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 quand cela était nécessaire. Une documentation Javadoc a été générée et est consultable dans le répertoire ./doc\_project/index.html. Afin de pouvoir être interprété par le plus grand nombre de personnes, l’ensemble du programme a été écrit 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 et de l’exécution d’évènement. 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 attributs sont des ArrayList qui est la structure de données la plus adaptée pour stocker les éléments initiaux du jeu. En effet, lors de la réinitialisation, les attributs étant immuables, seuls des accès à ces données est fait et le coût d’accès est en pour un ArrayList. Enfin, ces trois classes sont composées d’une liste d’éléments (Points, Boids, Cell) représentant l’état actuel de 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 quel que 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 par 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. Ceci est particulièrement utile dans le cas du modèle de Schelling où deux cellules peuvent avoir un changement d’état au même moment : l’une devient vacante tandis que l’autre devient habitée. 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 une Queue ayant pour éléments uniquement des tableaux unidimensionnels représentant les coordonnées de la cellule. En effet, cette liste doublement chaînée 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 Queue nous est apparue comme la plus efficace pour gérer la liste des logements vacants. La fonction remove() permet de supprimer tout en récupérant la tête de la liste en et l’insertion en queue possède une complexité équivalente. De plus, la classe Queue dispose, depuis Java 8, d’un prédicat permettant de faire une suppression conditionnelle sur tous les éléments de la liste chaînée. 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. Enfin pour la gestion des couleurs des logements habités, nous avons attribué à chaque couleur un état différent. Ainsi, toutes les couleurs sont générées aléatoirement et stockées dans une liste statique aux indices correspondant au numéro de l’état.

Toujours dans la partie calculatoire du projet, nous allons maintenant traiter le cas des boids. Tout d’abord, ces derniers étant composés de vecteurs, nous avons implémenté une classe Vector permettant de les manipuler aisément. Cette classe permet de réaliser différentes opérations arithmétiques sur les vecteurs.

Les cellules des trois jeux précédents sont représentées sur une grille tandis que les boids se déplacent dans le plan. Nous avons donc dû créer une nouvelle classe réalisant l’interface backend afin de gérer l’ensemble des boids à chaque étape de la simulation. Similairement à ce que nous avions fait pour les cellules, nous avons créé une première classe abstraite de boids permettant d’appliquer des règles communes à tous les boids. Puis, nous avons implémenté des classes filles permettant d’avoir des groupes de boids aux comportements différents. Parmi les règles communes à l’ensemble des boids, nous avons fait le choix de fixer les règles suivantes :

* Chaque boids doit rester au sein-même de la zone graphique et doit rebondir sur le mur.
* Les boids d’un même groupe doivent se séparer s’ils sont trop proches les uns des autres afin de ne pas se superposer.
* Une force simulant du vent est appliquée à chacun des boids afin de les mettre en mouvement perpétuel.

Comme précédemment, pour chacune des règles appliquées sur les boids, il sera laissé au lecteur de ce rapport la compréhension de celles-ci directement dans le code, commenté au préalable. La partie sur les évènements sera détaillée dans la suite du rapport. Néanmoins, il convient d’expliquer le choix de l’attribut typeOfBoids de la partie backend. Chaque type de boids possède un pas de temps spécifique. Afin que le système de gestion des évènements puisse connaître ce pas de temps, il est nécessaire de lui communiquer sa valeur pour un type donné. Pour implémenter cela, nous avons donc utilisé une HashMap composé d’une clé, le type de boids, et d’une valeur, le pas de temps. Ainsi, lors de l’initialisation de tous les boids, la structure de données HashMap contient une liste exhaustive des types de boids utilisés. L’utilisation d’une HashMap permet de conserver l’unicité de la valeur du pas de temps pour un type de boids donné.

Parlons désormais de la partie graphique qui gère l’affichage de l’ensemble des jeux. Toujours dans un principe d’encapsulation, nous avons choisi de créer une classe abstraite Simulator qui réalise l’interface gui.Simulable. Elle définit les méthodes next() et restart() de l’interface réalisée, elle déclare la méthode de dessin principale draw() et elle fait le lien entre le GUISimulator, l’EventManager et le Backend en permettant d’accéder à chacun.

