Dimitri Charneux

Rendu TPs SCI :

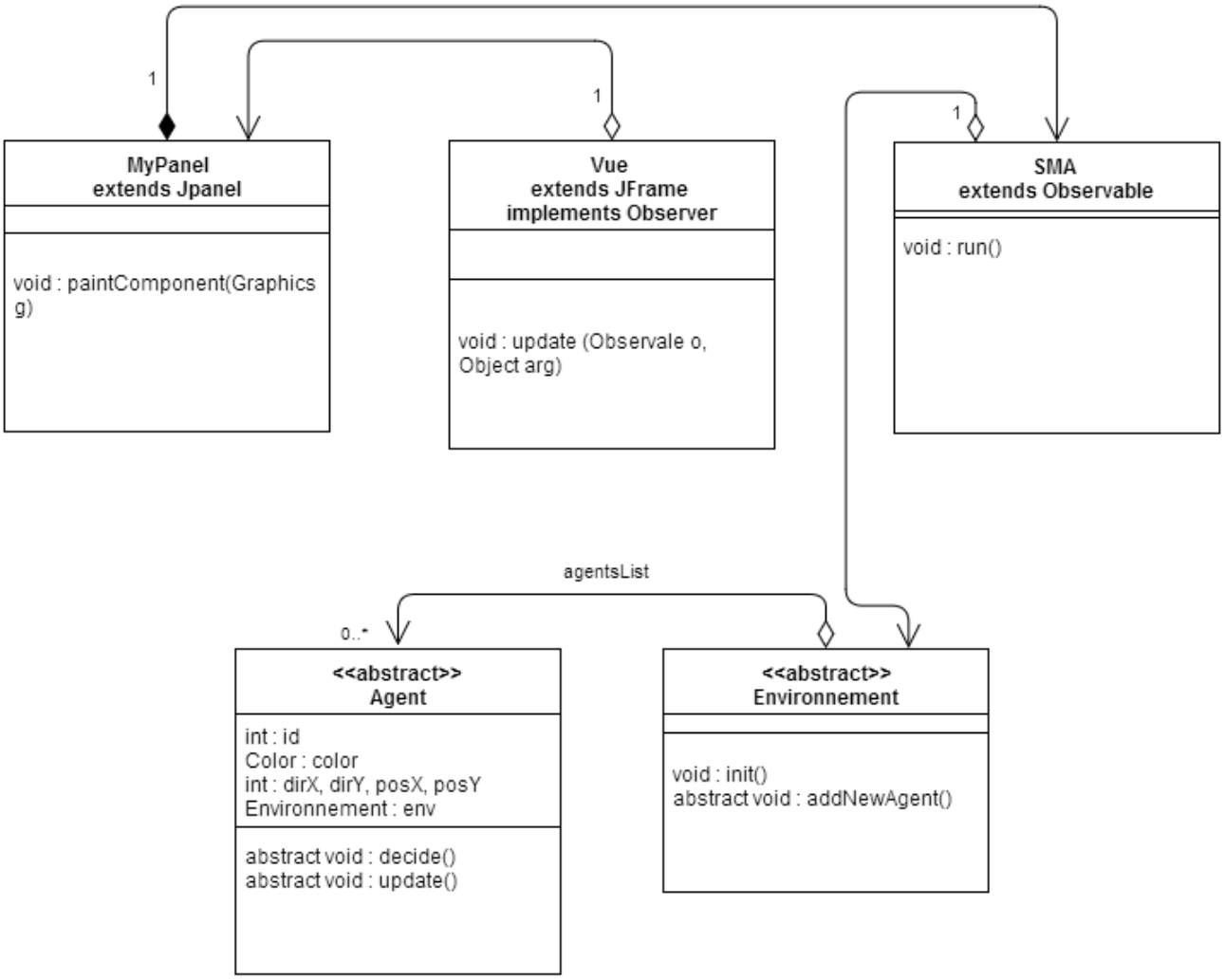
Table des matières

Introduction

Dans ce rapport, je vais vous présenter les 3 TPs que j'ai réalisé en SCI. Ces TPs ont consisté à créer plusieurs systèmes multi-agents où différents agents indépendants cohabitent dans un même environnement en 2 dimensions.

La vue

Dans ce projet, la vue est la même pour tout les TPs, elle se compose d'une classe ***Vue*** qui représente la fenêtre et d'une classe ***MyPanel*** qui va contenir tout les éléments graphiques.



