Université Lille 1

Master IAGL

IDL/Système Multi-Agents

Compte rendu de TPs
SMA

Auteurs:
Slimane El Ouadi
Sofiane Yousfi

Superviseur:
Dr. Philippe Matieu

10 Février 2016

Sommaire

Chapitre 1

Structure du projet

Afin de mieux comprendre la structure de notre projet, nous avons réalisé les diagrammes UML (*dépendance des packages et diagrammes des classes*) à l'aide d'un outil que nous avons nous même développé dans le cadre du cours de *Mr.Monperrus*.

1.1 Diagramme des paquetages (dépendances)

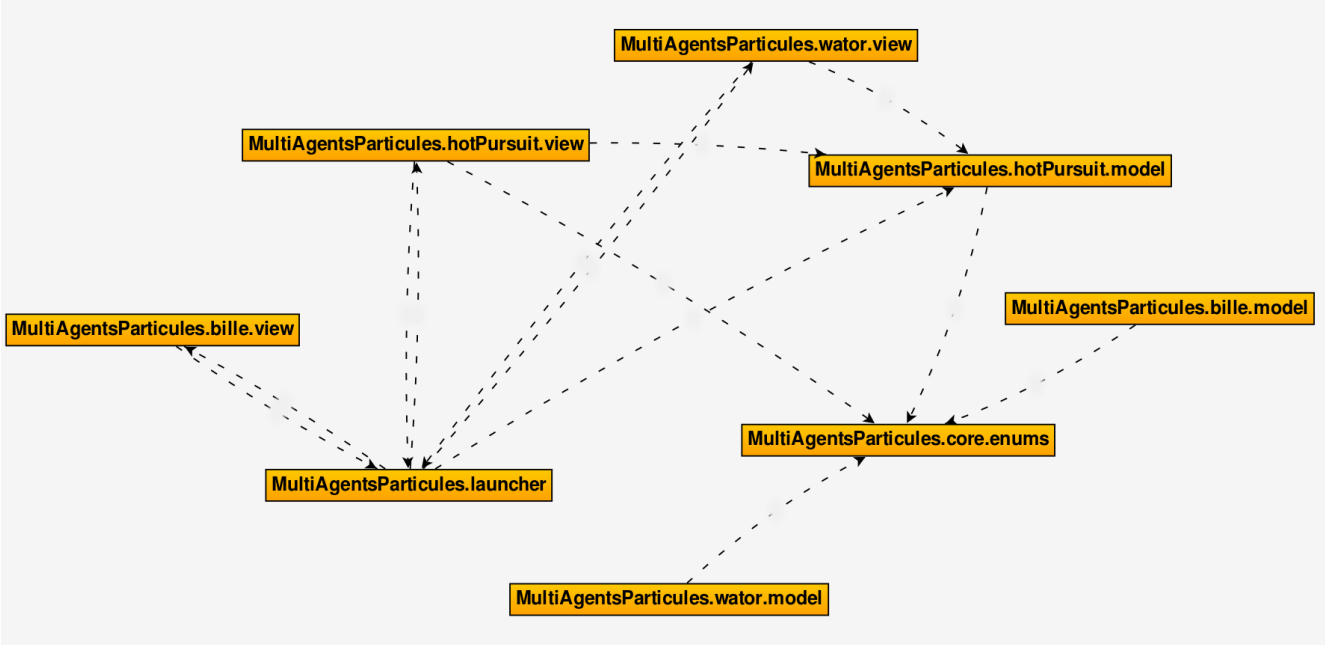

Figure 1.1: Diagramme des dependance entre les packages

La figure représente les dépendance entre tout les packages du projet. Les connexions entre les packages représentées par des flèches discontinues signifies un *import*

Le package *core* contient toutes les sources communes aux trois projets. Pour chacun des trois projets il y a un package *model* et un package *view*.

