Simulation Orientée-Objet de systèmes multiagents

Équipe 52

19 novembre 2021

1 Introduction

Ces quatres pages visent à présenter nos choix d'implémentation et les résultats obtenus aux problèmes posés dans le sujet de TP de POO. Dans ce rapport, il sera considéré que le lecteur connaît les différents sujets du projet, limitant les explications à fournir vis-à-vis de certaines parties du code. Il sera également considéré qu'il est à l'aise avec les éléments vus en cours de POO, nous permettant d'omettre certaines explications triviales (accesseurs, mutateurs, constructeurs des classes etc). Souhaitant simuler un cadre réel d'entreprise, nous nous sommes imposés d'écrire l'ensemble du code de la manière la plus universelle possible. Pour cela, il a été entièrement implémenté en anglais, le *coding style* Java est respecté, et les variables ont un nom le plus explicite possible. Des commentaires précisent le rôle des différentes parties du code, et l'API des classes peut être générée avec la commande *javadoc* (voir fiche 0 du cours). Les différentes parties du projet seront traitées dans trois parties distinctes, comprenant toutes une explication de la structure de l'implémentation, des justifications, et une présentation des résultats obtenus. Dans la suite, les méthodes seront appelées par leur nom, et non leur prototype (il faudra donc être attentif aux cas considérés), et les fichiers par leur nom de classe principale. Le code a été séparé dans différents packages, comme spécifié sur le diagramme UML disponible dans l'archive de rendu et qu'il est fortement conseillé de regarder avant de s'attaquer au code. Ainsi, tous nos fichiers de tests et ceux décrits dans le sujet sont dans le package *tests*. Les packages *math* et *util* comportent des objets ou outils mathématiques utilisés dans les sujets du TP (par exemple une classe représentant un vecteur et comprennant des méthodes associées).

2 Premier Simulateur : Jeu de Balles

Cette partie concerne les classes *Balls*, *BallsSimulator* et les fichiers de tests associés. Même si elle est facile à comprendre, le lecteur est conseillé de bien la lire pour comprendre le fonctionnement et la séparation des parties graphique et calcul, qui sera également faite dans les implémentations suivantes par soucis de clarté. On implémente la classe *Balls* comme une réalisation de *GraphicalElement*, pour l'utiliser plus tard comme argument dans la méthode *addGraphicalElement* de la classe *GUISimulator* (voir l'API). Cela explique la redéfinition de la méthode *paint* dans le code. Les méthodes *translate* et *reInit* reposent sur l'utilisation de la méthode *translate* de la classe *Oval* et sur un choix arbitraire de coordonnées. Le fichier de test *TestBalls* valide l'implémentation réalisée.

Afin d'animer les balles, on réalise la classe *BallsSimulator* comme un *Simulable*, permettant d'utiliser une instance de cette classe comme attribut lors de l'instanciation d'un *GUISimulator*. Cela induit la redéfinition des méthodes *restart* et *next*. Cette dernière calcule les déplacements des balles en translatant les coordonnées de chacune par le vecteur *velocity* associé, et modifié au fur et à mesure pour que les balles restent dans le rectangle formé par les paramètres *x*, *y*, *width* et *height* de la classe *BallsSimulator*. Quand la coordonnée d'une des balle est en dehors de ce rectangle, la coordonnée correspondante de son vecteur de déplacement est changée pour son opposée. Enfin, l'affichage est réalisé dans la classe *TestBallsSimulator*. Pour cela, il suffit d'instancier une variable de type *GUISimulator* avec un attribut de classe *BallsSimulator* (réalisation de *Simulable*), et d'y ajouter les balles comme élément graphique (avec *addGraphicalElement*). On retrouve ici la séparation des parties *calcul* et *graphique* de la simulation. En effet, les méthodes de dessin sont dans la classes *GUISimulator*, tandis que les méthodes de calcul sont dans *BallsSimulator*.