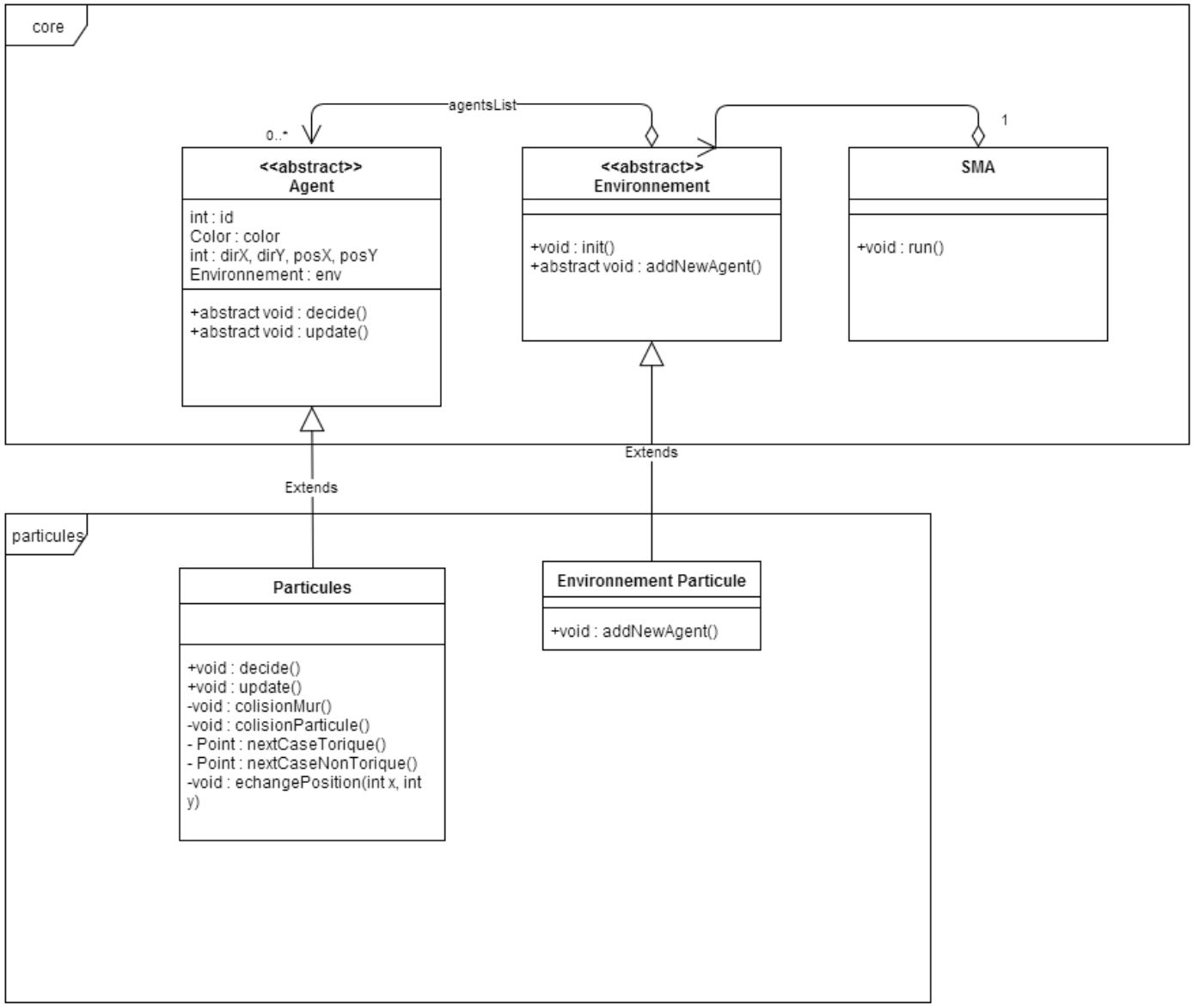
Comme vous pouvez le voir dans cet UML, La vue fonctionne avec le modèle Observable/Observer. La classe ***SMA*** va être observée par la classe ***Vue***. À chaque fois que le ***SMA*** va changer, la ***Vue*** va appeler sa méthode *update* qui va elle même appeler la méthode *paintComponent* du ***MyPanel.*** Cette méthode va dessiner chaque agent présent dans l'environnement du SMA en fonction de leur couleur.

