Simulation d'une équipe de robots pompiers

Thibaud Backenstrass, Robin Moussu, Amanda Sambath

19 novembre 2014

Introduction

L'objectif de ce TP est la mise en application des différentes notions vues en cours de Programmation Orientée Objets en Java. Il s'agit de simuler une équipe de robots-pompiers évoluant de manière complètement autonome dans un environnement contraint, ayant pour mission d'aller éteindre des incendies, idéalement le plus rapidement possible.

Les premières parties du TP étaient très guidées, ce qui fournissait une « base » de code sur laquelle travailler. Dans ce document, nous allons présenter les différents choix de conception qui ont été faits tout au long de ce TP en justifiant l'architecture de notre programme (organisation des classes) et en expliquant succinctement le principe des algorithmes utilisés. Enfin, nous terminerons par un « état des lieux » de notre code pour expliciter les tests effectués et les fonctionnalités implantées.

1 Choix de conception et architecture des classes

Nous avons utilisé pour tout le projet un seul package Java, toutes nos classes étant dans le même dossier `src`. Nous avons par ailleurs maintenu un fichier par classe afin de faciliter la compilation du programme.

1.1 Patrons de conception

Notre code est implémenté selon plusieurs patrons de conception *(design patterns)* principaux. Nous allons rapidement expliquer leur intérêt ici, mais sans rentrer dans les détails d'implémentation.

Le patron **Modèle-Vue-Contrôleur (MVC)** régit l'ensemble de notre code. Il se traduit par le fait que la partie affichage (visualisation) est séparée de la partie données (modèle de simulation) et de la partie contrôleur (prises de décisions et validation), cette séparation étant faite par l'écriture de classes bien distinctes. Plus précisément :

— l'utilisation des interfaces Java et du polymorphisme (classes abstraites) permet de définir un certain nombre de traitements autorisant une manipulation des objets sans connais-

sance de leur nature réelle : interface graphique, validation d'un déplacement, action sur un robot, ...
— lors de l'exécution d'une méthode sur un objet, les autres objets en sont avisés si nécessaire : ceci est rendu possible grâce à la propagation des exceptions et à l'exécution de méthodes de signalement d'événements sur des objets (par exemple, signaler au Simulateur que le Manager a correctement exécuté l'événement),
— certains aspects du programme ont été conçus de manière à être interchangeables : propriétés des robots, managers, ...

Le manager qui implémente le calcul du plus court chemin est conçu selon le patron ***Non-Virtual Interface***. Une superclasse publique définit des méthodes concrètes qui peuvent s'appliquer dans le cas général et qui sont utilisables depuis l'extérieur de la classe. Ces méthodes sont ensuite redéfinies dans des cas particuliers définis au sein des sous-classes (éventuellement privées). Le déplacement d'un robot peut par exemple être motivé par la nécessité de remplir son réservoir ou celle d'éteindre un incendie, l'appel se fera dans tous les cas grâce à une méthode `doAction()` qui se chargera d'appeler les méthodes compétentes pour éxécuter l'action souhaitée.

Enfin, le patron **Stratégie** s'applique tout particulièrement aux managers : le code a en effet été conçu pour permettre une interchangeabilité des managers par la modification d'une seule ligne de code dans la classe principale. Les managers peuvent ainsi être le point d'entrée de tests réalisés sur la simulation (voir plus bas).

1.2 Architecture des classes

Notre conception du code a suivi les patrons mentionnés plus haut : nous avons créé de nombreux objets chargés chacun d'une tâche spécifique, et qui délègue le code qui ne leur appartient pas à des sous-classes ou à d'autres objets. C'est par exemple le cas entre le Simulateur et le Manager : coder une seule classe aurait été possible, mais il ne revient pas au manager d'exécuter des événements ! C'est également pour l'algorithme du plus court chemin, qui a été codé séparément du Manager.

Une telle construction génère plus de code et de fichiers, mais présente plusieurs avantages, parmi lesquels une plus grande facilité à tester les fonctions et une lisibilité accrue : chaque objet se charge uniquement de ce qu'il est légitimement en mesure de faire.

2 Algorithme du plus court chemin

L'algorithme utilisé pour le calcul du plus court chemin est l'algorithme dit « A-star », qui permet de trouver rapidement l'un des plus courts chemins permettant de rejoindre deux nœuds d'un graphe. Dans cet algorithme, chaque case de la carte est représentée par un nœud, et la distance séparant deux nœuds est d'autant plus grande qu'un robot donné mettra de temps à passer d'une case à sa voisine : le poids de chaque arrête est donc inversement proportionnel à la vitesse de déplacement du robot sur la case courante. Certaines cases de la carte sont inaccessibles à certains robots : les nœuds correspondants du graphe ne seront pas reliés et seront ignorés par l'algorithme.

