Simulation Orientée-objet de systèmes multiagents

TP en temps libre de Programmation Orientée Objet

Novembre 2017

1 Un premier simulateur : jouons à la balle

Généralitées
Dans cette partie, une balle est un objet caratérisé par un vecteur de position et un vecteur vitesse. A chaque appel à next, nous calculons le nouveau vecteur vitesse et nous additionons celui-ci à la position pour obtenir la nouvelle.

Spécificitées
Les particularités de notre implémentation: gestion des rebonds contre les bords du gui (fontion *updateVelocityBorder* et rebonds des balles entre-elles (fonction *updateVelocityImpact*).

2 Des automates cellullaires

Dans toutes les questions suivantes, la taille des cases s'adapte au nombre de cases, et l'écran du Gui s'adapte à la taille de l'écran de l'utilisateur.

2.1 Le jeu de la vie de Conway

Pour cette partie, le code se décompose de la manière suivante :

1. Une classe abstraite UNIT, qui nous évite de redéfinir les accesseurs et les modificateurs dans chaque classe qui hérite de UNIT, mais qui oblige aussi chaque sous classe à redéfinir la méthode *changeState()*, spécifique à chaque type de systèmes multiagents. Cette classe est nécessairement abstraite : il n'est pas possible d'instancier un objet de type Unit, puisque l'on ne sait pas exactement ce que représente une unité. De plus, comment faire passer un objet de type Unit dans l'état suivant si l'on ne sait pas de quelle "unit" il s'agit ? En effet, une cellule à deux états n'a pas les mêmes règles de changement d'état qu'une cellule à N états. D'où l'utilisation de l'abstraction ici. En outre, cette classe réalise l'interface Cloneable, car il nous est parfois nécessaire de recopier un ArrayList d'objet dans un autre. Une simple égalité entre deux élèments de l'ArrayList produit une copie de référence, et non pas une copie d'objet, d'où l'intérêt de réaliser l'interface Cloneable : elle nous fournit une méthode *clone()*, qui nous permettra de cloner des objets d'une ArrayList vers une autre.

2. Une classe CELL, qui hérite de UNIT. En effet, une cellule est une unité qui ne peut prendre que deux états : 1 pour vivant, 2 pour mort.

3. Une autre classe abstraite, UNITS, qui va nous permettre de créer des ArrayList d'objets qui représenteront le plateau. Cette classe utilise le principe de généricité. En effet, la classe est déclarée de la façon suivante : "abstract class Units<T extends Unit>". Cela signifie que T peut être n'importe quel objet héritant de Unit, et ainsi nous pourrons créer des ArrayList de type de cellules différentes, en remplaçant simplement T par le type de cellule voulu. Cette classe sera donc très utile dans les questions suivantes. Cela permet de plus la réutilisation de certaines méthodes tel que *reInit()*, commune à tout type de cellule.
   En outre, l'abstraction est nécessaire, puisque, par exemple, la méthode *nextState()* est spécifique à chaque type de cellule et doit donc être redéfinie pour chacunes d'entre elles.

4. Une classe CELLS qui hérite de UNITS<CELL>. Le principe de généricité s'applique donc ici. Le plateau est alors représenté par un ArrayList *board*, dont chaque élément est un objet de type Cell. Il est uniquement nécessaire dans cette classe de redéfinir la méthode *nextState()*, la méthode *reInit()* étant héritée de UNITS<T>.

5. Une classe CELLSSIMULATOR, qui se contente de mettre à jour notre simulateur graphique. En effet, à chaque fois que le bouton suivant est enclenché, on réinitialise le simulateur puis pour chaque cellule de notre plateau, c'est à dire pour chaque élèment de notre ArrayList *board*, on affiche une cellule de la couleur correspondant à son état sur le simulateur graphique. Cette classe hérite de la classe abstraite SIMULATOR.

6. Quelques détails sur la classe SIMULATOR. Elle aussi utilise les principes de généricité et d'abstraction, pour les mêmes raisons que UNITS : pour que la classe SIMULATOR soit utilisable quelque soit le type de cellules, et sachant que nous avons besoin d'un ArrayList de cellules représentant le plateau, la généricité est nécessaire. De plus, on ne peut instancier un objet de type SIMULATOR, puisque l'on ne sait pas quel type de cellule il doit simuler. La classe est donc abstraite.
   Dans chaque question de cette partie, on pourra faire hériter nos classes *SIMULATOR de cette classe, et ainsi factoriser le code. Les méthodes spécifiques à chaque type d'objet (la mise à jour du gui n'est pas le même pour une cellule "classique" que pour une cellule à N états!) sont déclarées abstraites, et sont ainsi redéfinies dans les sous classes.

