Projet proies-prédateurs

Léo Creuse

6 décembre 2017

Table des matières

1 Introduction

Ce projet entre dans le cadre de l'enseignement de la programmation orientée objet en deuxième année. Le langage de programmation utilisé est donc java 1.8. Le but du projet est d'implémenter une simulation d'un système proies-prédateurs, selon les consignes disponibles sur la page LMS du cours associé. Le projet se découpe donc en quatre grandes parties, chacune ayant pour but de mettre en place des fonctionnalités spécifiques. Il est important de noter à ce stade que certaines fonctionnalités sont encore expérimentales, voire non testés, cela sera précisé à chaque fois dans la présente notice, ainsi que les raisons ayant empêché la finalisation de ces fonctionnalités.

La présente notice a pour but d'expliquer les choix effectués lors de la conception du code, les classes utilisées, et les autres spécificités du langage java utilisées. Le projet s'est déroulé en trois phases : une phase de conception "sur papier" de la simulation : l'ensemble des classes, et le liens entre elles ont été définis, donnant lieux à des diagrammes UML préliminaires, puis une phase d'implémentation, où certaines modifications ont dues être apportées aux classes définies à partir du diagramme UML initial, puis enfin une phase de synthèse pour déterminer s'il y avait des redondances, ou des approches peu efficaces facilement corrigibles dans la simulation.

Il est important de noter que l'ensemble du fonctionnement du code ne sera pas détaillé dans cette notice, mais des informations complémentaires se trouvent dans les commentaires du code.

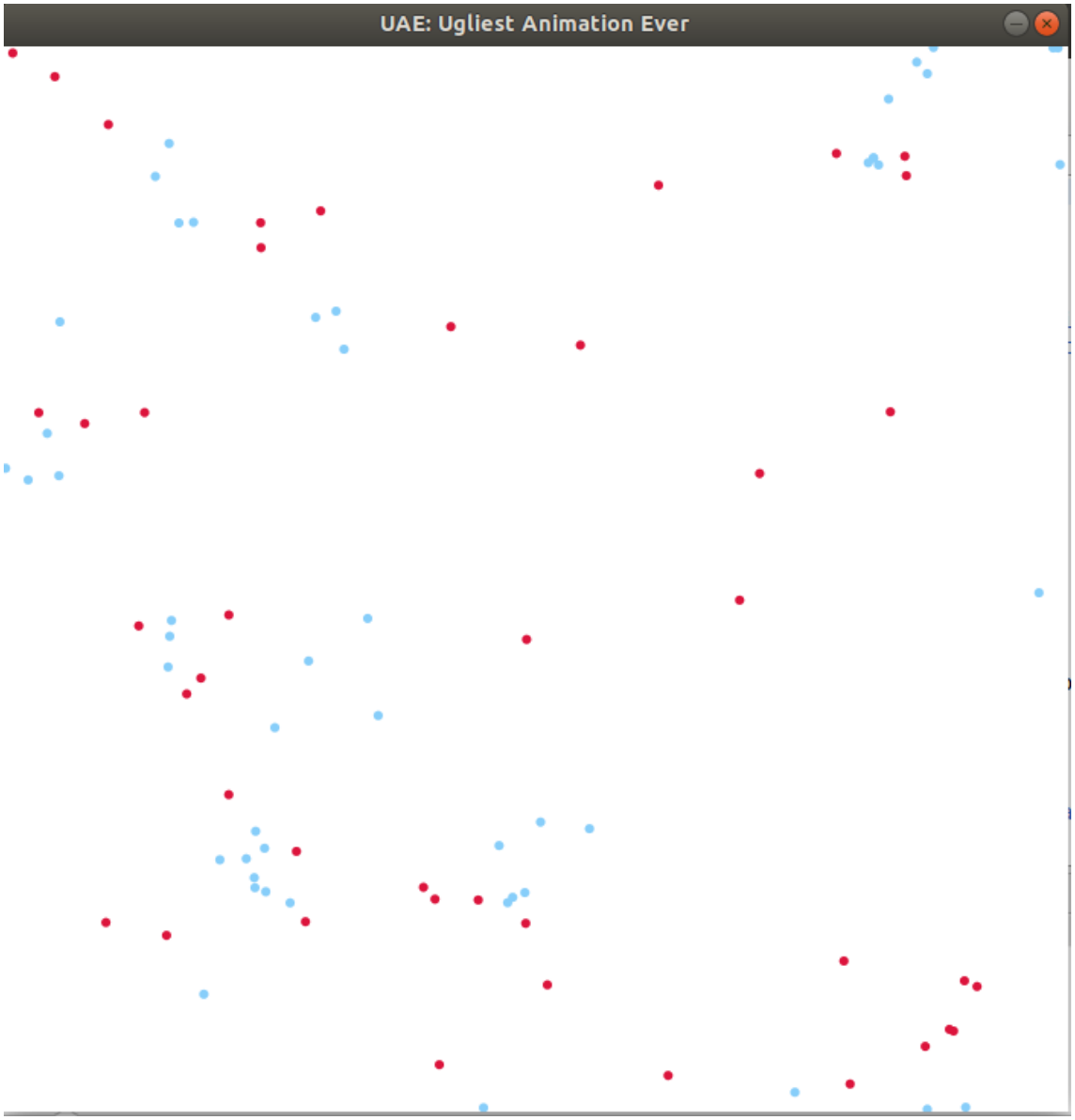

FIGURE 1 – Vue d'une exécution de la simulation

2 Architecture de la simulation

2.1 Contraintes

L'architecture de la simulation était premièrement soumise à quelques contraintes, puisque certains fichiers de base étaient fournis, et non modifiables. Ainsi, la simulation devait renvoyer des éléments de type `Circle`. D'autre part, le serveur simple étant aussi fourni, et non modifiable, il fallait que le client d'interconnexion possède certains flux de données particuliers, notamment une connexion en TCP. Enfin, il fallait que le code s'exécute suffisamment rapidement pour que l'animation soit agréable, voir compréhensible lors qu'elle était regardée.