TP1 : Particules

Principe

Le premier système multi-agents créé a été le TP particules. Dans ce système, plusieurs agents nommés « particules », cohabitent et interagissent entre eux à travers des rebonds simples. L'objectif de ce TP a été de réaliser un premier système multi-agents qui nous a servir de base pour les prochains TPs. Voici le diagramme de classes de ce TP:

UML et explications



Les classes ***Agent, Environnement*** et ***SMA*** sont communes à tout les TPs de ce projet. La classe ***Agent*** représente un agent de base qui possède une couleur et qui ne fait rien. Il contient une méthode abstraite *decide* qui va servir à choisir ce que va faire l'agent en fonction de son environnement, ainsi qu'une méthode *update* qui va effectuer l'action décidée par *decide*.

La classe ***Environnement*** contient un tableau à 2 dimensions contenant des agents. Elle contient également une méthode *init* qui va initialiser les agents grâce à la méthode abstraite *addNewAgent*. Cette méthode est abstraite pour permettre aux environnements spécifiques au

différents TPs de créer des agents qui leur son propre. Pour le TPs particules, cette méthode va créer des agents de type particules.

La classe ***SMA*** va servir à donner la parole aux agents. Elle contient une méthode *run* qui parcourt la liste des agents et exécute leur méthode *decide*, puis, quand tout les agents ont été appelés, Le classe ***SMA*** va appeler leur méthode *update*.

Une autre classe commune à tout les TPs, mais qui n'est pas présente dans l'UML, est la classe ***Parameters*** qui va contenir toutes les informations concernant le programme comme la taille de la grille d'agents ou le temps d'un tour.

Deux autres classes sont nécessaires au TP particules, la classe ***EnvironnementParticule*** qui va servir à créer les agents de type ***Particule,*** et la classe ***Particule*** qui va être un agent se déplaçant continuellement dans une direction et qui va se cogner aux murs ou aux autres particules.

Pour donner vie à ce comportement, la méthode *decide* de ***Particule*** va détecter les changements de direction en regardant si la particule va se cogner à un obstacle lors de son prochain déplacement. Si l'environnement est torique, la méthode *decide* va appeler *collisionMur.* Cette méthode va vérifier si la particule se cogne à un mur, et va changer la direction en fonction de l'emplacement du mur si il y a collision. La méthode *decide* va ensuite appeler *colilsionParticule* pour voir si la particule se cogne à une autre particule. Si c'est le cas, les deux particules vont s'échanger leur direction.

La méthode *update* va juste effectuer le déplacement dans la direction décidée par *decide* en allant chercher la prochaine case grâce aux méthodes *nextCaseTorique* ou *nextCaseNonTorique*.

Ce programme est capable de faire cohabiter plus de 20000 particules tout en gardant un taux de rafraîchissement raisonnable.

Maintenant que nous avons vu comment a été réalisé le premier TP, Nous allons maintenant passer au TP wator.

