

Exercice 01-04 –

Conception préliminaire

1. <u>CONCEPTION PRÉLIMINAIRE</u>	<u>2</u>
1.1. L'AGENT	3
1.2. LE FACTORY D'AGENT	4
1.3. L'ESPACE DE TORE	4
1.4. LE VISUALISEUR D'UN ESPACE DE TORE	5
1.5. L'IHM du joueur	5

1. Conception préliminaire

Avant de réaliser les diagrammes de classe à partir du besoin et du diagramme de communication, il est nécessaire de faire une conception préliminaire afin de préciser ou simplifier le besoin, de mettre en évidence certaines contraintes, de caractériser les classes principales et leurs rôles.

Le nombre d'agent créés par joueur doit être le même pour une partie donnée.

Afin de simplifier le codage, nous choisissons de ne pas coder cette contrainte. Elle reste donc une recommandation d'usage. D'autant que cela permet de faire des tests avec des populations d'agent quelconques.

Il n'y a pas de contrainte du nombre de joueur maximum bien que dans la réalité il faudrait le limiter en fonction de la taille de la grille, du nombre d'agents créés par joueur et des capacités de l'ordinateur. Nous ne limitons pas le nombre max de joueur. Dans la pratique il peut y avoir au moins jusqu'à 3 à 4 joueurs.

Il devrait être impossible que deux joueurs puissent avoir la même couleur. Cela reste une recommandation d'usage. Chaque joueur choisit une couleur différente des autres joueurs avant de créer ses agents.

Chaque agent est un thread dont le traitement (run) réalise le traitement suivant :

```
Tantque le nombre de vie de l'agent > 0 faire
  Demander à l'espace de tore de faire un déplacement
  dans une direction donnée;
Fintantque
```

Etant donné que deux agents ne doivent pas occuper en même temps une même case de l'espace, il est nécessaire de centraliser et de synchroniser le traitement de déplacement (synchronized) et la mise en attente d'un thread (wait).

Chacun des agents réalise ses déplacements successifs avec un certain rythme afin que les déplacements soient visibles par un œil humain. Cette fréquence est réalisée en attendant une certaine durée entre chaque déplacement. Ce temps d'attente doit être toujours le même quel que soit l'agent. Ainsi tous les agents ont la même vitesse de déplacement (sinon ce serait tricher).

Par contre ce temps d'attente ne doit pas se trouver dans la méthode synchronisée de déplacement mais il doit quand même être centralisé (le même pour tous et impossible de tricher). Ainsi chaque agent appellera dans un premier temps une méthode centralisée non synchronisée qui appelle la méthode synchronisée de déplacement, et en retour attends le temps d'attente donnée. Ainsi, ce n'est pas parce qu'un agent attend le prochain cycle que les autres ne peuvent pas faire aussi leur demande de déplacement.

Nous supposons que le nombre de vie initial de tous les agents est le même, égal à 10 par défaut. Cette valeur est centralisée et sera donc la même pour tous les agents sans possibilité de tricherie.

Un joueur pourra connaître le nombre de morts de sa population d'agent. Cette information peut être, dans un premier temps, une simple information statistique, mais elle peut être utilisée pour qu'un joueur puisse adapter le comportement de ses agents en train de se battre en jouant sur des paramètres de déplacement de ses agents.

Il est à noter que cette information peut être calculée par chaque joueur puisqu'il a accès aux attributs de tous ses agents. Il connaît le nombre d'agent qu'il crée, le nombre

de vie de chacun, ... Mais le plus simple est de faire calculer cette information par le système qui sera ainsi disponible pour tous les joueurs sans effort de codage de leur part.

Il ne doit pas être possible de créer d'agents supplémentaires en cours de combat.

Lors de sa création, grâce à une méthode synchronisée (sur le même verrou que le déplacement), l'agent est positionné sur une position aléatoire dans la grille.

