**HÉRAÏZ-BEKKIS Daphné**

# Projet
# Fondement des SYstèmes Multi-Agents

# RAPPORT

Paris
9 Mai 2016

## ***Sommaire :***

# i. Introduction

Dans la nature, les êtres vivants ont toujours eu à s'organiser pour survivre : pour chasser chez les humains, se déplacer chez les antilopes, garder la maisonnée chez les manchots… Les espèces n'ayant pas validé la contrainte d'organisation n'ont de fait pas vécu longtemps. L'organisation se construit selon les qualités et défauts des inddividus, et selon leurs capacités physiques, mentales et d'adaptation.

De nos jours, il nous est aussi possible d'observer cette même forme d'organisation au sein des êtres humains, les différentes professions se complétant mutuellement : un facteur ne pouvant survivre sans un boulanger qui lui a besoin d'un producteur qui lui a besoin d'un garagiste qui lui a besoin d'un physicien…

Avec le développement de plus en plus courant de nouvelles technologies, de plus en plus spécialisée, intelligente, autonome et extravagante, avec des décénies de recherche derrière et à venir, les comportements des êtres vivants sont de plus en plus copiés (humains et animaux) : se baser sur la capacité qu'ont certains scarabées dans les déserts à récupérer l'eau à l'aide de leur carapace pour construire des filets optimaux de récupération des eaux, construire des oiseaux de fer géants basés sur la physique qu'utlise naturellement les oiseaux pour déplacer un ensemble de personnes par les airs, ou encore copier l'habilité qu'on les êtres vivants à s'organiser pour survivre selon leur environnement extérieur et leur contact avec les autres pour créer des agents intelligents autonomes.

De par ce dernier exemple, le projet qui va suivre fait état du développement d'un système simplifié et peu avancé d'auto-gestion d'agents. En effet, il s'agit d'agents, c'est-à-dire d'entités informatiques (donc virtuelles) interragissant avec et selon leur environnement, se mouvant sur une plateforme spécifique, dans une carte qui leur est initialement inconnue et avec pour simple but de ramasser le plus de trésors en collaboration et au total avec leurs camarades. Chacun peut ainsi se retrouver avec des choix différents, dans des situations différentes mais toujours avec le même objectif commun : accumuler le plus grand trésor commun.

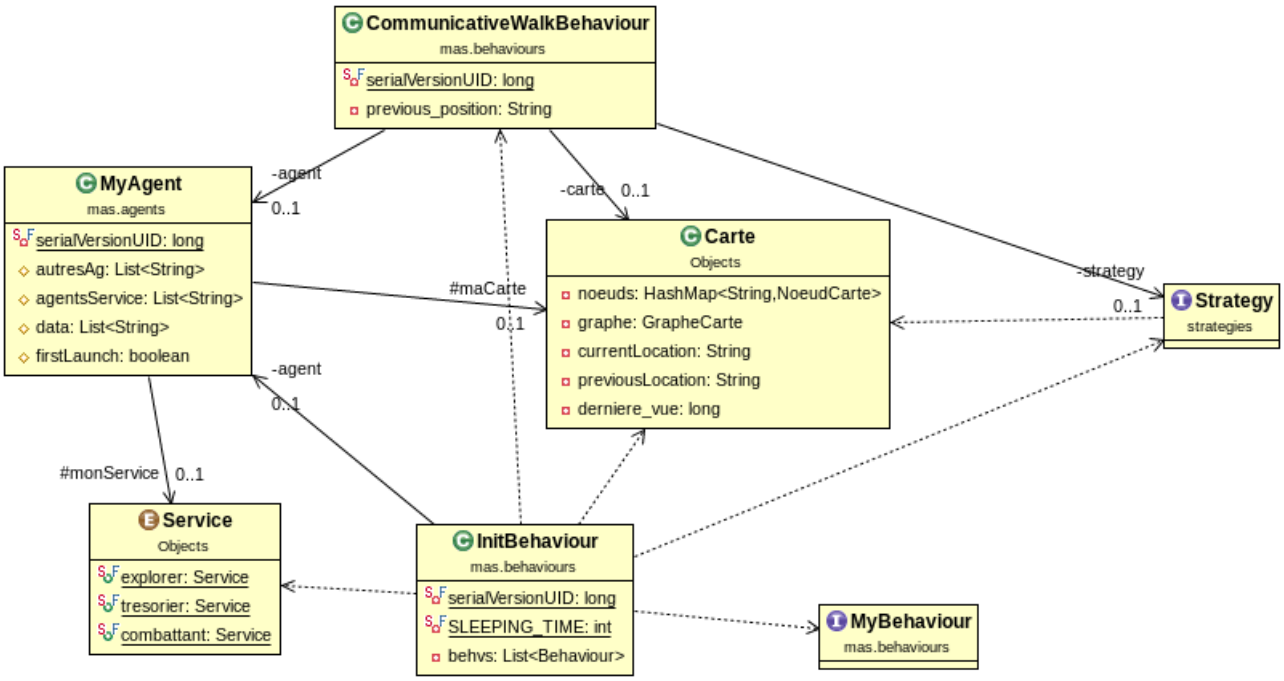
Le rapport qui va suivre présente des choix et solutions réfléchies mais n'ayant pas tous et toutes été implémentés dans le projet concret en archive, qui, comme dit plus haut, n'est pas très avancé.

