Université de Lille

MASTER 2 INFORMATIQUE
Premier semestre

Simulation centrée invididus

Rapport

Particles, Wa-Tor, Hunter

BARCHID Sami

Année académique 2019-2020

Introduction

Ce rapport présente le travail réalisé pour la création d'un framework de simulation multi-agents. Ce travail est composé de trois parties :

- **Particles** : simulation multi-agents de billes qui se collisionnent dans un environnement.

- **Wa-Tor** : simulation multi-agents représentant l'évolution de populations de requins (prédateurs) et de poissons (proies).

- **Hunter** : simulation multi-agents représentant un système similaire au jeu « Pacman », où l'utilisateur humain est représenté dans la simulation.

La même architecture modulable de code, écrit en Java, est utilisée pour les trois travaux. **Le rapport suppose donc que le lecteur, s'il désire reproduire les exécutions reportées, est capable de compiler et exécuter un programme JAVA 12 sur sa machine**. Il est également admis que l'utilisateur se trouve à la racine du projet qui a été livré avec ce rapport.

Table des matières

Architecture de code

Cette partie du rapport présente l'architecture générale du framework construit pour mettre au point les trois projets différents. Pour ce faire, je présenterai le diagramme UML général accompagné d'un texte d'explication de chaque classe importante.

Packages

Le framework est composé de 4 packages différents :

- **core** : package de base qui contient l'ensemble des classes génériques à hériter pour créer un système multi-agents avec le framework.

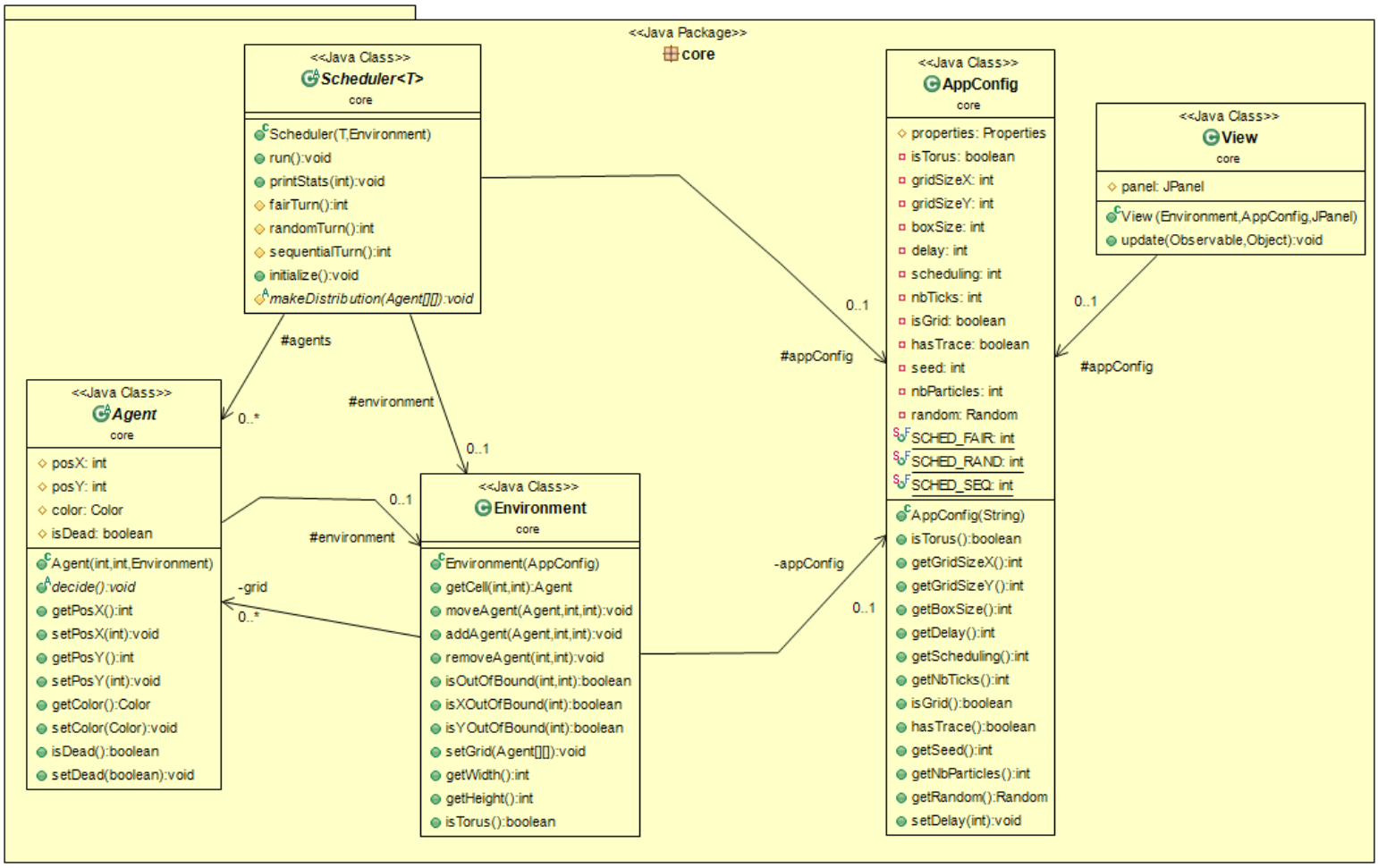
- **particles:** package qui contient l'implémentation concrète du système « Particles».

- **wator** : package qui contient l'implémentation concrète du système « Wa-tor ».

- **hunter** : package qui contient l'implémentation concrète du système « Hunter ».

Ici, seul le package « core » (base de l'architecture) sera expliqué. Les autres packages seront traités dans leur partie propre.

Classes



- **AppConfig** : classe instanciée une unique fois au début de l'exécution qui lit le fichier properties contenant tous les paramètres du système multi-agents.

- **Environment** : classe instanciée une unique fois au début de l'exécution et qui représente l'environnement dans lequel évolue les agents du système. C'est donc dans cette classe qu'est stockée la grille 2D des agents ainsi que les méthodes permettant de déplacer/ajouter/retirer ceux-ci dans cette grille. *Note : pour la suite du TP, l'environnement ne sera jamais override étant donné que les fonctions de base suffisent au fonctionnement de n'importe quelle simulation.*

- **Scheduler<T extends AppConfig>** : classe instanciée une unique fois au début de l'exécution et qui a pour rôle d'organiser la distribution et l'exécution des agents dans l'environnement.

