TP de Programmation Orientée Objet

Carlos Mendes, Marcos Ribeiro, Raphael Amarante

14 novembre 2018

1 Introduction

Le projet a comme objectif le développement de nos compétences en programmation orientée aux objectes. Pour cela on implémente un programme Java qui, avec la représentation d'une carte qui possède des cases avec des différents types de terrain ; une liste de coordonnées d'incendies et une autre de robots (de caractéristiques différentes) ; gère les opérations de ces robots de manière articulée pour que tous les incendies soient supprimés, de préférence le plus vite possible.

2 Conception du Projet

On a divisé le projet dans 6 packages, chacun contenant un groupe de classes qui sont fortement reliées, avec son diagramme. Une explication du contenu de chaque bloque est donnée ci-dessous :

2.1 Simulation

Dans ce bloque on a les classe les plus importantes du projet, celles qui vraiment permettent le (correct) fonctionnement de la simulation et qui doivent être expliquées en profondeur.

ChefPompier
Classe qui fait le contrôle des actions des robots. Pour le robot en question elle trouve l'incendie le plus proche de lui avec la fonction **assignerIncendie**. Elle trouve aussi le chemin le plus court pour qu'un robot peut faire son remplissage avec la fonction **assignerRemplissage**. Ce qu'elle fait dans ces deux cas est d'appeler la fonction qui calcule le chemin et, une fois trouvé, appeler la séquence d'événements qui feront les actions de déplacements et ensuite d'intervention ou de remplissage.

DonnesSimulation
Classe qui contient la carte, la liste de robots et la liste d'incendies.

LecteurDonnees
Classe qui fait la lecture du fichier de données et crée la carte, les cases, les robots, et les incendies.

PathCalculator (en utilisant PairDoubleCase et PairListCaseDouble)
Classe qui calcule le chemin le plus courte pour une destination désirée. Elle a seulement une méthode public qui reçoit la carte, le robot en question, et le case de destination, et retourne un objecte **PairListCaseDouble** qui contient la séquence en ordre des cases et les coûts qui composent le chemin. Tout les autres éléments de la classe sont privés. **PairDoubleCase** est seulement une classe d'éléments qui contient une clé et une valeur pour meilleur implémenter une **PriorityQueue** dans le calcul du chemin.

RobotsPompiers
Classe qui contient **main**. Fait la lecture du fichier, crée un nouveau GUISimulator et un nouveau Simulateur. Ensuite, commence la simulation avec **start**. En cas de fichier illisible, inconnu, ou hors format, on lance une exception.

RobotsPompiersException
Classe extension de la classe java **Exception**.

Simulateur
Méthodes Principaux :
- start : appelle le **chefPompier** pour assigner chaque robot à l'incendie les plus proche et appelle la méthode **draw** ;
- restart : fait la relecture du fichier d'entrée, met en place les données et appelle **start** ;
- next : vérifie s'il y a des incendies pas encore supprimés avec la méthode **checkIncendies**. Ensuite, si la simulation n'est pas encore finie (méthode **simulationTerminee**), elle incrémente la date avec **incrementeDate** ; traite l'action de chaque robot ; exécute les événements ; vérifie s'il y a quelque robot en repos ; et, enfin, dessine la carte ;
- draw : dessine toute la carte avec les incendies, les robots, les terrains et les différentes couleurs, forme et tailles de chaque élément ;
- executerEvenements : vérifie pour chaque événement si sa date est la date actuelle et, si positif, l'exécute ;
- checkRobotsAuRepos : si le robot en question est en arrête et il n'a pas des événements pas encore réalisés (fait avec la méthode **evenementSuivant**), on vérifie s'il doit remplir ou intervenir. Ensuite on appelle le **chefPompier** pour trouver le mieux chemin ;

- evenementSuivant : vérifie s'il y a un événement quelconque pour le robot en question dans la date suivante ;

2.2 Simulation.Carte

Ce package contient tout ce qui concerne la carte, comme les cases, les incendies, le terrain et la couleur qu'on associe à chaque chose.

Carte
Méthodes Principaux :
- getVoisin retourne la case voisin dans la direction désirée ;
- voisinExiste retourne vrai si le voisin dans la direction désirée existe ; sinon retourne faux ;
- isNearEau : retourne vrai si la case en question a un voisin dont la nature est "EAU" dans quelque direction ; sinon retourne faux.

Case
Définit la case.

Couleur
Classe enum qui définit les couleurs suivantes : RED (incendie), BLUE (eau), GREEN (foret), BROWN (roche), WHITE (terrain libre), PURPLE (habitat).

Incendie
Méthodes Principaux :
- eteindreIncendie : diminue la quantité des litres nécessaires pour éteindre l'incendie.

NatureTerrain
Classe enum qui définit la nature des terrains suivantes : EAU, FORET, ROCHE, TERRAINLIBRE, HABITAT.

2.3 Simulation.Evenement

Ici on a les classes qui définissent les événements possibles ainsi que la direction de cette action (NORD, SUD, EST, OUEST), si elle existe. On a **Evenement** comme la classe père, et **Deplacer, Intervenir, Remplir** comme les classes filles.

Evenement
Fait l'association entre le robot qui demande l'événement et sa date de début.

Direction
Classe enum qui a une seule méthode. C'est le méthode static **getDirection**, qui vérifie quelle est la direction de voisinage entre deux Cases, une source et une destine.

Deplacer, Intervenir, Remplir
Ces classes ont le méthode **execute** qui change l'état du robot et ajoute l'action a être fait par ce robot.

