amU Faculté des sciences Aix Marseille Université

Année 2024-25

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

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 les bonnes pratiques pour le développement logiciel :

— 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.

Le problème à résoudre pour ce TP reste globalement le même que celui du TP précédent. Néanmoins, les spécifications ont légèrement changées (formats de fichiers différents) et deux nouveaux types de grilles sont introduits en plus de la grille rectangulaire.

Le but de ce TP sera donc de modifier trois fois le code en créant à chaque fois une branche dans le git :

— la première modification et donc la première branche correspondra à la mise en place d'outil de mesure de couverture du code par les tests ;
— la deuxième modification et donc la deuxième branche correspondra à l'implémentation des deux nouveaux types de grilles ;
— la troisième modification et donc la troisième branche correspondra à l'implémentation de la prise en charge.

2 Utilisation des branches

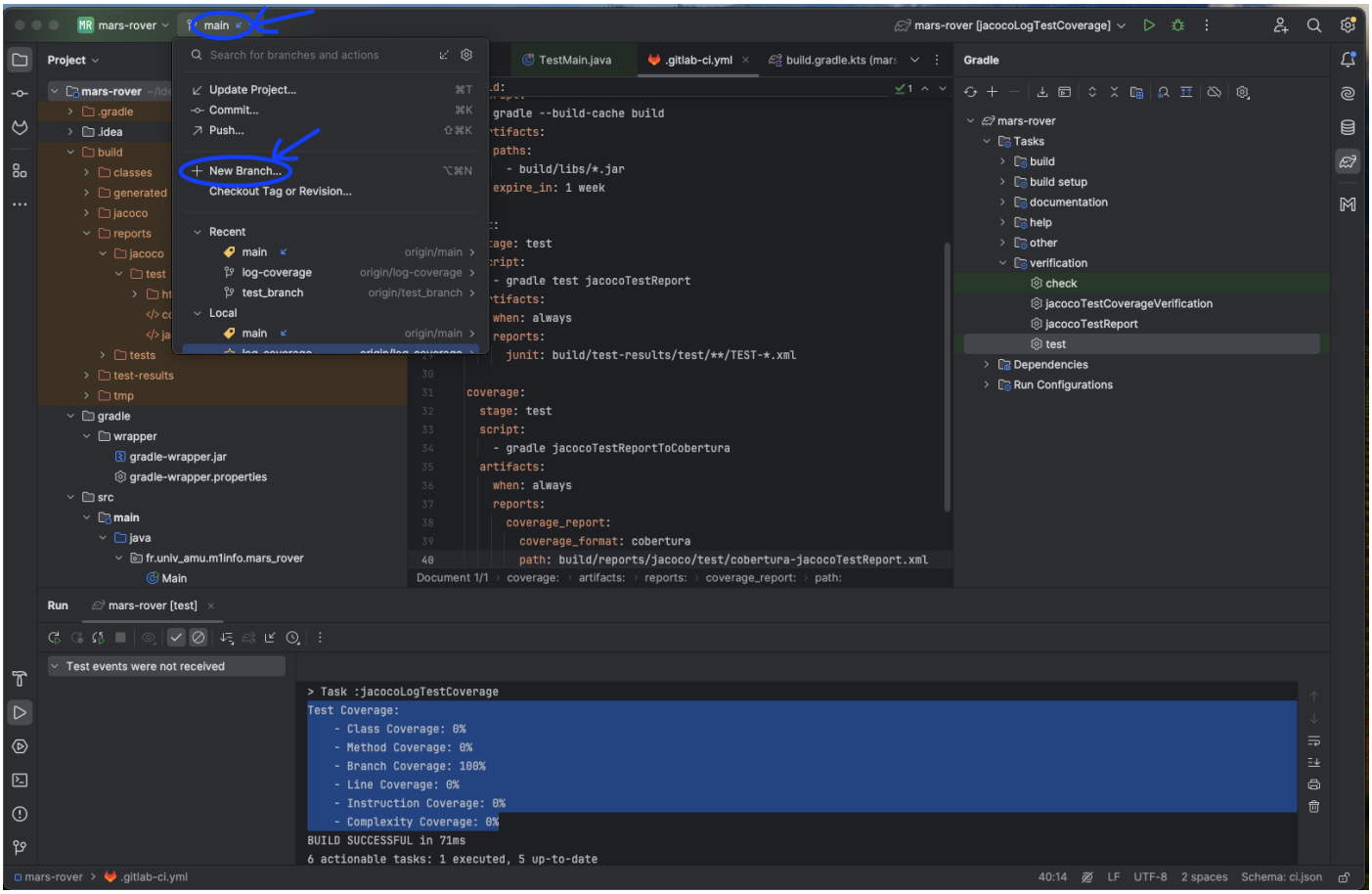
Une bonne pratique de développement pour des projets utilisant la gestion de versions est de créer des branches pour introduire les modifications du code. Grâce aux branches, les équipes de développement logiciel peuvent apporter des modifications sans affecter la branche principale (*main*). L'historique 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 trois branches (une branche par fonctionnalité) et vous devez donc créer une branche pour ajouter la couverture par les tests. Pour gérer des branches, on vous propose deux manières de procéder soit directement via IntelliJ IDEA ou bien en ligne de commandes.

2.1 Les branches avec IntelliJ IDEA

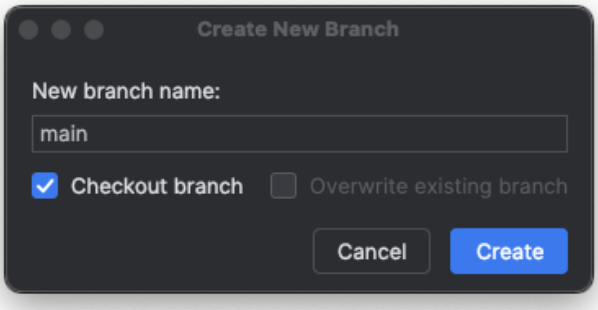
2.1.1 Création de branche

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.