1.2 Diagramme général de classes

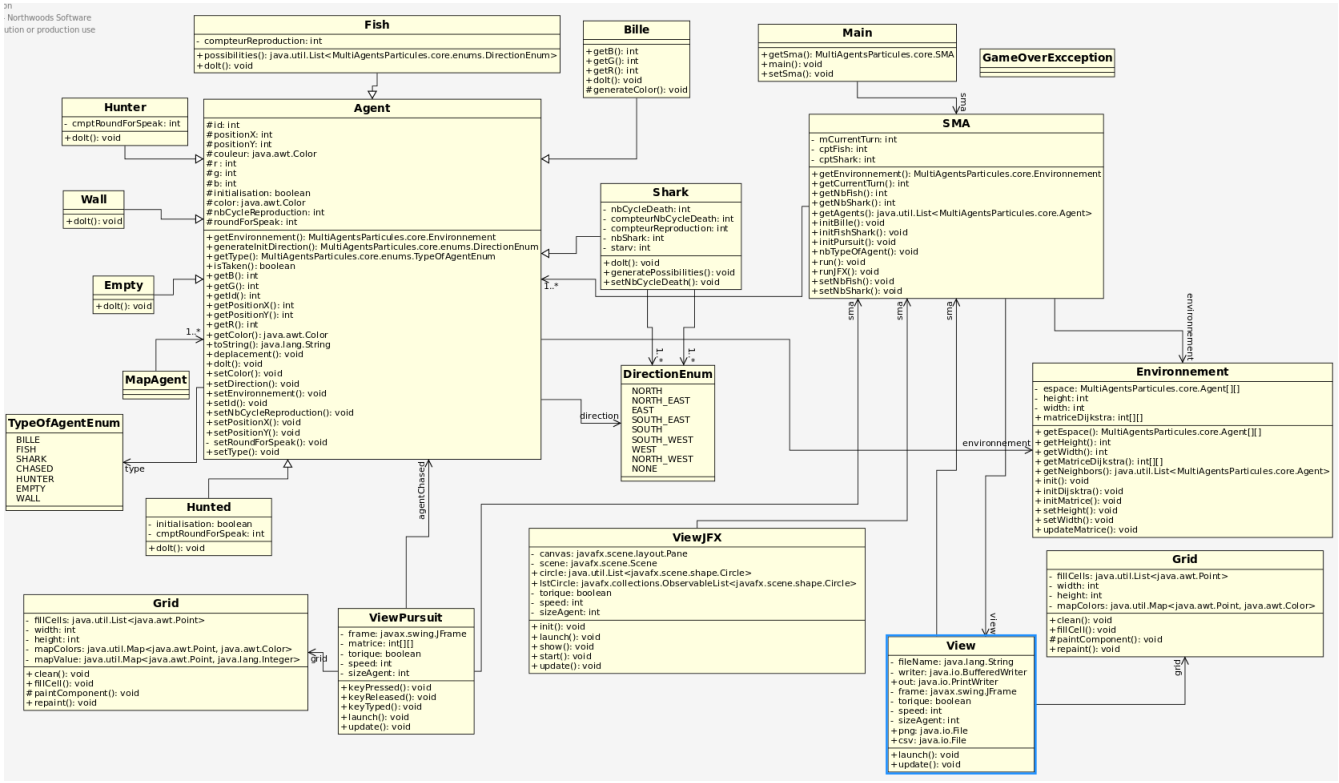

Figure 1.2: Diagramme de classes general

Les classes *Bille, Fish, Shark, Hunted, Hunted, Empty* étendent de la classe *Agent* qui rassemble les proprietés générales d'un agent. Les directions de déplacements sont spécifiés en tant qu'énum dans DirectionEnum, quand à l'environnement, il est spécifié dans *Environnement*

