PROJET PROGRAMMATION ORIENTEE OBJET 2016 : ROBOTS

Vincent Kylian

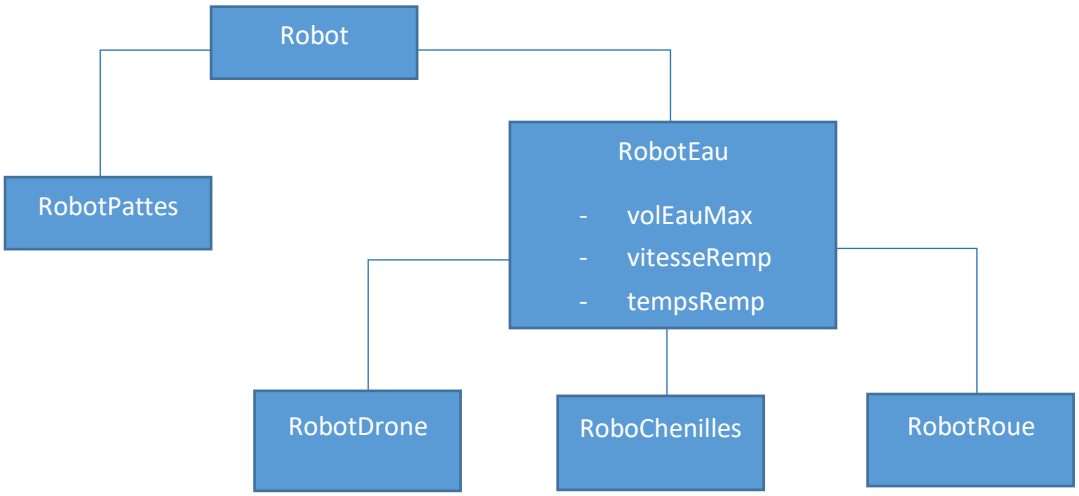
Buthod-Garçon Tony

Bouton Paul

I-Choix des structures de données

Hiérarchie des robots

Lors de ce projet, nous avons réfléchi notre choix concernant la structure de données la plus adaptée à notre problème. Dans un premier temps, nous avons décidé de ne pas faire hériter directement tous les robots de la classe Robot mais de créer une classe abstraite intermédiaire :



Cette structure de données nous a semblé la plus logique pour plusieurs raisons :

- Le robot à patte n'ayant pas à se remplir, il est cohérent qu'il ne partage pas les attributs volEauMax, vitesseRemp et tempsRemp.
- La création de la classe RobotEau nous permet de factoriser notre code en utilisant le principe d'héritage pour RobotChenilles, RobotDrone et RobotRoue.

Néanmoins, nous avons rencontré plusieurs problèmes lors de l'implémentation des évènements : Pour certains évènements, l'utilisation des attributs de RobotEau était nécessaire. Cependant, un évènement étant associé à un Robot et non pas à un RobotEau, nous devions recourir à des casts.

C'est pour cette raison que nous avons décidé de supprimer la classe RobotEau et de revenir à une structure de donnée plus classique en remontant ses attributs (ceux-ci ont des valeurs incohérentes mais connues pour le RobotPattes (tempsRemp = -1 par exemple).

Liste d'évènements

Pour l'implémentation des évènements, notre choix s'est porté sur l'utilisation de listes chainées (LinkedList en java). Chaque évènement a pour attribut sa date d'exécution et le robot auquel il est associé. Lors la simulation, les évènements sont chainés par date d'exécution croissante puis exécutés un par un selon les modalités suivantes :

- Si l'exécution d'un évènement est impossible (on demande à un robot patte de se remplir par exemple), alors une exception est levée et on passe à l'évènement suivant sans exécuter l'évènement courant ayant levé l'exception.
- Si à la date d'exécution de l'évènement, le robot est occupé (Robot.dateOccupée > évènement.date), alors l'exécution de cet évènement est ignorée et on exécute l'évènement suivant.

ChefRobot et Simulateur

La classe ChefRobot, nous permet de mettre en place différentes stratégies pour éteindre les incendies. Lors de notre projet, nous avons seulement implémenté la deuxième stratégie (Stratégie plus évoluée). Ainsi, à chaque appel de next(), le ChefRobot parcourt la liste des incendies et leur affecte un robot libre s'il y en a un de disponible (le plus proche). La classe ChefRobot possède donc un attribut robotsLibres, une liste chainée de robots, qui est gérée comme suit :

- Au début, tous les robots sont libres.
- Lorsqu'un robot est affecté à un incendie, il est retiré de la liste des robots libres par le chef.
- Lorsqu'un robot a fini d'éteindre un incendie, il se rend automatiquement au point d'eau le plus proche et envoie un message au chef une fois son remplissage terminé.
- A la réception du message, le chef ajoute le robot à la liste des robots libres et lui affectera un incendie lors du prochain appel de next().

En ce qui concerne le simulateur, ce dernier possède un attribut chefRobot qui vaut initialement null et qui n'est pas mis à jour par le constructeur de Simulateur. Ainsi, l'utilisateur peut choisir d'effectuer une simulation en utilisant une stratégie (il crée un ChefRobot et lui donne en paramètre le simulateur) ou bien il peut décider d'ajouter lui-même ses évènements (dans ce cas l'attribut Chef de simulateur reste à null).

II-Implémentation des différents Tests

1-Tests

Pour tous nos évènements, nous avons réalisé des tests afin de vérifier que les comportements des robots étaient bien ceux attendus. Cela nous a notamment permis de vérifier le bon fonctionnement de nos exceptions (Lors d'un déplacement incohérent par exemple).

2-dateOccupée

Il nous a semblé utile ici de détailler l'utilisation de l'attribut dateOccupée présent dans la classe robot. Ce dernier a été mis en place pour le besoin des différents tests réalisés lors de notre projet.

Lorsqu'un robot éteint un incendie ou bien lorsqu'il se rempli, il ne doit pas pouvoir se déplacer. Par exemple, lors du remplissage, le robot ajoute lui-même un évènement à la liste d'exécution : l'évènement ReservoirRempli. Ce dernier est ajouté à une date calculée en fonction du temps de remplissage du robot et passe le volume d'eau du robot à son maximum lors de son exécution. Ainsi, si le robot ajoute cet évènement (on lui a demandé de se remplir) puis se déplace pendant qu'il se rempli, il sera plein alors même qu'il n'a pas fini son remplissage (l'évènement ReservoirRempli sera exécuté puisque qu'il aura été chainé). Cela n'étant pas cohérent, le robot refusera tous les ordres lorsqu'il se rempli, c'est le rôle de l'attribut dateOccupée.

NB : Ce choix d'implémentation n'est pas idéal pour des stratégies d'intervention évoluées (si le feu restant est de faible intensité, pas besoin de remplir le robot au maximum). Cependant, il ne pose pas de problème pour l'implémentation des deux stratégies proposées.

III-Résultats obtenus

A l'issue de ce projet, nous avons obtenu une simulation fonctionnelle. Nos robots, sous les ordres d'un chef, sont capables d'éteindre tous les feux présents sur la carte. De plus, lors de l'exécution de nos différents tests, nous n'avons pas remarqué d'écarts quant aux comportements des robots par rapport aux stratégies définies.