2.1.2 Utilisation des branches

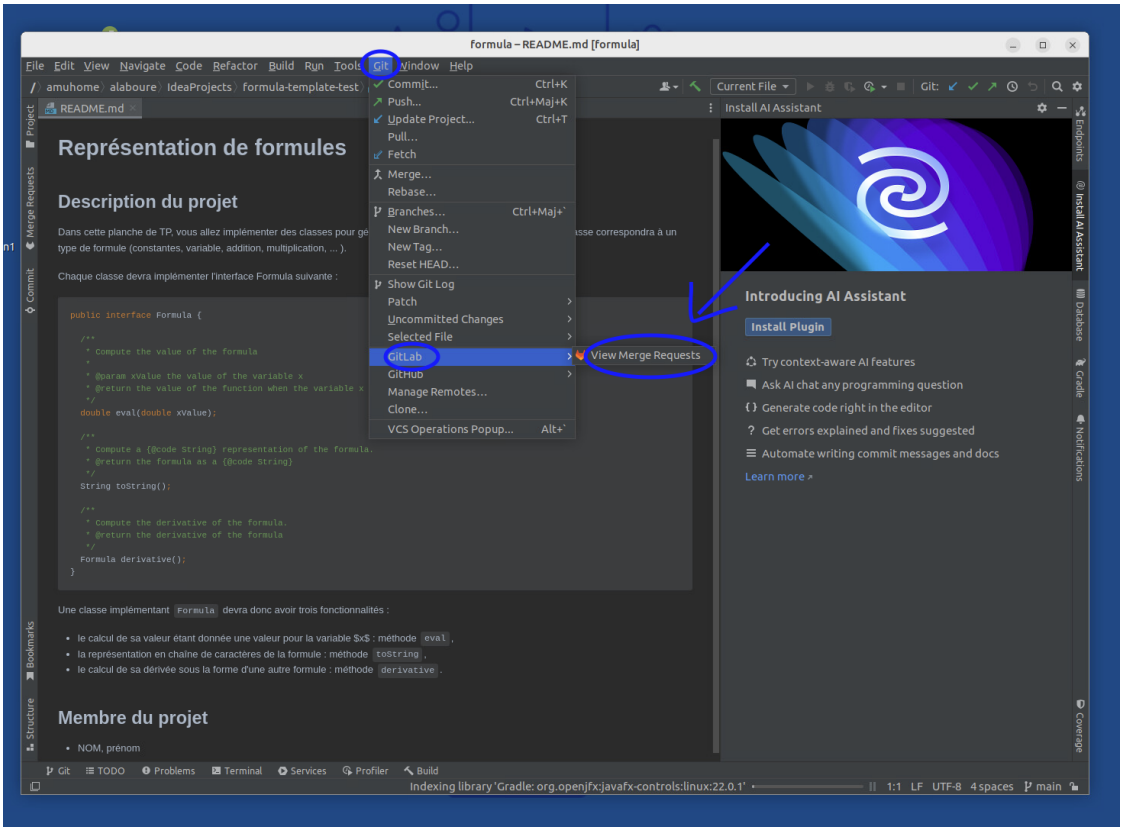
À l'aide du même menu, vous avez aussi accès aux différentes branches de votre projet. Pour chaque branche, vous pouvez

- 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 ...`.

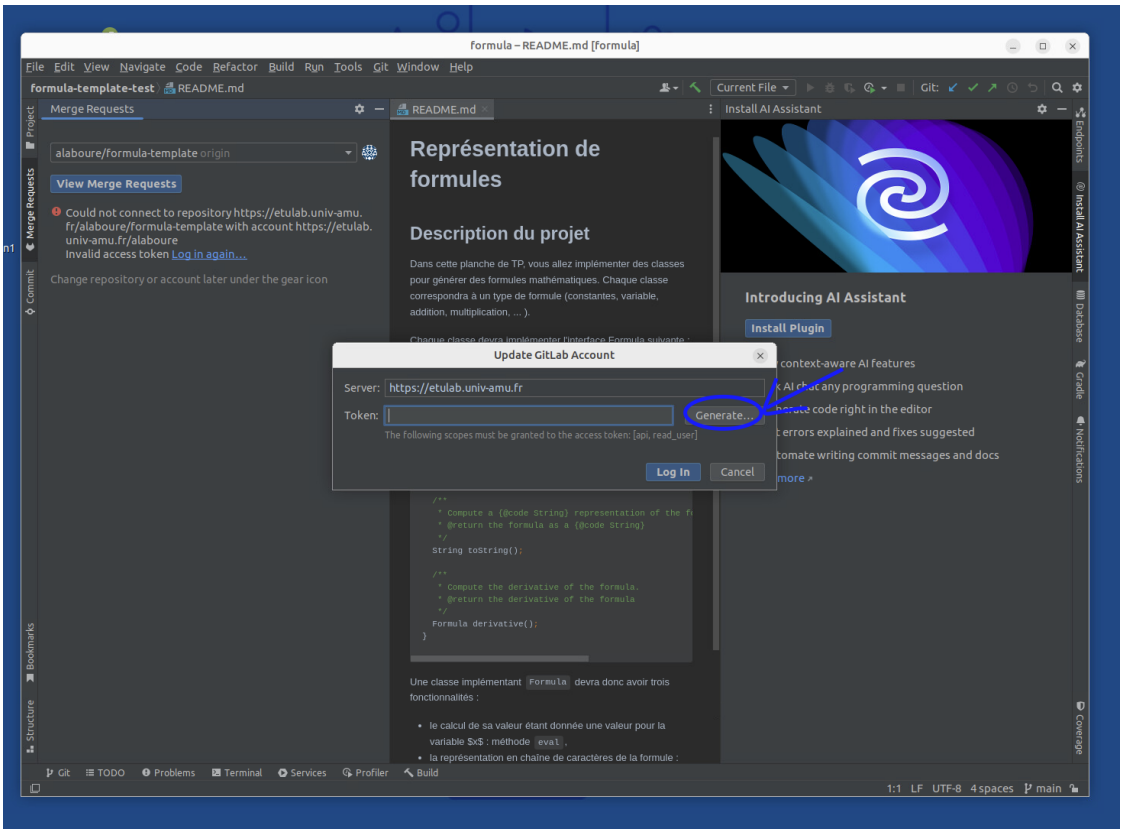
Lorsque vous êtes sur une branche, tous les *commits* et *pushs* impactent celle-ci.