Chapitre 2

Billes / Particules

2.1 Diagramme UML

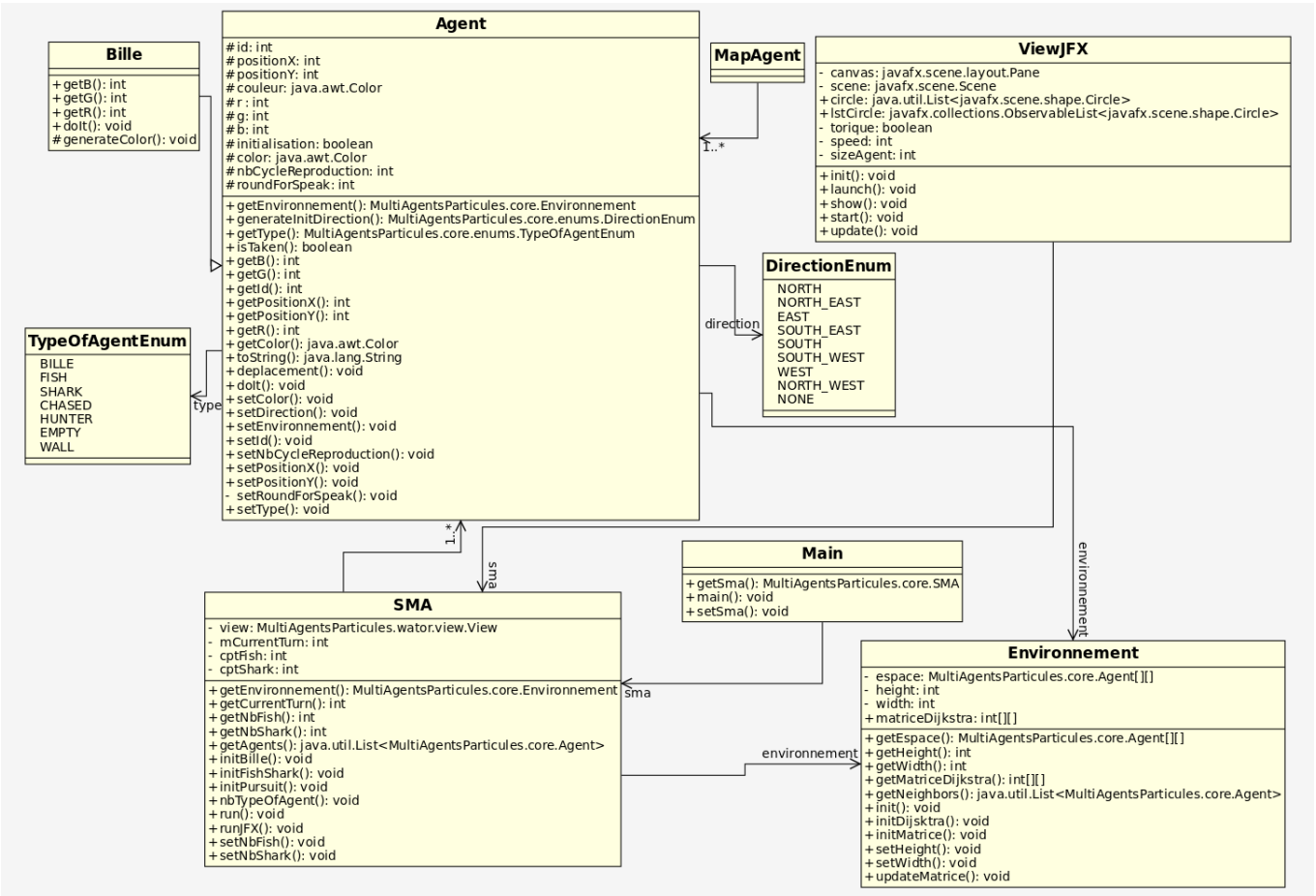

Figure 2.1: Diagramme de classes Bille

2.2 Description

2.2.1 Caractéristiques

Ici il n'y a qu'un seul type d'agent, qui est Bille. Une bille possède des coordonnées (x,y), sa propre couleur générée aléatoirement lors de l'instanciation de l'objet. Une bille peut se déplacer dans l'une des 8 directions.

2.2.2 Comportement

La toute première direction est choisie à l'initialisation de l'environnement. Une fois cette direction choisie, la bille garde la même direction sauf si elle croise une case non vide ou si elle tape le mur dans le cas non torique. Dans ces deux cas, la bille qui fait appel à sa methode *doIt()* en premier va dans la direction opposée si la case est non vide, sinon elle reste sur place pour le tour en cour jusqu'a qu'une case autour se libère. Ensuite la bille va dans une case aléatoire parmi toutes ses possibilités, c'est à dire parmi toutes les cases vides autour. Pour ce projet, le mode torique et non torique ont été géré.

2.2.3 Résultats et performance

Pour ce qui est de la vue, pour ce projet nous utilisons JavaFX. Le principe est simple, au lancement du projet, la vue récupère les positions des billes et on déssine un cercle pour chaque position (x,y) récupérée de la couleur de l'agent. En terme de performance, nous arrivons à avoir un rendu légèrement fluide avec 50 000 particules de taille 5 réparties dans une grille de 2500 X 2500 mais totalement fluide avec 40 000 particules.