    - *makeDistribution* : méthode destinée à être override pour implémenter les créations des différents agents spécifiques au système à implémenter.

- **View** : classe instanciée une unique fois et qui a pour rôle de gérer l'affichage de la simulation pour l'utilisateur. Cette classe est un Observer du Scheduler, ce qui permet de garder à jour la vue après chaque tour de parole.

- **Agent** : classe abstraite qui représente un Agent évoluant dans le système.

    - *decide* : méthode abstraite à override dans chaque nouvelle classe d'Agent du système pour implémenter tous les comportements possibles de l'agent.

Paramètres de base

Un système multi-agents de base accepte un fichier properties contenant les différentes options qui suivent :

- `gridSizeX` (int) : taille de l'environnement (en nombre de cases) dans l'axe des X.

- `gridSizeY` (int) : taille de l'environnement (en nombre de cases) dans l'axe des Y.

- `boxSize` (int) : taille en pixels d'un carré de la grille de l'environnement.

- `delay` (int) : délais (en millisecondes) pendant lequel le système « attend » entre deux tours de parole des agents.

- `scheduling` (int) : entier indiquant la stratégie d'exécution du tour de parole des agents. Les valeurs de ce paramètres sont les suivantes :

    - **0 :** « fair scheduling », c’est-à-dire que, pour chaque tour de parole, tous les agents sont exécutés une et une seule fois, suivi d’un « shuffle » de l’ordre de parole pour le prochain tour.

    - **1 :** « random scheduling », c’est-à-dire que les agents sont exécutés au hasard, sans garantie que tout le monde ait été consulté pendant le tour de parole.

    - **2** : « sequential scheduling », c’est-à-dire que les agents sont exécutés l’un après l’autre dans le même tour de parole.

- `seed` (int) : nombre correspondant au sel permettant de réaliser un pseudo-aléatoire afin de pouvoir reproduire à l’identique une simulation. Si ce paramètre vaut 0, l’aléatoire est total (et donc l’expérience n’est pas reproductible).

- `nbParticles` (int) : nombre d’agents dans le système.

- `nbTicks` (int) : nombre de tours de parole au bout duquel la simulation prend fin.

- `isTorus` (boolean) : flag qui indique si l’environnement est torique ou pas.

- `hasTrace` (boolean) : flag qui indique si des stats sont affichées sur la sortie standard pendant l’exécution de la simulation.

- `isGrid` (boolean) : flag qui indique si l’environnement s’affiche comme une grille.

Particles

Cette partie présente le système multi-agents implémentant le systèmes de billes se collisionnant dans leur environnement.

Mode d’emploi

Fonctionnement

- Les agents Particles sont représentés par des cercles noirs.
- Quand deux agents Particles entrent en collision, ils passent à la couleur rouge jusqu’au prochain tour. **Important : lorsque deux particules se collisionnent, elles échangent leurs directions.**
- Quand un agent Particle entre en collision avec le mur, il passe à la couleur rouge jusqu’au prochain tour. **Important : lorsqu’une particule touche un mur, sa direction devient l’inverse de l’originale.**

Si l’affichage de stats est activé pour la simulation, le système imprime deux types d’informations (format CSV) :

- `Tick;<nombre de collisions pour le tour de parole passé>;` (après chaque tour de parole)
- `Agent;<position en X>;<position en Y>;<pas en X>;<pas en Y>;` (pour chaque agent)

Exécution

À partir de la racine du projet, exécuter les commandes suivantes :

Paramètres

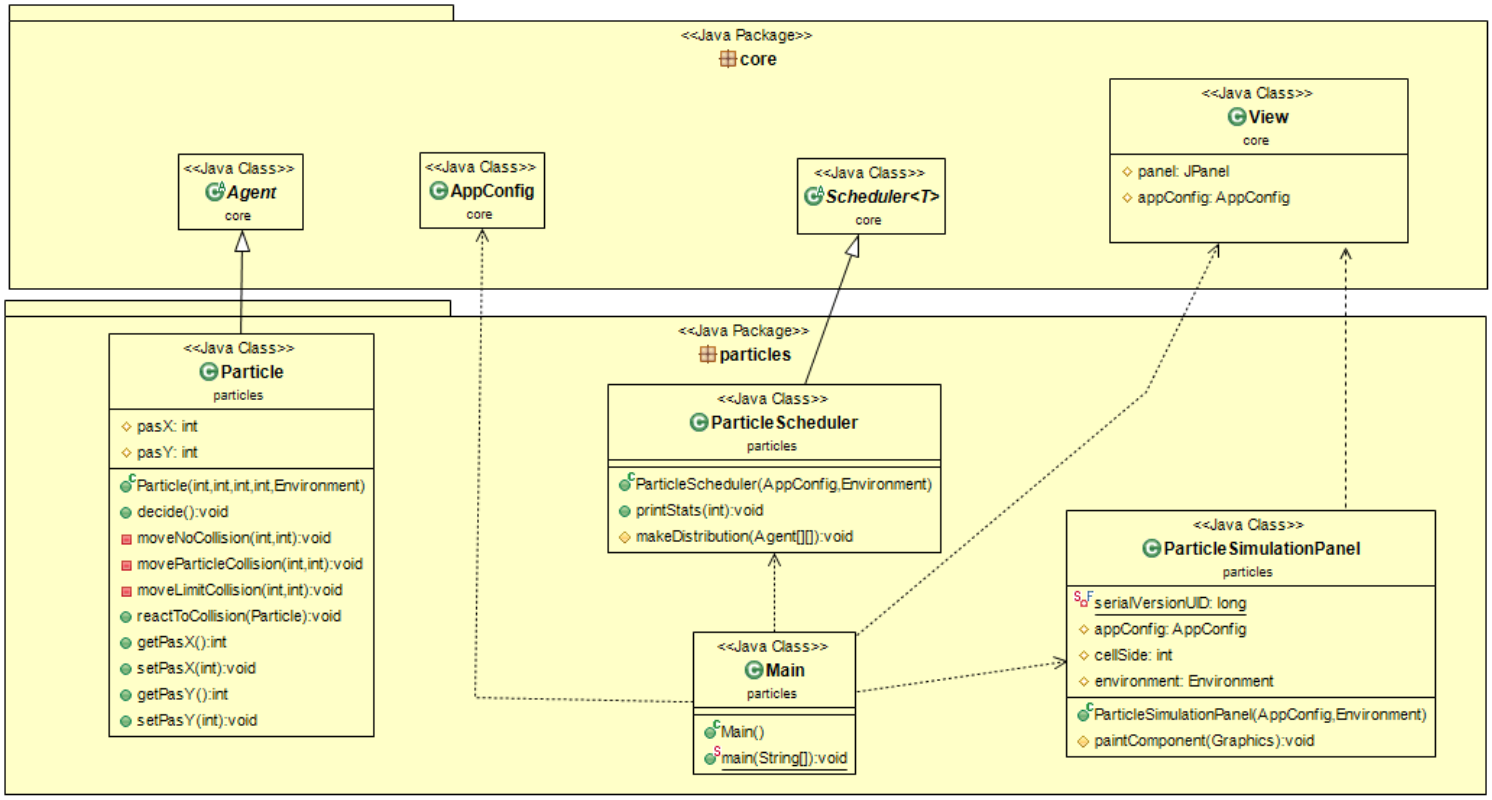
Un fichier « `particles.jar` » est présent à la racine du projet remis avec ce rapport.

