amU Faculté des sciences Aix Marseille Université

Année 2025-26

Domaine Sciences et Technologies
Master informatique
Génie logiciel : TD 1
Codes UE : SINA09EL
Mars Rover (1/3)

Le but de ce TD est de concevoir un simulateur pour des *rovers* de la NASA explorant un terrain. Un terrain martien doit être parcouru par les *rovers* afin que leurs caméras embarquées puissent obtenir une vue complète du terrain environnant et la transmettre à la Terre. Le terrain est modélisé sous la forme d'une grille qui peut être rectangulaire ou toroïdale (détails ci-après). La grille a une largeur l et une hauteur h quel que soit son type. La position d'un *rover* est représentée par un triplet composé des deux coordonnées entières $0 \leq x < l$ et $0 \leq y < h$ et d'une orientation correspondant à un des quatre points cardinaux (Nord, Sud, Est ou Ouest).

Pour contrôler un *rover*, la NASA donne une suite finie de commandes. Les commandes possibles sont :

- *tourner à gauche* : fait tourner le *rover* de 90 degrés vers la gauche (sens inverse des aiguilles d'une montre) ;
- *tourner à droite* : font tourner le *rover* de 90 degrés vers la droite (sens des aiguilles d'une montre) ;
- *avancer* : indique au *rover* d'avancer d'une case dans le sens de son orientation.

L'entrée du problème pourra décrire plusieurs *rovers* en donnant pour chacun les coordonnées initiales de celui-ci (coordonnées d'atterrissage) ainsi que son orientation. Chaque *rover* agit indépendamment l'un après l'autre. On supposera qu'il n'y pas d'interactions entre les *rovers* (deux *rovers* peuvent être aux mêmes coordonnées au même moment). Le but étant d'explorer le terrain, la sortie contiendra le pourcentage de coordonnées explorées par les *rovers* (une paire de coordonnées étant considérée comme étant explorée si elle correspond à des coordonnées dans la trajectoire d'un des *rovers*, y compris leurs coordonnées de départ) ainsi que la position (coordonnées et orientation) finale de chaque *rover* (position à la fin de l'exécution des commandes ou position juste avant la destruction du *rover* si celui-ci est détruit).

1 Grille rectangulaire

Une grille rectangulaire correspond à la grille de base représentant un rectangle. Les coordonnées $(0, 0)$ correspondent au coin sud-ouest de la grille alors que les coordonnées $(l - 1, h - 1)$ correspondent au coin nord-est de la grille (l étant la largeur de la grille et h sa hauteur). On suppose que tout mouvement faisant sortir le *rover* de la grille rectangulaire entraîne sa destruction et une fois détruit il n'exécutera pas les commandes suivantes.

2 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 :