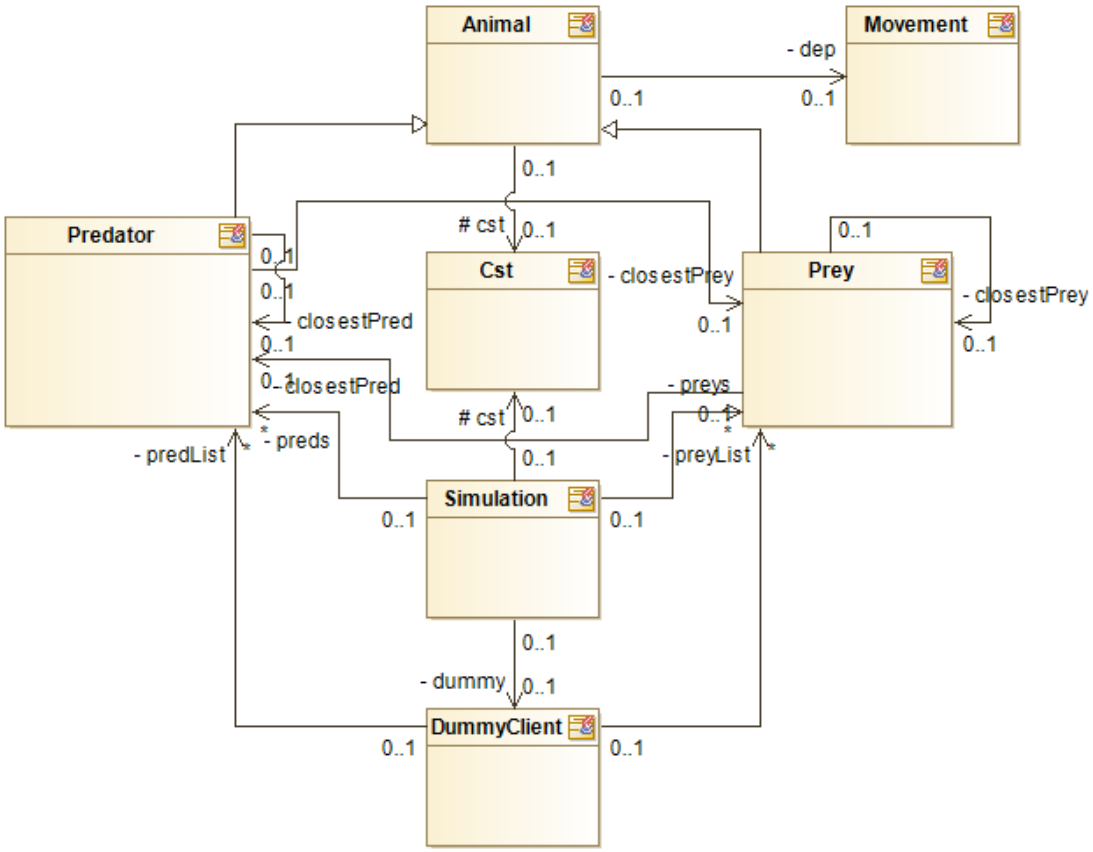

FIGURE 2 – Diagramme de classe global de la simulation

2.2 Structure globale

La structure globale de la simulation est représentée dans la figure 2 page 4. Le principe de cette simulation est que la classes `GraphicalDisplay` crée une instance de `Simulation`, et appelle périodiquement une méthode `update()` de simulation afin d'effectuer un pas de simulation, et récupère l'ensemble des éléments à afficher via la méthode `getElement()`, devant retourner une liste de `Circle`.

J'ai donc choisi pour se faire de créer une classe `Animal`, héritant de la classe `Circle` fournie par javafx pour obtenir une représentation générale de ce que pourrait être un animal évoluant au sein de la simulation, et deux classes `Prey` et `Predator` pour représenter les proies et les prédateurs plus spécifiquement. Ces deux classes héritent de `Animal`. On retrouve donc dans `Simulation` deux listes, une de proies, et une de prédateurs, pour représenter les créatures présentes dans la simulation à l'étape courante. Le diagramme possède une classe `DummyClient`, qui ne sert qu'à gérer les sorties et entrées des créatures aux frontières de la simulation, et permet de garder un code extrêmement proche lors du passage à un client d'interconnexion. Cette classes sert donc de "client fictif" d'où le nom de la classe. Enfin, une classe `Cst` stocke l'ensemble des constantes du problème, telles que définies au début de l'exécution de la simulation. L'utilisation d'une classe à part me permet de plus simplement retrouver le fichier à modifier pour régler la stabilité du problème, et même d'appliquer un algorithme du gradient pour tenter de trouver

les paramètres optimaux.

Le déroulement d'une étape de simulation sera détaillé dans la section consacrée à l'explication de la classe `Simulation`.

3 Les classes implémentées

Les diagrammes de classes présentés ici sont les diagrammes des classes dans leur état le plus amélioré, cependant les différentes versions (version de base, version avec interconnexion simple client, version avec interconnexion multi-client, version avancée) seront rendues avec cette notice.

Les liens vers les classes prédéfinies dans java (ou plus généralement extérieures au package du projet) ne sont pas représentées, cela apparait sous forme de `<no type>`, car les diagrammes finaux on été générés à partir du code java, celui-ci ayant beaucoup été modifier depuis le diagramme UML de base. Quand le type de retour n'est pas simple à déterminer (par ex la fonction `toString()`) le type de retour sera précisé.