2.1.3 Merge request

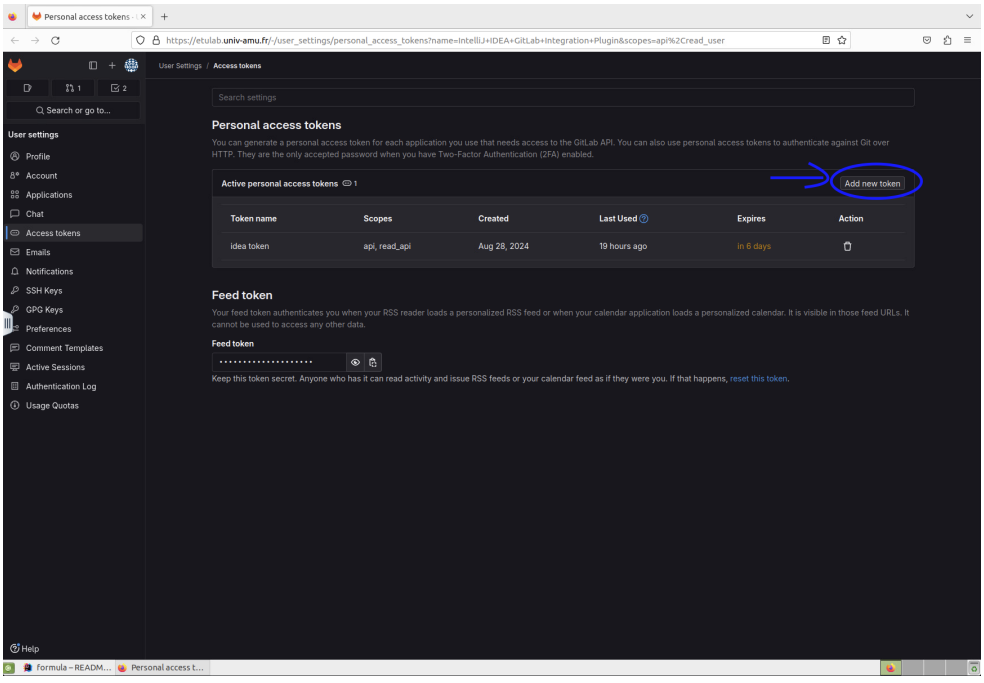
Une fois qu'une fonctionnalité est finalisée au sein d'une branche, la prochaine étape est la fusion à la branche principale *main*. Pour cela, il faut, sur *gitlab*, passer par une demande de fusion appelée *merge request*. Vous pouvez utiliser *IntelliJ IDEA* pour créer une telle demande. Pour cela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



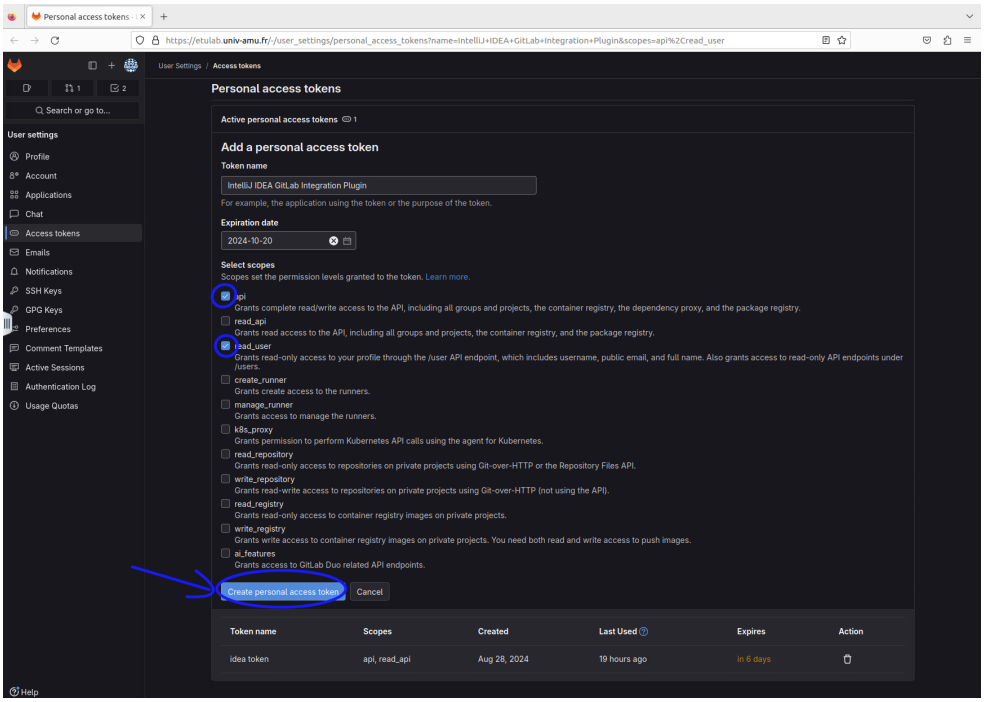
La première fois que vous faites cette opération, on vous demande de créer un jeton d'accès afin de pouvoir vous connecter à l'API *Gitlab*. Le bouton *generate* devrait automatiquement vous ouvrir la page web.



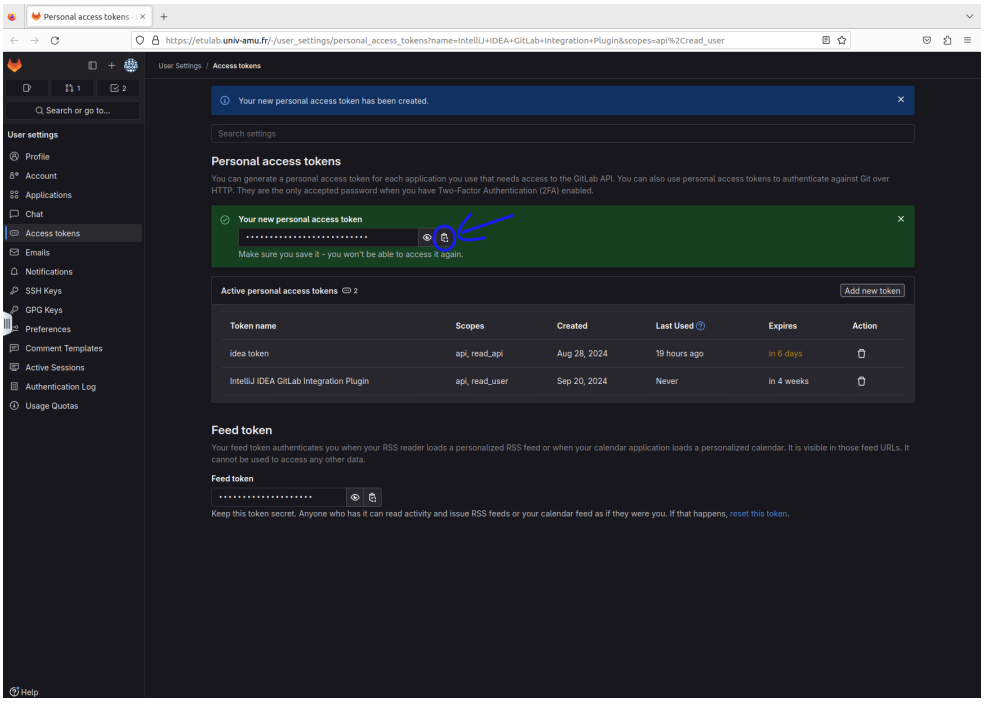
Une fois que vous avez cliquez sur le bouton *generate*, vous devriez avoir l'affichage suivant (aussi accessible via *Access token* dans votre profil *etulab*). Il vous faut maintenant cliquez sur le bouton *add new token*.



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.