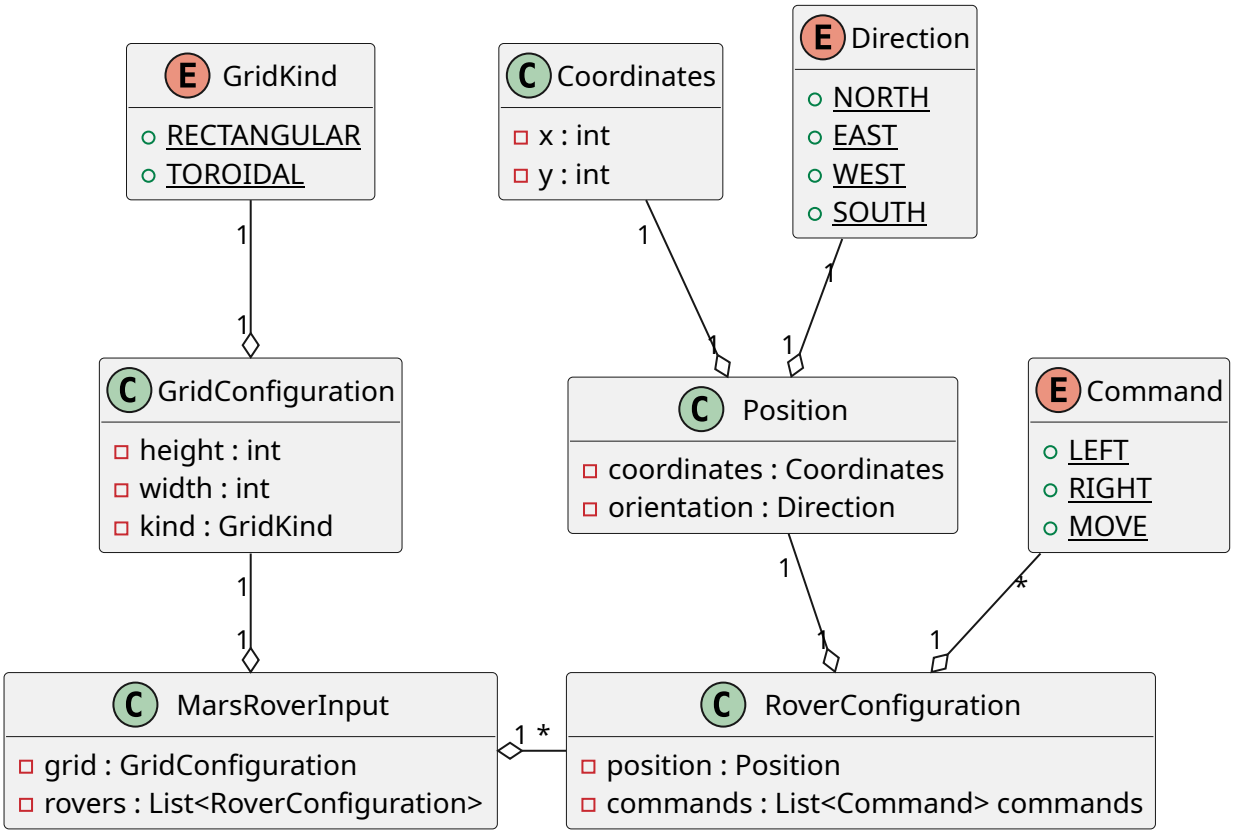
Position\Opération	NORTH	SOUTH	EAST	WEST
(0, 0)	(1, 0)	(3, 0)	(0, 1)	(0, 3)
(1, 0)	(2, 0)	(0, 0)	(1, 1)	(1, 3)
(1, 1)	(2, 1)	(0, 1)	(1, 2)	(1, 0)
(2, 0)	(3, 0)	(1, 0)	(2, 1)	(2, 3)

3 Spécification des entrées/sorties

L'entrée du problème contiendra les informations suivantes :

- Dimensions (largeur et hauteur) de la grille ;
- type de la grille : toroïdale ou rectangulaire ;
- la position de départ de chaque *rover* :
  - coordonnées initiales ;
  - orientation initiale.

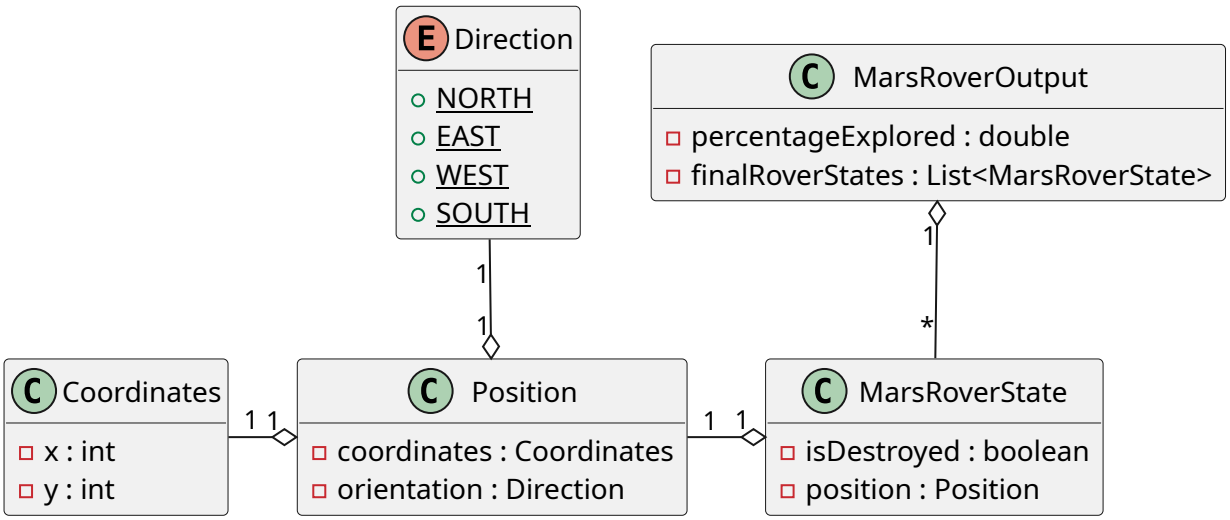
Pour simplifier, on supposera que l'entrée du problème est donnée par une instance de la classe `MarsRoverInput` ayant la structure suivante (on supposera que pour chaque attribut de classe il existe un accesseur dédié) :