À partir de la racine du projet, exécuter la commande suivante :

- `java -jar particles.jar [path fichier properties]`

Si aucun fichier properties n'est fourni, un fichier properties par défaut est cherché par le programme : `particles.properties`. Ce fichier est déjà existant à la racine du projet et peut être modifié.

Architecture



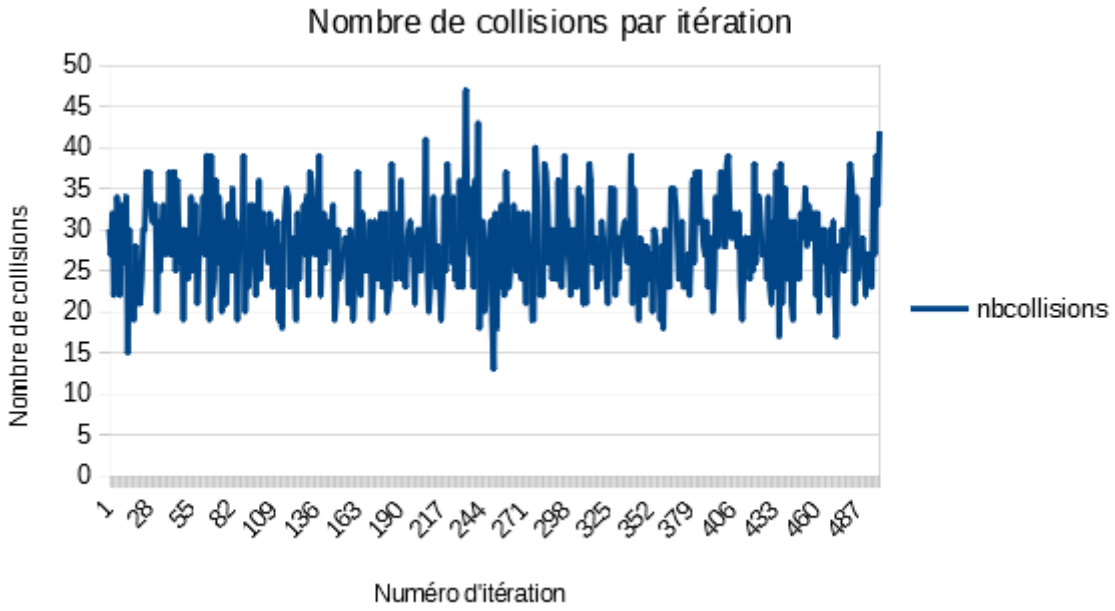
- **Particle** : seul agent du système.
- **ParticleSimulationPanel** : classe qui étend JPanel et qui sert à gérer l'affichage de la simulation (c'est cette classe qui va contenir les appels aux méthodes paint de Java Swing).

- **Main** : classe qui contient la méthode main et qui se charge de faire les instanciations avant de lancer la simulation.

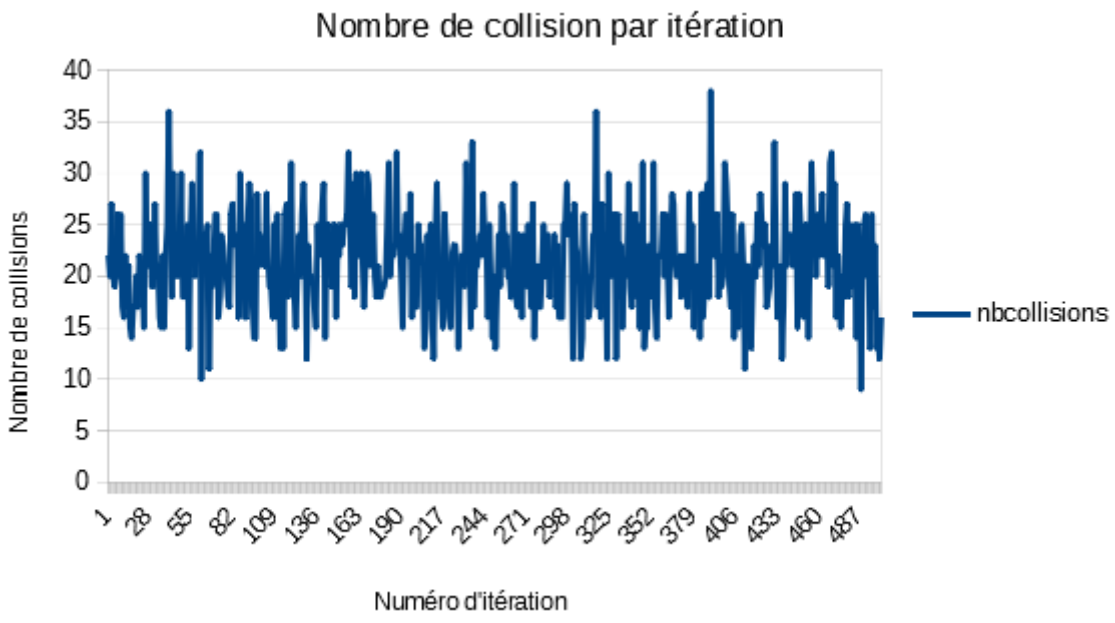
Résultats

Nous allons comparer le nombre de collisions pour chaque tour de parole de 500 particules dans un espace torique (et non-torique) pour des tailles d'environnement de plus en plus réduites.