2.4 Simulation.Evenement.Action

Ce groupe de classes implémente l'action elle-même de chaque événement, c'est-à-dire, le processus de mise-à-jour des valeurs des éléments en question en appelant les méthodes d'incendie ou de robot. On a **Action** comme la classe père, et **Deplacement, Intervention, Remplissage** comme les classes filles.

Action
Bref, les objectes de cette classe ont une date initial et une date final. Elle possède une méthode abstrait **finir**, implémentée dans les classes fils.

Deplacement
Cette classe a comme attribut la direction de déplacement. La méthode **finir** est ici implémentée comme la mise à jour de la position du robot.

Intervention
Cette classe a comme attributs l'incendie en question, un boolean qui vérifie si l'incendie est fini, et un autre qui vérifie si le réservoir d'eau est vide. La méthode **finir** est ici implémentée comme le suivante : d'abord on vérifie si c'est la fin de l'incendie et en cas négatif on seulement dissocie cet incendie comme cible du robot en question (pour qu'un autre puisse intervenir) ; ensuite on vérifie si le réservoir du robot en question est vide et on appelle **chefPompier** pour trouver le chemin pour faire une remplissage ou une autre intervention.

Remplissage

La méthode **finir** est ici implémentée comme la complète remplissage du robot en question et, directement après, l'appel au **chefPompier** pour l'assigner un incendie.

2.5 Simulation.Robot

Ce bloque-ci englobe les types de robots ainsi que son état. D'abord on a divisé les robots entre Drone et Terrestre, dont la plus grande différence est le type de remplissage, les drones sur l'eau et les terrestre à côté d'eau. Ensuite on a divisé Terrestre entre Roues, Chenilles, Pattes. Tout était fait avec les principes d'héritage et d'abstraction. De plus, on a la classe TypeRobot qui contient les valeurs défaut de chaque type.

TypeRobot
Classe enum qui contient les valeurs défaut de chaque type de robot : DRONE, ROUES, CHENILLES, PATTES.

EtatRobot
Classe enum qui contient les états des robot : INTERVENTION, DEPLACEMENT, REMPLISSAGE, ARRETE.

Robot
Méthodes Principaux :
- addAction : vérifie si le robot est encore en train de faire quelque chose et, en cas négatif, appelle une des méthodes suivantes ;
- addDeplacement, addIntervention, addRemplissage : méthodes qui créent une action avec ses dates de début et de fin ;
- traiterAction : vérifie le type d'action et appelle la méthode deverserEau ou remplirReservoir . Ensuite, si l'action est fini, elle change l'état du robot à ARRETE.

Drone, Terrestre, Roues, Chenilles, Pattes
Méthodes Principaux :
- getVitesse : retourne la vitesse du robot selon son type et selon la nature du terrain sur lequel il est.

2.6 Tests

Dans ce package on a fait le test initial pour vérifier le correct fonctionnement de l'algorithme du chemin le plus court en utilisant la classe **ShortestPathTester**.
Les autres tests sont fait seulement en changeant le fichier d'entrée pour vérifier le comportement de la simulation.

3 Décisions et Fonctionnement

Pour implémenter le programme on a pris quelques décisions importantes :
- Le seule type de robot qui peut traverser l'eau est le Drone ;
- Un robot assigné à un incendie est le seule qui peut faire son intervention jusqu'au moment où son réservoir est vide et il faut le remplir. On comprend, donc, que quand un robot aux pattes est assigné il est le seule à faire l'intervention de cet incendie puisque il n'a jamais besoin de remplir son réservoir ;
- Pour faciliter les calculs de temps de déplacement d'un case à l'autre on considère tout le terrain de la source, et pas la moitié de chaque case ;
- La vitesse de déplacement est mesurée en mètres/seconde, et la vitesse d'intervention et de remplissage sont mesurées en litres/seconde ;
- Le format du fichier est le même décrit dans l'annexe A de l'énoncé du projet.

Ainsi, pour faire fonctionner la simulation, il faut compiler les paquets et exécuter la méthode main de la classe **RobotsPompiers** en passant le chemin du fichier avec les donnés d'entrée. Donc, dans la fenêtre qui nous est montrée, on peut configurer la vitesse d'exécution, exécuter la simulation pas à pas ou tout de suite, la redémarrer ou quitter.

4 Résultats

Avec des différents fichiers il est possible de vérifier le correct fonctionnement de la simulation sous différents conditions. Ci-dessous une liste de certaines choses qu'on peut observer dans les tests :
- Si on a un type de terrain qui traverse la carte entière et qui n'est pas accessible à un type de robot, certaines robots seront confinés dans son zone, et peut être n'aideront pas dans l'intervention de quelques incendies ;
- On vérifie le changement de vitesse en terrains spécifiques pour robots spécifiques ;
- On vérifie la taille des incendies en décroissance jusqu'au moment qu'il disparaît ;
- On vérifie qu'on peut lire du fichier des vitesse pour les robots différentes de ses défauts ;
- On vérifie qu'il peut y avoir des cas impossibles.

Extension et Améliorations : il est toujours possible d'améliorer le programme. C'est possible d'implémenter un algorithme plus robuste du **chefPompier** pour vraiment trouver les décisions les plus efficaces pour supprimer les incendies. C'est possible aussi de faire multiples pompiers intervenir dans un même incendie. Finalement, comme extension, on pourrait ajouter une troisième dimension dans la carte (en imaginant des bâtiments avec différents étages en feu), ou ajouter la dispersion des incendies.