TP2 : Wator

Principe

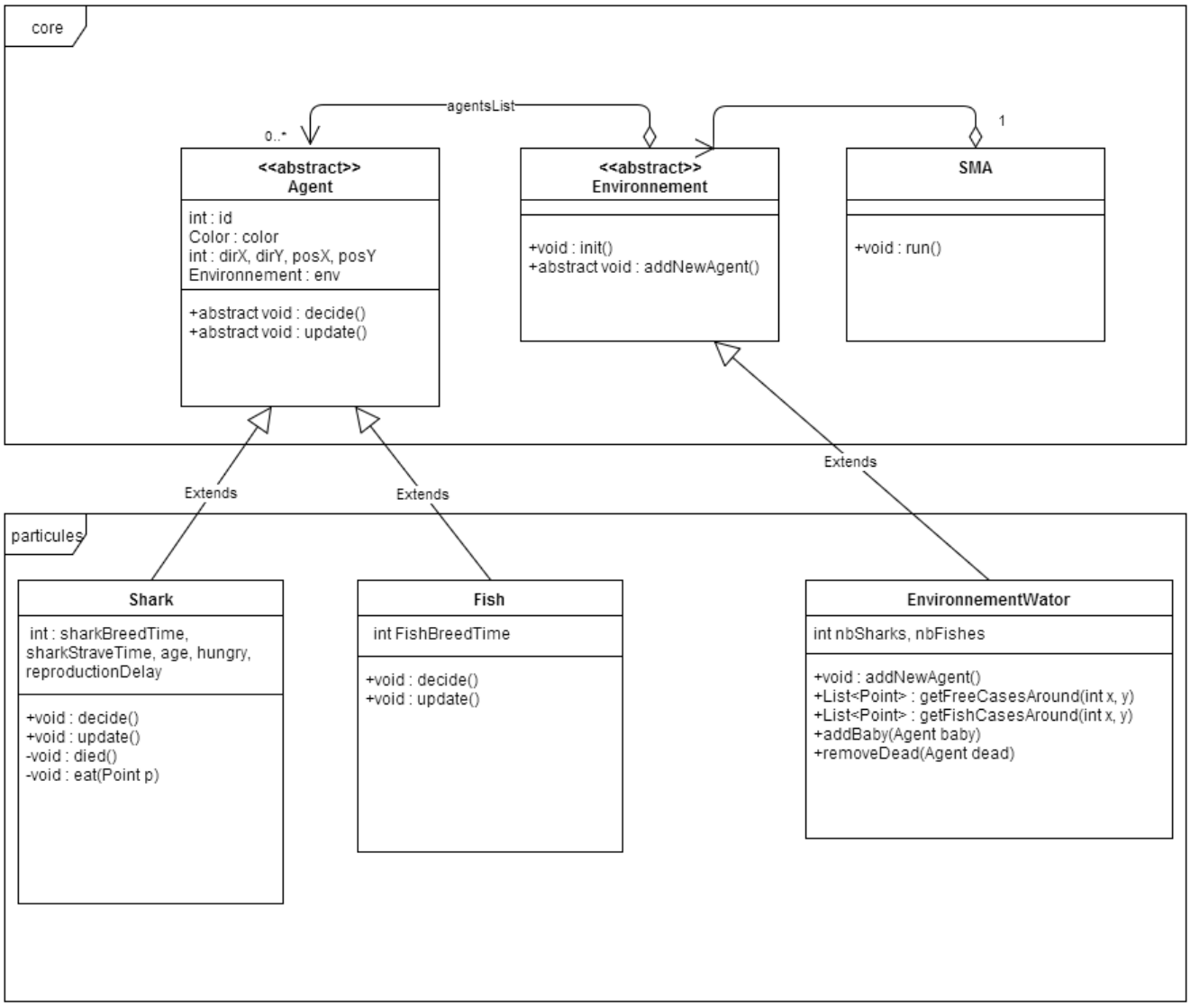
Le second système multi-agents créé a été le wator. Ce système consiste à simuler une population sous marine constituée de requins et de poissons. Les comportements des poissons et des requins vont être différents.

Les poissons vont se contenter de se déplacer si il y a une case libre autour d'eux et de se reproduire si ils se sont déplacés et ont atteint leur maturité.

Le comportement des requins est un peu plus compliqué. Ils vont vouloir en priorité manger un poisson si il en ont un autour d'eux. Ils vont ensuite se comporter comme un poisson en se déplaçant sur une case libre si ils n'ont pas mangés et en se reproduisant si ils se sont déplacés. Un autre point spécifique au requin est qu'il va mourir si il n'a pas mangé depuis un certain nombre de tour.

La difficulté de ce TP a été de régler l'âge de maturité des poissons et des requins ainsi que la durée de vie sans manger des requins afin d'avoir une simulation qui survit sans qu'une espèce ne s'éteigne.

UML et explications



L'environnement utilisé pour ce TP ajoute quelques méthodes utilitaires à l'environnement de base. L'***EnvironnementWator*** possède deux méthodes *getFreeCasesAround* et *getFishCasesAround* qui vont respectivement renvoyer les cases libres ou contenants un poisson autour de la case de coordonnées x et y. Elle possède également des méthodes permettant d'ajouter un nouvelle agent ou d'en retirer un en cas de naissance ou de mort d'un agent.

La méthode *addNewAgent* va quant à elle ajouter les ***Fishes*** et les ***Sharks*** souhaités à notre tableau d'agents.

Pour les ***Sharks***, la méthode *decide* est assez différente de ***particules***. Elle va tout d'abord vérifier si le requin peut survivre grâce à la nourriture qu'il a mangé. Ensuite, la méthode va regarder si il y a des poissons aux alentours grâce à *getFishCasesAround* et va en choisir un au hasard qui se fera manger par le requin. Si aucun poissons n'est disponible, *decide* va appeler la méthode *getFreeCasesAround* pour trouver un endroit libre où se déplacer.

La méthode *update* du requin va ensuite effectuer le déplacement choisi par *decide* et va ensuite regarder la dernière fois que le requin s'est reproduit pour savoir si il peut se donner naissance ce tour ci. Si c'est le cas et si le requin s'est déplacé, un nouveau requin va apparaître à l'endroit où était le requin au début du tour.

Les méthodes *decide* et *update* des poissons sont équivalente à celle des requins, homis l'absence de gestion de nourriture.

Paramètres optimaux et courbes

Pour obtenir une simulation équilibrée, plusieurs paramètres ont été testés. Il a fallut analyser les conséquences de chaque paramètres pour obtenir un équilibre idéal.

