APOO

Simulation d'une équipe de robots pompiers

Koenig Romain
Leymarie Valmon
Van Eenaeme Bastien
Gaudron Mathieu
Domec Mehdi
De Vos Tim
Jodin Romaric
Carren Nicolas

2 mars 2013

Chapitre 1

Présentation du projet

Ce projet a pour but la réalisation d'un logiciel de gestion d'une équipe de robots pompiers codé en Java pour une entreprise inconnue. Nous aurons à notre disposition 4 sortes de robots (robot volant, robot chenille , robot à roues et robot à pattes) ayant tous des attributs différents, notamment au niveau de leur vitesse (qui dépendra du relief du terrain) et de la présence d'un réservoir d'eau (le robot à pattes fonctionnant au gaz). Notre programme devra être en mesure de lire la carte fournie par l'entreprise (pour de référencer la position et l'intensité des feux) afin de faire évoluer de maniére optimale et autonome notre équipe de robots pompiers.

Chapitre 2

Le main

2.1 Les différentes parties

En Java, un programme fonctionne grâce à l'imbrication de plusieurs classes. Le main nous sert de point d'entrée dans notre programme de simulation. La solution retenue par le groupe concernant le main, a été de le garder simple et épuré, pour laisser la gestion des différents entités de notre programme à d'autres classes plus spécialisées (notamment le simulateur et le manager). Le schéma d'imbrication des différentes classes est le suivant :

C'est dans le main que vont être instanciés les objets nécessaires au fonctionnement du programme. Pour ce faire, on importe différentes classes : simulator.* qui permettront d'instancier le simulateur (cf paragraphe suivant pour le fonctionnement plus précis du simulateur) ; carte.* qui regroupent les classes permettant la gestion de la carte ; l'interface graphique, simple, qui est celle fournie dans le projet. Une carte est générée à partir d'un fichier contenant les informations fournies par l'entreprise pour la simulation des robots pompiers. On y lit la taille de la map (un rectangle de X*Y cases) ainsi que les différents types de terrains associés à chacune des cases de la carte, mais également le nombre de robots présents sur celle-ci, et bien sûr les incendies et leurs intensités que les robots devront éteindre. Le main instancie donc le simulateur et le manager qui fonctionneront de manière autonome par la suite. Il génère également une carte à partir d'un nom passé en variable correspondant au fichier de la carte sélectionnée. Il lance ensuite le SimulationPlayer qui nous permettra de controler le temps discret de la simulation, puis il démarre l'affichage de la map grâce à l'interface graphique.

Chapitre 3

Le manager

3.1 Choix de la solution

Le Manager est une entité qui va réfléchir à une répartition optimale de nos robots pompier sur la carte. Pour cela, le manager considère la liste de tous les robots disponibles, et crée une liste de tous les feux de la map. Pour chaque feu de la liste n'ayant pas encore été traité par le manager, celui-ci calcule le temps mis par chaque robot disponible pour aller éteindre ce feu. Ensuite, le manager élit le plus rapide d'entre eux et crée l'événement de déplacement, qu'il envoie dans la file d'événement du simulateur. Si un robot se trouve à porté d'un feu, le manager envoie l'événement d'extinction au simulateur et si un robot a déja été retenu pour éteindre un feu ou aller se remplir, le manager le fait simlplement avancer d'une case. Enfin, si un robot n'a plus d'eau dans son réservoir, soit il est sur une case adjacente à un point d'eau, dans quel cas on lui demande de se remplir, soit le manager calcule le chemin au lac le plus proche, et crée un événement afin qu'il s'y rende.

3.2 Les méthodes

Le manager posséde plusieurs méthodes pour assurer une gestion autonome des robots.

listeFeu liste tous les feux de la map.

caseEnCours est la méthode qui permet de savoir si un robot se dirige déjà vers la case en feu.

traiteRobot est notre méthode principale. Elle permet d'ajouter à la file d'événement du simulateur le déplacement des robots, leur remplissage et l'extinction des feux.

caseEau permet de savoir si on est sur une case adjacente à une case de remplissage.

peutRemplirouEteindre est une méthode qui permet de vérifier que le robot est a côté d'un feu ou d'un point d'eau. Si le bouléen "eteindre" donné en paramétre est à "vrai", c'est qu'o, cherche un point d'eau. Sinon on cherche un feu.

lacPlusProche permet de calculer et de retourner le chemin le plus court au lac le plus proche en utilisant notre algorithme de recherche de chemin le plus court (Djkstra).

3.3 Les tests

Pour tester notre Manager, nous avons effectué une batterie de tests dans le fichier TestManager.java. Avant tout, nous avons généré notre propre map 5x5 contenant 3 feux. Nous avons testé la fonction de recherche et de listing des cases en feux "listeFeu". Nous avons ensuite crée une liste de 3 robots disponibles placés à des endroits différents de la carte. Enfin, nous avons testé la fonction traiteAllRobots dans différents cas (pas de feu, un seul robot disponible, un seul robotcapable d'éteindre le feu et enfin traiteAllRobots avec tous les robots) et avons vérifié son bon fonctionnement.

Chapitre 4

Le simulateur

4.1 La liste d'évènement

Le principe du simulateur est d'ordonner les différents évènements qui vont s'éxécuter sur la carte.
Pour ce faire, il faut dans un premier temps une structure liste. Les packages Java comprennent ces structures élémentaires, c'est pourquoi il suffit d'importer ces packages et d'utiliser ces listes (Voir LinkedList donnée sur le site d'Oracle). Le temps est discret, choix imposé par l'énoncé, ainsi chaque évènement se verra attribué une date, ainsi qu'un temps d'éxecution. On insère chaque évenement dans la liste d'évenement et ce dernier est classé dans cette liste en fonction de sa date, les évènements avec les dates les plus tôt en premièrs et les évenements avec les dates les plus tard en derniers. Sachant que la date alias 'currentDate' est tenu à jour après chaque éxecution d'évènement grâce à la date actuelle de ce dernier plus la date d'éxecution de l'évènement.