Environnement taille 100x100



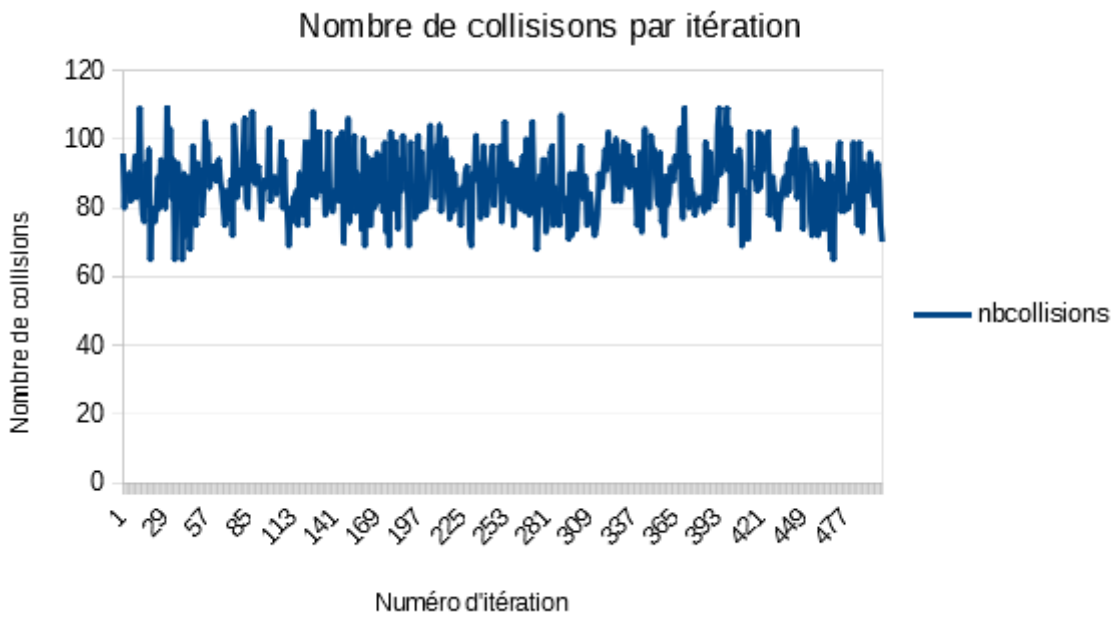
Non-torique



Torique

On remarque que, de manière logique, il y a plus de collisions en environnement non-torique puisque les particules se heurtent aux murs. La suite des mesures est faite en torique

Environnement 50x50



On remarque que le nombre de collisions est nettement plus important à dans un environnement plus restreint (ce qui est encore une fois logique vu qu'on a mois de place pour le même nombre de billes).

Wa-Tor

Cette partie présente le système de simulation Wa-Tor.

Mode d’emploi

Fonctionnement

- Les Fishs sont représentés par des cercles verts. Un Fish qui est enfanté par un autre est représenté par un cercle jaune jusqu’au tour suivant.

- Les Sharks sont représentés par des cercles rouges. Un Shark qui est enfanté par un autre est représenté par un cercle rose jusqu’au tour suivant.

- Si le print des stats est activé, les informations imprimées sur l’output standard sont :

  - `Agent;<Death ou Birth>;Fish;`**`<coord. X de la mort>;<coord Y de la mort>`**

  - `Tick;`**`<numéro de l’itération>;<nombre de Shark>;<nombre de Fish>`** : Après chaque fin de tour.

- **Important** : les Sharks et les Fishs possèdent une espérance de vie aléatoire obtenue à la naissance. Lorsque le Shark ou le Fish atteint la fin de son espérance de vie, il meurt de vieillesse.

Exécution

Un fichier « `wator.jar` » est présent à la racine du projet remis avec ce rapport.

À partir de la racine du projet, exécuter la commande suivante :

- `java -jar wator.jar [path fichier properties]`

Si aucun fichier properties n’est fourni, un fichier properties par défaut est cherché par le programme : `wator.properties`. Ce fichier est déjà existant à la racine du projet et peut être modifié.

De plus, trois fichiers properties sont déjà présents et permettent de lancer des simulations pré-configurées selon les différents scénarios possibles :

- `wator-fish-win.properties`: simuler un cas de figure où tous les Sharks meurent et les Fishs abondent.
- `wator-apocalypse.properties`: simuler un cas de figure où tous les Fishs meurent puis tous les Sharks.
- `wator-stable.properties`: simuler un cas de figure où les Fishs et les Sharks sont en équilibre.