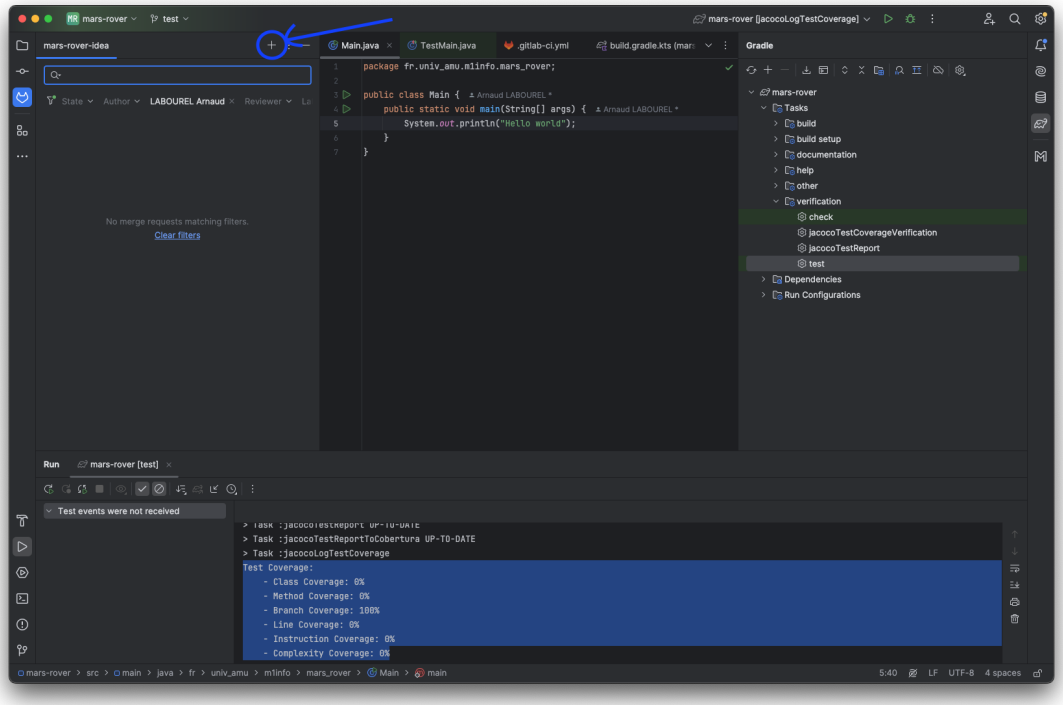


Une fois le *token* créé, il vous suffit de le copier en cliquant sur le bouton dédié.



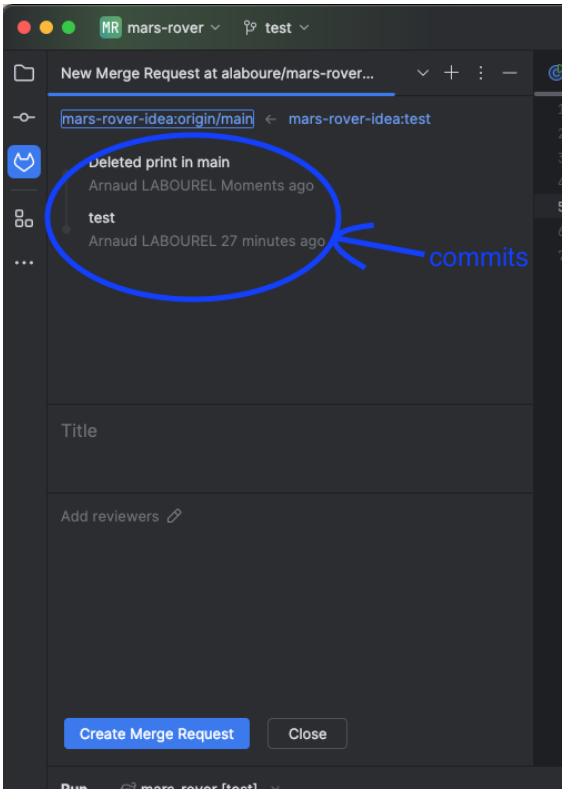
Finalement, il vous suffit de coller ce que vous avez copié dans le champ *Token* de la fenêtre d'*IntelliJ IDEA* pour pouvoir vous connecter avec le bouton *Log in*.

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 :