3.1 La classe `Animal`

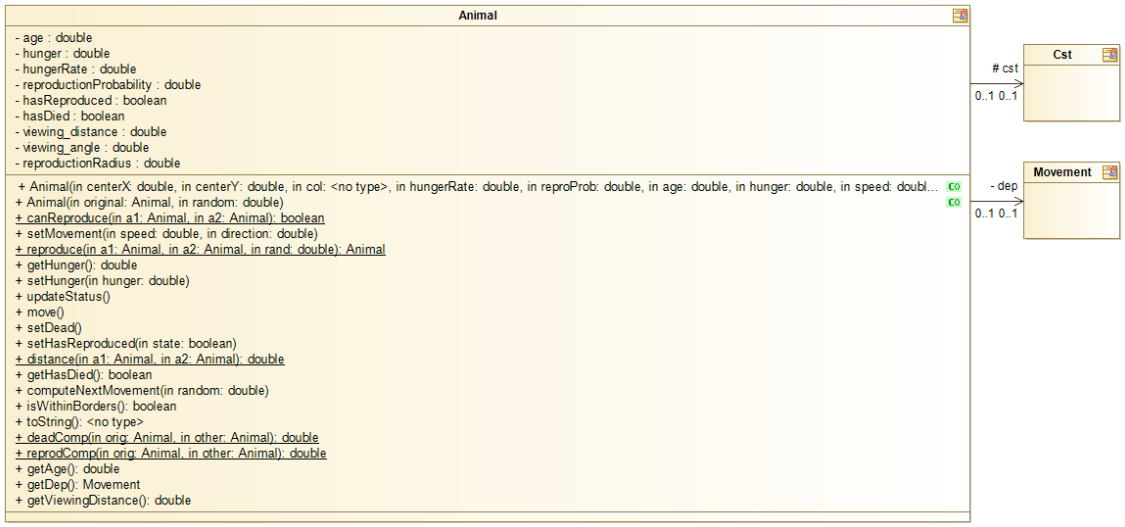

FIGURE 3 – Diagramme de la classe `Animal`

Cette classe représente les créatures les plus générales pouvant se trouver dans l'espace de simulation. Son diagramme de classe est représenté figure 3 page 5. L'héritage depuis la classe `Circle` de javafx n'est pas ici représentée, mais est néanmoins essentielle, car ce sont les attributs `CenterX` et `CenterY` de `Circle` qui définiront la position de l'animal. Les autres attributs comprennent notamment l'age et la faim, augmentant monotonnement, et entrainant la mort de l'animal si ces attributs atteignent une valeur limite ; un taux de faim, un déplacement, de classe `Movement` qui stocke les information de vitesse et direction de l'animal, des constantes telles que le rayon de reproduction, des paramètres de champ de vision (distance et angle), ainsi que des attributs servant à suivre l'état de l'animal au cours d'une étape.

Les méthodes disponibles comprennent un constructeur explicite, un constructeur "à bébé", servant à dupliquer un animal, mais ayant un age nul, et une faim nulle, des méthodes générales pour modifier, ou accéder à certains attributs, des méthodes de modification de statut de l'animal (faim, age, stratégie de déplacement, état de reproduction, vie ou mort), ainsi qu'un certain nombre de méthodes statiques, permettant la reproduction entre deux animaux, de trouver diverses distances selon des critères particuliers : `distance` donne la distance euclidienne entre deux animaux ; `deadComp(Animal ordig, Animal other)` renvoie la distance entre deux animaux en prenant en compte le champ de vision, et la vie ou non de l'autre animal : si l'animal passé en paramètre `other` est mort, ou n'est pas dans son champ de vision, alors la distance est supérieure à deux fois la taille de la simulation, permettant de trouver facilement l'animal viviant le plus proche dans le champ de vision. La fonction `reprodComp()` remplie la même fonction, mais prend en compte l'état de reproduction de l'autre l'animal.

