Documentation Technique

Introduction

Ce document présente les choix de conception que nous avons faits pour le projet de simulation d'une équipe de robots pompiers dans un environnement naturel. Le but est d'expliquer les décisions architecturales, d'illustrer l'utilisation des classes et méthodes choisies, et de fournir un aperçu des tests effectués ainsi que des principaux résultats obtenus.

Choix de Conception

Architecture Générale

L'application est structurée selon une architecture orientée objet, divisée en plusieurs packages pour assurer une meilleure organisation et modularité. Les principaux packages incluent :

- **robots** : Contient les classes représentant les différents types de robots (ex. : `Roue`, `Pattes`).
- **carte** : Définit la structure de la carte et des terrains (ex. : `Case`, `Carte`).
- **simulation** : Gère les événements et la logique de simulation (ex. : `Evenement`, `Simulateur`, `ChefRobotPompier`).
- **plus_court_chemin** : Gère le calcul du plus court chemin (ex. : `aetoile`).
- **io** : Gère la lecture de données.

Les différentes textures graphiques se trouvent dans le dossier */src/ressources*.

Conception Orientée Objet

- **Encapsulation** : Les attributs des classes sont protégés pour garantir l'intégrité des données, avec des méthodes d'accès pour gérer les modifications.
- **Héritage et Polymorphisme** : La classe abstraite `Robot` est utilisée comme base pour les différentes implémentations de robots (ex. : `Roue`, `Pattes`). Cela permet de simplifier le code et de gérer les robots de manière générique dans le simulateur.
- **Redéfinition** : Les méthodes spécifiques, comme `deplacer` et `eteindre`, sont redéfinies dans chaque sous-classe pour gérer les comportements propres à chaque type de robot.

Utilisation des Classes et Méthodes

Package robots

Dans ce package, nous avons choisi de faire de la classe `Robot` une classe abstraite. En effet, chaque robot a forcément un type spécifique, mais tous les types partagent des méthodes similaires. De plus, la superclasse `Robot` est très utile lors de l'utilisation de l'algorithme A*. Cet algorithme prend en argument une carte, un robot, et une destination, rendant ainsi le polymorphisme nécessaire. Une énumération permet de lister tous les types de robots.

Package plus_court_chemin

Ce package ne contient qu'une seule classe, *aetoile*. Nous avons quand même décidé d'utiliser un package par souci de clarté. Cela facilite aussi l'ajout d'un nouvel algorithme de plus court chemin (par exemple, Dijkstra) sans trop de modifications.

Package simulation

Ce package regroupe toutes les classes permettant de créer des événements, de dessiner la carte et les cases, et de gérer l'algorithme de résolution du problème.

Package carte

Ce package contient toutes les classes en rapport avec la carte. Nous avons notamment deux énumérations qui nous permettent de lister les différentes directions et natures de terrain.

Description des Tests et des Algorithmes

Explications des Algorithmes

Pour notre projet, nous utilisons deux algorithmes : A* pour la recherche de plus court chemin, et *ChefRobotPompier* pour la résolution du problème.

L'algorithme *ChefRobotPompier* recherche les robots disponibles. S'il y en a, il sélectionne celui qui se rendra le plus rapidement au premier incendie de la liste. De plus, quand les réservoirs des robots sont vides, l'algorithme identifie la case d'eau la plus proche pour qu'ils puissent se remplir.

Stratégie de Test

Nous avons choisi de réaliser une batterie de tests au fur et à mesure de l'avancement du projet. Pour exécuter les différents tests, toutes les commandes sont répertoriées dans un Makefile. Pour les utiliser, veuillez vous référer à la documentation utilisateur jointe au projet (*documentation_user.pdf*).

Scénarios Testés

Nous avons testé différentes parties du code. Pour vérifier la lecture des documents, le test *Invader* a été fourni. Nous avons ensuite testé notre gestionnaire d'événements avec *testEvenement*, qui simule le scénario de la montée du drone. *testChemin* effectue un test rapide sur le fonctionnement de notre algorithme de plus court chemin. Enfin, dans *Main*, nous générons et résolvons le problème complet en utilisant notre algorithme *ChefRobotPompier*.

Résultats Obtenus

Les résultats obtenus sont les suivants : - **Invader** : Les tests confirment que la lecture des données fonctionne correctement. - **testEvenement** : La gestion des événements fonctionne, et une erreur est générée lorsqu'un robot tente de sortir de la carte. - **Main** : Lors de l'exécution, les robots parviennent à éteindre les incendies. Cependant, l'algorithme *ChefRobotPompier* n'est pas optimisé : les robots ne se dirigent pas toujours vers l'incendie le plus proche, et lorsqu'ils cherchent une source d'eau, ils se remplissent automatiquement à la case (2,2) si aucune source n'est accessible. Une amélioration de la gestion de ce cas serait nécessaire. Comme précisé dans la documentation utilisateur, il faut mettre un temsp de 200ms entre 2 affichages car sinon nous avons le drone qui se téléporte parfois.

Conclusion

En conclusion, nous sommes satisfaits du projet. Nous avons réussi à implémenter une solution fonctionnelle avec un affichage graphique simple mais esthétique. Cependant, avec plus de temps, nous pourrions optimiser l'algorithme *ChefRobotPompier*.

Pour plus de détails techniques, utilisez la commande suivante pour générer la Javadoc du projet :

```
make javadoc
```