La sortie du problème contiendra les informations suivantes :

— Pourcentage de la grille explorée ;
— l'état final de chaque *rover* :
  — un booléen indiquant s'il a été détruit ;
  — coordonnées finales ;
  — orientation finale.

Pour simplifier, on supposera que la sortie du problème est donnée par une instance de la classe `MarsRoverOutput` ayant la structure suivante :



Notez que les classes ci-dessus ne sont que des classes pour récupérer les données à partir d'un fichier (en entrée) ou écrire des données dans un fichier (en sortie) et ne sont pas forcément les classes à utiliser pour résoudre le problème de *mars rover*.

Question 1 : Quelles sont les interfaces et les classes qui vous sembleraient utiles de définir pour résoudre ce problème de *mars rover* ?

Question 2 : Dessinez un diagramme de classes du simulateur de mars *rover* avec les deux types de grilles.

On souhaite maintenant rajouter une nouvelle commande en plus de trois déjà présentes qui permettrait au *rover* de reculer, c'est-à-dire de se déplacer dans la direction inverse de son orientation. Par exemple un *rover* orienté vers le nord qui recule, se déplace vers le sud tout en gardant son orientation vers le nord.

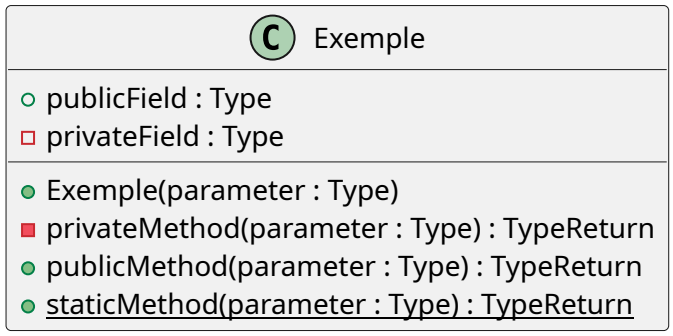
Question 3 : Modifiez le diagramme de classes de la question précédente afin de prendre en compte le fait que les *rovers* puissent reculer.

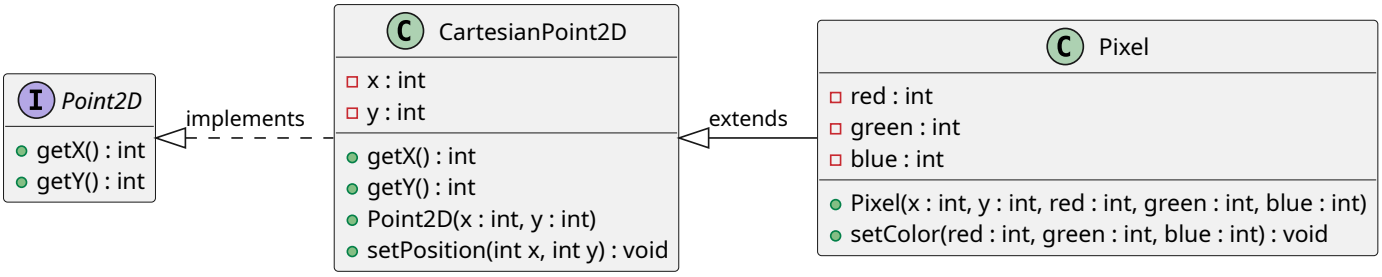
On souhaite maintenant prendre en compte la présence d'obstacles dans les terrains. Des obstacles (présents sur certaines cases) détruisent tout *rover* s'y déplaçant (y compris pour l'atterrissage si l'obstacle est présent sur