3.2 La classe `Predator`

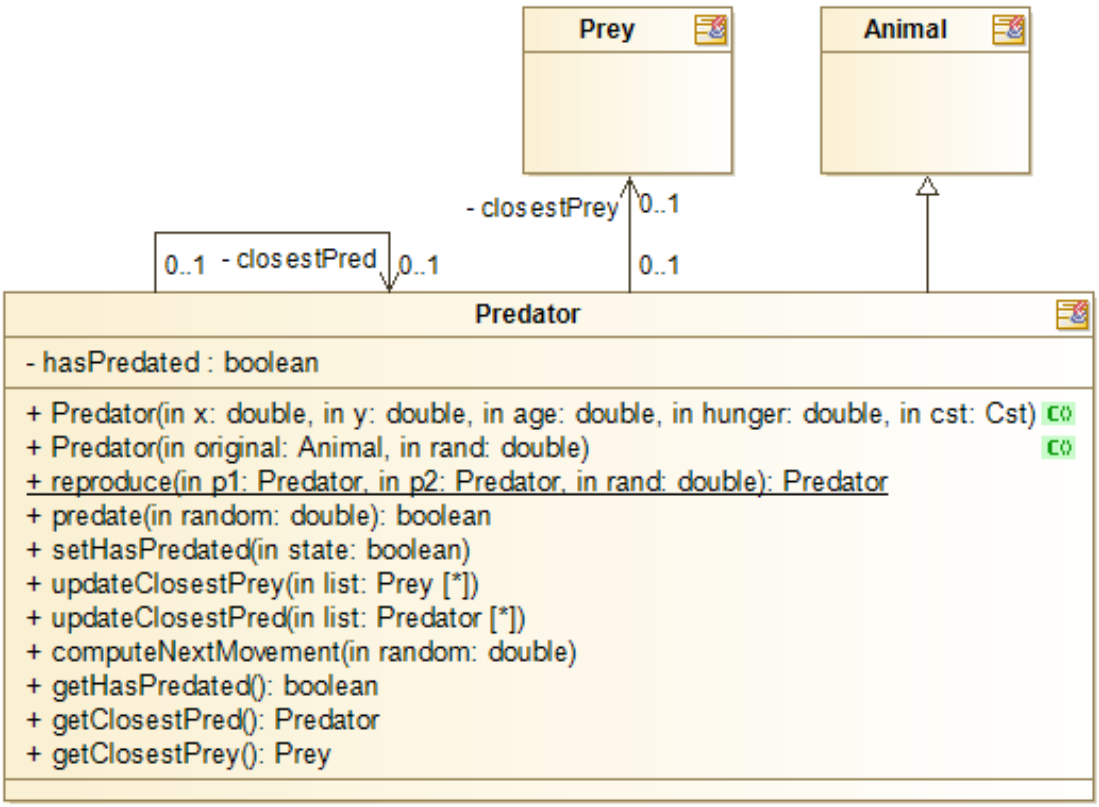

FIGURE 4 – Diagramme de la classe `Predator`

La classe `Predator` hérite de la classe `Animal`, et sert à spécifier le comportement des animaux en tant que prédateurs. Elle est représentée figure 4, page 6. C'est pourquoi, cette classe redéfinit un certain nombre de de méthode existantes dans animal, afin de spécifier le comportement propre aux prédateurs (notamment,

initialisation d'un certain nombre d'attributs avec les constantes propres aux prédateurs). Cependant, comme les prédateurs peuvent effectuer des action en plus, certaines méthodes supplémentaires ont été écrites : la méthode `predate()` permet à l'animal de réinitialiser sa faim, en tuant la proie la plus proche si possible, et si cette proie existe. La méthode `computeNextMovement()` met à jour la stratégie de déplacement du prédateur selon cette règle : La vitesse est proportionnelle à la faim du prédateur (coefficient négatif, plus le prédateur a faim, moins il va vite) ; si il y a une proie dans le champ de vision ($\pi/3$ actuellement) alors le prédateur sprinte (vitesse doublée) vers elle, sinon il se dirige en ligne droite à vitesse normale.

Pour pouvoir sélectionner la proie à chasser, et le prédateur avec qui se reproduire, chaque prédateur possède comme attribut la proie et le prédateur le plus proche de lui, au sens du champ de vision pour la proie et de la reproduction pour le prédateur (cf classe `Animal`). Ceux-ci sont déterminés en utilisant les algorithmes de tri génériques offerts par l'API java 8, en utilisant comme fonction d'ordre les fonction `deadComp()` et `reprodComp()` définies dans la classe `Animal`.

Dans cette implémentation du problème chaque prédateur se reproduit et mange au plus une fois par étape de simulation.

3.3 La classe `Prey`

La classe `Prey` est le pendant de la classe `Predator` pour les proies. Les fonctions y sont similaires, si ce n'est que la stratégie de déplacement consiste en fuir en sprintant le prédateur dans son champ de vision (π actuellement), sinon de se déplacer dans une direction aléatoire à vitesse normale. Son diagramme de classe est représenté figure 5 page 7

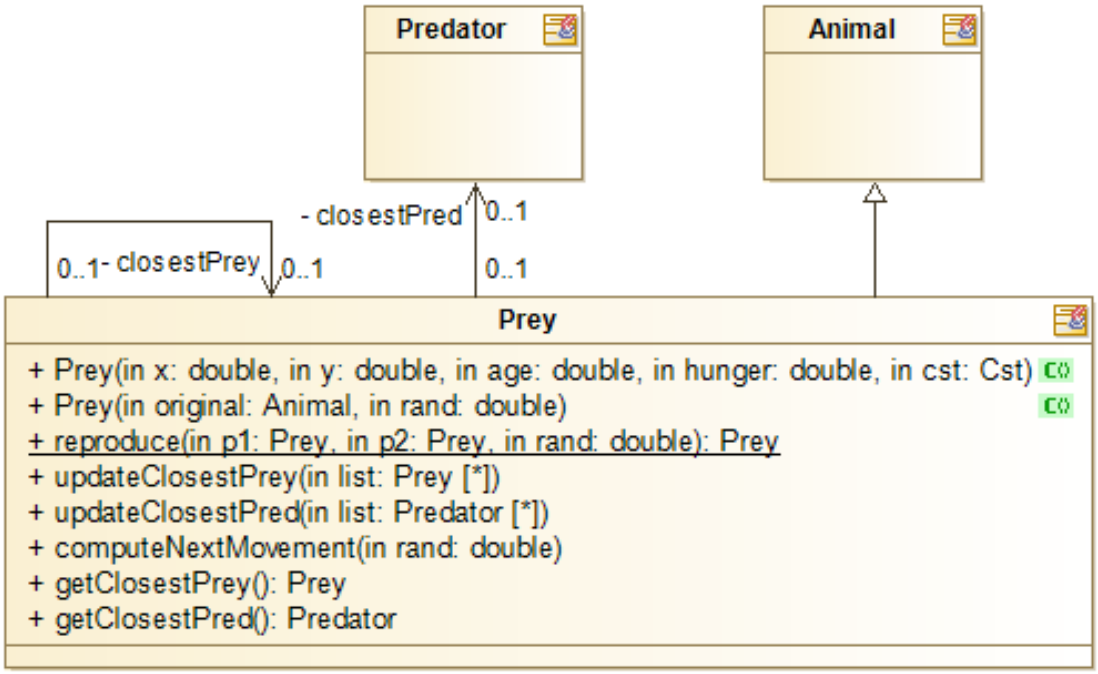

FIGURE 5 – Diagramme de la classe `Prey`

3.4 La classe `Simulation`

La classe `Simulation` contient l'ensemble des méthodes et attributs nécessaires à l'itération d'une étape de simulation. Son diagramme de classe est représenté figure 6, page 8.