Paramètres

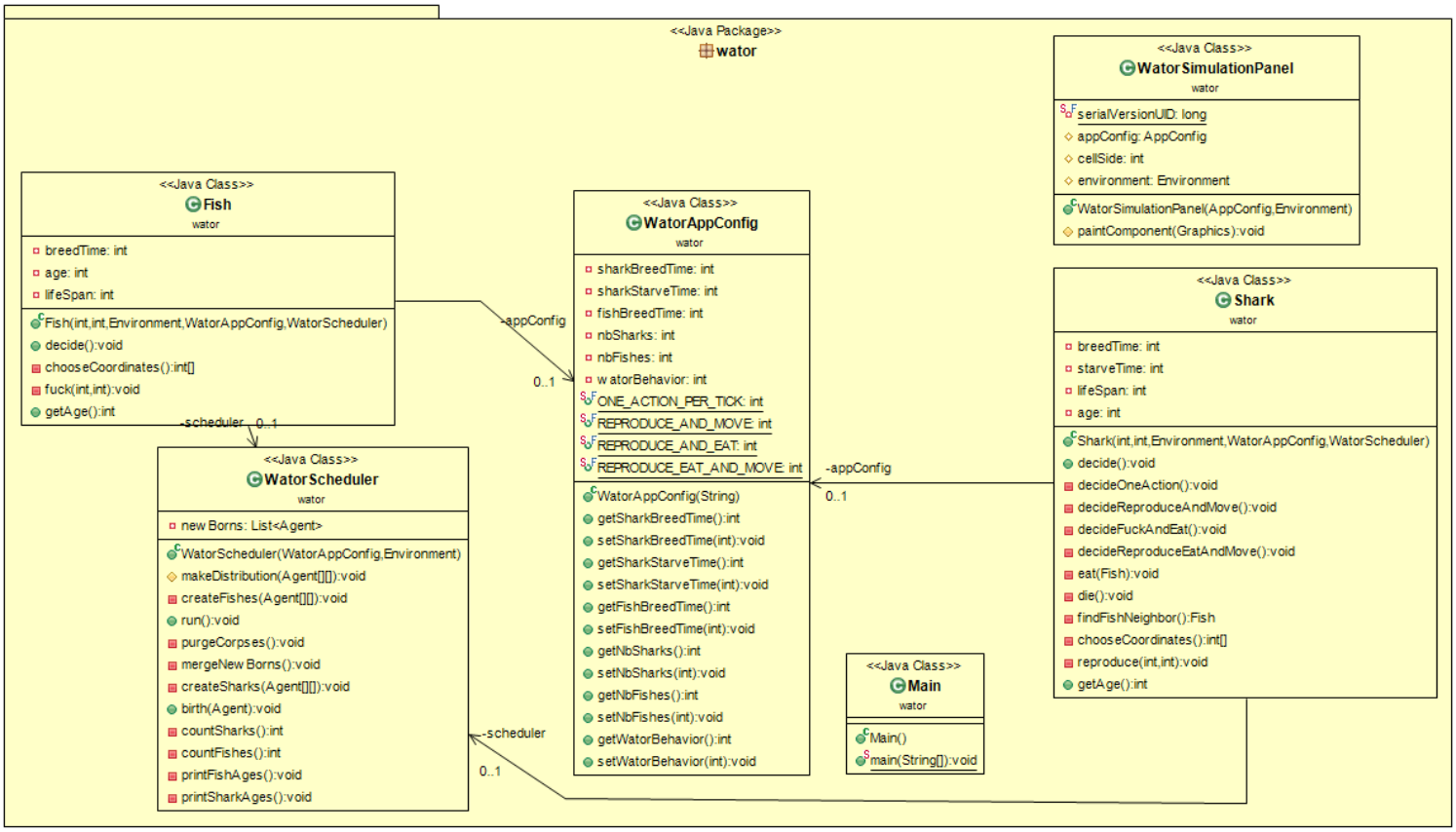
En plus des paramètres de base, Wa-Tor contient les paramètres suivants :

- `nbFishes` (int) : nombre de Fishs au départ de la simulation.
- `nbSharks` (int) : nombre de Sharks au départ de la simulation.
- `sharkBreedTime` (int) : nombre de ticks au bout duquel un Shark peut se reproduire.
- `fishBreedTime` (int) : nombre de ticks au bout duquel un Fish peut se reproduire.
- `sharkStarveTime` (int) : nombre de ticks sans manger au bout duquel un Shark meurt de faim.
- `watorBehavior` (int) : entier indiquant la variante de comportement du Shark. Les valeurs possibles sont :

- **3 :** un Shark peut se déplacer, manger et se reproduire pendant le même tour.

- **2 :** un Shark peut, soit se reproduire, soit manger pendant un même tour.

Architecture



- **Fish** : classe héritant de core.Agent et qui implémente le comportement des Fishs dans le système.

- **Shark** : classe héritant de core.Agent et qui implémente le comportement des Shars dans le système.

- Les autres classes héritent des classes générales du package core.

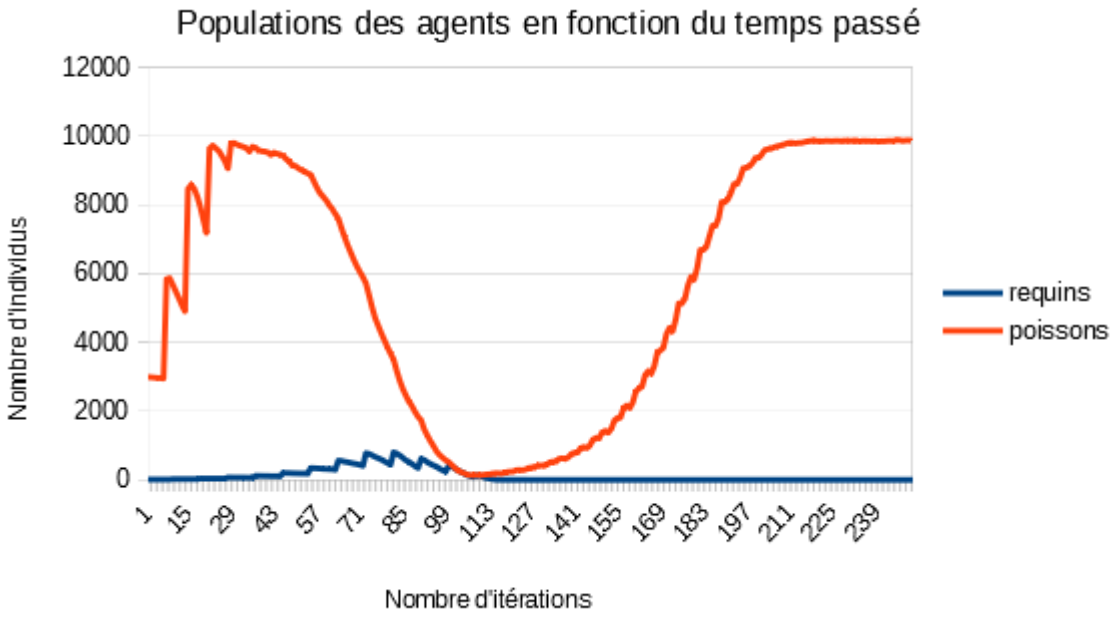
Résultats

Je commencerai par montrer des configurations pour illustrer les trois différents cas finaux possibles dans Wa-Tor. Ensuite, je montrerai des statistiques sur la population des agents dans un environnement stable.