Chapitre 3

Wator

3.1 Diagramme UML

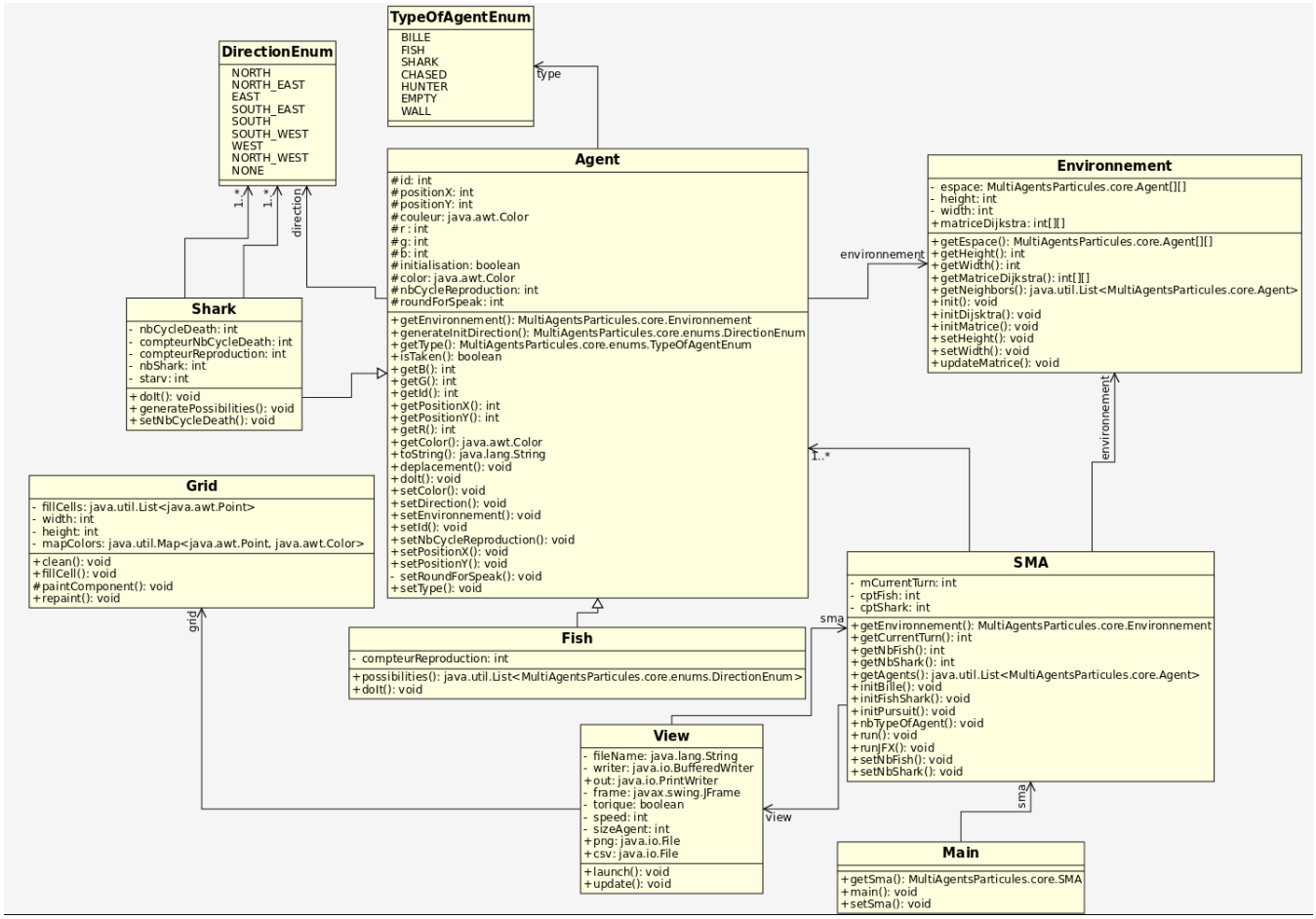

Figure 3.1: Diagramme de classes Wator

3.2 Description

3.2.1 Caractéristiques

Il y a deux types d'agents. Le type Fish et le type Shark. Pour ces deux types nous avons des caractéristiques communes : la couleur, le nombre de cycle avant reproduction et le nombre de cycle. Le type Shark possède en plus un nombre de cycle avant de mourir.