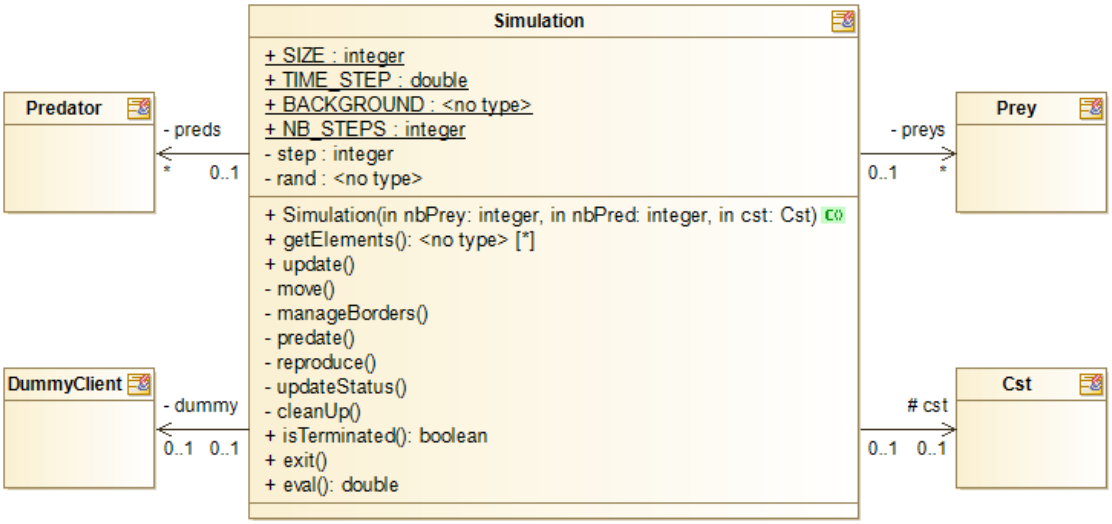

FIGURE 6 – Diagrammme de la classe `Simulation`

Lors de l'exécution de la simulation, seule une instance de la classe doit être créée. Le constructeur de cette classe prend en paramètre le nombre de proies et prédateurs à créer, uniformément répartis, et une instance de la classe `Cst`, servant à donner l'ensemble des constantes du problème, sauf la taille de la simulation, le taux de rafraîchissement de la simulation et le nombre d'étapes de simulation, gardés dans la classe pour ne pas modifier la classe `GraphicalDisplay`. Les différentes méthodes ont des noms relativement explicites quand à leur but : `move()` déplace l'ensemble des animaux, en mettant à jour la stratégie de déplacement, `manageBorders()` s'occupe de récupérer les animaux hors-simulation, et de les envoyer au client (factice ou non) pour que les échanges avec le serveur soient faits. Vient ensuite la méthode `predate()` qui donne l'opportunité à tous les prédateurs de manger, puis `reproduce()` permet à tous les animaux de se reproduire si possible. Ensuite `updateStatus()` met à jour toutes les informations d'age et de faim, et enfin `cleanUp` se charge de retirer l'ensemble des animaux morts au cours de l'étape des deux listes `preys` et `preds` de proies et prédateurs respectivement. La méthode `getElements` concatène ces deux listes sous forme de liste de `Circle` et les renvoie, pour affichage graphique. Enfin, la méthode `eval()` est une fonction d'évaluation de la stabilité de la simulation sur 480 cycles (20s à 24 images par secondes).

Les autres attributs de cette classe sont `rand` de type `Random`, pour générer l'ensemble des nombres aléatoires nécessaires à la simulation (direction des proies, probabilités de reproduction, direction des nouveaux-nés), et `dummy` ou `client` servant à la gestion des frontières.

3.5 La classe `Cst`

La classe `Cst` sert à contenir l'ensemble des constantes du problème. Son diagramme de classe est représenté figure 7 page 9. Les attributs sont tous nommés de manière explicite et ont comme seule propriété `final` afin de pouvoir être initialisés lors de l'exécution : la classe possède deux constructeurs, un constructeur explicite permettant de définir l'ensemble des constantes lors de l'exécution, et un constructeur par défaut, reprenant les paramètres donnant la simulation la plus stable jusqu'à présent.