Tous les requins meurent

Tous les requins meurent et les poissons pullulent. La configuration de la simulation est présentée ainsi que le graphique d'évaluation des populations.

```
gridSizeX=100
gridSizeY=100
boxSize=10
delay=20
scheduling=2
seed=139098
nbParticles=3010
nbTicks=250
isTorus=true
hasTrace=true
isGrid=true
nbFishes=3000
nbSharks=10
sharkBreedTime=8
fishBreedTime=6
sharkStarveTime=5
watorBehavior=3
```

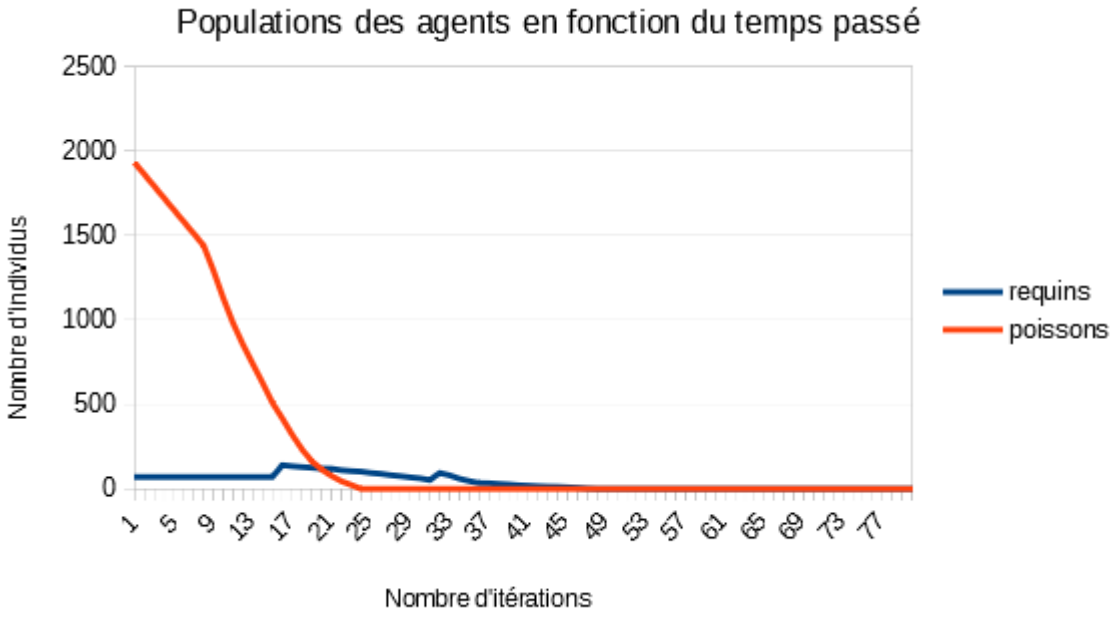


On remarque que la population de requinsa disparu lorsque le stock de poissons était extrêmement faible, ce qui a pu laisser pulluler les poissons par la suite, faute de prédateur.

Tout le monde est mort

Les requins mangent tous les poissons puis meurent lorsqu'il n'y a plus rien à manger. La configuration de la simulation est présentée ainsi que le graphique d'évaluation des populations.

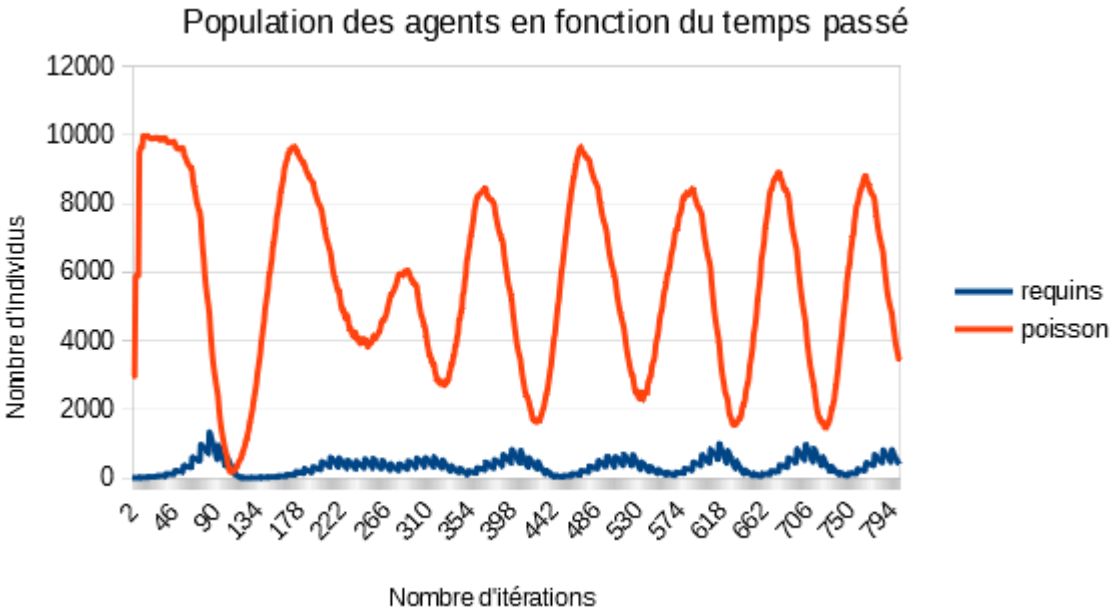
```
gridSizeX=50
gridSizeY=50
boxSize=20
delay=20
scheduling=2
seed=139098
nbParticles=2
nbTicks=80
isTorus=true
hasTrace=true
isGrid=true
nbFishes=2000
nbSharks=70
sharkBreedTime=15
fishBreedTime=40
sharkStarveTime=15
watorBehavior=3
```



On remarque bien ici la suppression de tous les poissons, qui entraîne une mort des prédateurs.

