Grenoble-INP Ensimag, UGA 2ème année 2023-2024

Équipe numéro 65

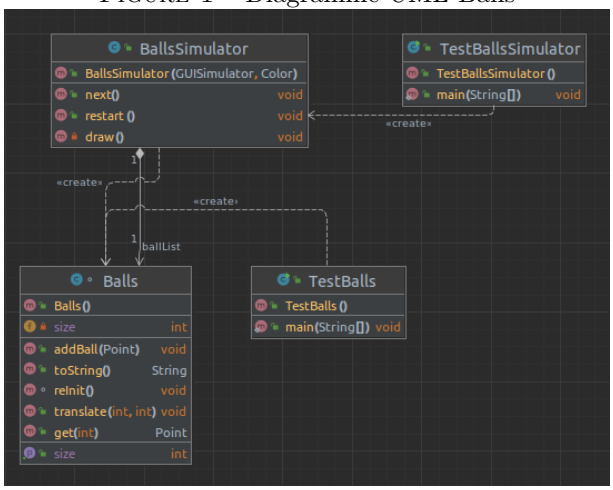
TPL Java POO

Tout notre code est documenté dans notre archive, dans le dossier doc-projet (en lançant index.html).

I. Les Balles

Pour implémenter cette partie, le sujet était assez détaillé et guidé donc nous n'avons rien tenté d'exotique. Le seul choix à faire était le choix de la `Collection` dans la classe *Balls* afin de manipuler l'ensemble de balles. Ainsi, notre choix s'est tourné vers une `ArrayList<Point>()` pour les raisons suivantes : Grâce à cela, la méthode `get()` est en $\Theta(1)$ et pour le fait que la méthode `add()` est déjà implémenté (sachant que l'on souhaite toujours ajouter en fin de liste, cela n'est pas coûteux). Nous avons également implémenter les rebonds sur les bords de la fenêtre.

FIGURE 1 – Diagramme UML Balls



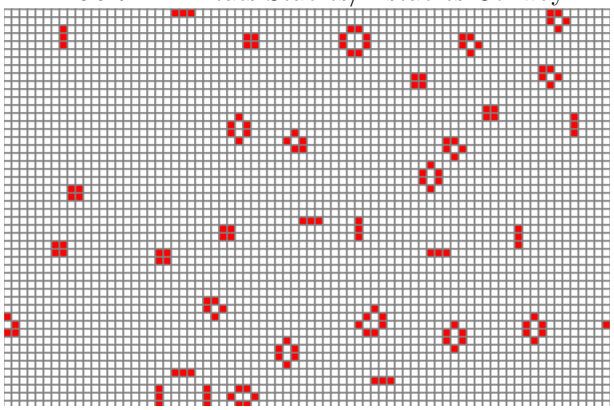
Ainsi, pour gérer les balles, nous avons :
- le fichier *Balls.java* qui implémente la classe *Balls*.
- le fichier *BallsSimulator.java* qui implémente l'interface *Simulable* afin d'obtenir un affichage graphique.
- le fichier *TestBalls.java* qui affiche sur la sortie standard un objet de type *Balls* une fois initialisé afin de vérifier que tout se passe bien.
- le fichier *TestBallsSimulator.java* qui permet d'obtenir un aperçu graphique de la simulation.

II. Jeu de la vie de Conway

Pour implémenter le jeu de la vie, nous avons choisi d'utiliser une matrice de Cellules (Tablaux de tableaux) de tel sorte que $m_{i,j}$ est égale à la cellule qui à pour coordonnées (i,j) à une translation près qui sera géré par la méthode `coordplan()` sur l'affichage graphique. Pour éviter des résultats incorrects lors de la modification de l'état des cellules lorsque l'on passe de l'état t à l'état $t+1$ nous avons deux matrices, une *current* et une *next*. *Current* nous permet d'avoir accès aux états de toutes les cellules avant modifications que nous faisons dans *next*. une fois les modifications faites, on met à jour *current* avec *next*.