3.2.2 Comportement

Le poisson a comportement basique, il se déplace dans une case aléatoirement parmi les cases vides extraites ses huit directions. Si aucune case n'est vide, le poisson ne bouge pas. Une fois le nombre de cycle de reproduction atteint, si le poisson peut se déplacer, il pose son bébé à sa propre position puis il se déplace et un nouveau cycle commence. sinon il abandonne le bébé, et un nouveau cycle commence.

Le requin quand à lui possède un comportement basique si il n'y a pas de poisson dans ses huit cases autour. Il choisit une case au hasard s'il peut se déplacer sinon il ne bouge pas. Si il y a un ou des poissons dans les cases autour, il en choisit une au hasard parmi celles ou il y a des poissons. Le requin mange le poisson. Pour la reproduction, le principe est le même que pour les poissons. Nous considérons ici que le requin peut manger, se déplacer et déposer un bébé pendant un même tour. Lorsque le requin mange, son compteur de vie ne décrémente pas. Pour ce projet, le mode torique et non torique ont été géré.

3.2.3 Résultats et performance

Pour cette vue, c'est java Swing qui a été utilisé mais le principe reste le même que pour javaFX. Nous obtenons un équilibre entre les poissons et les requins pour les paramètres suivants qui sont les paramètres par défaut :

```
nbFish = 2000
nbShark = 1
nbCycleRepFish = 2
nbCycleRepShark = 5
nbCycleDeathShark = 3
roundForSpeakFish = roundForSpeakShark = 1
width = 600
height = 600
speed = 50
sizeAgent = 5 (pas par défaut)
torique
```