Le mode combat est un booléen qui par défaut est à faux. Ainsi une partie se déroule en deux étapes :

- une première étape pendant laquelle chacun des joueurs crée des agents de couleurs différentes (une couleur différente par joueur). Le mode combat est à faux et les agents ne se battent pas c'est-à-dire que si un agent essaye de se déplacer sur un agent ennemie (de couleur différente) alors le déplacement est refusé. Si la case est libre, le déplacement se fait et est réussi.

- une deuxième étape qui commence dès que le système passe le mode combat à vrai (via l'IHM de contrôle). Dans ce cas, si un agent essaye de se déplacer sur un agent ennemi (de couleur différente) alors le nombre de vie de l'agent ennemi est décrémenté, l'agent reste à sa place et cela est vu comme un déplacement réussi.

Quel que soit l'étape, si un agent essaye de se déplacer vers une case occupée par un agent de sa famille (même couleur) alors le déplacement est refusé. Mais cela compte pour un temps de déplacement

Quel que soit l'étape, si l'agent essaye de se déplacer vers une case libre alors le déplacement est réussi. L'agent se déplace.

Quand le nombre de vie d'un agent passe à 0 alors l'agent meurt : le thread de l'agent se termine.

La partie s'arrête quand on est en mode combat et qu'il n'existe dans la grille que des agents d'un même joueur, il est alors le gagnant.

Il peut exister le cas où il reste très peu d'agent de couleurs différentes et ils n'arrêtent pas de se poursuivre sans arriver à se rencontrer pour se battre. Dans ce cas, si on est en mode combat et que pendant N cycles, il n'y a eu aucune vie décrémentée alors la partie s'arrête et le gagnant est celui qui a le plus d'agents. En cas d'égalité d'au moins N (≥ 2) joueurs, si $N < \text{nombre de joueurs}$ alors les N joueurs sont déclarés vainqueurs et les autres joueurs sont déclarés perdants; si $N = \text{nombre de joueurs}$, alors on déclare un match nul.

Le système n'interdit pas que 2 joueurs se liguent contre un autre, par exemple, en privilégiant ses déplacements au profit d'un autre.

Chaque joueur est avant tout un programmeur qui code sa propre IHM et donc conçoit les algorithmes de déplacement de ses agents dont le code est injecté dans le système. On peut donc imaginer des IHM complexes (différentes pour chaque joueur) permettant de créer des agents avec des comportements différents en fonction de certaines situations, et même des boutons permettant de changer le comportement de ses agents en cours de combat.

Ainsi, la perspicacité de codage du joueur fait partie de la règle du jeu. Ainsi, ce sera, sûrement, celui qui aura imaginé les meilleurs algorithmes de combat qui gagnera.

1.1. L'agent

Un **agent** est un composant dynamique (thread) de notre architecture.

Cet agent :

- maintient son état local (coordonnées x et y de sa position, le nombre de ses vies)
- **est un objet distant** dont l'interface distante permet de :
 - connaître son nombre de vie
 - changer son nombre de vie
 - connaître sa position x y
 - changer sa position x y
- réalise dans le thread cycliquement (tempo millisecondes) un déplacement dans une grille de tore.

Le principe que nous adoptons sera le suivant :

- une classe **Agent** abstraite qui crée un thread dont le run consiste à appeler une méthode, **deplacer**. La méthode **deplacer** réalise par défaut un déplacement aléatoire.
- une classe d'agent développée par un joueur hérite de cette classe abstraite et surcharge la méthode **deplacer**. A sa charge donc de réaliser sa stratégie de déplacement.

Dans la classe **Agent** sera codé des déplacements types, simple que chaque joueur pourra utilisés s'il le souhaite : se déplacer aléatoirement, se déplacer vers l'ennemie le plus proche, se déplacer d'une manière groupée avec ses semblables, se déplacer à l'opposé de ses ennemis (fuir), ...

1.2. Le factory d'agent

Les agents sont créés, à la demande d'un joueur, par un **factory d'agent** (qui est un objet distant). Il y a autant de factory que de joueur.