Ainsi, pour implémenter le jeu de la vie, nous avons
- le fichier *Cell.java* qui implémente la classe *Cell* qui sera utile pour Immigration et Shelling.
- le fichier *CellularAutomate.java* qui implémente le jeu de la vie.
- le fichier *CellSimuator.java* qui implémente l'interface *Simulable* du jeu de la vie afin d'obtenir un affichage graphique.
- le fichier *TestCellSimulator.java* qui permet d'obtenir un aperçu graphique de la simulation.

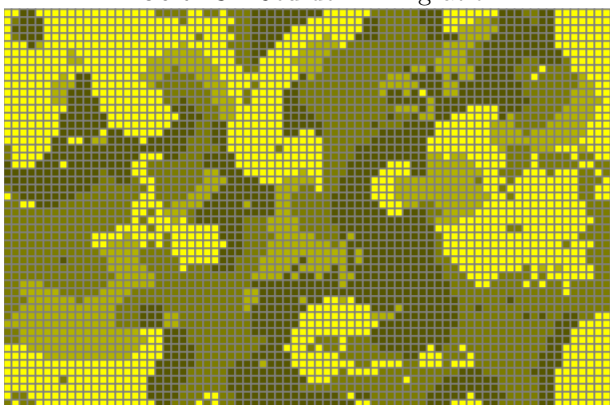
FIGURE 2 – Etats Stables/Instables Conway



III. Jeu de l'immigration

Pour implémenter le jeu de l'immigration, nous avons du généraliser la classe *Cell* dans le but de faire `extends` de notre classe *CellularAutomate* (cf. Figure 2) car le principe est exactement le même. Il suffit de redéfinir uniquement `nextTurn()` et `countNeighbor()`. Il y a seulement quelques différences lors de l'implémentation de l'interface simulable où par exemple cette fois ci, nous utilisons la méthode *darker()* de la classe *Color* pour obtenir un dégradé de couleur en fonction des états des cellules.

FIGURE 3 – Jeu de l'immigration



Les résultats obtenues correspondent bien à nos attentes. De plus, si on augmente le nombre d'états dans lequel peuvent être les cellules, on se rend compte qu'après seulement quelques itérations, plus rien ne bouge ce qui est cohérent car il est très peu probable pour que les conditions de changements d'état d'une cellule soient réunies.

IV. Modèle de Shelling

De la même manière que pour le jeu de l'immigration, nous avons fais un `extends` pour récupérer les méthodes qui n'ont pas besoin d'être implémenté de nouveau. Néanmoins cette fois-ci, comme inqué dans le sujet, nous avons du ajouter un attribut de type `Queue<Point>` pour stocker les places vacantes. Cette collection était parfaite pour remplir les exigences que nous avions grâce à ses méthodes `remove()` et `add()` pour modéliser le déménagement des familles sans qu'il y est de conflit entre chaque famille voulant s'installer ailleurs. Les résultats nous paraissent très cohérents.