2.2 Le jeu de l'immigration

Chaque cellule a maintenant n états. Grâce aux principes d'abstraction et de généricité introduits à la question précèdente, la structure du code pour cette question est très similaire. En effet :

1. Une cellule à N états est une unité, par conséquent CELLNSTATES hérite de UNIT. On prend soin de redéfinir les méthodes abstraites de UNIT. On constate bien ici l'intérêt de l'abstraction : la méthode *changeState()* est différente selon l'unité.

2. La classe CELLSNSTATES hérite de UNITS<CELLSNSTATES>. Le plateau est alors représenté par un ArrayList d'objets CellNStates, et il est donc uniquement nécessaire de redéfinir la méthode *nextState()* dans cette classe.

3. De même que pour la classe CELLSSIMULATOR, la classe CELLSNSTATESSIMULATOR hérite de la classe SIMULATOR, et ainsi, seule la méthode *majGui()* est à redéfinir. On peut à nouveau remarquer l'efficacité des principes d'abstraction et de généricité.

2.3 Le modèle de Schelling

Encore une fois nous n'avons pas besoin de rajouter grand chose á nos précédentes classes pour cette nouvelle question :

1. Une habitation est une cellule á N états avec 1 état signifiant que l'habitation est vacante (état 0). La classe CELLSNSTATES hérite aussi de UNIT, et seule la méthode *changeState()* est á redéfinir.

2. La classe HABITATIONS hérite de UNITS<HABITATIONS>. L'unique différence avec les cellules est un attribut seuil en plus, permettant de décider du changement d'état.

3. De même que pour la classe CELLSSIMULATOR, la classe HABITATIONSSIMULATOR hérite de la classe SIMULATOR, et ainsi, seule la méthode *majGui()* est à redéfinir.

Spécificitées
Le choix de l'habitation vacante choisie pour migrer est aléatoire. Pour le test, nous avons mis autant d'habitations vacantes que non vacantes afin de garantir le bon fonctionnement du programme.

3 Un modèle d'essaims : les boids

3.1 Programmation d'un système de boids

Généralitées sur la classe Boid
Dans cette partie, un boid a plusieurs caractéristiques. Tout d'abord, il est caractérisé par un vecteur de position, un vecteur de vitesse et un vecteur d'accélération. Ainsi, la classe Vector créée dans la première partie de ce projet est réutilisée ici. Il est de plus soumis à des forces, qui auront un impact sur son vecteur accélération, donc sur son vecteur vitesse et position. Nous avons rajouté quelques contraintes : le boid a une vitesse maximale (*maxSpeed*), un angle de vision réduit à moitié (le boid ne voit que ce qui est devant lui), une distance minimale entre lui et ces voisins (*desiredSeparation*), une distance qui définit sa notion de voisinnage (*preferredDist*), et une limite de force qui peut lui être appliquée (*maxForce*). Toutes ces caractéristiques définissent les attributs de la classe Boid créée.

Spécificitées de la classe Boid
Comment se déroule le calcul du vecteur accélération d'un boid? Aprés avoir calculé les vecteurs des forces exercées par les boids voisins sur le boid actuel, nous avons décidé d'accorder un poids plus important a certaines forces qu'à d'autres. Sachant que tous les vecteurs forces résultats sont normalisés, il suffit de multiplier ces vecteurs résultats par des coefficients plus ou moins élevés selon le degrés d'importance de la force. C'est que nous faisons au seins de la méthode *flock()* de la classe Boid. De plus, nous avons rajouter une force "migrate" au boid, que l'on ajoute en fin de calcul à son vecteur accelération, afin de donner un peu plus de dynamisme à nos boids! Pour finir, nous avons ajouté une force "border", qui s'applique sur les boids quand ceux-ci se rapprochent trop d'un bords du gui. Cette force est progressive et augmente en x puissance 6 quand le boid continu de se rapprocher. Cette force pousse donc les boids à rester Ã l'intérieur de l'écran. Si jamais ceux-ci s'Ã©chappent malgrÃ© tout, alors ils rÃ©apparaissent dans l'écran du coté opposé.

Généralitées sur la classe BoidsSimulator
Pour animer nos boids, on crée une classe BoidsSimulator qui réalise l'interface Simulable. A l'instar des autres classes *Simulator, elle redéfinit les méthodes *next()* et *restart()*. Elle est de plus munie de deux méthodes de mise à jour de l'interface du simulateur : *initGui*, qui est uniquement appelée dans le constructeur de BallsSimulator pour afficher la position initiale des boids, et *updateGui()* qui affiche les nouvelles positions des boids.