Pour chaque type de jeu, une classe est créée et hérite de Simulator : pour les balles, on a BallsSimulator ; pour la grille, on a GridSimulator ; pour les boids, on a BoidsSimulator. Chacune de ces classes doit initialiser un Backend, les évènements principaux (on en reparlera plus bas) et définir la méthode draw() qui dessine le contenu de la fenêtre considéré vide à l’appel. Pour les balles, on utilise le Backend pour récupérer la position des balles et ajouter un gui.GraphicalElement “Oval” au GUISimulator pour chacune. Pour la grille, on fait la même chose avec Rectangle. Pour les boids, nous avons créé notre propre GraphicalElement nommé Triangle. Ce dernier réalise l’interface GraphicalElement, permet de dessiner un triangle isocèle par la méthode paint(Graphics2D g2d) et est créé en prenant en paramètre deux Vector correspondant aux coordonnées de la base et de la pointe du triangle. Un autre constructeur plus adapté aux boids est disponible, prenant en paramètre la hauteur du triangle et les coordonnées de la pointe du triangle.

Concernant le gestionnaire d’évènements EventManager, ce dernier doit pouvoir gérer l’ajout et la suppression d’évènements pendant le parcours de ces derniers. Pour cela, nous avons quatre listes d’évènements de type ArrayList<Event> : eventList pour la liste des évènements à exécuter à leur date respective, eventsInit pour la liste des évènements initiaux initialisée lors de la première itération et qui permet de réinitialiserla simulation, eventsToAdd pour la liste des évènements qu’on ajoute, et eventsToRemove pour la liste des évènements qu’on supprime. La méthode next() ajoute à eventList les évènements de eventsToAdd, parcourt eventList et exécute les évènements dont la date est passée, puis enlève ces derniers avant d’incrémenter la date de l’EventManager. La méthode restart() réinitialise les évènements en les remplaçant par les évènements initiaux.

Nous avons un évènement principal SimulatorEvent, qui hérite de Event, faisant le lien entre le GUISimulator, le Backend, le Simulator et l’EventManager. Il nécessite d’être initialisé avec le Simulator, et sa méthode execute() appelle dans l’ordre la méthode reset() du GUISimulator, la méthode step() du Backend, la méthode draw() du Simulator, et la méthode addEvent(Event e)de l’EventManager en s’ajoutant lui-même afin d’avoir une animation perpétuelle.

Maintenant que nous avons parlé de toute la structure, comment peut-on créer une simulation ? Il suffit simplement d’initialiser un GUISimulator, d’initialiser un Simulator (choix entre BallsSimulator, GridSimulator et BoidsSimulator) puis d’exécuter la méthode setSimulable prenant en paramètre notre Simulator créé. Un test est disponible pour chaque jeu réalisé : TestBallsSimulator pour des balles qui rebondissent, TestGameOfLife pour le jeu de la vie, TestImmigration pour le jeu de l’immigration, TestSchelling pour le modèle de Schelling, et enfin TestBoidsSimulator pour un simulateur de Boids. Certains de ces tests sont aléatoires, et tous ont les paramètres de la simulation modifiables.

[La façon dont on exécute le tout dans un test]

Pour conclure, ce projet nous a permis de découvrir et renforcer nos connaissances dans la programmation orientée-objet. Nous avons également pris conscience de l’importance de la phase de réflexion en implémentant notamment le diagramme de classes. Ce dernier permet d’avoir un visuel sur la structure complète du code et ainsi de prendre du recul sur les problèmes rencontrés. Néanmoins, comme tout projet informatique, des axes d’amélioration sont envisageables. Par exemple, lors de l’application des règles sur les boids, la liste complète des boids est parcourue à chaque étape. Pour pallier cela, il est possible de faire une optimisation algorithmique en ne comparant que les boids suffisamment proches à l’aide d’un algorithme récursif « diviser pour mieux régner ». Enfin, un autre choix d’implémentation aurait pu être fait pour la gestion des logements vacants. Il aurait été en effet possible d’utiliser une structure ArrayList pour les stocker. Le nombre de logements vacants étant constant, il aurait suffi de modifier un élément de la structure pour un indiquer le nouvel emplacement d’un logement vacant, une opération en . Cela nous amène à une complexité équivalente à celle présente dans notre projet avec une liste chaînée.

**#TODO** : ajouter le pdf du diagramme UML lors de la génération du rapport