FIGURE 4 – État Stable Modèle de Shelling

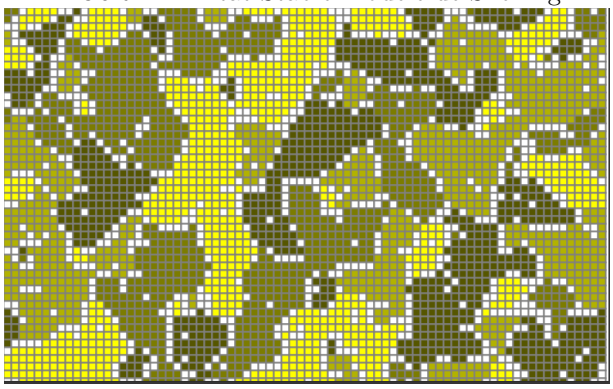
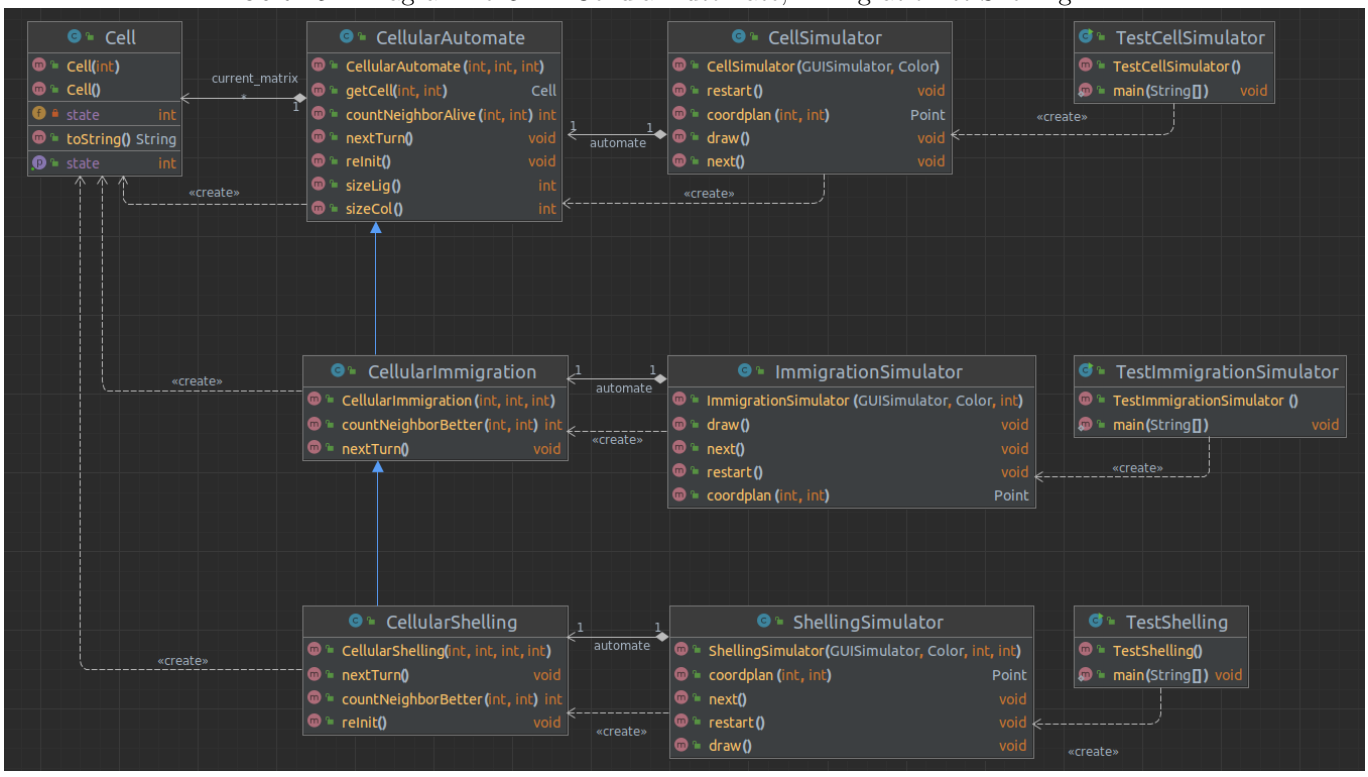