3 Automates Cellulaires

En regardant le diagramme UML, le lecteur verra que les classes *Area* et *Entity* sont utilisées pour l'implémentation des Automates Cellulaires et des Boids. Ces deux classes sont à l'origine de la séparation des parties graphique (*Entity* et filles) et calcul (*Area* et filles). Dans cette partie, *Area* est la classe mère de *GridArea*, qui représente la grille dans laquelle les automates évoluent, et la classe *Entity* est la classe mère de *Case*, qui représente une case de cette grille.

Dans la classe *Entity*, on retrouve du code relatif à la position d'un élément sans sens particulier. La classe *Case* (sa fille) est enrichie d'une couleur, largeur, hauteur et de trois attributs de classe *State*. Ceux-ci correspondent aux différents états (courant, suivant et initial) de la case, et sont utilisés pour la mettre à jour, ou la réinitialiser. De manière classique, pour faire évoluer la grille, le changement de chaque case sera stocké dans son attribut *nextState*, avant de mettre à jour son attribut *currentState* (avec la méthode *update*) lorsque tous les changements de la grille auront été calculés. Cela assure que l'état de toutes les cellules à l'étape $t + 1$ ne dépende que de l'état des cellules voisines à l'étape t. Ces changements d'états sont déterminés par la méthode *calculate* de la classe *Case*, qui prend en paramètre la liste des cases voisines, charge leur état courant, et utilise la méthode *nextState*, spécifique à chaque automate, pour stocker le prochain état dans l'attribut *nextState*.

Dans la classe *Area*, on retrouve une *ArrayList* d'entités graphiques. Pour cette partie, elles correspondent aux cases (de classe *Cases*) qui composent la grille. Elle est une réalisation abstraite de *Simulable*. Ainsi, on retrouve la redéfinition des méthodes *restart* et *next* dans sa classe fille *GridArea*. La première ne fait que charger l'état initial de chaque case dans l'état courant et remet à jour la grille. La seconde parcourt toute la grille, appelle pour chaque case la méthode *calculate* décrite plus haut, puis met également à jour la grille (états + couleurs des cases). Enfin, on retrouve pour chaque automate cellulaire une classe spécifique qui est fille de *GridArea*, et qui se charge de "remplir" la grille de cases correspondantes. Ces différentes classes seront traitées dans les parties dédiées. La dernière partie commune de code est la classe abstraite *State*, qui correspond à un certain état (ayant donc une certaine valeur à un certain moment), et dans laquelle on retrouve la méthode abstraite *nextState* décrite au paragraphe précédent et propre à chaque automate.

3.1 Jeu de la vie de Conway

Le lecteur aura remarqué la présence de la classe *ModuloState* sur le diagramme UML. Elle est mère des classes *ConwayState* et *ImmigrationState* car l'état d'une cellule évolue toujours de manière modulaire (d'où la méthode *increaseValue*). La méthode *nextState* de la classe *ConwayState* calcule les états suivants comme spécifié sur le sujet. Enfin, la classe *ConwayArea* est utilisée pour créer une grille de cellules de *Conway*. Pour chacune d'entre elles, on créé trois états distincts (mais au départ de même valeur), une case les contenants, et on ajoute cette dernière à la collection *ArrayList* des entités graphiques de la classe *Area*. En lançant le programme de test *ConwayTest*, le lecteur pourra observer des *structures stables*, des *vaisseaux*, et s'il est plus chanceux des *canons* ou des *puffeurs*[1].

3.2 Jeu de l'immigration

Pour cette partie, presque rien ne change. Les classes *ImmigrationArea* et *ImmigrationState* sont identiques à quelques éléments près. Le fichier de test *ImmigrationTest* en confirme l'implémentation.

