Simulation d'une équipe de robots pompiers

TP de Programmation Orientée Objet

Matthias Bouderbala, Philémon Fieschi et Alexis Bigé

1 Introduction

Ce compte-rendu permet de résumer le contenu du code du projet. Le code a été écrit par les trois membres du groupe. Il s'organise de la manière suivante dans le dossier `src` :

- le package `data` : les classes de données (carte, robots, incendies...)
- le package `gui2` : le simulateur et son scénario, liés à `gui.jar`
- le package `events` : les classes d'évènements
- le package `chemin` : la classe qui définit un chemin
- le package `io` : la lecture et création de données
- le package `strategie` : les chefs pompiers (élémentaire et avancé)
- les tests de base

2 Choix de conception

2.1 Le choix des classes de données

Bien guidés par le sujet, nous avons implémenté les différentes classes de données : `Carte ; Case ; DonneesSimulation ; Incendie`. Les méthodes ont été ajoutées au fur et à mesure selon les besoins pour les différentes étapes du projet.

2.2 Le choix des méthodes des robots

Le package `robot` contient tous les types de robots, qui référencent une classe principale `Robot`. Cette dernière contient les méthodes de base pour les robots ainsi que celles, essentielles, pour les différentes actions des robots. Nous avons choisi de laisser le robot gérer ses déplacements, quand il reçoit un ordre : `deplacementCase ; ordreRemplissage ; ordreIntervention`, il ajoute ensuite lui-même les évènements au simulateur en prenant en compte le temps d'action.
Parmis ces méthodes, on pourra remarquer :

— `Dijkstra` : renvoie le plus court chemin selon l'algorithme de Dijkstra, implémenté à partir de la version décrite sur la page `https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra`. Nous avons choisi cet algorithme parce que nous l'avons vu l'an dernier et savons qu'il est plutôt efficace. Nous avons fait en sorte d'utiliser un maximum de collection pour optimiser son calcul. Nous avons aussi utilisé de la mémoisation.

— `ordreIntervention` : gère le déplacement (si nécessaire) et l'intervention du robot sur un incendie dans le simulateur.

— `intervenir` : gère l'intervention directe du robot sur un incendie (le robot a auparavant géré son déplacement).

— `ordreRemplissage` : gère le déplacement (si nécessaire) et le remplissage du robot dans le simulateur. Il utilise la méthode `choisirCaseEau` qui permet d'obtenir la case où se déroulera le remplissage.

Toutes ces actions sont ainsi ajoutées au simulateur selon le temps que le robot prend, temps calculé selon la nature du robot, sa vitesse, la nature du terrain et en partant du principe que le robot parcourt la moitié de la case où il est et la moitié de la case où il va.

2.3 Le choix de la classe Chemin

Afin de faciliter la recherche d'un plus court chemin, nous avons implémenté une classe `Chemin` qui contient deux `List<>` :

— `List<Case>` : une liste de cases (ordonnées selon la date du déplacement).

— `List<Long>` : une liste de dates (ordonnées).

Avec des méthodes classiques pour ajouter ou récupérer des éléments dans Chemin, nous avons pu utiliser ce type de données pour implémenter la recherche d'un plus court chemin dans `Robot`.

2.4 Le choix du simulateur et de l'implémentation des évènements

Notre classe `Simulateur` contient un attribut de type `Scenario` qui permet de gérer la séquence d'évènements qui s'exécutent au cours du temps. Ce `Scenario` contient simplement une `ArrayList<Evenement>` qui permet de garder les évènements ordonnés par date et ainsi d'en ajouter continuellement au mileu ou à la suite. Héritées de la classe abstraite `Evenement`, les classes `DeplacementUnitaire ; EvenementMessage ; Intervention ; Remplissage` permettent d'effectuer les actions indiquées au robot. En effet, c'est le robot lui-même qui a ajouté ces évènements au simulateur, après avoir reçu des ordres.

2.5 Le choix de la stratégie

Nous avons implémenté deux stratégies, à l'aide de deux chefs pompiers, sous-classes de `Chef`, où sont implémentées les principales méthodes liées à la connaissance de la carte en temps réelle (incendies restants, robots occupés...) :

- `ChefElementaire` : la plus grosse partie du travail était déjà effectuée auparavant avec la gestion des évènements par les robots.
- `ChefAvance` : le chef associe à chaque robot l'incendie le plus proche, et si tous les robots sont occupés, le chef attend un peu avant de recommencer

Pour cette dernière stratégie, nous avions fait l'inverse (le robot le plus proche pour chaque incendie), mais cela n'apportait pas grand chose par rapport à la stratégie élémentaire.

3 Tests et résultats obtenus

3.1 Premiers tests : la lecture des données et les classes de données

Nous avons choisi de tester nos classes et méthodes au fur et à mesure. En premier lieu, il fallait s'assurer du bon fonctionnement des méthodes des classes de données, dont la principale : `DonneesSimulation`. Le fichier `TestCreationDonnees.java` contient ainsi nos premiers tests sur ces classes de données. Nous n'avons pas eu de soucis particulier à ce niveau.

3.2 L'affichage graphique

Notre affichage, géré dans la classe `Simulateur`, était plutôt basique au départ, avant d'être progressivement amélioré avec des fichiers `.png` faits à la main. Il est certain que le projet ne repose pas sur la qualité de l'affichage, mais il nous paraissait évident d'implémenter une interface qui soit agréable à visionner.

3.3 Les évènements

Au départ, nous avions implémenté davantage d'évènements, en différenciant par exemple s'il s'agissait d'un déplacement vers une case voisine ou non, dans une direction particulière, etc...
Cela nous était utile pour tester le simulateur et l'organisation des évènements (voir les tests évènements `.java`).
Puis, nous avons gardé seulement les évènements utiles lorsque le robot a du gérer ses déplacements après avoir reçu un ordre.

3.4 La stratégie

Nous n'avons pas eu de soucis particuliers pour tester les deux stratégies, avec deux fichiers de test : `TestStrategieElementaire` et `TestStrategieAvancee`. Les tests nous ont permis d'améliorer un maximum la stratégie avancée, qui au début n'était pas très différente de notre stratégie élémentaire.

4 Conclusion

Ce projet nous a permis de nous améliorer en Java, de travailler certaines notions comme l'héritage, dans une situation concrète. Nous avons pu effectuer ce projet en équipe malgré l'absence d'heures communes et des emplois du temps différents : nous sommes un ISI, un SEOC et un MMIS.