4.2 L'éxecution

En ce qui concerne l'éxécution, c'est simple à chaque fois qu'on appel l'éxecution du simulateur (c'est à dire l'éxecution des différents évènemements contenu dans la liste du simulateur), on prend le première élement de la liste, on l'éxecute, puis on prend le suivant et ainsi de suite, on s'assure ainsi d'éxecuter tous les évènements dans le bon ordre. Le simulateur se contente donc d'ordonner la liste d'évènements et de l'éxécuter, c'est au manager d'insérer les bon élements avec la bonne date dans la liste du simulateur.

Chapitre 5

Les différentes classes

5.1 Les classes abstraites

Deux classes abstraites Pour pouvoir implémenter tous les différents types de robot dont nous avons besoins, nous nous sommes servis de deux classes abstraites. Une classe robot qui englobe toutes les classes de robot et qui définie les attributs, accesseurs, mutateurs, et méthodes abstraites communs à tous les robots. Les attributs directionActu et cpc ne seront pas utilisés dans les méthodes implémentées dans les classe héritières de la classe Robot. Mais ces attributs doivent êtres gardés en mémoire car ils sont uniques pour chaque robot et des opérations sont faites dessus. Une classe RobotRes qui englobe tous les robots possédant un réservoir et donc qui exclue les robots à pattes puisque ces robots utilisent du gaz et ne possèdent pas de réservoir. Cette classe RobotRes hérite de la classe abstraite Robot décrite ci-dessus. Elle définie les attributs, accesseurs, mutateur (seulement sur l'attribut reservoir) et méthodes en rapport avec l'utilisation du réservoir par les robots concernés. Ces méthodes sont : -remplir qui génère un événement de début de remplissage et un de fin de remplissage en fonction du temps mit par le robot pour se remplir (spécifique à chaque type de robot). -tmpEteindre qui renvoie la durée d'une salve du robot (spécifique à chaque type de robot) si l'intensité du feu qu'il essaye d'éteindre n'est pas nulle et que le réservoir du robot n'est pas vide. Dans le cas contraire des exceptions seront levées. -eteindre qui va diminuer l'intensité du feu visé d'une valeur égale à son efficacité pour chacune de ces salves et met à jour la contenance du réservoir. Si le robot ne possède plus assez d'eau pour réaliser une salve, l'intensité du feu diminue de 1 et le réservoir est vidé.

5.2 Les classes de robots

RobotPattes La classe RobotPattes hérite de la classe Robot. Elle a comme attributs (déjà implantés dans sa classe mère) : la position du robot, sa vitesse, son efficacité et sa disponibilité. Elle implémente quatre méthodes (Les méthodes

tmpEteindre et eteindre sont différentes de celles implémentées dans la classe abstraite RobotRes du fait de l'absence de réservoir pour cette classe) : -toString qui fait un résumé des caractéristiques du robot au moment de l'appel de cette méthode. -tempsDeplacement qui calcul le temps de parcours entre deux cases. Pour un robot à pattes, sa vitesse est constante sur tous les terrains et pour n'importe quelle pente. Si une pente est trop raide, le robot patte ne pourra pas avancé et une exception sera levée. Un robot patte ne peut pas traverser une case lac. -tmpEteindre qui renvoie la durée d'une salve du robot patte (20 secondes) si l'intensité du feu qu'il essaye d'éteindre n'est pas nulle. Dans le cas contraire une exception sera levée. -eteindre qui va diminuer l'intensité du feu visé.

RobotVolant La classe RobotVolant hérite de la classe RobotRes, elle même héritée de la classe Robot. Elle a comme attributs (déjà implantés dans ses classes mère et grand-mère) : la position du robot, sa vitesse, son efficacité, sa disponibilité et tous les attributs liés à la possession d'un réservoir qui sont : la capacité totale du réservoir (tailleReservoir), la contenance actuelle du réservoir (reservoir), le temps mit par le robot pour remplir son réservoir en entier (tmpRemplissage), le temps mit par le robot pour se vider d'une salve (tmpVidage), et le volume d'une salve (salve). La classe RobotVolant se sert des mêmes méthodes toString et tmpDeplacement que la classe robotPattes décrites ci-dessus à ceci près qu'un robot volant possède plus d'attributs à décrire qu'un robot à pattes. De plus, comme l'indique son nom, le robot vol et n'a donc pas de contrainte sur l'inclinaison ou la nature du terrain dans la fonction temDeplacement. Toutes les autres méthodes de cette classe sont implémentées dans sa classe mère et ont déjà été décrites (cf Classe abstraites/Classe RobotRes).

RobotChenille La classe RobotChenille hérite de la classe RobotRes, elle même héritée de la classe Robot. La classe RobotChenilles possède les mêmes attributs que la classe RobotVolant décrite ci-dessus, seules les valeurs de ces attributs changent. Les méthodes sont également identiques à l'exception de tmpDeplacement. La méthode tmpDeplacement implémentée dans la classe RobotChenille prend en compte l'inclinaison de la pente et lève une exception si celle-ci est trop importante. Une exception est également levée si le robot tente de traverser un terrain d'eau. Si le robot traverse un terrain répertorié comme forêt, sa vitesse est diminuée et donc son temps de déplacement augmenté.

RobotRoues La classe RobotRoues hérite de la classe RobotRes, elle même héritée de la classe Robot. Cette classe possède la même implémentation que la classe RobotChenille.