3.3 Modèle de Schelling

Cette partie comporte plus de spécificités. Dans la classe *SchellingState* on retrouve le seuil K et deux collections *ArrayList* correspondant aux cases vacantes (de valeur 0) initiales et à un certain moment de la simulation. Lors de l'initialisation des cases dans la classe *SchellingArea*, toutes celles qui sont vacantes y sont ajoutées avec la méthode *addVacantCase*. Etant générées de manière aléatoire, on peut supposer que pour les dimensions raisonnables d'une grille, il y aura toujours un nombre suffisant de cellules vacantes. Nous avons choisi de mettre confondre l'état suivant et l'état courant afin de faire évoluer la grille dynamiquement. Ainsi, une case qui devait déménager à l'étape t peut ne finalement pas le faire, si certains de ses voisins ont déjà déménagé avant le calcul de son état futur (si les raisons

1. Si l'envie le prend, le lecteur est invité à consulter la page wikipédia du jeu de la vie pour comprendre les structures citées : `https://fr.wikipedia.org/wiki/Jeu_de_la_vie`

qui font qu'elle voulait déménager ne sont plus là, il n'y a pas de raison qu'elle déménage). Enfin, on retrouve une redéfinition de la méthode *restart* dans la classe SchellingState pour réinitialiser la liste des cellules vacantes. La méthode *move* se charge de faire "déménager" une case lorsque la méthode *nextState* estime que cela est nécessaire. Elle retire une case vacante choisie aléatoirement dans la liste *currentVacantCases*, en change la valeur pour celle qui déménage et y ajoute la nouvelle case vacante. Avec un seuil K égale à 3, les tests convergent assez rapidement vers des formes stables pour un nombre de couleur inférieur à 6 (attribut de la classe *SchellingArea*). On observe des ilots formés par des cases de même couleur. Au délà d'un nombre de couleur égale à 6, cela est très lent mais finit par converger. Naturellement, augmenter le seuil K augmente la vitesse de convergence. Enfin pour un seuil K égale à 2, le test ne semble plus converger pour un nombre d'état supérieur ou égale à 4 (ou trop lentement pour être observé).

4 Un modèle d'essaims : les boids

Commençons par expliquer le fichier de test *BoidsTest* associé à cette partie. Le simulable *area*, de classe *AgentArea* est d'abord créé, avec le booléen *optimized* de valeur *true*. Cela autorise d'utiliser des fonctions optimisées pour le calcul des interactions sur les agents. Deux groupes de "proies" sont créés, et sont ajoutés comme groupes d'individus à *area*. Ensuite, on ajoute une force de séparation entre ces deux groupes. Un groupe de prédateur est créé (triangles oranges dans la simulation), et est autorisé à "manger" les "proies" du groupe 1. Le paramètre 30 correspond au délai avant de pouvoir "manger" les "proies" de ce groupe.

Encore une fois, la partie graphique, située dans la classe *Agent*, est fille de la classe *Entity*. On y retrouve des attributs propre à un agent, comme sa vitesse, sa vision, sa couleur, son état (vivant ou non) etc. La redéfinition de la méthode *paint* représente chaque agent comme un triangle sur l'interface graphique. La méthode *isViewing* utilise des méthodes du package *math* pour déterminer si un agent en voit un autre comme indiqué sur le sujet.

Pour la partie calcul, la classe *EventArea* est fille de la classe *Area*. Elle est enrichie d'un attribut de classe *EventManager* et de méthodes permettant d'intégrer le gestionnaire d'évènements discrets à notre simulateur. Dans cette classe, on retrouve la collection java *PriorityQueue* pour stocker les évènements, permettant de sortir l'évènement le plus faible en priorité. Afin d'avoir une notion d'ordre sur ces évènements, la classe *Event* est une réalisation de la classe *Comparable*. Le test *TestEventManager* confirme l'implémentation des deux classes précédentes. Dans l'implémentation, la classe *AddInteractionEvent* est spécifique au groupe de prédateurs. Elle est utilisée pour manger une "proie" spécifique dans la simulation. De plus, *AgentGroupUpdateEvent* se charge d'effectuer le pas de simulation pour le groupe en question en lui appliquant, toutes les intéractions ainsi que la seconde loi de Newton en prenant en compte le pas de mise à jour propre à ce groupe. Enfin, *AreaUpdateEvent* se charge d'appliquer les interactions globales (c'est à dire les *GlobalInteraction* exercées par l'environnement comme le vent via *WindForce* par exemple).