Cet algorithme est une variante optimisée de la méthode de Dijkstra en ce qu'il n'analyse pas tous les nœuds du graphe (coût quadratique), mais il recherche un chemin de coût minimal dans une direction globale donnée. Pour des graphes proches de la complétude (peu de nœuds inaccessibles et donc d'obstacles), cet algorithme se montre très performant.

L'implémentation de cet algorithme se base sur une fonction d'heuristique utilisée pour calculer la distance qui sépare le nœud courant du nœud à atteindre. Dans notre programme, cette fonction se content d'additionner l'écart absolu en abscisse et en ordonnée entre les cases, sans tenir compte de leur taille ou de la vitesse du robot sur le terrain en question :

$$h(x) = |dx| + |dy|$$

.

Cette fonction est simple à calculer et donne, d'après nos tests, de bons résultats sur la cartes mises à disposition.

3 État du projet

3.1 Tests effectués

En parallèle du développement de notre code, nous avons mené des tests unitaires pour nous assurer de la bonne conception de notre programme. Les classes des deux premières parties du sujet n'ayant pas posé de difficultés majeures, nous nous sommes contentés d'écrire quelques classes de tests, comme par exemple `TestLecteurDonnees`, `TestAfficheSimulation`, ...

En revanche, les parties III et IV du sujet mettaient en œuvre des algorithmes plus lourds dont la validation du bon fonctionnement devait faire l'objet de tests plus poussés. Nous avons alors utilisé la bibliothèque de tests unitaires Junit sur les classes `Case`, `Astar`, `ManagerDynamique` (le manager implémentant l'algorithme du plus court chemin), ...

En plus de Junit, nous avons écrit plusieurs managers chargés de tester différents scénarios sur les cartes proposées. Ceci nous a permis de valider le fonctionnement du programme pour différentes configurations initiales. En revanche, nous n'avons pas envisagé d'autres cartes que celles proposées par le sujet.

3.2 Ce qui fonctionne

Le programme que nous rendons fonctionne dans la mesure où l'algorithme du plus court chemin est implanté et où les robots se déplacement, éteignent des incendies et remplissent leur réservoir en eau lorsque cela est nécessaire. Dans l'état actuel, tous les robots se dirigent vers le feu dont ils sont le plus proche pour aller l'éteindre ; de ce fait, plusieurs robots peuvent se retrouver à éteindre le même incendie. Nous ne pensons pas que ceci soit gênant étant donné que chaque robot possède ses propres durées d'interventions.

3.3 Ce qui ne fonctionne pas

Le `ManagerDynamique` rendu implémente l'algorithme de plus court chemin mais ne tient pas compte de la durée des événements : il exécute en effet les actions directement sur les

robots, sans passer par la création d'un événement pour le simulateur. On obtient ainsi une simulation dépourvue de l'aspect temporel, mais où l'on peut constater le bon déplacement des robots. De fait, les robots se déplacent très rapidement d'une case à l'autre et se remplissent instantanément.

Dans leur calcul du plus court chemin, les robots ne tiennent ainsi pas compte des ralentissements qu'ils subissent en traversant certaines zones. En revanche, ils ont connaissance des cases qui leur sont inaccessibles et ne s'y rendent pas. Dans les cartes où certains robots sont entourés de cases inaccessibles derrière lesquelles se trouvent les incendies, ceux-ci ne bougent pas, comme demandé.

Conclusion

Au cours de ce TP, nous avons pu mettre en application les différents concepts vus en cours de Java et les intégrer dans un cadre plus concret et plus vaste que les TD faits en classe.

Cependant, les apprentissages de ce TP vont bien au-delà de la simple connaissance du langage Java. Ainsi, nous avons pu avoir une première expérience en conception objets, ce qui représente une rupture par rapport à tous les modes de programmation vus précédemment : il est parfois assez difficile de penser objets ! Cependant, on s'aperçoit bien vite qu'une telle conception facilite grandement les interactions entre les différents segments de code informatique...

Par ailleurs, ce TP a été l'occasion pour nous de découvrir de nouveaux outils et de nouveaux algorithmes : on pourra notamment citer l'usage du gestionnaire de versions Git et la programmation de l'algorithme A-Star.

Enfin, un travail de cet ampleur mobilise trois personnes sur des aspects parfois éloignés d'un même programme : on note des divergences dans la vision de l'objectif, dans les moyens à mettre en œuvre pour y arriver. Ces différences de conception nécessitent des périodes de mise au point et impliquent une grande communication entre les membres de l'équipe. Même si ce travail ne relève pas à proprement parler de la gestion de projet, il passe par la nécessité de fixer un cadre au travail de chacun.