



GRENOBLE INP - ENSIMAG

PROJET

PROGRAMMATION ORIENTÉ OBJET (POO)

VENDREDI 22 NOVEMBRE 2019

Étudiants :

Gwenaël Leger

Ayoub Bouhachmoud

Swann Philippe

Table des matières

1	INTRODUCTION	2
2	CONCEPTION	2
2.1	DÉROULEMENT DU PROJET	2
2.2	SCHÉMA DE CONCEPTION	2
3	PREMIÈRE SIMULATION	2
3.1	BALLS	2
4	AUTOMATES CELLULAIRES	2
4.1	LE JEU DE LA VIE DE CONWAY	3
4.2	JEU DE L'IMMIGRATION	3
4.3	LE MODÈLE DE SÉGRÉGATION DE SCHELLING	3
5	LE MODÈLE DE BOIDS	3
5.1	GESTION DES EXTRÉMITÉS	3
5.2	INTÉGRATION DE L'EVENT MANAGER	4
6	TEST ET RÉSULTATS	4
7	CONCLUSION	4

1 INTRODUCTION

L'objectif du TP est d'arriver à simuler des systèmes multiagents. Pour cela nous avons tout d'abord réalisé une simulation basique de balles rebondissantes pour nous familiariser avec les simulations graphiques. Ensuite, nous nous sommes intéressés à 3 systèmes de type automate cellulaire avant de finir par la simulation d'un système de mouvement d'essaims auto-organisés.


2 CONCEPTION

2.1 DÉROULEMENT DU PROJET

Durant tout le projet, nous avons respecté le principe d'encapsulation en masquant les attributs pour garantir l'intégrité des états des objets.

Dans un premier temps, nous avons implémenté les simulations de balles et les 3 automates cellulaires de façon naïve. C'est à dire de manière totalement indépendante. Cependant lors de leur implémentation, nous y avons vu des similitudes. C'est pourquoi nous avons utilisé l'héritage pour factoriser le même code présent dans plusieurs objets. Ensuite nous avons fait la simulations des boids et nous nous sommes rendu compte qu'il aurait été possible de faire évoluer la conception des automates cellulaire pour faire en sorte que les cellules soient des agents indépendants comme nous l'avons fait dans boids.

2.2 SCHÉMA DE CONCEPTION

uml.png

3 PREMIÈRE SIMULATION

3.1 BALLS

Tout d'abord, nous avons créé la classe Balls fonctionnant avec deux vecteurs de de Point, l'un est au temps t et l'autre en $t+1$. La classe BallsSimulator réalise l'interface Simulable grâce à un "implements" de gui.simulable et en redéfinissant next() et restart().

4 AUTOMATES CELLULAIRES

Pour la partie automate cellulaire, nous avons fait hériter les différents simulateurs d'une classe abstraite MultiAgentDiscretSimulator permettant de factoriser notre code. En effet, les next() et restart() (méthodes permettant d'implémenter une simulable) et quelques autres méthodes, ne sont définies qu'une seule fois dans la classe mère MultiAgentDiscretSimulator.

4.1 LE JEU DE LA VIE DE CONWAY

Dans la conception du jeu de la vie de Conway, nous avons juste à créer une classe Conway-Simulator héritière de MultiAgentDiscretSimulator et d'y implémenter une méthode evaluer() qui réponde aux fonctionnements du jeu de la vie.

4.2 JEU DE L'IMMIGRATION

De même, nous avons créé une classe ImmigrationSimulator héritière de MultiAgentDiscretSimulator et y avons implémenté une méthode evaluer() adaptée aux consignes du jeu de l'immigration.

4.3 LE MODÈLE DE SÉGRÉGATION DE SCHELLING

Toujours dans le même esprit de factorisation de code nous avons implémenté une classe SegregationSimulator héritière de MultiAgentDiscretSimulator avec une méthode evaluer() propre aux règles de la ségrégation de Schelling. Le modèle de ségrégation de Schelling étant un peu plus complexe que les deux précédents, nous avons dû ajouter quelques méthodes et objets supplémentaires.

5 LE MODÈLE DE BOIDS

Dans un premier temps, on a essayé d'implémenter les boids comme étant une sous classe de Ball. Un agent boid était dans un point. Pour qu'un boid change sa position (location), il fallait ajouter à sa position le vecteur de vitesse. Au début, on est parti un modèle où tous les attributs de boids location, velocity et acceleration ainsi que les forces appliquées étaient de type Point. Après on s'est rendu compte que c'est pas pratique à manipuler. On a donc créé une classe Vecteur avec des méthodes plus adaptées à notre conception. La gestion de la simulation se faisait, dans un premier temps, à partir de BoidsSimulator qui crée les agents et les affiche sur l'écran. Afin de donner une âme à nos objets inanimés, on a créé des groupes de boids, des prédateurs, représentés par la classe Predateur, et des proies, représentés par la classe Food. Ces deux classes héritent de la classe mère Boid, qui s'est baptisé dorénavant AbstratBoid. De plus, on a une classe ClassicBoid qui peut générer des boids aux comportements différents en changeant des paramètres.

5.1 GESTION DES EXTRÉMITÉS

On a deux modes de gestion des extrémités du panneau d'affichage. Un mode où les boids peuvent les traverser et donc communiquent d'une extrémité à l'autre, et un autre où ils rebondissent.

5.2 INTÉGRATION DE L'EVENT MANAGER

Après avoir créé la classe `EventManager`, qui implémente `Simulable`, la classe `AbstractBoidsEvent`, ancien `BoidsSimulator`, étend `Event`

6 TEST ET RÉSULTATS

Nous avons fait des tests manuels pour chaque simulation et ils se sont tous avérés concluants, répondant bien à toutes les consignes du sujet.

7 CONCLUSION

Lors de ce premier projet de Programmation Orienté Objet réalisé en Java nous avons pu apprécier les avantages d'un tel langage. Avec l'utilisation de l'héritage, nous avons pu factoriser notre code et donc le simplifier de sorte qu'il y ait peu de lignes de code et que la compréhension de notre code soit optimale. Le principe d'encapsulation c'est avéré être une sécurité pour notre code, afin de ne pas accéder et manipuler des variables à des endroits inappropriés.