle.kts` :

```
plugins {
    id("jacoco")
    id("org.barfuin.gradle.jacocolog") version "3.1.0"
}
```

Il vous faut aussi configurer votre *build* pour que les rapports soient générés après les tests et indiquer que les rapports dépendent des tests :

```
tasks.test {
    finalizedBy(tasks.jacocoTestReport) // report is always generated after tests run
}
tasks.jacocoTestReport {
    dependsOn(tasks.test) // tests are required to run before generating the report
}
```

Une fois cela fait vous pouvez relancer la tâche test de *gradle*. Vous devriez obtenir un affichage similaire à

l'affichage suivant, vous donnant pour le projet la couverture par les tests suivant différentes métriques (détails au lien suivant) :

```
Test Coverage:
    - Class Coverage: 0%
    - Method Coverage: 0%
    - Branch Coverage: 100%
    - Line Coverage: 0%
    - Instruction Coverage: 0%
    - Complexity Coverage: 0%
```

`JaCoCo` produit aussi un rapport détaillé qui est accessible via le fichier `build/reports/jacoco/test/html/index.html` dans votre projet. Ce rapport détaille la couverture par *package*, par classe et par méthode. La couverture d'une méthode donne la couverture de chaque ligne via un code couleur :

— Vert : la ligne de code est couvert par un test ;
— Rouge : la ligne de code n'est couvert par aucun test ;
— Jaune : la ligne de code est partiellement couvert par un test, par exemple un `if` pour lequel tous les cas possibles ne sont pas couvert.

Pour finir, il est possible de fournir un taux de couverture au serveur *gitlab* en ajoutant un champ `coverage` au *job* `test` défini dans le fichier `.gitlab-ci.yml` comme ceci (en gardant la configuration existante du *job*) :

```
test:
  stage: test
  coverage: '/    - Instruction Coverage: ([0-9.]+)%/'
```

Cette ligne va extraire le pourcentage affiché pour la couverture des instructions. Vous pouvez changer la métrique de couverture pour n'importe quel autre des métriques de JaCoCo que vous souhaitez.

Une fois le *push* effectué, vous devriez avoir une colonne *coverage* dans l'onglet *build* → *jobs* de votre projet sur etulab.

# 7 Fonctionnalités optionnelles

Si vous avez fini d'implémenter les fonctionnalités décrites précédemment, vous pouvez ajouter des fonctionnalités supplémentaires au simulateur *mars-rover*, comme les suivantes :

— **Ajout d'obstacles sur le terrain :** On considère qu'il y a maintenant des obstacles (présents sur certaines cases) détruisent tout *rover* s'y déplaçant (y compris pour l'atterrissage si l'obstacle est présent sur ses coordonnées initiales). Le format de fichier d'entrée doit être modifié afin de pouvoir rajouter une liste de coordonnées correspondant aux cases contenant des obstacles. La sortie du simulateur devra dorénavant indiquer si le *rover* est détruit ou pas à la fin de la simulation.
— **Capacité d'exploration augmentée pour les *rovers* :** Chaque *rover* a désormais un rayon d'exploration qui lui permet d'explorer à tout moment lors de sa trajectoire toutes les cases qui sont à une distance

de sa case inférieure ou égale à son rayon d'exploration. Un rayon d'exploration de zéro correspond à la configuration des TP précédents. Un rayon d'exploration égal à un permet au *rover* d'explorer toutes les cases voisines à une des cases de sa trajectoire. Un rayon d'exploration égal à deux permet d'explorer les case à distance deux (voisine des voisines).

— **Interface graphique pour la visualisation :** Le but est d'ajouter une interface permettant de visualiser les trajectoires d'exploration par les *rovers*. Le terrain sera représenté par une grille, l'affichage des cases devant permettre de distinguer si la case est explorée et/ou si elle contient un ou plusieurs *rovers*. L'interface devra permettre de charger un fichier de configuration puis de lancer la simulation soit pas à pas ou bien via une animation.

**Tâche 4 :** Pour chaque fonctionnalité que vous souhaitez implémenter, créez une branche. Une fois que la branche est opérationnelle, fusionnez la avec la branche principale.