Les paramètres suivants donnent une simulation équilibrée : `SharkBreedTime = 6, SharkStarveTime = 5, FishBreedTime = 2.`

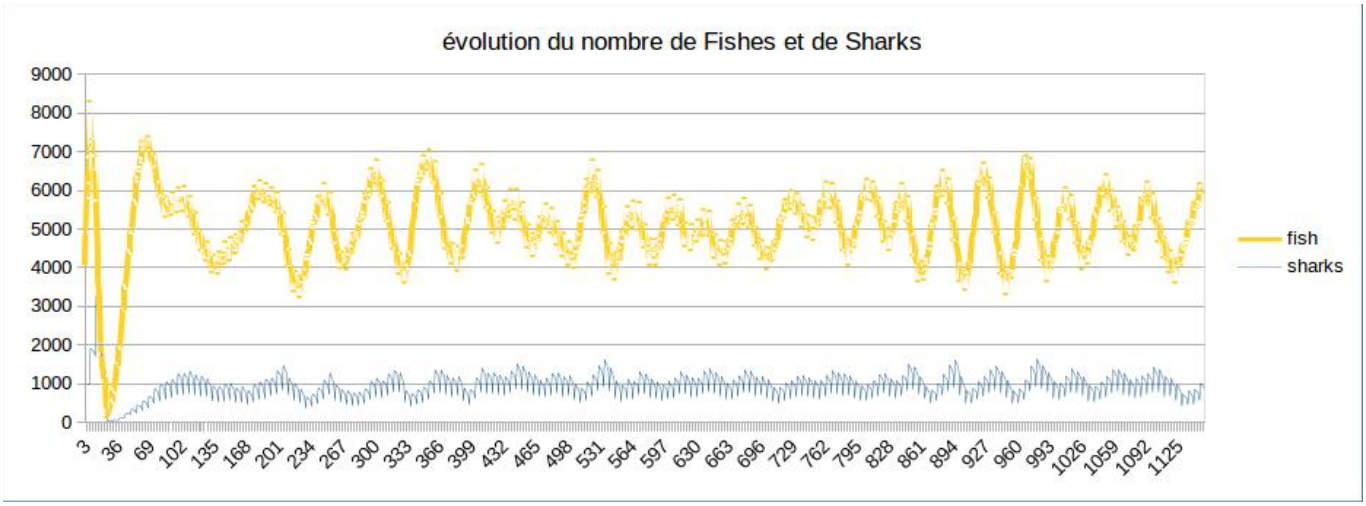
Si nous augmentons le temps de survit sans nourriture des requins, ils parviennent à rattraper les poissons et à les manger, les poissons n'ont plus le temps de se reproduire et leur population est éradiquée. L'effet est le même si nous diminuons l'âge de maturité des requins car le nombre de requins sera trop élevé pour que les poissons survivent.

La population de poisson est également éradiquée si nous diminuons leur âge de maturité. En effet, les requins ont alors toujours de la nourriture et ne meurent pas, leur population grandit et ils finissent par manger tout les poissons et mourir de faim, les deux espèces sont alors éteintes.

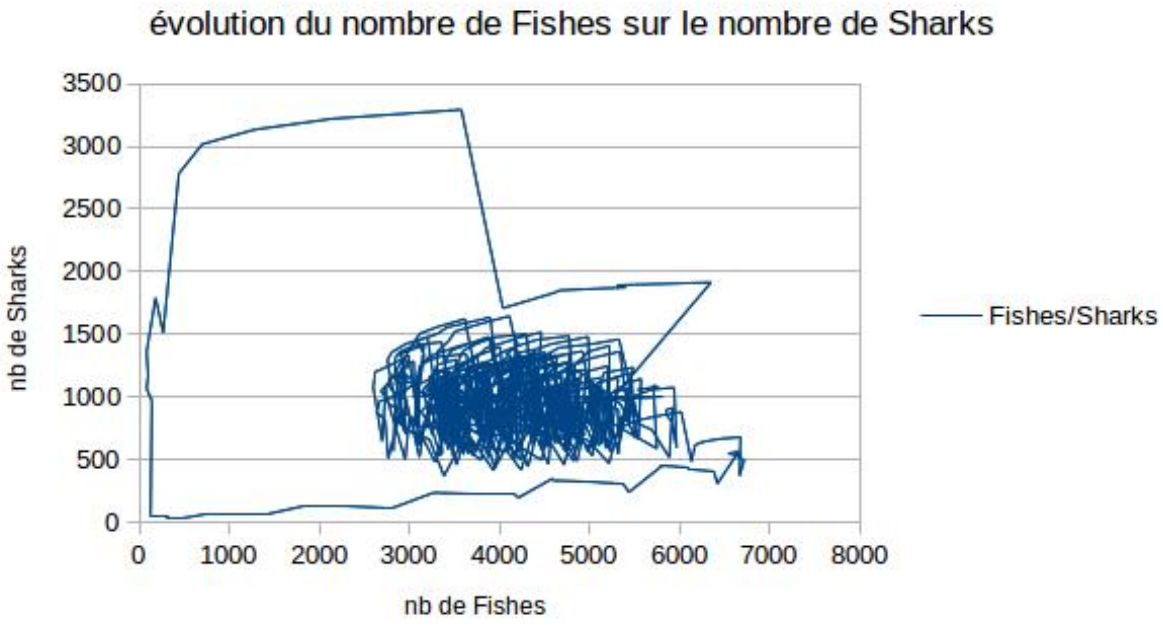
À l'inverse, si nous diminuons le temps de survit des requins ou augmentons leur âge de maturité, ils n'arrivent pas à se déplacer jusqu'aux poissons et meurent de faim au début de la simulation.