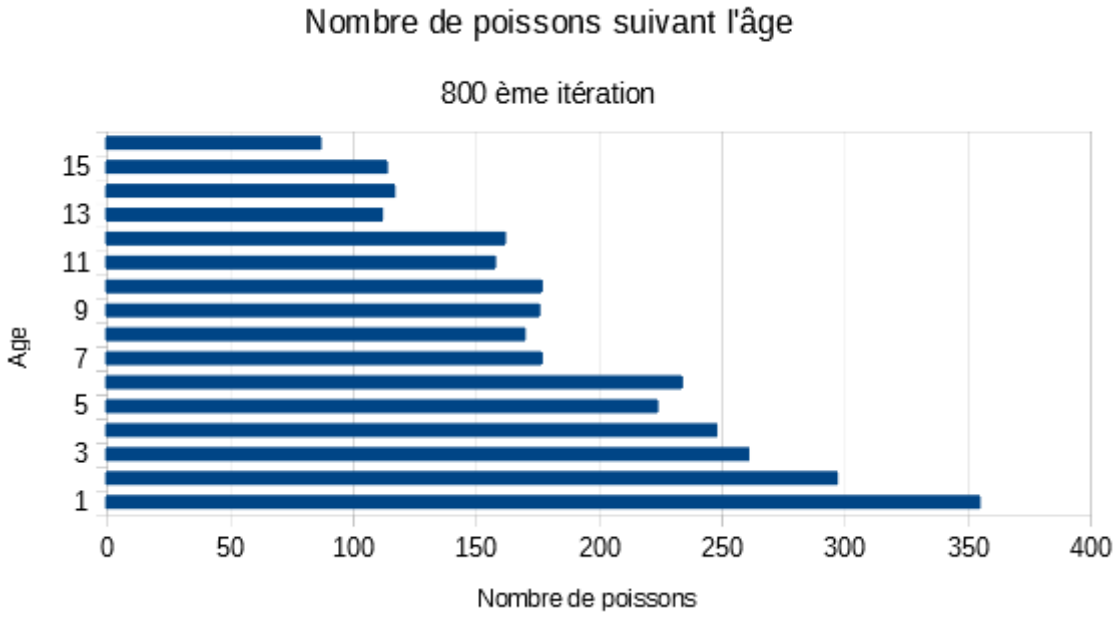
Populations stables

```
gridSizeX=100
gridSizeY=100
boxSize=10
delay=20
scheduling=2
seed=139098
nbParticles=3010
nbTicks=800
isTorus=true
hasTrace=true
isGrid=true
nbFishes=3000
nbSharks=10
sharkBreedTime=8
fishBreedTime=3
sharkStarveTime=5
watorBehavior=3
```



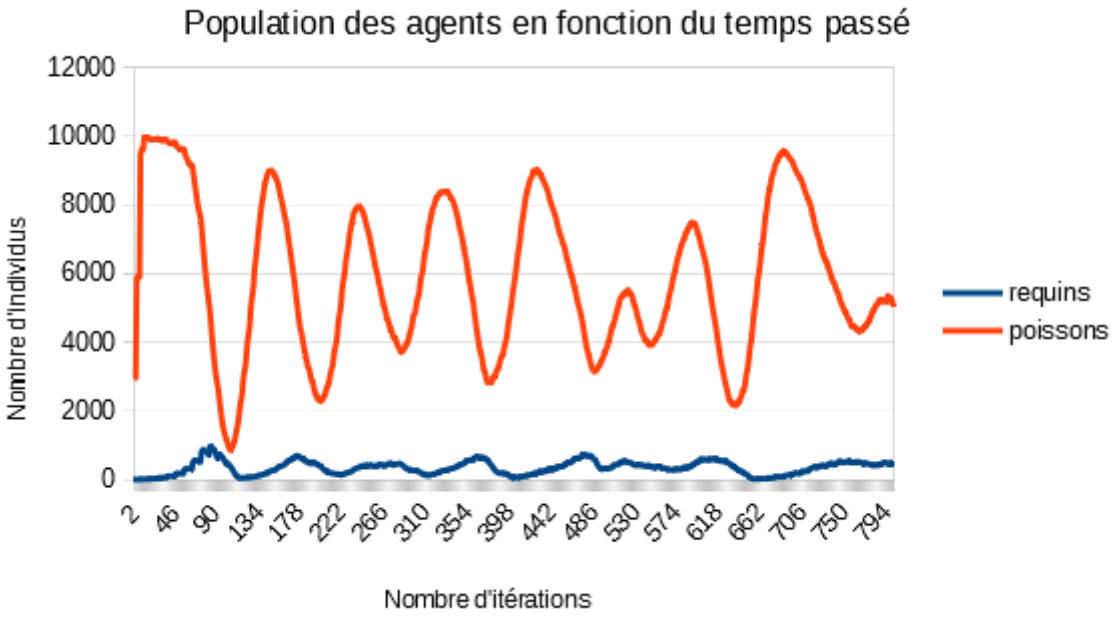
On remarque l’apparition d’une fonction sinus (légérement « creneaulée » même si ça se voit mal à cause du nombre d’individus pour les poissons) pour les poissons et les requins. On voit aussi une sorte de « décalage » entre les deux sinus, montrant le lien entre les deux populations : quand il y a un grand nombre de requins, les poissons sont vites beaucoup mangés puis les requis meurent, faute de nourriture.

On prend ensuite un extrait de la pyramide des âges (avec la population des poissons) pour voir que l’on obtient bel et bien une répartition comme celle vue au cours (beaucoup de jeunes, et de moins en moins d’individus suivant l’âge).



Changer le comportement des Sharks

Dans les simulations précédentes, seul la stratégie du « tout faire le même tour » a été testée. On peut également tester le comportement du « soit manger soit se reproduire » pour les requins, pour la simulation stable.



On remarque que les variations du nombre de poissons sont plus réduites, que le nombre de poissons reste plus haut même dans les périodes de creux et que la population de requins ne décrit plus une sinusoidale marquée.

Hunter

Cette partie présente le système multi-agents implémentant un simili au jeu de Pacman.

Mode d’emploi

Fonctionnement

- L’Avatar est piloté grâce aux touches **ZQSD**. Il est représenté par un cercle jaune.
- Les Hunters sont représentés par des carrés rouges. Si un Hunter arrive à rattraper l’Avatar, il se colore en blanc et le jeu s’arrête.
- Les Defenders sont représentés par des carrés verts.
- Les Walls sont représentés par des carrés noirs.
- Le Winner est représenté par un carré rose. Si l’Avatar arrive à prendre le Winner, il se colore en blanc et le jeu s’arrête.
- La touche **ESPACE** permet de recommencer une simulation en plein milieu du jeu.
- La touche **O** accélère la vitesse des Hunters.
- La touche **P** ralentit la vitesse des Hunters.
- La touche **L** accélère la vitesse de l’Avatar.
- La touche **M** ralentit la vitesse de l’Avatar.
- La touche **U** accélère la vitesse de la simulation.
- La touche **I** ralentit la vitesse de la simulation.

