Sebastien HERBRETEAU
Alexandre DONNE
Victor BROS

# Rapport Projet Java

## 1 Organisation du code

Nous avons créé huit packages, dont un de tests sur lequel nous reviendrons.
Le package **Decision** contient le chef robot et les différentes stratégies qu'il peut utiliser.

**Environnement** contient tout ce qui est relatif à la carte (Case, Nature, Itineraire ..).
**Evenements** contient tous les événements.
**IO** contient le lecteur de données, ainsi que la classe **DonneesSimulation** et le simulateur.
Robots contient tout ce qui est relatif au robot, avec dedans un autre package **StratégieDeplacement** pour les différentes stratégies.
Nous avons décidé de séparer les classes dans des packages en fonction de leurs rôles et de ce qu'elles représentent.

Nous avons créé une énumération **TypesRobot** qui possède les différents types de robot ainsi que les particularités de chaque type (l'image sur l'interface graphique, le temps de rechargement), ce qui allège le corps des différentes classes robots.
La classe robot a une méthode getType() qui renvoie le type, avec celui-ci on peut récupérer d'autres informations.

Notre classe abstraite **AbstractRobot** contient ainsi plusieurs méthodes abstraites à redéfinir, comme ce getType() et aussi le getTempsOperation() qui renvoie le temps que mettra le robot pour déverser une quantité d'eau.

Pour d'autres détails sur les classes nous avons fait le diagramme de classes de différents packages de notre projet.

## 2 Simulation de scénarios

Pour permettre d'animer nos robots sur la carte, il est nécessaire d'utiliser un simulateur. Celui-ci contient une liste d'évènements datés qui seront exécutés au fur et à mesure de la simulation.
Cette liste est en fait une **PriorityQueue**, cette structure a été utilisée car elle permet de trier les événements selon leurs dates, et quand nous exécutons un événement nous le dépilons, ainsi il est supprimé et ne sera pas exécuté de nouveau.

Nous avons tout d'abord créé une classe abstraite **Evenement**. Celle-ci a pour attribut une date et pour méthode abstraite execute(). Si l'on prend pour illustrer la classe **EvenementDeplacer**, l'appel de cette méthode a pour effet de modifier la position du robot qu'elle a pour attribut.

Chaque appel à next() déclenche une série de tâches :
- On exécute tout d'abord les événement du simulateur à la date courante. Cela a pour effet de modifier la position des robots si des déplacements étaient programmés à cette date. Des incendies peuvent également être pris en charge ou bien encore des robots peuvent passer de l'état libre à l'état occupé.
- On prend des décisions qui s'imposent compte tenu de la situation actuelle. Par exemple, si un robot est dans l'état libre et qu'un incendie n'est pas pris en charge, le « chef des pompiers » lui confie la mission de s'y rendre au plus vite et de verser l'eau nécessaire à son extinction. Chaque ordre du chef correspond en réalité à l'ajout d'événements datés dans le simulateur qui ne seront exécuté que plus tard (prendre une décision ne modifie pas la position du robot mais peut modifier sa disponibilité).
- Cette prise de décisions suit la stratégie que le chef utilise. Nous avons utilisé le patron stratégie.
Ainsi nous avons une **StrategieGlobaleSimple** et une **StrategieGlobaleOptimisee** que nous choisissons au début du programme.
- On incrémente ensuite la date courante lorsque toutes les décisions possibles ont été prises.
- Enfin on dessine la situation actuelle avec l'appel à draw().

## 3 Troisième partie : calculs de plus courts chemins

Pour qu'un robot puisse effectivement se rendre sur une case où se trouve un incendie, il est nécessaire qu'il sache calculer le plus court chemin d'une case à une autre. Nous avons dans cette partie décider d'implémenter l'algorithme de Dijsktra.
Nous avons utilisé le patron stratégie pour pouvoir éventuellement changer de stratégie (de déplacement) sans changer le coeur du code.
Nous passons par une interface **IStrategieDeplacement** que notre **StrategieDijkstra** implémente. Les robots ont en attribue une **IStrategieDeplacement**.