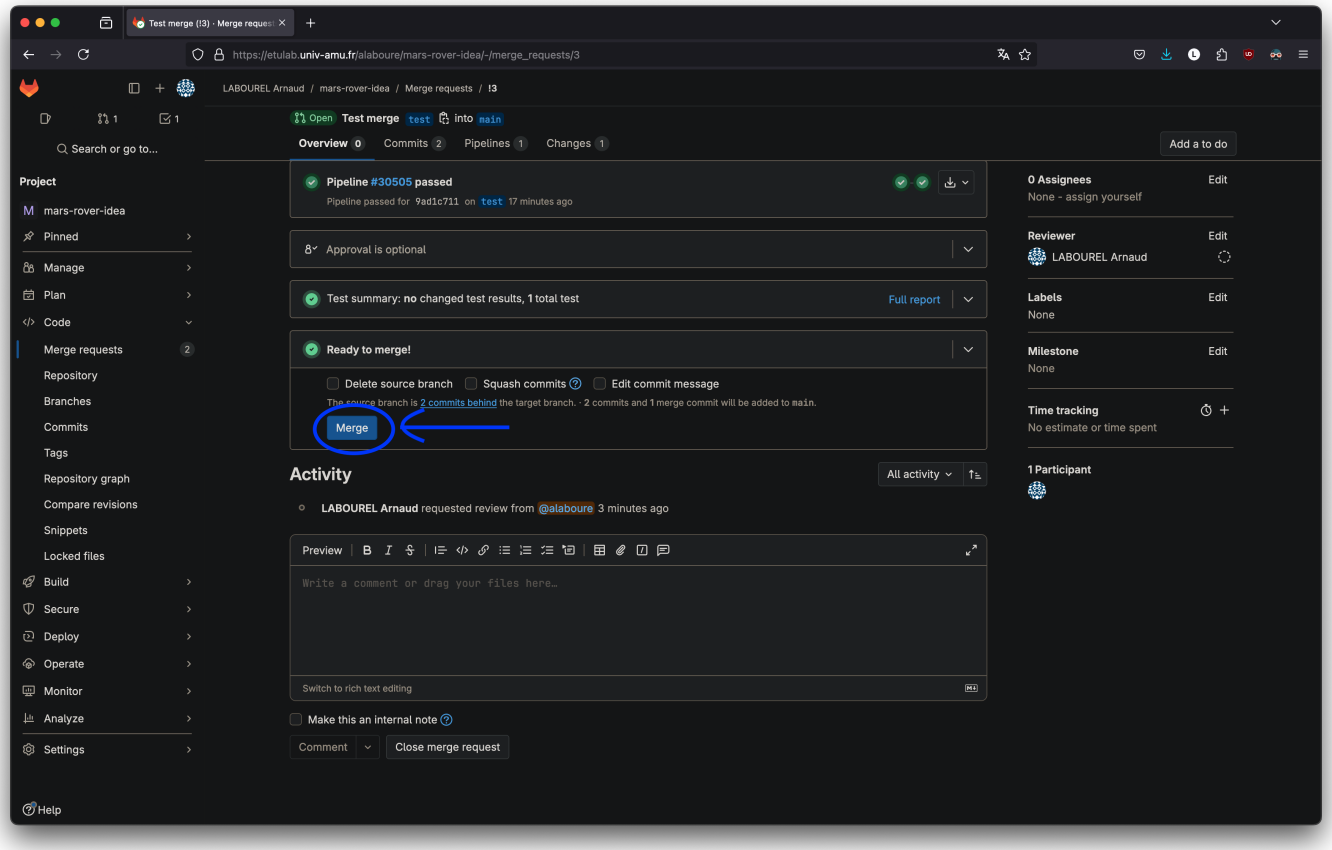


L'onglet devrait changer et contient les éléments suivants (de haut en bas) :

— 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*.



2.2 Les branches en ligne de commandes

2.2.1 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.

2.2.2 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 :

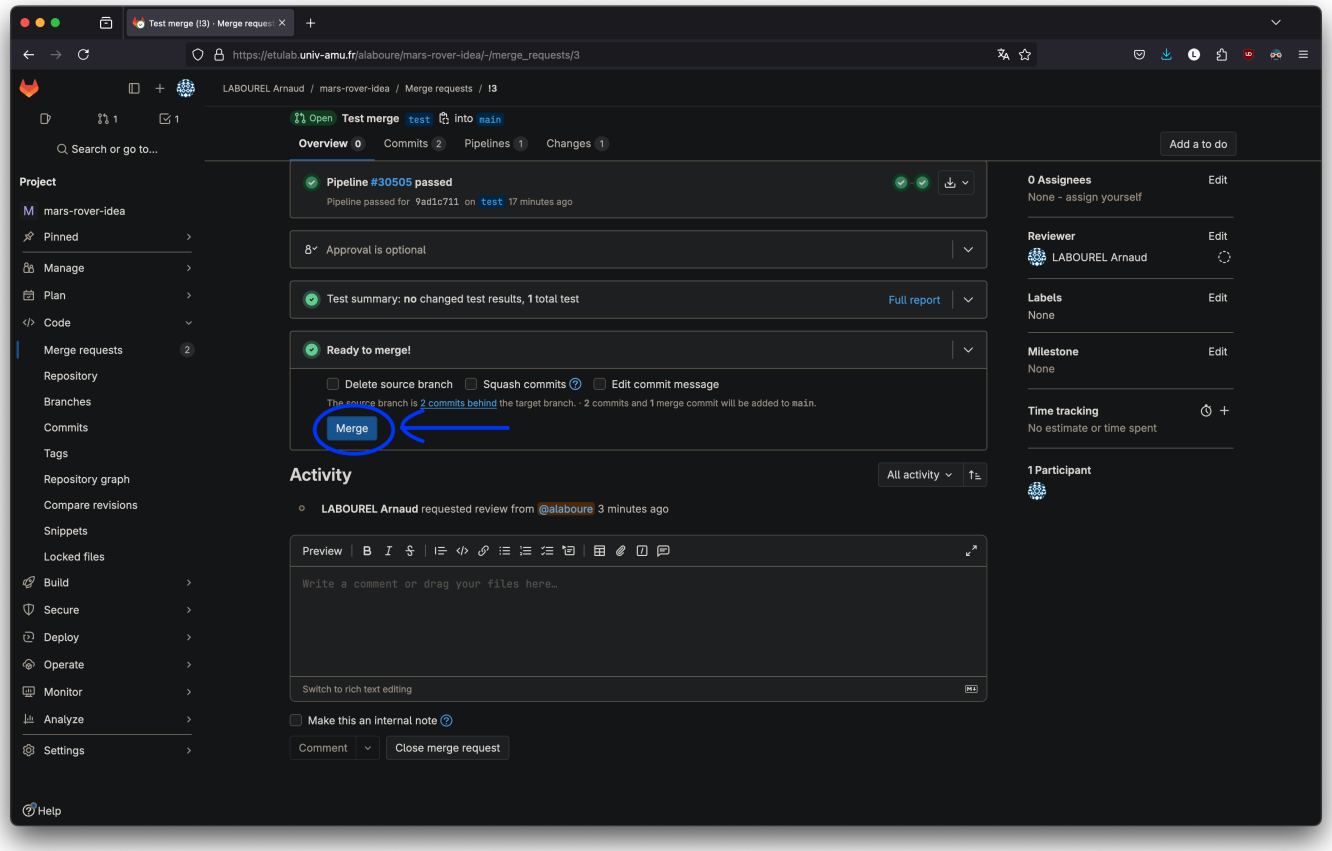
```
git push --set-upstream origin new_branch
```

2.2.3 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*.



3 Couverture de code par les tests

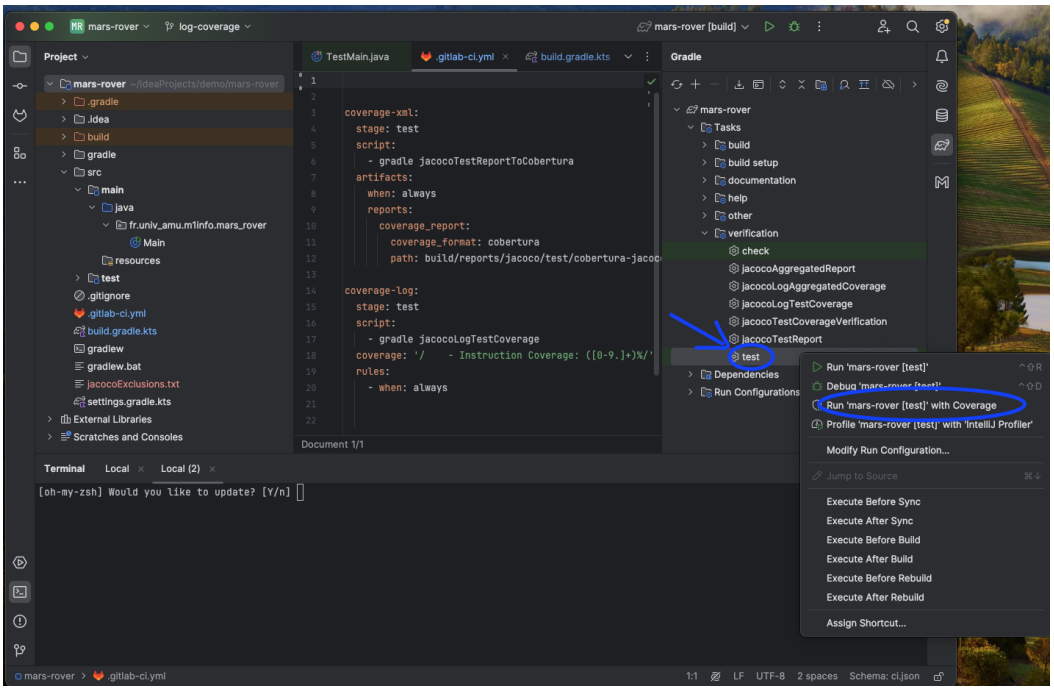
3.1 Description

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.

