Dans ce TP de programmation orientée-objet en Java, nous nous sommes intéressés à la simulation graphique de systèmes multi-agents. Dans une démarche d’ingénieurs responsables, nous avons réalisé l’ensemble du programme en respectant le *coding style* fourni par Java. Nous avons également pris le soin de commenter le code où cela était nécessaire. Afin de pouvoir être interprété par le plus grand nombre de personnes, l’ensemble du programme a été implémenté en anglais. Comme demandé dans le sujet, le principe d’encapsulation est respecté afin de garantir l’intégrité des objets manipulés.

Afin de structurer au mieux notre projet, nous avons réalisé un diagramme UML représentant chacune de nos classes et les liens qui les unissent entre elles. Ce diagramme a été mis à jour tout au long de l’avancement dans le sujet. Pour une meilleure visibilité, il est consultable dans les annexes du rapport.

Concernant la structure du code, nous avons fait en sorte de rendre ce dernier le plus générique possible. Pour cela, nous avons utilisé des interfaces, des classes abstraites ainsi que l’usage du polymorphisme lors de l’héritage.

Dans un premier temps, afin de gérer la partie calculatoire de l’ensemble des jeux, nous avons créé une interface Backend avec seulement deux méthodes. Ceci afin d’encapsuler au maximum cette partie du projet pour que le simulateur puisse l’exploiter sans se soucier de son contenu. Les deux méthodes présentes, reInit() et step(), sont celles exploitées par le GUISimulator lors de l’appui sur les boutons de l’interface graphique. Ainsi, l’interface Backend est réalisée par seulement trois classes : pour les balles, pour les boids ainsi que pour les trois jeux sur grille. Chacune de ces trois classes possède un ou plusieurs attributs permettant de réinitialiser le jeu dans sa configuration initiale lors de l’appel de la fonction reInit(). Ces trois classes sont composés d’une liste d’éléments (Points, Boids, Cell) spécifiques à chacun des jeux.

Concernant les boids et les cellules composant la grille, nous avons fait le choix d’en faire des classes abstraites. En effet, cela nous permet d’avoir un niveau d’abstraction modifiable selon nos besoins. Pour les cellules, nous nous sommes rapidement rendus compte qu’elles possèdent toutes la même structure quelque soit le jeu, c’est-à-dire : des coordonnées et un état. La seule différence entre les jeux de grille est le comportement de la cellule à l’état suivant. C’est donc la méthode nextState() qui est spécifique pour chacun des jeux. Nous avons donc décidé de mettre en place le polymorphisme afin de gérer au mieux les cellules. Pour cela, la grille gérant le jeu ne manipule que des objets de type Cell sans se soucier de leur spécificité. Trois classes filles implémentent donc la classe Cell pour chacun des jeux : jeu de la vie, jeu de l’immigration et modèle de Schelling. Ainsi, de part le polymorphisme, lorsque la grille utilise la méthode nextState() afin de connaître le prochain état d’une cellule, la méthode exécutée sera une de celles présente dans les trois classes filles.

Du fait que l’état des cellules à l’étape dépend de l’état des cellules à l’étape , dans la méthode step() de la grille, les futurs états des cellules sont récupérés dans un premier temps. Dans une logique d’optimisation logicielle, nous avons fait en sorte que seules les cellules modifiées soient mises à jour. À l’issue de l’ensemble des calculs pour chacune des cellules, les nouveaux états sont appliqués sur les cellules. Concernant, les algorithmes de calculs d’états suivant pour une cellule donnée, il sera laissé au lecteur de ce rapport la compréhension de ceux-ci directement dans le code, commenté au préalable.

Concernant la classe CellSchelling utilisée pour les cellules du modèle de Schelling, quelques explications d’implémentation sont nécessaires. Afin de gérer les logements vacants, nous avons décidé de stocker ceux-ci de façon statique dans la classe. Ainsi, lors d’une même étape dans le jeu, chacune des cellules modifie ou non cette liste et il ne peut y avoir de conflits d’utilisation d’un même logement vacant par exemple. Nous avons choisi d’utiliser un ArrayList ayant pour éléments uniquement des tableaux unidimensionnels représentant les coordonnées de la cellule. En effet, cette liste étant statique, nous avons fait le choix de ne pas directement mettre les cellules au sein de la structure afin que l’utilisateur ne puisse pas les modifier manuellement. La structure de données ArrayList, nous est apparue comme la plus efficace pour gérer la liste des logements vacants. Le coût d’accès à une donnée est en et l’insertion en queue possède une complexité similaire. De plus, la classe ArrayList dispose, depuis Java 8, d’un prédicat permettant de faire une suppression conditionnelle sur tous les éléments de la liste. Ainsi, lorsqu’une cellule passe d’un état vacant à un état habité, ce prédicat est utilisé pour supprimer cette dernière de la liste des logements vacants.

# Annexes

#TODO : mettre le diagramme UML