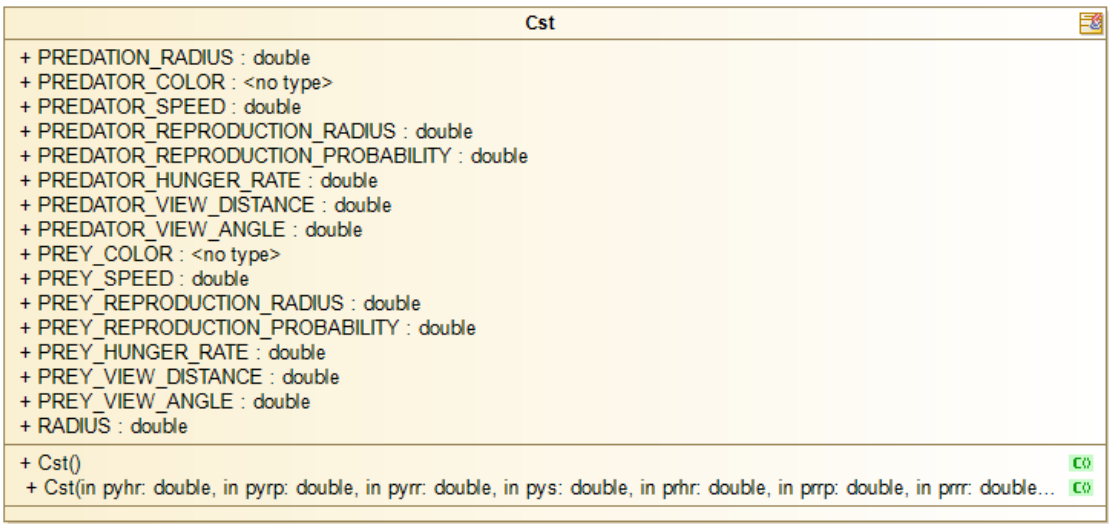

FIGURE 7 – Diagramme de la classe `cst`

3.6 La classe `DummyClient`

La classe `DummyClient` sert de "faux client" afin de minimiser la modification du code de simulation lors du passage au client d'interconnexion. Son diagramme de classe est présenté dans la figure 8 page 10. Son fonctionnement consiste à stocker les proies et prédateurs qui sortent de la simulation, via les méthodes `addPrey()` et `addPred()`, calculer les nouvelles coordonnées pour chaque proie, ou prédateur, puis les renvoie via les méthodes `getPreys()` et `getPreds()`.

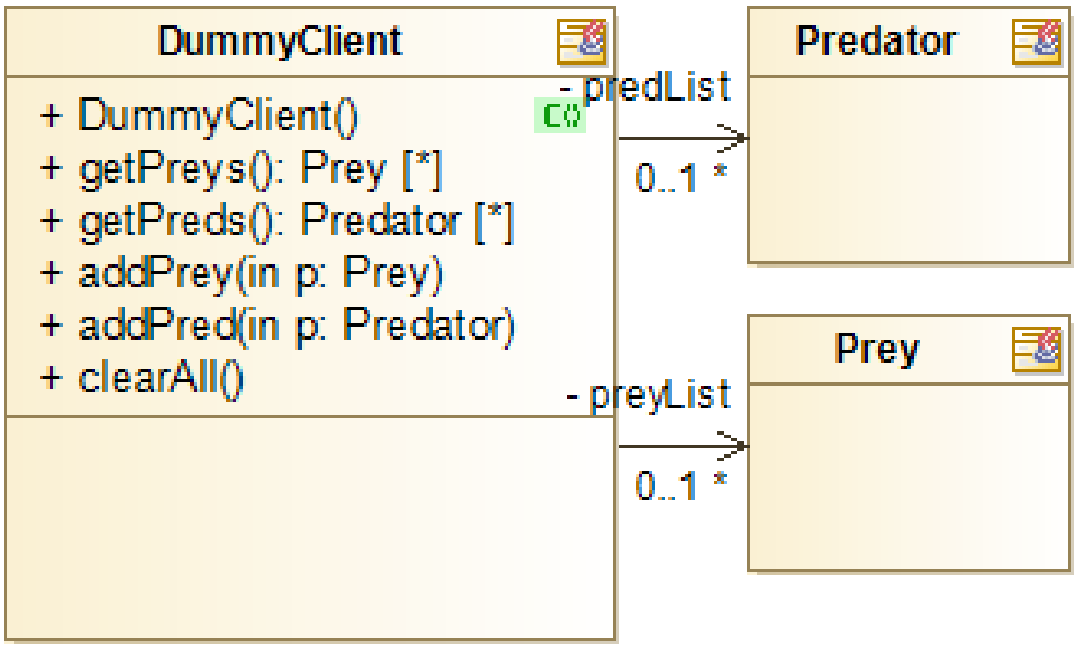

FIGURE 8 – Diagramme de la classe `DummyClient`

3.7 La classe `InterconnectionClient`

La classe `InterconnectionClient` permet la communication avec un serveur pour gérer la sortie et l'entrée d'animaux dans la simulation. Elle est représentée figure 9 page 11.

Cette classe interagit de la même manière avec la classe `Simulation` (d'ailleurs, une interface aurait pu être réalisée à cette fin). Le client utilise le même protocole de communication pour un `SimpleServer` ou un `MultiServer`, en suivant le protocole définit dans le cahier des charges de la simulation. Cette classe effectue une opération de mise à l'échelle entre 0 et 1 des coordonnées, pour pouvoir fonctionner en accord avec les serveurs.