L'interface distante de ce factory permet de :

- Créer N agents de couleur C
- Retourne le taux de mortalité (nombre d'agent mort / nombre d'agent créé)

1.3. L'espace de tore

Le déplacement des agents est centralisé dans un objet distant, un **espace de tore** qui maintient en local une grille NbX x NbY dont chaque case mémorise :

- La couleur de l'agent
- La référence distante à l'agent.

Chaque agent demande à cet espace de tore de réaliser son déplacement suivant les principes suivants :

- Le traitement de déplacement appelle une méthode synchronisée afin que deux agents ne réalisent pas un déplacement en même temps
- Ensuite le traitement se met en attente un certain temps paramétrable.
- En entrée du traitement de déplacement, on a la couleur de l'agent, sa position courante (x,y) et une direction (parmi les 8 directions)
- L'espace de tore calcule dans un tore la position de destination (px,py)

- Ensuite il réalise l'algorithme suivant :
 - si la case de destination n'est pas occupée par un autre agent alors
il met à jour la grille en déplaçant le contenu de la case.
 - sinon
 - si on est en mode combat alors
 - si la position de destination est occupée par un agent de couleur différente alors
il décrémente le nombre de vie de cet agent (à la charge de l'agent de se tuer si son nombre de vie est à 0)
 - sinon rien
 - sinon rien
- Finsi
- La méthode retourne la nouvelle position de l'agent, null si le déplacement n'a pas été possible. Si différent de null alors l'agent change ses attributs x,y.
Une valeur null retournée permet à l'agent de réaliser (au coup suivant) éventuellement une variante de déplacement si le déplacement demandé n'a pas été possible.

De plus cet espace de tore, centralise les informations suivantes qui peuvent être changées à distance par l'ihm de contrôle :

- le mode combat (vrai, faux)
- le temps d'attente entre chaque déplacement d'un agent
- le nombre de vie initial de tous les agents

L'interface distante de cet OD permet de :

- demander de réaliser un déplacement
- changer la valeur de tempo
- demander le nombre de vie initial
- demander le mode combat
- connaître la taille de la grille (NbX NbY)
- connaître la position de tous les agents d'une couleur donnée ou tous
- connaître la position de tous les agents qui ne sont pas d'une couleur donnée

Ces deux dernières méthodes permettent à un agent de calculer (stratégie) son sens de déplacement en fonction de la position des autres agents (amis ou ennemie)

L'espace de tore utilise une ihm locale, **IHM de l'espace de tore**, pour saisir la taille de la grille (NbX NbY), la valeur de tempo et le mode combat/non combat. Cette IHM exécute l'OD.

1.4. Le visualiseur d'un espace de tore

Pour visualiser le déplacement des agents, un **visualiseur d'un espace de tore**, indépendant des autres composants affiche dans une grille dessinée la position de chaque agent.

Pour rafraichir cette grille, il existe deux solutions :

- soit l'IHM est prévenu à chaque déplacement d'un agent de sa position précédente et suivante (et sa couleur)
- soit l'IHM demande cycliquement à l'espace de tore la couleur et la position de tous les agents et rafraichit sa grille en conséquence.

Nous décidons de coder les deux solutions dans deux ihm différentes afin de faire un test de comparaison des deux solutions.

1.5. L'IHM du joueur

Chaque joueur utilise une IHM, IHM de joueur. Cette IHM lui permet de :

- se connecter à son factory
- demander au factory de créer ses agents (*).
- choisir sa couleur
- affiche diverses informations statistiques (taux de mortalité des agents par ex) pour suivre l'évolution de ses agents.

(*) Nous envisageons que chaque joueur développe sa propre IHM et ses propres classes d'agent dans lequel il code ses stratégies de déplacement.

Dès lors il est nécessaire qu'un factory (qui est une même classe pour tous les joueurs mais une instance différente pour chaque joueur) charge dynamiquement les classes qui ont été développées par chacun des joueurs.

Remarque Java : il est nécessaire que le path de package et d'accès aux classes d'agent d'un joueur ne soient pas les mêmes qu'un autre joueur.