Spécificitées de la classe BoidsSimulator
Le vecteur vitesse de chaque boid du ArrayList *boids* est normalisé puis multiplié par 5.5 afin de... En outre, deux éléments graphiques sont ajoutés au simulateur : la "tête" du boid, représentant sa direction, ainsi que son corps.

3.2 La cohabitation, c'est plus difficile...

3.2.1 Un gestionnaire à évenements discrets

Spécificitées
Chaque évènement a une date et un gestionnaire d'évènements, qui sera principalement utilisé pour lister l'intégralité des évènements. Ces deux caractéristiques sont les attributs de la classe Events. Events réalise l'interface Comparable : il est donc nécessaire de redéfinir la méthode *compareTo()*, qui est utilisée pour ordonner les élèments d'une collection selon un critère particulier, ici la date. D'ailleurs, nous avons choisi la collection TreeSet pour stocker les évènements, d'où l'intérêt d'avoir introduit la relation d'ordre *compareTo()* dans la classe Events. Ainsi, les éléments de Treeset<Events> sont ordonnés, c'est à dire triés de la date la plus récente à la plus vieille.

3.2.2 Modification de votre simulateur

Pour intégrer un gestionnaire d'évènements à notre simulateur, nous avons tout d'abord crée des classes d'évènements spécifiques à chaques systèmes multiagents. En effet, les appels aux méthodes des systèmes d'agents (comme la méthode *nextState()* par exemple) sont en fait réalisés dans la méthode *execute()* des événements, et comme ces méthodes ne sont pas les mêmes pour les boids, les balls ou les cellules, une nouvelle classe d'évènements est nécessaire pour chacun d'entre eux. C'est pourquoi,pour les cellules, nous

créons la classe EventChangeColor, sous classe de Event. Celle-ci utilise le principe de généricité introduit dans les questions précèdentes ainsi que l'abstraction. De ce fait, dans la méthode *execute()* de EventChangeColor, lorsque l'on effectue *cells.nextstate()*, *cells* étant de type *private T*, le compilateur appelle la méthode *nextState()* adapté au bon type de cellule.
Dans un second temps, pour intégrer un gestionnaire d'évènements à notre simulateur, nous devons ajouté celui-ci à chaque classe de type *Simulator. C'est pourquoi la classe Simulator (qui concerne les cellules) a un attribut *protected EventManager eventManager*. Il est initialisé dans le constructeur de Simulator avec un premier élément : un évènement EventChangeColor de date 0. Ensuite, comme demandé dans l'énoncé, le méthode *next()* de Simulator ne fait plus directement appel au la méthode *nextState* du type de cellule correspondant, mais c'est le gestionnaire d'évènements qui s'en charge, en appelant dans sa méthode *next()*, la méthode *execute()* pour le bon type d'évènement, dans notre cas le "execute" de EventChangeColor.
En outre, pour avoir une animation qui "dure", on crée de nouveaux évènements à la volée dans la méthode *execute()* de chaque sous classe de Event.
Il en est de même pour les boids : on crée un nouveau type d'évènement, EventMove, qui redéfinit la méthode *execute()* car spécifique à chaque type d'agent. Le gestionnaire d'évènements est intégré dans BoidsSimulator de la même manière que dans Simulator.

3.2.3 Simulation de plusieurs groupes de Boids

Dans cette partie, on crée un nouveau type de Boid (donc une nouvelle classe) : Boid2. Cette classe hérite de Boid puisque les objets de type Boid2 sont des boids. Les caractéristiques de ces boids sont différentes : ils sont plus éloignés les uns des autres et les voisins sont considérés dans un rayon plus large.
Pour simuler cette nouvelle classe, on crée la classe MultiSimulator, qui hérite de BoidSimulator. Les boids, quelques soient leur type (Boid ou Boid2), sont stockés au seins d'un même ArrayList *boids*. Lorsqu'un traitement particulier est à effectuer selon le type de Boid, comme par exemple pour l'affichage (un boid de type Boid est affiché en rouge, un boid2 en bleu), on utilise la méthode *getClass()* pour savoir de quel type de Boid il s'agit.
En outre, un nouveau type d'évènement est nécessaire, puisque la méthode *execute()* est spécifique à chaque système d'agent. En effet, nous avons choisi pour les boids Boid2, un pas de temps 2 fois plus grand entre l'éxécution de deux évènements que pour les boids Boid. Ainsi, l'ajout d'évènements à la volée dans la méthode *execute()* est différente : les dates vont de 2 en 2.
On peut vérifier visuellement que notre implémentation fonctionne: les boids de type Boid2 sont plus lents, bien plus écartés, et les essaims sont plus important en taille, puisque les voisins sont considérés dans un rayon plus large.