La simulation est viable si nous augmentons l'âge de maturité des poissons jusque 4, au-delà, les requins se retrouvent éloignés des poissons et meurt de fin avant de les atteindre.

La courbe d'évolution ci-dessous, représentant le nombre de poisson et de requin par rapport au temps, montre que les deux populations évoluent suivant un cycle. Le nombre de requins évolue de la même façon que celui des poissons mais avec un petit décalage. Cela s'explique car le nombre élevé de poissons permet au requins de manger, se reproduire et ne pas mourir. Quand le nombre de requins est trop haut, il n'y a plus assez de poissons pour les nourrir, le nombre de requin diminue donc jusqu'à ce que celui des poissons ré-augmente.



La courbe ci-dessous montre quant à elle le nombre de poissons sur le nombre de requins. La forme circulaire de cette courbe confirme les résultats obtenues avec la courbe précédente.



Ce TP nous a permis d'apprendre à régler les paramètres de nos agents pour avoir un environnement viable où aucun agent ne disparait.

Maintenant que ce TP vous a été présenté, nous allons passer au dernier TP concernant un jeu interactif basé sur cette architecture.

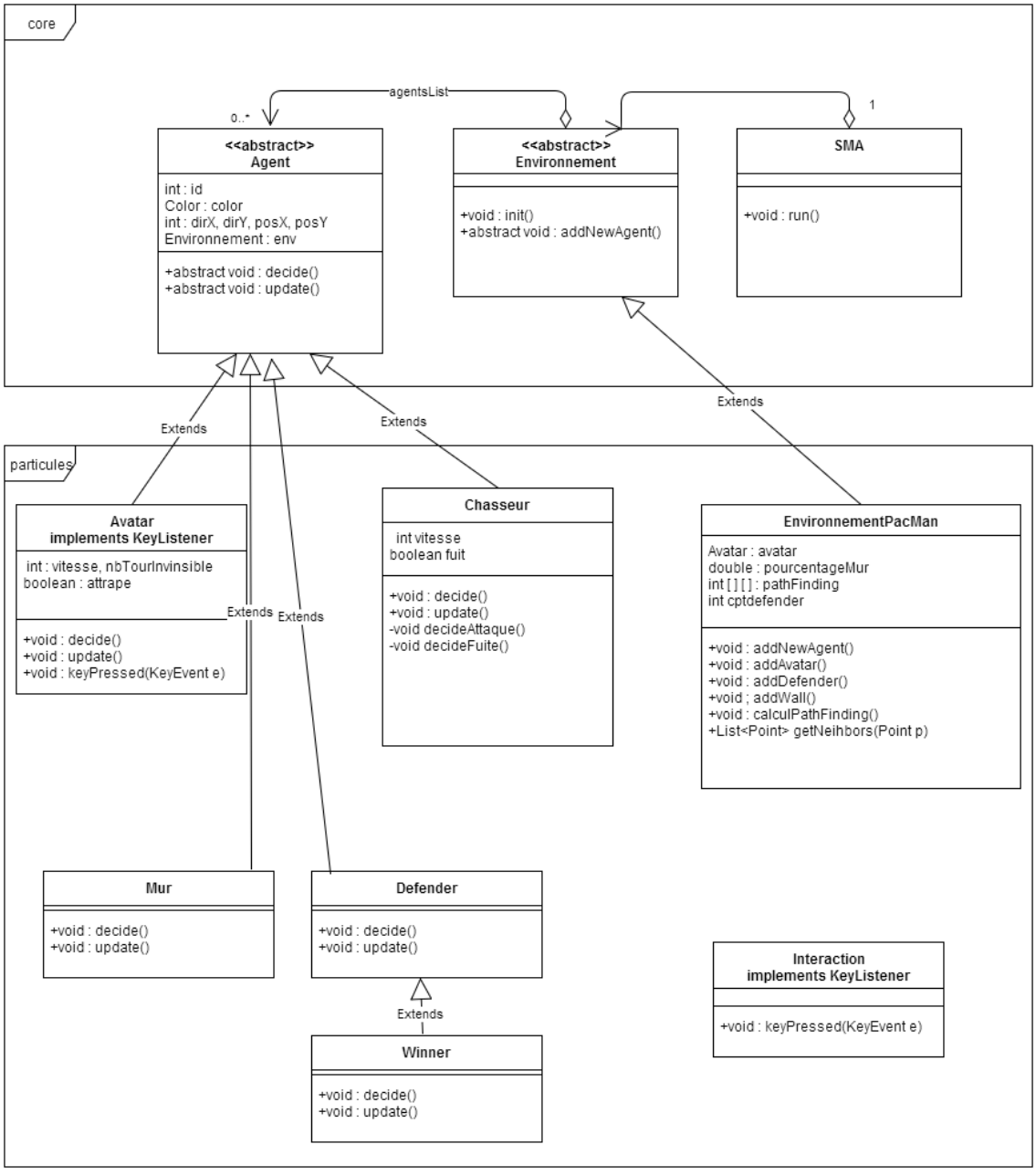
TP3 : Pac-Man

Principe

Le dernier TP qui a été réalisé a été un jeu de Pac-Man. Ce jeu consiste a déplacer un personnage nommé ***Avatar*** pour survivre a des agents nommés ***Chasseurs*** qui essayent de nous attraper. Pour gagner, le joueur doit récupérer des ***Defenders*** afin de faire apparaître un ***Winner*** qu'il faudra également attraper pour gagner la partie.

Ce TP nous a permis de réaliser un système interactif où des agents autonome réagissent au mouvement de l'agent contrôlé par le joueur.

UML et explications