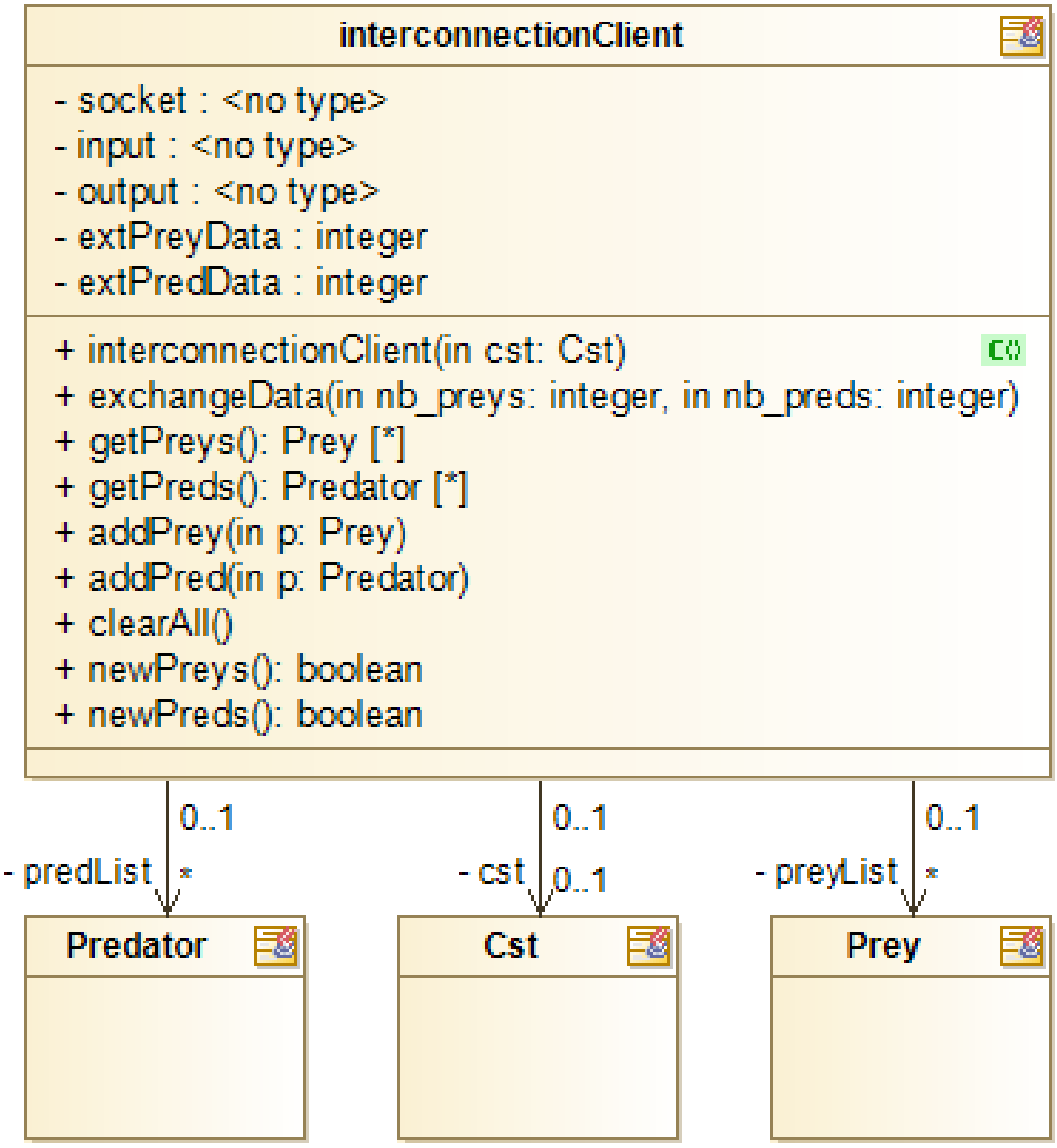

FIGURE 9 – Diagramme de la classe `InterconnectionClient`

3.8 La classe `MultiServer`

Cette classe représente un serveur capable d'inter-connecter un nombre carré de clients. Son diagramme est représenté en figure 10 page 12.

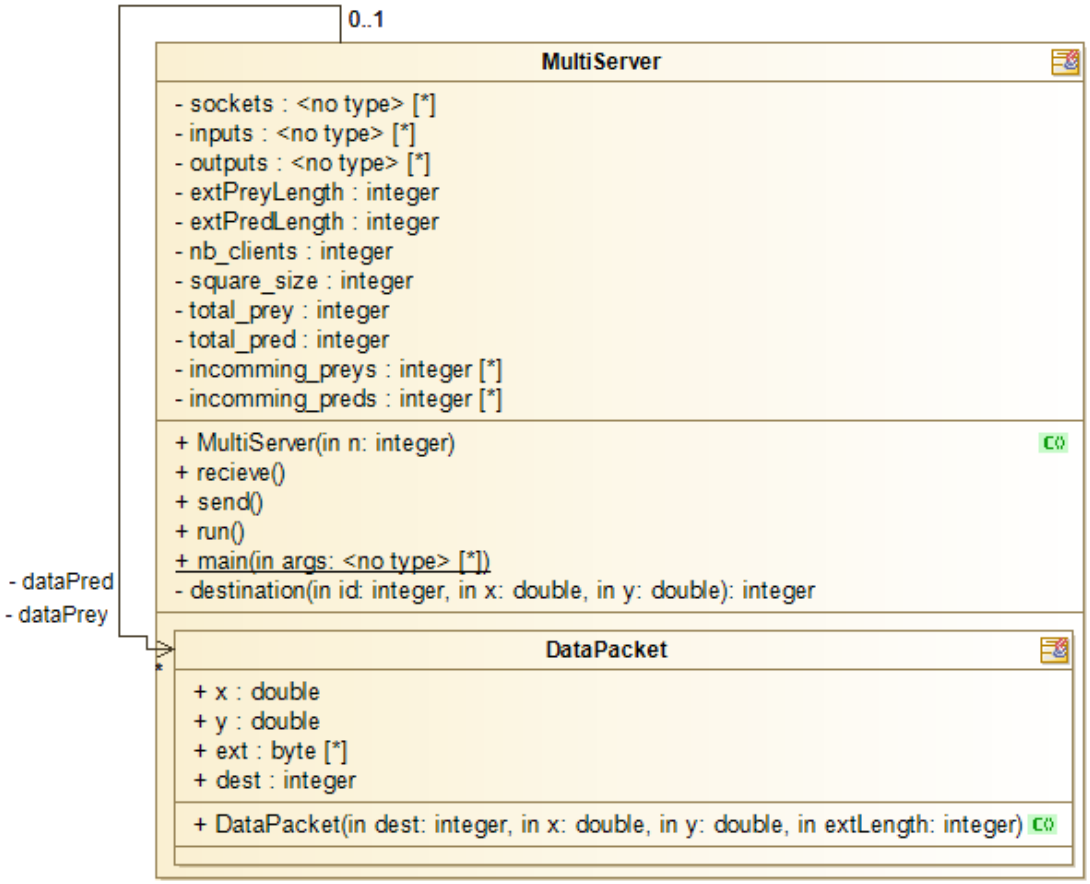

FIGURE 10 – Digramme de la classe `MultiServer`

Ce serveur comporte une classe interne, `Packet` qui lui permet de plus simplement gérer les flux de données entre les différents clients. Par nature du traitement des données par le serveur, comme le serveur traite d'abord l'ensemble des animaux entrants, puis ensuite seulement renvoie les animaux aux bon clients, et les fonction de lecture de données étant bloquantes, cela synchronise l'ensemble des clients à chaque cycle.