FIGURE 5 – Diagramme UML CellularAutomate, Immigration et Shelling



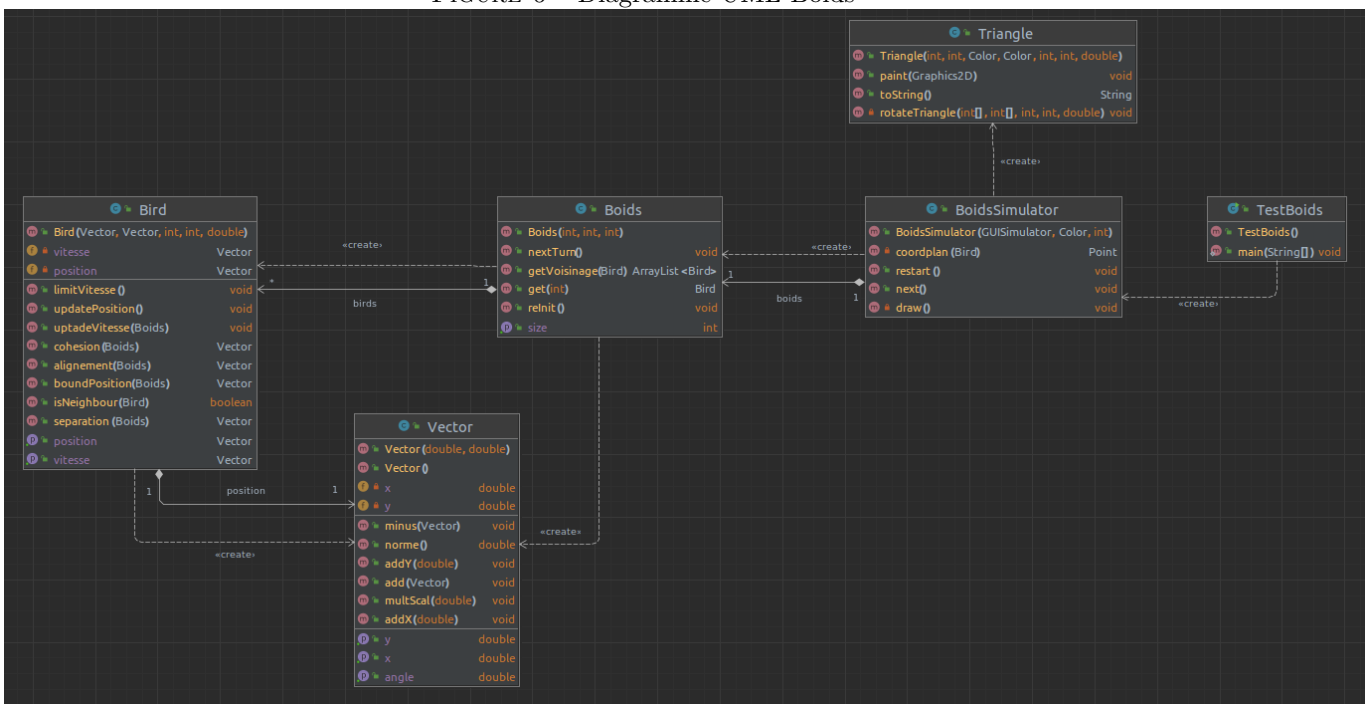
V. Les Boids

Comme le sujet le conseillait, nous avons suivi la même logique que pour les automates cellulaires pour l'implémentation d'une simulation de boids. Ainsi, notre simulation se structure en trois grandes parties : les classes Bird et Boids permettant de calculer la simulation (au moyen de Vector), la classe BoidsSimulator permettant l'affichage graphique (utilisant au passage Triangle)de la simulation et la classe TestBoids pour lancer un test de notre simulation.

Nous avons en premier lieu créée Bird, c'est notre classe la plus important pour cette simulation car nous avons décidé de faire les calculs intrinsèques au déplacements des boids, pour cela nous avons d'abord suivi le pseudocode fourni par le sujet. Mais nous nous sommes rendus compte de comportements inadaptés dû notamment à des erreurs et imprécisions de celui-ci, nous nous sommes donc concentrés sur notre propre réflexion, en refaisant au passage de la physique et décider de coder une classe Vector pour simplifier la lisibilité de notre code.

Ensuite, nous avons fait la classe Boids qui ne comporte pas de parties complexes car elle ne fait, principalement, qu'itérer sur les méthodes de Bird.

FIGURE 6 – Diagramme UML Boids



Conlusion

Pour conclure, nous avons réussi à simuler la plupart des systèmes multi-agents sur lesquels nous nous sommes penchés (type Automate Cellulaire et mouvement d'essaims auto-organisés). Néanmoins par manque de temps, nous n'avons pu finir le gestionnaire à évènements discrets ce qui nous aurez menés vers le modèle de proies-prédateurs. Durant tout le projet, nous avons gardé en tête les aspects fondamentaux de la POO tels que l'encapsulation, la délégation ou encore le principe d'héritage. Tout au long de ce projet, nous avons eu l'occasion de progresser d'un point de vue technique, personnel mais aussi dans la gestion de projet de groupe.