Pour ce TP, nous utilisons encore une fois un environnement différent. Celui-ci permet d'ajouter au jeu tout les types d'agents qui lui sont nécessaire. L'***EnvironnementPacMan*** contient également une méthode *calculPathFinding* qui va calculer le chemin le plus cours vers l'avatar. Le calcul du pathFinding est réalisé à chaque déplacement de l'***Avatar*** et a été fait de la manière suivante :

```
initialiser le numero de chaque case à -1;
initialiser une linkedList (list FIFO);
pour la case où se situe l'avatar {
     mettre son nombre à 0;
     l'ajouter à la LinkedList;
}
tant que la linkedList n'est pas vide {
     pour chaque voisin v de e {
          si numéro de v = -1 ou > à numéro de e +1 {
               numéro de v = numéro de e + 1;
               ajouter v à la linkedList
          }
     }
}
```

Concernant les ***Agents***, ils sont au nombre de 5 dans ce jeu. Les trois Agents les plus simples sont le ***Mur***, le ***Defender*** et le ***Winner***. Le ***Mur*** a un comportement vide tandis que les ***Defender*** et ***Winner*** compte les tours pour disparaître et faire apparaître un nouveau ***Defender/Winner*** quand leur temps d'apparition est écoulé. Ils ne se déplacent donc pas.

L'agent ***Avatar*** possède un comportement assez simple lui aussi. Il va détecter les frappes du clavier et retenir la dernière direction indiquée par l'utilisateur grâce au flèches directionnelles. La méthode *decide* va alors vérifier, quand elle sera appelée, si l'agent peut aller dans la direction indiqué ou si il est bloqué par un ***Mur*** ou par les bord de la carte. Si il peut se déplacer, la méthode *update* va effectuer le déplacement. Elle va également regarder si l'***Avatar*** s'est déplacé sur un ***Defender***. Dans ce cas, l'***Avatar*** passera en mode « attrapé », les ***Chasseurs*** le fuiront et le prochain ***Defender*** apparaîtra. Si 4 ***Defenders*** sont attrapés, un ***Winner*** apparaîtra pour offrir la victoire au joueur si celui-ci l'attrape.