4 Les différentes simulations codées

Le code rendu comporte quatre packages, pour les différentes améliorations apportées.

4.1 Simulation de base

Cette version de la simulation ne comporte pas d'améliorations notable, mais peut être une limitation de l'age minimum de reproduction à 2 secondes, pour limiter les évolutions exponentielles. Cette version de la simulation est "stable" (pas d'extinctions, ni de reproduction incontrôlées) sur au moins 480 cycles (20 sec à 24ips).

4.2 Simulation avec `SimpleServer`

Cette version intègre uniquement le serveur simple en plus. Le problème avec ce serveur est la lenteur des communications. En effet, il est impossible de faire fonctionner la simulation à plus de 12 Hz. Les paramètres n'ont pas été modifiés pour avoir une simulation stable dans cette configuration.

Il faut lancer le serveur avant de lancer le client, en précisant le numéro de port dans la commande si le port par défaut n'est pas désirable.

4.3 Simulation avec `MultiServer`

Cette version intègre le multi-serveur. Bien que le code ait été soigneusement conçu, il n'a pas pu être entièrement et complètement testé, car l'affichage graphique ne fonctionne pas bien avec plusieurs fenêtres ouvertes à la fois (sur mon ordinateur en tout cas), et la cadence d'images est encore ralentie, car il y a plus de trafic, proportionnel nombre de clients.

Il faut lancer d'abord le serveur, en précisant le nombre de clients se connectant (un carré, sinon le serveur quittera ; 1 par défaut) dans la ligne de commande, puis lancer les clients.

4.4 Simulation avancée

Cette version finale, n'utilise pas de serveur, afin d'avoir la performance associé au travail en local pur. Les proies fuient les prédateurs dans leur champ de vision de 180 degrés en sprintant (vitesse doublée), et les prédateurs visent la proie la plus proche dans leur champ de vision de 60 degrés sinon avancent en ligne droite. Leur vitesse est proportionnelle à leur faim. L'ensemble des animaux doivent aussi attendre 5 ans (secondes en temps réel) pour pouvoir se reproduire, et les enfants partent dans une direction aléatoire afin d'éviter les reproduction exponentielles chez les prédateurs. Cette simulation est très aléatoire en ce qui concerne la stabilité : les proies peuvent s'éteindre, puis les prédateurs aussi, les prédateurs peuvent s'éteindre, puis les proies aussi, ou alors les proies s'éteignent, et les prédateurs se multiplient indéfiniment, de même en échangeant prédateur et proies, ou la simulation reste stable sur plus de 40 secondes.

5 Un mot sur la classe `Gradient`

Cette classe a été codée afin de pouvoir trouver les paramètres de stabilité optimaux sans trop d'efforts : en effet sur la version de base de la simulation il est

possible de jouer sur 9 paramètres indépendants, et sur la version avancé 13 paramètres sont contrôlables via la classe `Cst`, mais en rajoutant l'age de mort et l'age minimal de reproduction pour proie et prédateur, le nombre de prédateurs et proies présentes au début dans la simulation, cela nous amène jusqu'à 19 paramètres, ce qui n'est pas optimisable à la main. Cette classe codée simplement mobilise les acquis de première année, et cherche à mettre en place une optimisation de la fonction d'évaluation de la classe `Simulation` par méthode de gradient à pas constant. Les résultats fournis par la méthode n'ont pas été satisfaisant pour le moment, mais cela peut s'expliquer par plusieurs raisons :

- La fonction à optimiser n'est pas déterministe, et son gradient non plus par la même occasion
- Les temps d'exécution de la fonction d'évaluation et du calcul du gradient ne permettent pas de laisser l'algorithme tourner sur un nombre suffisant d'itérations
- Le pas n'est peut être pas choisi correctement, je n'ai pas pris le temps de calculer un pas de temps correct
- les contraintes n'ont pas été implémentées sur les paramètres : certains paramètres passaient négatif, ce qui est absurde.

Ces raisons ont donc conduit à rapidement abandonner son utilisation.

6 Améliorations possibles et conclusion

Cette simulation pourrait être d'avantage enrichie, il est en effet possible de complexifier le comportement des animaux à souhait. Cependant des améliorations peuvent être apportées au code actuel. Premièrement pour rendre le code d'avantage générique, les classes `DummyClient` et `InterconnectionClient` auraient pu réaliser une interface, afin de pouvoir créer plus facilement d'autres clients, qui auraient permis notamment l'exécution de la simulation sur plusieurs cœurs, (similaire aux pratiques se faisant en simulation numérique de dynamique des fluides). D'autre part, le protocole UDP aurait pu être choisi pour la communication entre client et serveur. En effet, si les simulation fonctionnent avec beaucoup d'animaux, la perte occasionnelle de quelques créatures n'est pas grave, et cela aurait permis des communications beaucoup plus rapide, et donc des taux de rafraichissement de l'affichage proche des 24Hz (c'est ce qui se fait pour les serveurs de jeux vidéos, tournant jusqu'à 60Hz). Enfin, plutôt que de coder une méthode du gradient inefficace, et incomplet, une recherche aléatoire de paramètres aurait pu être plus efficace.

Le code fourni contient donc un mélange de simulations fonctionnelles ou non, selon différents degrés. des informations supplémentaires sur le fonctionnement de chaque classe, des algorithmes utilisés est consultable dans les commentaires du code.

Fin de la notice.