Rapport TPL de Programmation Orientée Objet

Équipe n°77

1. Présentation du projet

Dans le cadre de ce projet, nous avons développé une application en **Java** simulant une équipe de robots pompiers opérant de manière autonome dans un environnement naturel. L'objectif de ces robots était d'éteindre des incendies sur une carte.

Ce TP nous a permis d'explorer et de mettre en œuvre des concepts fondamentaux de la programmation orientée objet, tels que l'*encapsulation*, l'*héritage*, la *délégation* et l'utilisation avancée des collections en **Java**.

Voici un résumé des étapes principales :

- Créations des différents objets après lecture de la carte
- Contrôle des déplacements des robots sur une carte.
- Implémentation d'algorithmes de recherche de chemin.
- Conception de stratégies d'organisation pour optimiser les interventions.

Concernant l'organisation, nous avons utilisé un dépôt Git pour centraliser les travaux et le logiciel *Trello* pour faciliter la gestion du projet.

2. Création des différentes classes

(a) Classe abstraite `Robot`

Cette classe regroupe les points communs entre les différents robots (*vitesse*, *temps d'intervention*, *volume*, etc.) ainsi que les méthodes communes d'accès et de modification. Tous les types de robots héritent de la classe robot et redéfinissent les méthodes abstraites et les constructeurs selon leur particularité.

(b) Classes `Carte`, `Incendie`, et `Case`

Ces classes sont définies selon les spécifications du sujet avec les méthodes de set et get qui vont avec.

(c) Classe `Simulateur`

Elle initialise les éléments graphiques, les événements et les données de simulation. Elle fournit des méthodes pour ajouter des événements, copier des objets et réafficher les éléments graphiques. Elle gère aussi les méthodes `next` et `restart`.

(d) Classe abstraite `Événement`

Cette classe représente un événement dans la simulation. Elle définit les attributs communs et les méthodes pour gérer les événements tels que la date d'exécution et la méthode abstraite `execute()` qui correspond aux actions spécifiques de l'événement. Elle est conçue pour être étendue par des classes spécifiques (EvenementChef, EvenementExtinction, EvenementRemplissage, EvenementDeplacement).

(e) Classe `DonneesSimulation`

Elle regroupe toutes les données (`robots`, `incendies`, et `carte`). Les robots sont stockés dans un tableau et les incendies dans un `HashSet` pour une gestion rapide.

(f) Classe `CalculateurChemin`

Elle calcule le chemin optimal entre deux points, en utilisant des algorithmes comme *Dijkstra* et *BFS*.

(g) Classe abstraite `Chef`

Cette classe représente un type de chef pour la simulation. Comme Événement, elle définit les attributs communs et les méthodes qui devront être redéfinies par ses filles(ChefSimple, ChefEvolue, ChefCollaboratif).

3. Gestion des événements

(a) Création des événements

Les événements sont représentés par des classes qui héritent de la classe abstraite Evenement. Chaque type d'événement a sa propre classe, par exemple :

- **EvenementDeplacement** : représente un déplacement d'un robot.
- **EvenementExtinction** : représente l'extinction d'un incendie par un robot.
- **EvenementRemplissage** : représente le remplissage du réservoir d'un robot.
- **EvenementChef** : represente l'execution d'instruction donnees par un chef pompier.

(b) Ajout des évènements

Les événements sont ajoutés au simulateur à l'aide des méthodes `ajouteEvenement()` et `remplaceEvenement()` de la classe Simulateur. Ces méthodes ajoutent les événements à une file de priorité (PriorityQueue) qui trie les événements en fonction de leur date d'exécution(appelons cette file file1). La différence entre ces 2 méthodes est "assez" importante car elle influe sur la possibilité de recommencer une simulation (bouton restart). En effet, `ajouteEvenement` ne doit être utilisé que dans les classe Test pour spécifier au simulateur quels sont les evenement principaux(ceux qu'on veut réellement effectué et pas des événement appelé par d' autre événement afin de continuer les tâches). Ces événements sont en fait copiés dans une autre file de priorité(notons file2) afin de pouvoir effectuer un éventuel restart en plus d'être mis dans file1 pour la simulation actuelle. Quant a `remplaceEvenement`, elle ajoute les événements(ceux appelés par d'autres événements) dans file1. Ceux ci ne sont pas copier dans la file de priorité file2 car jugés non indispensable vu ques les événements principaux peuvent les rappelles lors de la prochaine simulation.