Pour l'agent ***Chasseur***, la méthode *decide* va regarder si l'***Avatar*** est en mode « attrape ». Si c'est le cas, elle va appeler la méthode *decideFuite* et sinon, elle va appeler *decideAttaque*.

La méthode *decideAttaque* va regarder le nombre de case séparant le ***Chasseur*** de l'***Avatar*** et sélectionner la case libre la plus proche. La méthode *decideFuite* va faire exactement l'inverse et choisir la case la plus éloignée de l'***Avatar***.

Ensuite, la méthode *update* va vérifier que l'***Avatar*** n'est pas présent sur la case où le ***chasseur*** veut se déplacer. Si il est présent sur cette case, la partie est terminée et le joueur a perdu. Sinon, le ***Chasseur*** va se déplacer sur cette case.

Ce jeu donne aussi la possibilité au joueur de changer certains paramètres de la partie à la volée. Par exemple, en appuyant sur A, le joueur peut accélérer la vitesse des ***Chasseurs***. Il peut également la ralentir avec Z, accélérer ou ralentir l'***Avatar*** avec O et P ou accélérer/ralentir la totalité du jeu avec W et X.

Ces modifications vont être réalisées par la classe ***Interaction*** qui va détecter les frappes du joueur sur le clavier et changer les paramètres en conséquence.

Ce TP nous a permis d'utiliser des interactions avec l'utilisateur dans un environnement composé d'***agents*** autonomes.

Voilà pour ce dernier TP, je vais maintenant vous expliquer comment lancer le programme et quel paramètres vous pouvez modifier grâce aux lignes de commande.

Exécution

Pour exécuter le TP, vous devez entrer la commande « java -jar SCI_Charneux_dimitri » dans un terminal.

Pour choisir le tp que vous voulez lancer, il faut ajouter un des arguments suivants juste après la commande de lancement : - « particules » pour le TP sur les billes.
- « wator » pour le TP wator.
- « pacman » pour le dernier TP.

Vous pouvez également entrer un des arguments suivants pour modifier les paramètres de base du programme :

gridSizeX [nombre] pour modifier le nombre de cases en X (100 par défaut).
gridSizeY [nombre] pour modifier le nombre de cases en Y (100 par défaut).
canvasSizeX [nombre] pour modifier la largeur de la fenêtre (adaptative par défaut).
canvasSizeY [nombre] pour modifier la hauteur de la fenêtre (adaptative par défaut).
boxSize [nombre] pour modifier la taille d'une case (5 par défaut).
delay [nombre] pour modifier le temps de déroulement d'un tour (50 par défaut).
scheduling ["equitable"|"sequentiel"|"aleatoire"] pour modifier le mode de scheduling (équitable par défaut).
nbTicks [nombre] pour modifier le nombre de tour, mettre 0 pour infini (0 par défaut).
grid [booléen] pour afficher ou non la grille (true par défaut).
trace [booléen] pour afficher la trace (true par défaut).
seed [nombre] pour utiliser une graine pour le générateur aléatoire, mettre 0 pour aléatoire (0 par défaut).
refresh [nombre] pour modifier la fréquence de rafraîchissement de la fenêtre, mettre 1 pour rafraîchir à chaque tour (1 par défaut).
nbParticles [nombre] pour modifier le nombre de billes ou de chasseurs de la grille (30 par défaut).
torique [booléen] pour mettre la grille en torique ou non, indisponible pour le dernier mode (non torique par défaut).
nbSharks [nombre] pour modifier le nombre de shark de la grille (1000 par défaut).
nbFishes [nombre] pour modifier le nombre de fish de la grille (4000 par défaut).
sharkBreedTime [nombre] pour modifier le temps en tick que met un shark pour se reproduire (6 par défaut).
fishBreedTime [nombre] pour modifier le temps en tick que met un fish pour se reproduire (2 par défaut).
sharkStarveTime [nombre] pour modifier le temps en tick durant lequel peut survivre un shark sans manger (5 par défaut).

Vous pouvez également utiliser les paramètres par défaut de chaque TP en entrant « particuleDefaut », « watorDefaut » ou « pacmanDefaut » après le nom du Tp choisit.

Un exemple pour lancer le TP wator : « java -jar SCI_Charneux_dimitri wator watorDefaut »

Conclusion

Cette série de TPs nous a appris à réaliser des systèmes multi-agents où chaque agent à le contrôle de lui même. Nous savons maintenant comment est réalisé une intelligence artificielle basique et les conséquences que peut avoir une petite modification sur le système entier. Nous sommes également capable de réaliser des agents réagissant aux actions d'un utilisateur.