Pour cet algorithme nous avons besoin d'une représentation de la carte sous forme de graphe. Plusieurs solutions sont possibles ici. Nous avons pris le parti de représenter un graphe sous forme d'une matrice d'adjacence. Bien que cela n'est pas forcément très optimisé pour notre problème (une case ayant au plus quatre voisins, la matrice est pour essentiellement vide), cette représentation a le mérite d'être plus facile et rapide à implémenter.
Pour la carte du sujet (carte 8x8), on passe sur une représentation de graphe, grâce à la méthode toGraphe(), qui crée le graphe et donc en particulier une matrice 64 par 64 symétrique où **matrice[i][j]** donne le temps nécessaire pour un robot donné de se rendre de la case numéroté **i** à la case numéroté **j**.

Nous avons, en parallèle, implémenté l'algorithme de Dijkstra comme méthode de la classe **Graphe**. Cet algorithme renvoie ce que nous avons appelé un **Itineraire**. Cette classe regroupe une liste de couple (Case, temps). C'est cet itinéraire qui nous sera utile lors de la programmation des évènements de déplacement du robot.

De plus, pour permettre à un robot de se rendre sur la case de nature **Eau** la plus proche, nous avons créé une méthode spéciale appelé dijkstraEau(). Celle-ci fonctionne sur le même principe que l'algorithme de Dijkstra mais s'arrête dès qu'une case est marquée par l'algorithme. Comme la méthode dijkstra classique, l'algorithme renvoie un **Itineraire**, utile au robot pour programmer ses déplacement dans le simulateur.

## 4 Déroulement générale

La classe **Programm** est notre point d'entrée, elle crée le simulateur et démarre la simulation.
Le simulateur dessine l'interface, lit les données, les transmet au chef pompier qui prend les premières décisions.
Ensuite l'appuie sur **next** met à jour l'interface et les décisions.

## 5 Les stratégies

Nous avons implémenté deux stratégies différentes, la stratégie simple et une stratégie optimisée.
La simple parcourt les incendies qui n'ont pas été pris en charge par un robot et l'attribue au premier robot non occupé qui a un réservoir non vide.
La stratégie optimisée parcourt les incendies non pris en charge et l'attribue au robot le plus proche qui n'est pas occupé.
Dans les deux, un incendie n'est pris en charge que par un seul robot, et lorsque celui-ci a déversé la quantité d'eau qu'il pouvait sur l'incendie, il est libéré. S'il lui reste de l'eau dans son réservoir il se verra attribué un autre incendie (peut-être le même) sinon il va se recharger.
Si l'incendie n'a pas été éteint entièrement par le robot, il sera attribué à un autre robot (ou peut-être le même).

## 6 Tests & Maps

Nous avons créé un test unitaire en utilisant JUnit, qu'on peut lancer depuis le makefile.

Nous avons également créé des maps simples permettant de tester les décisions prises par le « chef des pompiers » et donc vérifier si celles-ci sont optimales sur ces scénarios:
-AllerRetour: Le robot doit faire 2 passages par l'incendie puisque son réservoir est trop petit pour un seul versement.
-Detour: Le robot doit faire un détour pour passer sur les **TERRAIN_LIBRE** et **HABITAT** au lieu des **ROCHE.**
-Incendies: La gestion de deux incendies.
-LigneDroite: La trajectoire le plus courte est la ligne droite.
-Obstacles: Un seul des deux robots peut aller éteindre l'incendie.

Nous avons construit deux autres maps **mazeOfFate-11x11.map** et **imagOfDespair-20x20.map** en plus du map pool existant.

Le **MANIFEST.md** contient tout ce qu'il faut faire pour tester notre programme.

## 7 Rendu

Dans le projet, le répertoire diagramme contient les diagrammes de classes de certains package.
Et le répertoire javadoc contient la javadoc (en HTML).
Le répertoire bin contient les librairies que nous utilisé, gui, junit et hamcrest.