(c) Exécution des événements

Le simulateur exécute les événements en fonction de leur date d'exécution. La méthode **next** de la classe Simulateur incrémente la date de la simulation et exécute tous les événements dont la date est inférieure ou égale à la date actuelle de la simulation.

4. Algorithmes

(a) Plus court chemin

i. Algorithme de Dijkstra

Cet algorithme calcule le chemin le plus court entre un robot et une destination. Les étapes sont :

- **Initialisation** : Créez une carte des temps (times) pour stocker le temps de déplacement minimal vers chaque case, initialisée à l'infini pour toutes les cases sauf la source, qui est initialisée à 0. Créez une carte des prédécesseurs (predecesseurs) pour stocker le chemin. Utilisez une file de priorité (PriorityQueue) pour gérer les cases à explorer, triée par le temps de déplacement.
- **Exploration** : Tant que la file de priorité n'est pas vide, retirez la case avec le temps de déplacement minimal (u). Pour chaque voisin de u, calculez le temps de déplacement alternatif (alt) en ajoutant le temps de déplacement de u à ce voisin. Si alt est inférieur au temps de déplacement actuel vers ce voisin, mettez à jour le temps de déplacement et le prédécesseur de ce voisin, puis ajoutez le voisin à la file de priorité.

- **Construction du chemin** : Une fois l'exploration terminée, construisez le chemin en remontant les prédécesseurs depuis la destination jusqu'à la source.

ii. Algorithme BFS

- **Initialisation** : Créez une carte des prédécesseurs (predecesseurs) pour stocker le chemin. Utilisez une file (Queue) pour gérer les cases à explorer. Utilisez

un ensemble (Set) pour suivre les cases visitées.

- **Exploration** : Tant que la file n'est pas vide, retirez la case en tête de file (u). Pour chaque voisin de u, si le voisin n'a pas été visité et que le robot peut le traverser ou qu'il s'agit d'une case d'eau, ajoutez le voisin à la file, marquez-le comme visité, et mettez à jour son prédécesseur. Si un voisin est une case d'eau, construisez le chemin en remontant les prédécesseurs depuis cette case jusqu'à la source.

(b) Stratégies de gestion par les chefs

La classe abstraite Chef représente un chef pompier qui gère les robots et les incendies. Elle définit les attributs communs et la méthode abstraite `gererInterventions` que les classes dérivées doivent implémenter. Chaque type de chef par la suite hérite le classe Chef. Afin que chaque chef puisse être rappelé par la suite après chaque intervalle de temps, on l'associe à un événement dans le simulateur. A chaque exécution de l'événementChef, le chef se rappelle lui même si les incendies ne sont pas encore éteintes.

- **ChefSimple** : implémente directement la stratégie du sujet en utilisant les classes définies. Il vérifie bien qu'un robot est disponible avant de lui attribuer un incendie (Ceci est accompli grâce à la `dataDeDisponiblité` qui est présente dans chaque robot).
- **ChefEvolue** : compare le temps de chaque chemin pour trouver l'incendie le plus proche. Et en plus quand les robots n'ont plus d'eau dans leur réservoir, un événement de remplissage est planifié.
- **ChefCollaboratif** : recherche pour chaque robot l'incendie le plus proche même si ce dernier a été déjà attribué à un autre robot. En effet, il autorise la collaboration entre les robots pour l'extinction.

5. Tests

- **Test lecteurs de données et évenemrnts :** On a testé la lecture de données avec la carte du sujet pour vérifier que l'instance `Donnéessimulation` se créait et qu'on avait bien les différents objets. De plus on a aussi testé les différents scénarios donnés dans le sujet. Le resultat était plûtot positif.

- **Test plus court chemin :** Le programme utilise le `CalculateurChemin` pour déterminer le chemin optimal qu'un robot doit suivre pour atteindre la position d'un incendie donné. Le chemin calculé est converti en une série d'événements, ajoutés au simulateur, pour déplacer le robot jusqu'à l'incendie.

- **Test chef :** Un objet Chef (Simple, Evolue, Collaboratif) est initialisé avec les données de simulation (carte, robots, incendies) et associé à un événement de type `EvenementChef`. Ce chef gère les interventions des robots pour éteindre les incendies de manière spécifique. Le code a bien fonctionné.