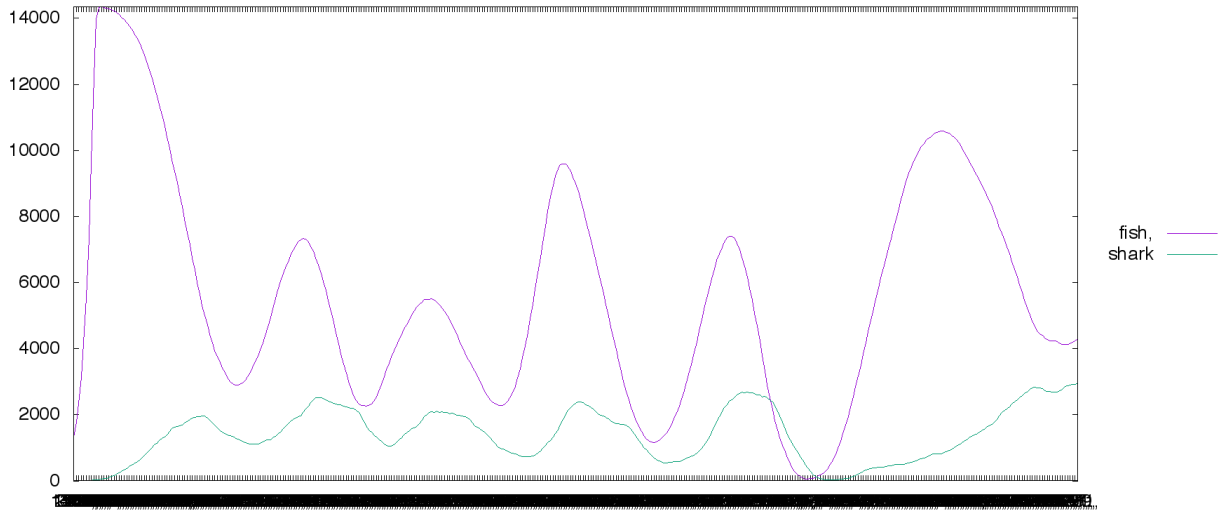

Figure 3.2: Evolution nb poissons nb requins

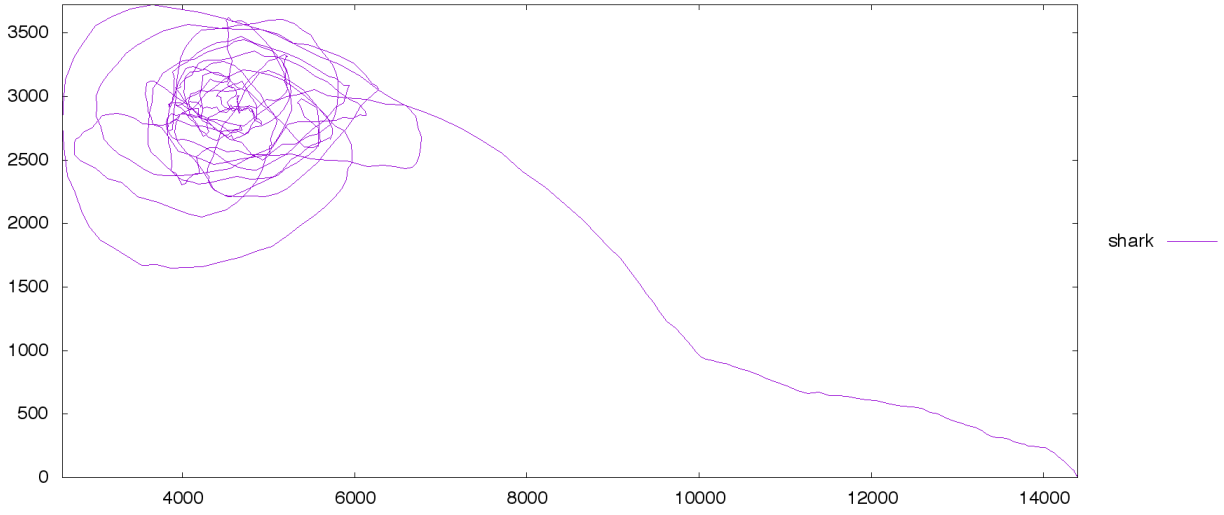

Figure 3.3: Evolution nb poissons nb requins

Chapitre 4

Poursuivant, poursuivi (HotPursuit)