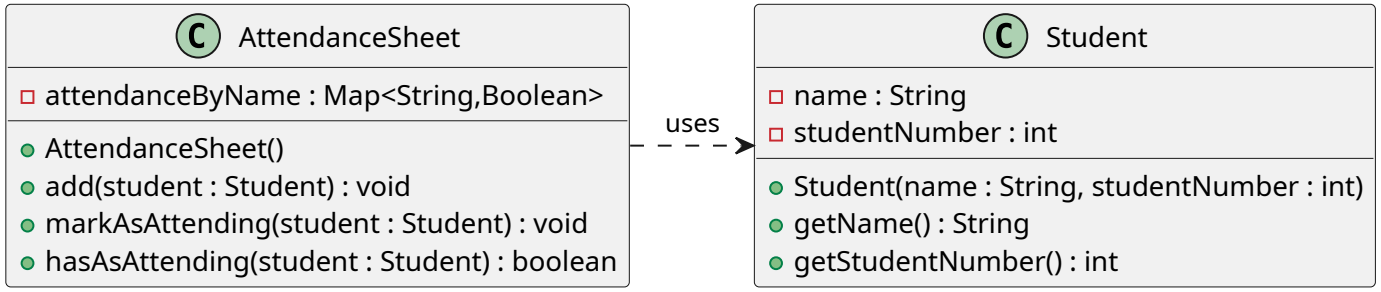
ses coordonnées initiales). L'entrée du simulateur a sortie du simulateur devra dorénavant indiquer si le *rover* est détruit ou pas à la fin de la simulation.

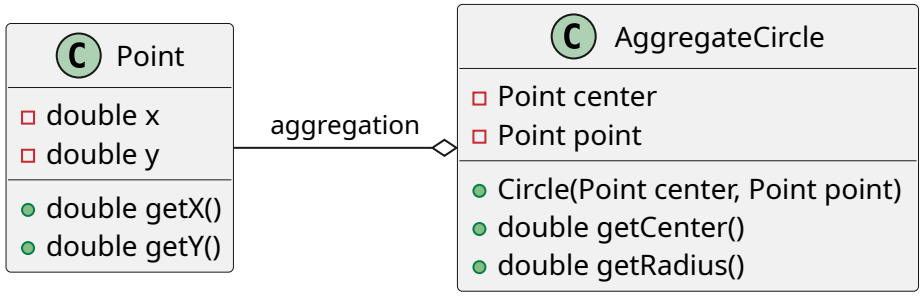
> **Question 4 :** Modifiez le diagramme de classes de la question précédente afin de prendre en compte le fait que des obstacles soient présents sur le terrain.

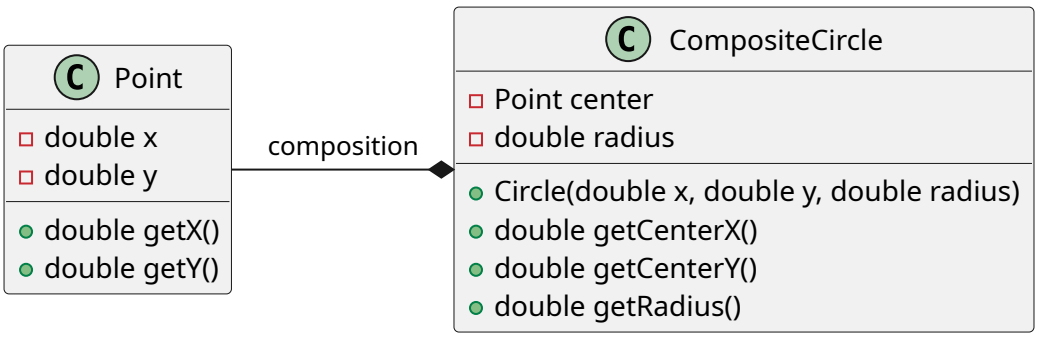
4 Rappels diagramme de classes










Point
double x
double y
double getX()
double getY()
composition
CompositeCircle
Point center
double radius
Circle(double x, double y, double radius)
double getCenterX()
double getCenterY()
double getRadius()