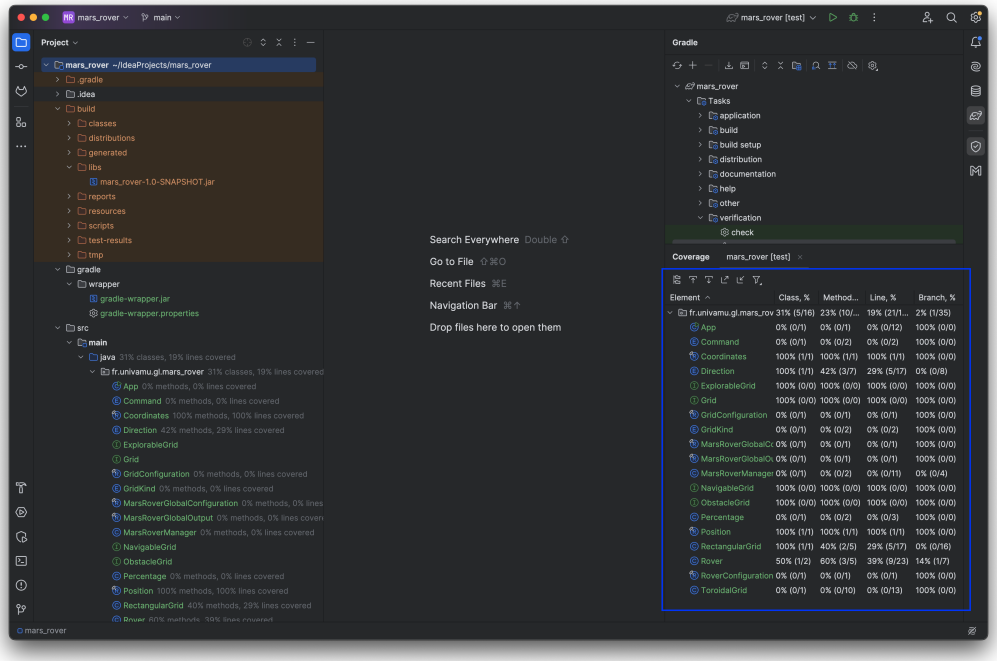
Pour ce TP, on vous demande de mesurer la couverture de votre code par les tests que vous avez écrits au TP précédent. Pour cela, on vous propose deux outils : JaCoCo

3.2 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.



3.3 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.

4 Nouveaux types de grilles

En plus de la grille rectangulaire, on considère deux types de grille toroïdale et sphérique.

4.1 Grille toroïdale

Une grille toroïdale correspond à une topologie ou le haut et le bas de la grille sont connectés et un robot sortant par le haut réapparait en bas de la grille (passant d'une coordonnée en y de $h-1$ à 0 avec h la hauteur de la grille et en gardant la même coordonnée en x) et inversement. De même, la gauche et la droite sont connectées et un robot sortant par la gauche réapparait à la droite de la grille (passant d'une coordonnée en x de 0 à $l-1$ avec l la largeur de la grille et en gardant la même coordonnée en y) et inversement.

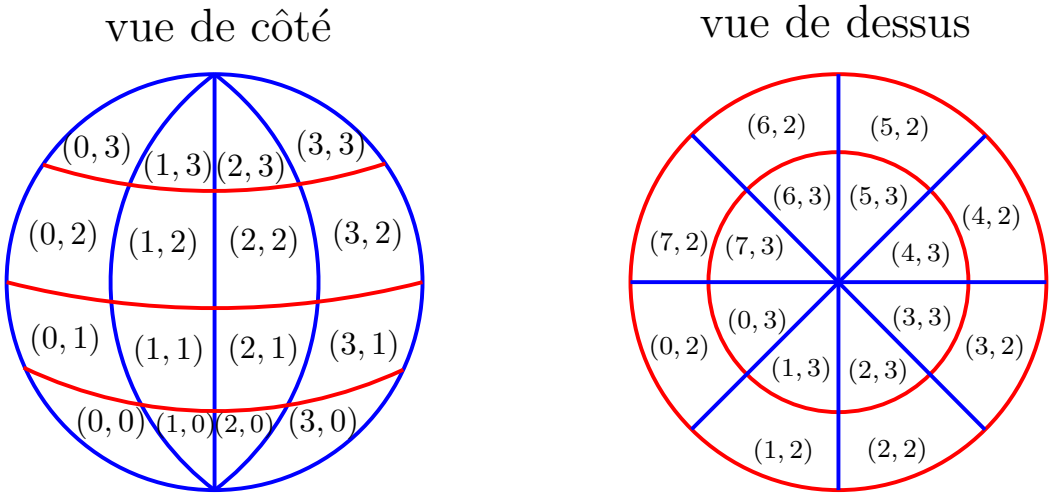
Le tableau suivant donne pour une grille 4×4 ($x \in \{0, 1, 2, 3\}$, $y \in \{0, 1, 2, 3\}$) la position après un mouvement dans la grille :

Position\Opération	NORTH	SOUTH	EAST	WEST
(0, 0)	(0, 1)	(0, 3)	(1, 0)	(3, 0)
(1, 0)	(1, 1)	(1, 3)	(2, 0)	(0, 0)
(1, 1)	(1, 2)	(1, 0)	(2, 1)	(0, 1)
(2, 0)	(2, 1)	(2, 3)	(3, 0)	(1, 0)

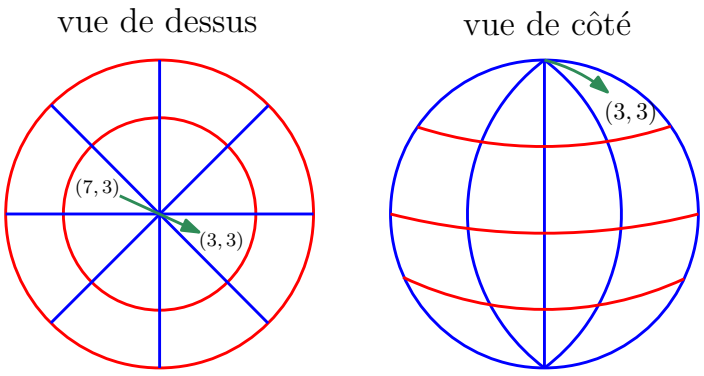
4.2 Grille sphérique

On souhaite définir une grille correspondant à une sphère. Pour cela, on définit un système de latitude et de longitude. La sphère est découpée en un nombre pair de parallèles (cercles horizontaux correspondant à l'intersection de la sphère avec un plan orthogonal à l'axe Nord-Sud) et de méridiens (demi-ellipses reliant le pôle Nord au pôle Sud). Dans ce modèle, x et y deviennent respectivement des représentations abstraites des longitudes et latitudes.

