Rapport de TPL de POO

Équipe 25

15 Novembre 2016

Dans le rapport, nous ne nous étendons pas sur les getters et setters car nous n'avons pas juger utiles de détailler leur utilité/fonctionnement

1 Les données du problèmes

1.1 Les robots

Pour l'implémentation des différents robots, nous avons créé une classe abstraite **Robot** qui regroupe les attributs communs à tous les robots (la position, volume du réservoir,la vitesse,etc...) ainsi que certaines fonctions communes :

- *caseLaPlusProcheAutour* qui retourne la case voisine de notre position la plus optimale pour accéder à une case destination qui peut être plus éloignée ainsi que le chemin pour y accèder.
- *ajouteDeplacementsVersDest* qui ajoute à la liste des événements tous les déplacements nécessaires pour atteindre la destination.
- *eteindreIncendie* qui permet d'ajouter un ou plusieurs événements nécessaires pour tenter d'éteindre un incendie.
- *seRecharger* de même que pour *ajouteDeplacementVersDest*, sauf que la destination est calculée pour être le point d'eau le plus proche et ajoute aussi l'événement pour recharger le robot.
- *cheminExiste* qui vérifie l'existence d'un chemin pour une destination.

Ensuite, nous avons créé 4 sous classes, une pour chaque type de robot :

- **Drone**. *Attribut :* la vitesse max.
- **RobotChenilles**. *Attributs :* la vitesse max, normale et en fôret.
- **RobotPattes**. *Attributs :* la vitesse normale et sûr des rochers.
- **RobotRoues**.

Chacune ayant ses constructeurs, ses getters/setters, et des fonctions telles que deverserEau() ou remplirReservoir() qui varient légèrement selon le type de robot.

1.2 La carte, les cases et les incendies

Nous avons d'abord créer 2 types : **Direction** (NORD,SUD,EST,OUEST) et **NatureTerrain** (EAU, FORET, ROCHE, TERRAIN_LIBRE, HABITAT). Ils nous seront utiles pour déplacer les robots, pour définir la nature du terrain, etc..
Nous avons aussi implémenté 3 nouvelles classes :

- **Case**
- **Incendie**
- **Carte**

La première, **Case**, implémente les attributs des cases de la carte (tels que la ligne, la colonne, la nature du terrain), ainsi que les getters/setters et une fonction *estAccessible* qui retourne vrai si un robot donné peut accéder à cette case.
La classe **Incendie** quant à elle, génère les attributs des incendies. Comme pour **Case**, en plus des getters/setters, il y a d'autres fonctions comme :

- *estEteint* qui permet de savoir si l'intensité d'un feu vaut 0.
- *eteindre* qui diminue l'intensité d'un feu d'un nombre passé en paramètre.

Enfin, la classe **Carte** implémente les attributs des cartes, un constructeur et des getters. Elle définit aussi une fonction *voisinExiste* permettant de savoir si un voisin existe pour une case et une direction données.

1.3 Le simulateur

Pour le simulateur, nous avons créé une classe **Simulateur**. Pour le moment, elle définit 4 fonctions en plus du constructeur :

- *dessineCase* va utiliser les informations de la carte à des coordonnées passées en paramètre et va alors afficher l'image selon le type de terrain.
- *dessineCarte* va récupérer les données de toute la carte, puis va parcourir chaque case pour les dessiner avec la fonction précédante.
- *dessineIncendies* va, une fois la carte construite, ajouter les incendies sur la carte.
- *dessineRobots* va, de même que *dessineIncendies*, ajouter les robots sur différentes positions de la carte, avec des images différentes pour chaque type de robot.

Lors de l'appel du constructeur, on appellera à tour de rôle ces fonctions pour dessiner les données initiales de la carte. Mais pour récupérer les données de la carte, nous avons du créer une classe **DonneesSimulation** qui se chargera de créer les cartes, cases, incendies et robots pour permettre au simulateur d'afficher ces données.

1.4 Test du simulateur

Pour tester l'affichage du simulateur, nous avons programmé un petit fichier de test **TestAffichage.java**. Ce programme lit un fichier contenant les données d'une carte, créer la fênetre graphique puis lance le simulateur avec les données lues précédemment. Ainsi, nous pouvons nous assurer que la carte affichée correspond à celle du fichier lu.

