Ensimag 2A POO - TP 2022/23

Réalisé par Paul Vernin, Marc Félix-Henry et Paul Bonmariage.

Installation du jeu

Pour lancer le jeu, il faut soit : - utilise un formidable IDE tel que Intellij IDEA :) (la configuration pour run est déjà faite, elle est dans .idea/runConfigurations/main_Main.xml) - utilise le Makefile :(

Input

La commande `java Main` prend 2 arguments : - argument 1 : le chemin vers la carte à charger - argument 2 : la stratégie à utiliser (`1` pour la stratégie 1, `2` pour la stratégie 2) - argument 3 : change le texture pack (`FORTNITE` active le texture pack Fortnite, `DEFAULT` active le texture pack par défaut) ### Output
Tous les `System.out.println` sont redirigés vers un fichier `logs/out.txt`.
Tous les `System.err.println` sont redirigés vers un fichier `logs/err.txt`.

Compte rendu

Choix d'implémentation

- Dans le sujet (page 3), il est expliqué que les robots éteignent le feu **sur lequel ils se trouvent**. Donc les robots à roues sont inutiles sur les maps où il n'y a du feu que sur des arbres.
- Les robots ne peuvent pas être assignés au même feu.

Stratégies

Stratégie 1 Notre stratégie 1 correspond à la stratégie 2 de l'énnoncé avec des optimisations décrites dans la suite du compte rendu.

Stratégie 2 La stratégie 2 implémente le multi-threading sur le calcul du chemin d'un robot vers un case d'eau. On observe clairement une différence de vitesse entre la stratégie 1 et la stratégie 2 sur la dernière carte qui possède beaucoup de point d'eau lorsqu'un robot veut aller se remplir.

Modèle MVC

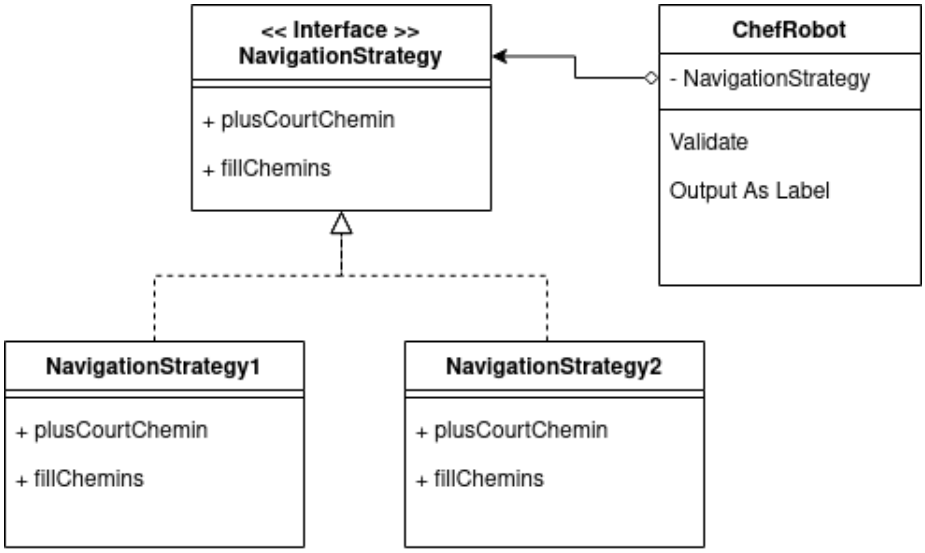
On s'est inspiré du très connu modèle MVC (Model View Controller) pour structurer notre projet. Le modèle MVC est un modèle de conception logiciel qui sépare les données d'une application, la logique métier et la présentation. Il est composé de trois éléments : le modèle, la vue et le contrôleur.
Ces trois partie (Vue, Modèle et Contrôleur) sont séparées dans des packages différents dans le package main.

Chef robot (Design Pattern Singleton)

Vu qu'il n'existe qu'un seul chef robot dans l'application nous avons utilisé le design pattern Singleton pour s'assurer qu'il n'y ait qu'une seule instance de la classe ChefRobot.
(En apprendre plus sur le design pattern Singleton : https://refactoring.guru/design-patterns/singleton)

Gestion des stratégies (Design Pattern Strategy)

En s'inspirant du design pattern Strategy nous avons implémenté deux stratégies pour calculer le plus court chemin.
La classe ChefRobot avait le rôle du Navigateur, c'est lui qui possède la stratégie dans ses attributs.



(En apprendre plus sur le design pattern Strategy : https://refactoring.guru/design-patterns/strategy)

Gestion des robots par le Chef robot (Design Pattern Observer)

On voulait éviter de surcharger d'appels les classes robots et incendies.
C'est-à-dire que dans une implémentation simpliste, le chef robot appellerait toutes les classes robots et incendies pour leur demander "Est ce que vous êtes disponibles ?" à chaque fois qu'on appelle `next()`
Donc si un robot était occupé pour une suite d'événements qui durerait 300 tours, le chef robot appellerait 300 fois la classe robot pour rien.

Pour palier à ce problème nous avons implémenté le design pattern Observer avec quelques modifications pour que cela colle à nos besoins.

(En apprendre plus sur le design pattern Observer : https://refactoring.guru/design-patterns/observer)

Une fois que le chef robot a fini d'assigner tous les robots à des incendies, il va se mettre en "off" et c'est un des robots qui va réveiller le chef robot pour qu'il re-assigne des robots non occupés à aller remplir leur réservoire ou à des incendies non occupé.
Donc à la place d'avoir une fonction next qui fait :

```
next():
    Pour chaque robot si il est disponible:
        Pour chaque incendie s'il n'est pas pris en charge:
            ...calculer chemins, etc...
```

On aura :

```
next():
    Si chef robot est "on":
        Pour chaque robot si il est disponible:
            Pour chaque incendie s'il n'est pas pris en charge:
                ...calculer chemins, etc...
```

Cela réduit la complexité de la méthode `next()` qui est beaucoup appelée par le simulateur.

Plus court chemin

On a implémenté l'algorithme A* pour aller du robot à un incendie ou à un endroit pour remplir son réservoir (un peu comme tout le monde).

Linting

Nous avons utilisé le linter SonarLint pour vérifier la qualité du code. Cela nous a permi de structurer notre code pour suivre les conventions de programmation en Java comme par exemple le nommage des variables, des fonctions, des classes, etc.
Et surtout faire en sorte que chaque membre de l'équipe suive les mêmes conventions de programmation pour avoir un code plus lisible et plus facile à maintenir.

JavaDoc

Au cas où, on a déjà généré la JavaDoc, elle est dans le dossier `doc/`.

Testing

Pour tester la partie modèle de l'application nous avons utilisé le framework JUnit afin de réaliser principalement des tests unitaires.
(Tester avec Intellij, c'est plus facile)