La figure ci-dessous illustre le concept de longitudes et latitudes sont la forme de couples (longitude, latitude) ainsi que les parallèles (en rouge) et les méridiens (en bleu) pour une grille 8×4 ($x \in \{0, 1, 2, 3, 4, 5, 6, 7\}$, $y \in \{0, 1, 2, 3\}$).



Pour la plupart des cas, les mouvements sont définis assez simplement, car la plupart des régions définies par ce système de coordonnées ont quatre côtés et donc quatre régions adjacentes (une par direction possible). Par exemple, la région $(0, 2)$ représentée dans la figure ci-dessus a quatre voisines, $(0, 3)$ au Nord, $(1, 2)$ à l'est, $(0, 1)$ au Sud et $(7, 2)$ à l'Ouest. Les régions touchant un des pôles n'ont que trois côtés et dans ce cas on considère que la région voisine dans la direction du pôle est celle opposé par rapport au pôle. Par exemple, la région considérée comme étant au Nord de la région $(7, 3)$ est la région $(3, 3)$. On considère que le robot traverse le pôle et se retrouve donc de l'autre côté. La région $(7, 3)$ a donc bien quatre voisines, $(3, 3)$ au Nord, $(0, 3)$ à l'Est, $(7, 2)$ au Sud et $(6, 3)$ à l'Ouest. La figure ci-dessous illustre le mouvement en vert du robot de la région $(7, 3)$ à la région $(3, 3)$.



Dans le cas d'un mouvement passant par un pôle, l'orientation du robot est inversée (un robot orienté vers le Nord traversant le pôle Nord devenant orienté vers le Sud et inversement un robot orienté vers le Sud traversant le pôle Sud devenant orienté vers le Nord).

5 Spécification des nouvelles entrées/sorties

Le format de fichier d'entrée pour la configuration des *rovers* est modifié pour être un fichier de type YAML.

Le fichier d'entrée a deux nœuds principaux :

- un nœud `grid` correspondant à la grille et donnant la largeur (`width`), hauteur (`height`) et type (`kind`) de la grille, le type de la grille peut être `RECTANGULAR` (rectangulaire), `TOROIDAL` (toroïdal) ou `SPHERICAL` (sphérique) ;
- un nœud `rovers` correspondant aux *rovers* et donnant pour chaque *rover* sous la forme d'une liste
  - sa position avec des coordonnées entières `x`, `y` et une `orientation` (`NORTH`, `EAST`, `WEST` ou `SOUTH`) et
  - les commandes à exécuter sous forme d'une liste d'éléments `LEFT`, `RIGHT` ou `MOVE` (même signification que les lettres `L`, `R` et `M`).

Par exemple, une configuration correspondant à une grille rectangulaire de taille 5×5 avec deux *rovers* :

- le premier de coordonnées $(1, 2)$, orienté vers le nord avec comme commandes `LMLMLMLMM` ;
- le second de coordonnées $(3, 3)$, orienté vers l'est avec comme commandes `MMRMMRMRRM` ;

aura le fichier de configuration config.yml suivant :

```yaml
grid:
  width: 5
  height: 5
  kind: RECTANGULAR
rovers:
  - position:
      coordinates:
        x: 1
        y: 2
      orientation: NORTH
    commands:
```

```
      - LEFT
      - MOVE
      - LEFT
      - MOVE
      - LEFT
      - MOVE
      - LEFT
      - MOVE
      - MOVE
  - position:
      coordinates:
        x: 3
        y: 3
      orientation: EAST
    commands:
      - MOVE
      - MOVE
      - RIGHT
      - MOVE
      - MOVE
      - RIGHT
      - MOVE
      - RIGHT
      - RIGHT
      - MOVE
```

Le fichier de sortie a deux nœuds principaux :

- un nœud `percentageExplored` correspondant au pourcentage de la grille explorée au format nombre entier suivi de `%` ;
- un nœud `finalRoverPositions` correspondant à la liste des positions finales des *rovers* (coordonnées entières `x`, `y` et une `orientation`)

```
percentageExplored: 28 %
finalRoverPositions:
- coordinates:
    x: 1
    y: 3
  orientation: NORTH
- coordinates:
    x: 4
    y: 3
  orientation: EAST
```