Exécution

Un fichier «`hunter.jar` » est présent à la racine du projet remis avec ce rapport.

À partir de la racine du projet, exécuter la commande suivante :

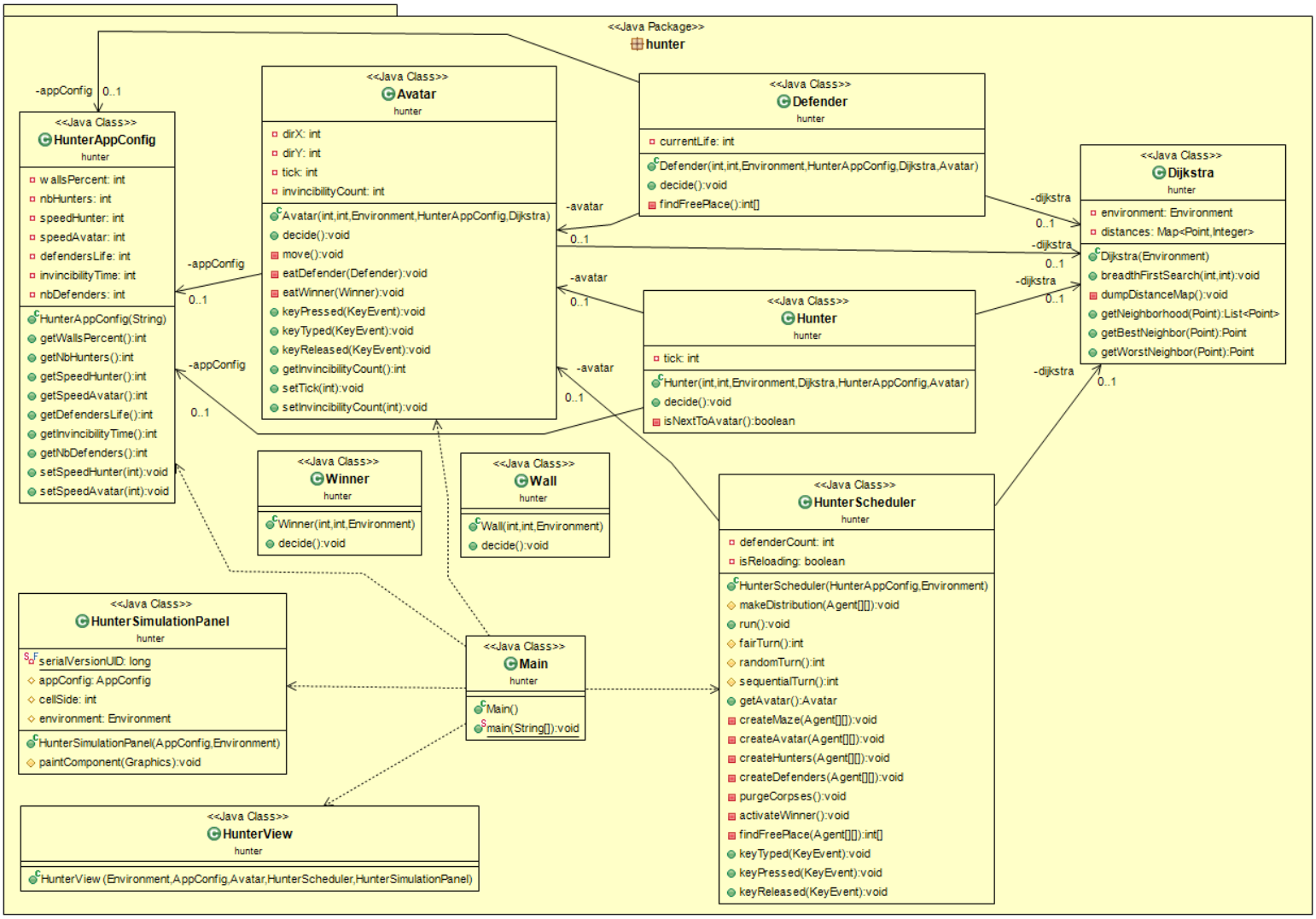
- `java -jar hunter.jar [path fichier properties]`

Si aucun fichier properties n'est fourni, un fichier properties par défaut est cherché par le programme : `hunter.properties`. Ce fichier est déjà existant à la racine du projet et peut être modifié.

Paramètres

- `wallsPercent` (int) : pourcentage de murs présents dans la grille.
- `nbHunters` (int) : nombre de chasseurs présents dans la simulation
- `speedHunter` (int) : nombre de ticks au bout duquel un hunter se met à faire une action. La vitesse maximale est donc de speedHunter = 1 et se réduit au fur et à mesure que speedHunter augmente.
- `speedAvatar` (int) : nombre de ticks au bout duquel l'Avatar se met à faire une action. La vitesse maximale est donc de speedAvatar = 1 et se réduit au fur et à mesure que speedAvatar augmente.
- `defendersLife` (int) : nombre de ticks au bout duquel un Defender change de position.
- `invincibilityTime` (int) : temps d'invincibilité (en nombre de ticks) octroyé par un Defender à l'Avatar lorsque celui-ci mange le mange.
- `nbDefenders` (int) : nombre de Defender présents dans la simulation.

Architecture



- **Avatar** : classe héritant de core.Agent contrôlé par l'utilisateur.

- **Hunter** : classe héritant de core.Agent qui pourchasse l'Avatar via l'algorithme de Dijkstra.

- **Wall** : classe héritant de core.Agent qui joue le rôle du mur (bloque tout passage sur la case où il se trouve et ne bouge pas).

- **Defender** : classe héritant de core.Agent qui joue le rôle des défenseurs qui sont pris par l'Avatar pour éloigner les Hunters et invoquer le Winner.

- **Winner** : classe héritant de core.Agent qui joue le rôle de la porte de la victoire pour l’Avatar. Le Winner est créé dans l’environnement quand l’Avatar a réussi à récupérer tous les Defenders.

- **Dijkstra** : classe instanciée en début d’exécution et qui sert à calculer le plus court chemin vers l’Avatar pour tous les Hunters. L’objet « Dijkstra » est partagé entre tous les Hunters et c’est l’Avatar qui lance la recherche sur l’instance de Dijkstra.

- Les autres classes héritent , quant à elles, de leur super classe respective du package core.