La classe *AgentArea* est fille de la classe *EventArea*. De manière analogue aux automates cellulaires, c'est elle qui contiendra les différentes entités graphiques de la simulation (de classe *Agent*). Son attribut *groups* contient des éléments de classe *AgentGroup*, représentant les différents groupes d'agents. Ainsi, dans le fichier de test décrit plus haut, *groups* contiendrait deux groupes de boids et un groupe de prédateurs. Ces deux "types" d'agents sont distingués par les classes *BoidGroup* et *PredatorGroup*, fille de la classe *AgentGroup*. Dans cette dernière, on retrouve des attributs spécifiques à un groupe d'agents, comme une liste d'interactions, et l'attribut *agentsGrid*, un tableau 2D contenant des *ArrayLists* d'entités graphiques (de classe *Agent*). Ce tableau 2D est représenté par la classe *BiDimensionalArray*, implémentée dans le package *util* et dont la *javadoc* permet de saisir le fonctionnement. Dans la classe *AgentGroup*, l'attribut *agentsGrid* représente un quadrillage de l'espace graphique dans lequel évolue les boids, calculé à partir de la distance de vision de ceux-ci. Il est utilisé, avec la méthode *getKey* qui associe chaque agent à une des cases du tableau 2D, dans la méthode *applyInteractionsOptimized*. Cette dernière calcule les interactions qui s'appliquent à chaque boids uniquement à partir des boids présents dans les cases voisines, plutôt que de vérifier chaque couple de boids de la simulation (on retrouve ici un raisonnement *diviser pour régner*). A noter qu'il est possible d'utiliser la méthode *applyInteractions*, non optimisée, en mettant le booléen *optimized* à *false* dans le fichier de test.

Concernant la partie interaction, la classe *ForceInteraction* représente une force pouvant s'exercer sur un agent. La méthode abstraite *computeForce* est spécifique à chacune des forces, et est redéfinie telle que décrite dans les sources du sujet pour les classes *AlignmentForce*, *CohesionForce* et *SeparationForce*. La classe *TrackingForce* a été imaginée

par nos soins, et applique une force sur les prédateurs pour les orienter vers des "proies". La classe *EatInteraction* a également été implémentée selon nos choix. Sa méthode *interaction* tue les agents considérés comme mangés (si la distance du "prédateur" et de la "proie" est inférieure à *3*radius*, et augmente la taille du prédateur en conséquence). De plus, la classe abstraite *Interaction* est mère des classes *ForceInteraction* et *EatInteraction*. On y retrouve le code commun à toutes les interactions, et notamment la méthode *applyOptimized*, servant à appliquer les interactions de manière optimisée comme décrite dans le paragraphe précédent. Si le booléen *optimized* est de valeur *false*, alors c'est la méthode *apply* qui est utilisée. Enfin, de manière analogue, les classes abstraites *GlobalInteraction* et *GlobalForce* réalisent les mêmes opérations mais à l'echelle de toute la simulation (il n'y a donc pas de version optimisée pour ces classes).

5 Conclusion

Pour conclure, ce projet nous aura sensibilisé aux spécificités da la *POO* en générale. En effet, il nous a fait réfléchir sur les notions de classes et d'objets, d'encapsulation, d'héritage pour factoriser au maximum le code et d'abstraction. L'évolution de notre code a été le fruit d'une réflexion méthodique et collective. Les différentes implémentations ont été faites en utilisation la méthode de *Développement Piloté par les Tests*, facilitée par les tests du sujets et ceux que nous avons imaginés. Enfin, il faut préciser que les performances de certaines implémentations sont encore améliorables. Par exemple, l'utilisation d'une *ArrayList* pour stocker les cases vacantes de la simulation du modèle de *Schelling* est un peu maladroite. En effet, le coût de la méthode *remove* est en $O(n)$ dans le pire cas, mais utiliser une autre collection (comme un *HashSet*) aurait demandé de changer une partie du code commun aux automates cellulaires. Cela nous a semblé acceptable sachant que pour les dimensions raisonnables d'une grille, la simulation reste fluide.