4.1 Diagramme UML

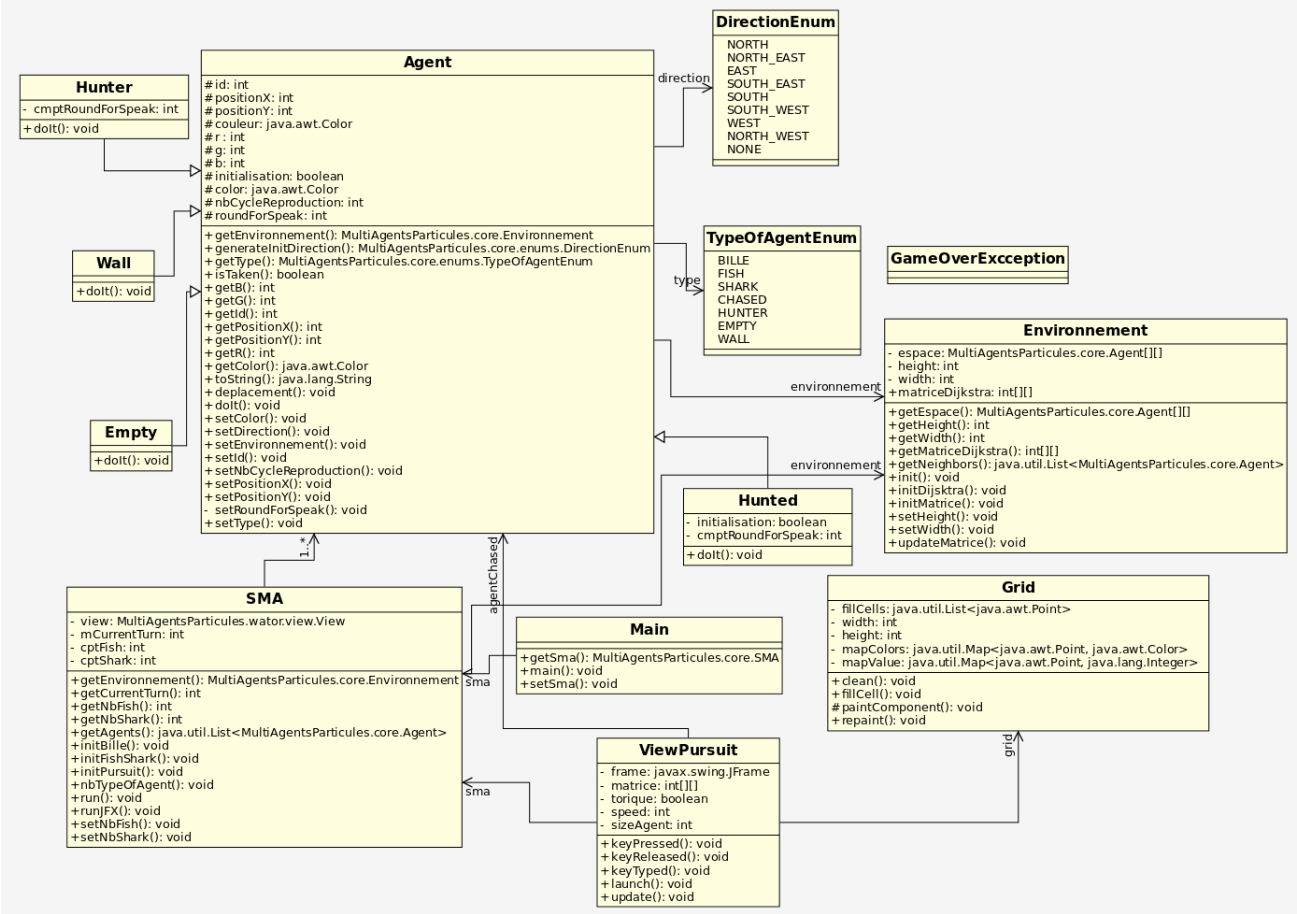

Figure 4.1: Diagramme de classes HotPursuit

4.2 Description

4.2.1 Caractéristiques

Il y a quatres types d'agent ici. Le type Hunted (de couleur verte) qui nous représente nous. C'est l'agent qui est chassé. Il possède simplement une direction et se déplace dans un environnement uniquement torique. Le type Hunter (de couleur rouge) qui représente les chasseurs sont des agents qui se déplacent constamment vers l'agent chassé en utilisant les plus courts chemins. Le type Wall (de couleur noir) qui représente un mur et qui ne bouge pas de sa position. Et enfin le type Empty qui est utilisé pour représenter les cases vides, il nous a été utile pour calculer la matrice de Dijkstra.

Les types Hunter et Hunted peuvent avoir un temps de parole différent.

4.2.2 Comportement

Pour le chassé, sa direction change en fonction des touches sur lesquelles on appuie. Nous avons choisis les touches UIO/JKL/,;: avec K qui permet de stopper l'agent. A chaque appelle de la méthode *doIt()* on initialise la matrice de Dijkstra et on calcule les plus court chemins.

Le chasseur quand à lui se déplace en suivant les valeurs des cases autour, le chasseur choisis toujours la case qui à la valeur la plus petite (qui est en réalité toujours une valeur inférieur de 1 à la case où il se trouve). Il vérifie que la case n'est pas un mur ou un chasseur. Si il tombe sur le chassé, nous lancons une exception qui correspond à la fin du jeu.

4.2.3 Résultats et performance

Nous avons testé le projet avec plusieurs modes (temps de parole, nombre de mur, taille de la grille...) et nous remarquons qu'a partir d'une grille 700 X 700 pour des agents de taille 10, le programme est ralenti. Mais augmenter le nombre de chasseurs pour une taille moindre ne pose pas de problème.

Chapitre 5

Commandes

Liste des paramètres généraux

```
"p" = projet : particles, wator ou hunt (par défaut)
"h" = hauteur : hauteur de la grille en pixel
"w" = largeur : largeur de la grille en pixel
"speed" = vitesse : temps d'attente entre chaque tour
"sa" = taille de l'agent dans la grille
"t" = torique si il y a présence de ce paramètre
```

Billes / Particules

```
"nb" = nombre de particules présent dans l'environnement
```

Wator

```
"nbF" = nombre de poissons
"nbS" = nombre de requins
"repF" = nombre de tour d'attente avant la reproduction du poisson
"repS" = nombre de tour d'attente avant la reproduction du requin
"death" = nombre de tour d'attente avant la mort du requin
"speakFish" = le poisson parle 1 fois sur [speakFish]
"speakShark" = le poisson parle 1 fois sur [speakShark]
```

Hunt

```
"nbH" = nombre de chasseurs
"nbW" = nombre de mur
"speakHunter" = le chasseur parle 1 fois sur [speakHunter]
"speakHunted" = le chassé parle 1 fois sur [speakHunted]
```

Exemple d'utilisation

```
java -jar SMA.jar -p wator -nbF 2000 -nbS 2 -t
```