2 Simulation de scénario

Comme conseillé dans le sujet, nous avons créé une classe abstraite **Evenement** qui implémente l'attribut de la date, un getter pour cette date et la méthode abstraite execute(). De plus, nous avons créé différentes classes filles, une pour chaque événement élémentaire, qui définissent un constructeur et la fonction execute() qui, selon les classes, fait une action différente :

- **Deplacement** pour le déplacement d'un robot. *Attributs :* un robot, une case destination et un carte.
- **Etat** pour changer l'état d'un robot (LIBRE, DEPLACEMENT,...). *Attributs :* un robot et un état.
- **Eteindre** pour éteindre un incendie. *Attributs :* un robot, un incendie et une carte.
- **Recharger** pour recharger le réservoir d'un robot. *Attributs :* un robot et une carte

execute() peut être assez simple et rapide, comme pour **Etat**, où il suffit d'appeler une fonction du robot, mais aussi plus longue et complexe comme pour **Eteindre**. Nous avons ensuite ajouté quelques fonctions au simulateur pour qu'il puisse gérer les événements tel que :

- *ajouteEvenement* qui ajoute un événement, selon sa date, dans la liste des événements à faire.
- *next* qui incrémente la date du simulateur et exécute les événements de la liste qui corresponde à cette nouvelle date. Un pas correspond à 1 seconde.
- *restart* qui permet de redémarrer le simulateur

Test des événements Pour les tests des événements, nous avons directement essayé avec notre **TestAffichage.java** et la stratégie élémentaire du chef pompier. Nous avons donc pu tester que le chef ne prenait pas de décision impossible et que le simulateur mettait bien à jour la fenêtre graphique lors d'événement.

3 Calculs de plus courts chemins

Nous avons décidé d'implémenter une nouvelle classe, **Calcul Chemin**, pour trouver le chemin le plus court. Il a fallu aussi créer une classe **Chemin** et **Sommet**.
La classe **Sommet** correspond à une case sur la carte mais avec plus d'attributs utiles tel que la liste de ses voisins et de leur poids, un booléen pour le marqué....
La classe **Chemin** permet d'associer un robot à une liste de sommets, d'avoir un tableau des temps de passage pour tous les points, d'ajouter un sommet à la liste,etc. Elle nous permet de mémoriser l'ensemble du parcours à faire par le robot.

Enfin, la classe **Calcul Chemin** implémente les fonctions nécessaires pour le calcul du chemin le plus court. Nous avons repris le principe de l'algorithme de Dijkstra pour le calcul de plus court chemin :

- *calculPoidsArc* retourne le poids pour passer d'une case à une case voisine.
- *sommetMin* retourne le sommet le plus facilement accessible.
- *dijksra* permet de calculer le plus court chemin entre une destination et une source.

Ainsi

4 Résolution du problème

Nous avons suivi le sujet et donc nous avons créer une nouvelle classe **ChefPompier**, mais aussi le type **EtatRobot** qui prend les valeurs DEPLACEMENT, ETEINDRE, LIBRE,... qui permet de connaitre l'état de chaque robot sur le terrain.
Le Chef pompier possède une liste des incendies non affectés, des points d'eau et des robots. Elles seront générées lors de la création du Chef pompier par une parcours de la carte ou des robots.
Nous avons implémenté la stratégie élémentaire, qui se comporte comme décrite dans le sujet :

1. Parcours de la liste des incendies non affectés, prend le premier sur la liste.
2. Parcours de la liste des robots. On regarde d'abord le type (DRONE,ROUES,...) pour définir la destination pour ce robot (si c'est un drone -> sur l'incendie, sinon sur une case à côté).
3. On vérifie qu'il existe bien un chemin pour ce robot et qu'il n'est pas occupé. Sinon on passe au robot suivant sur la liste.
4. On affecte le chemin au robot concerné, on calcul le temps de trajet pour arriver à destination et on retire l'incendie de la liste des incendies non affectés.
5. On recommence tant que la liste des incendies non affectés est vide ou que tous les robots sont occupés.