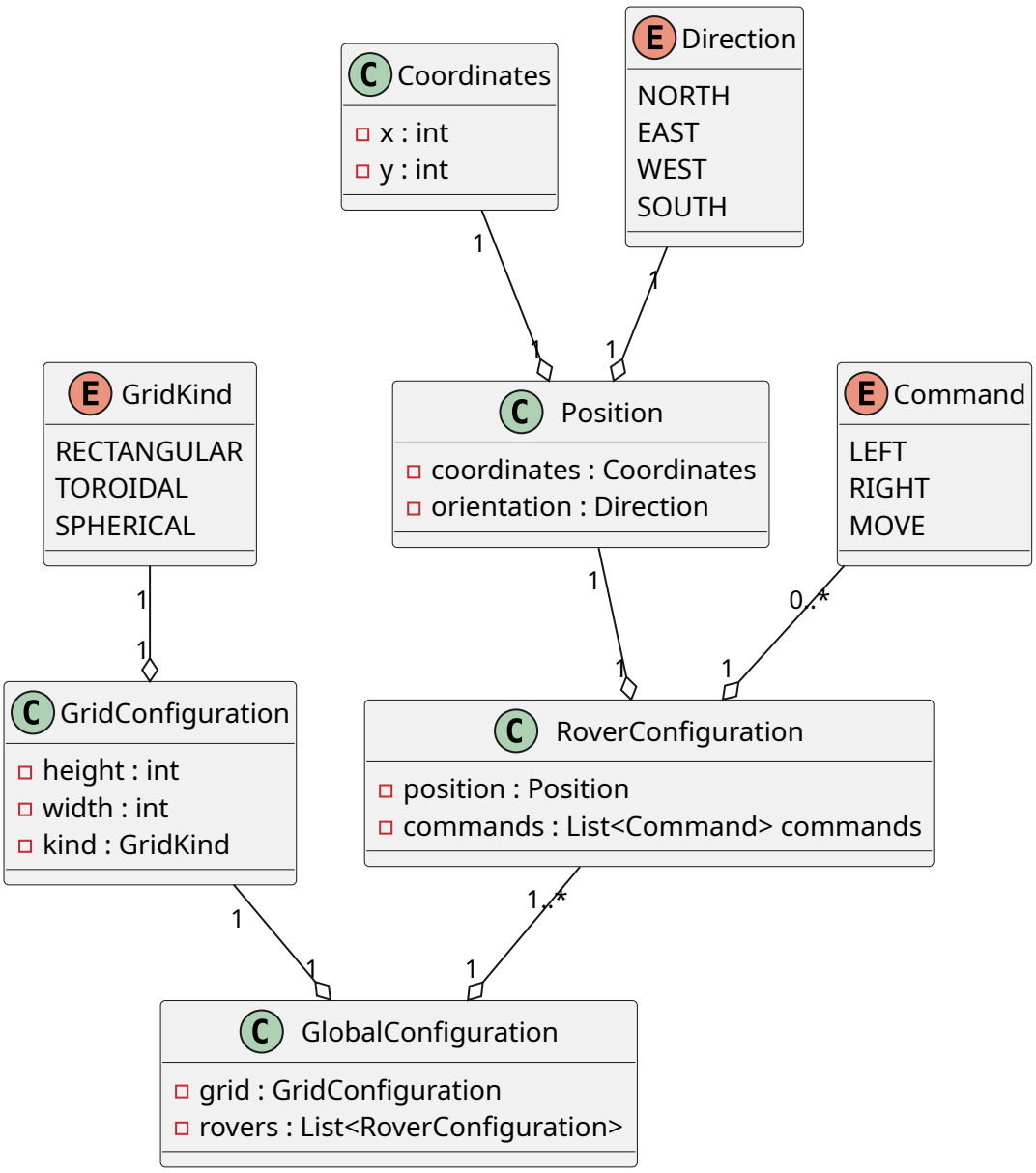
5.1 Utilisation de la bibliothèque Jackson

Pour la lecture des fichiers d'entrées et l'écriture des fichiers de sortie, on vous conseille fortement d'utiliser la *library* Jackson. Cette *library* permet de lire et écrire des fichiers au format *YAML* (et aussi *json*). Pour pouvoir l'utiliser, il vous suffit de rajouter la dépendance ci-dessous dans le fichier `build.gradle.kts` de votre projet.

```
dependencies {
    implementation("com.fasterxml.jackson.dataformat:jackson-dataformat-yaml:2.17.2")
}
```

5.1.1 Lecture de fichier

Pour lire un fichier, il faut d'abord créer une structure de classes correspondant à la structure en arbre du format de fichier *YAML*. Pour le format qui nous intéresse la structure doit être la suivante (les constructeurs évidents et les *getters* pour chaque attribut, avec le même nom, sont omis pour éviter de surcharger la figure) :



Afin de vous faciliter la déclaration de ces classes, on vous conseille d'utiliser les classes *record* de Java. Les *records* sont des classes spéciales qui permettent de définir succinctement des classes qui agrègent des valeurs de manière immuable. Pour les définir la syntaxe est similaire à celle des classes normales en remplaçant `class` par `record` et en spécifiant les attributs de la classe entre parenthèses après le nom du *record*.

Par exemple, on peut définir un record `Rectangle` avec deux attributs `length` et `width` avec le code suivant :

```
public record Rectangle(double length, double width) { }
```

Un *record* possède les caractéristiques suivantes :

— chaque élément de sa description est un attribut `private` et `final` ;
— un *getter* public est défini pour chaque élément ;
— un constructeur public qui possède la même signature que celle de la description qui initialise chaque

élément avec la valeur correspondante fournie en paramètre ;
— une redéfinition des méthodes `equals()` et `hashCode()` qui garantit que deux instances du record sont égales si elles sont du même type et qu'elles contiennent les mêmes attributs égaux.

Le record `Rectangle` est équivalent à la classe Java suivante :

```
public final class Rectangle {
    private final double length;
    private final double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    double length() { return this.length; }
    double width()  { return this.width; }

    // Implementation of equals() and hashCode(), which specify
    // that two record objects are equal if they
    // are of the same type and contain equal field values.
    public boolean equals...
    public int hashCode...

    // An implementation of toString() that returns a string
    // representation of all the record class's fields,
    // including their names.
    public String toString() {...}
}
```

Une fois que vous avez créé les classes, pour lire un fichier afin de créer un objet il faut :

1. Créer un `ObjectMapper` à partir d'une `YAMLFactory` ;
2. Ouvrir le fichier YAML à lire dans un `InputStream` ;
3. Lire le fichier afin d'obtenir l'objet correspondant à la racine du fichier *YAML* à l'aide de la méthode `readValue` d'`ObjectMapper` en donnant en argument l'`InputStream` et la classe de l'objet.

Cela donne par exemple le code suivant pour ouvrir un fichier `config.yml` situé dans le répertoire `ressources` du projet :

```
final ObjectMapper objectMapper = new ObjectMapper(new YAMLFactory());
try {
  final InputStream inputStream = App.class.getResourceAsStream("/config.yml");
  final GlobalConfiguration globalConfiguration =
    objectMapper.readValue(inputStream, GlobalConfiguration.class);
} catch (IOException e) {
  e.printStackTrace();
```

```
}
```

5.1.2 Écriture de fichier

Pour écrire un fichier *YAML*, il faut :

1. créer un `ObjectMapper` à partir d'une `YAMLFactory` (vous pouvez réutiliser celui utilisé pour la lecture) ;
2. ouvrir un nouveau fichier *YAML* à écrire dans un `OutputStream` ;
3. récupérer un `ObjectWriter` en appelant la méthode `writer` d'`ObjectMapper` ;
4. récupérer un `SequenceWriter` en appelant la méthode `writeValues` d'`ObjectWriter` en donnant l'`OutputStream` en argument ;
5. écrire le fichier en appelant la méthode `write`.

Cela donne par exemple le code suivant pour écrire un fichier `path/output.yml` avec le contenu d'un objet de type `GlobalOutput` :

```
ObjectMapper objectMapper = new ObjectMapper(new YAMLFactory());
try {
  GlobalOutput globalOutput = ...
  ObjectWriter writer = objectMapper.writer();
  ileOutputStream fos = new FileOutputStream("path/file.yml");
  SequenceWriter sw = writer.writeValues(fos);
  sw.write(globalOutput);
} catch (IOException e) {
  e.printStackTrace();
}
```