## ii. Architecture générale du code

Ci-dessous l'arborescence de fichiers du projet en archive :

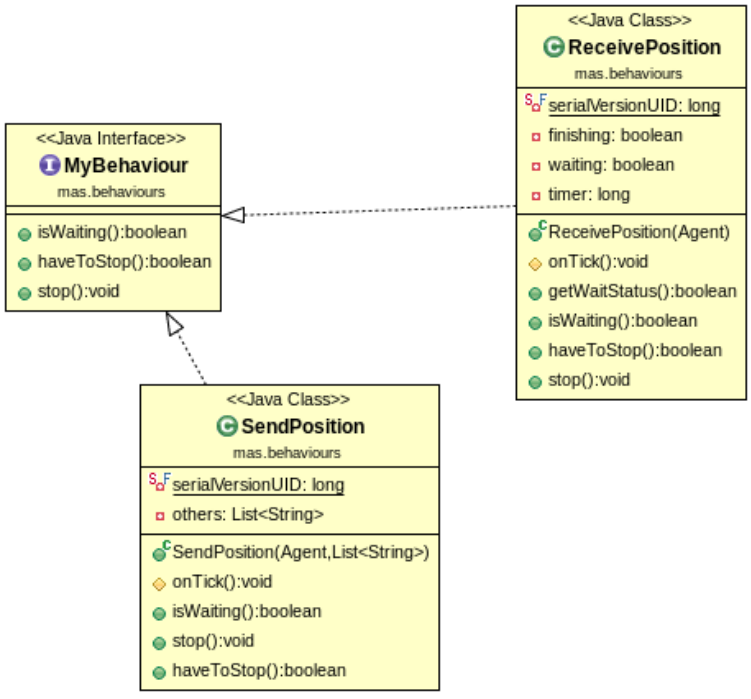
```
src/
├── Exceptions
│   ├── ExplorationClosedException.java
│   └── MeetingExpectedException.java
├── graphique
│   ├── TestCreateGraphGS.java
│   └── TestGraphS.java
├── mas
│   ├── agents
│   │   ├── DummyExploAgent.java
│   │   ├── DummyTalkingAgent.java
│   │   ├── DummyWumpusAgent.java
│   │   └── MyAgent.java
│   └── behaviours
│       ├── CommunicativeWalkBehaviour.java
│       ├── ConversationBehaviour.java
│       ├── ExploreBehaviour.java
│       ├── InitBehaviour.java
│       ├── MyBehaviour.java
│       ├── RandomWalkBehaviour.java
│       ├── ReceivePosition.java
│       ├── SayHello.java
│       └── SendPosition.java
├── Objects
│   ├── Carte.java
│   ├── GrapheCarte.java
│   ├── NoeudCarte.java
│   ├── Service.java
│   └── Tresor.java
├── princ
│   ├── PrincipalD.java
│   └── Principal.java
└── strategies
    ├── exploring
    │   ├── CommunicativeStrategy.java
    │   ├── NonVisitedDoubleCheckStrategy.java
    │   ├── NonVisitedSimpleStrategy.java
    │   ├── RandomStrategy.java
    │   └── ShorterWayStrategy.java
    ├── PickingTreasuresStrategy.java
    └── Strategy.java
```

Ci-après le diagramme UML général du code, présentant certaines modifications au sujet de l'héritage d'un agent.



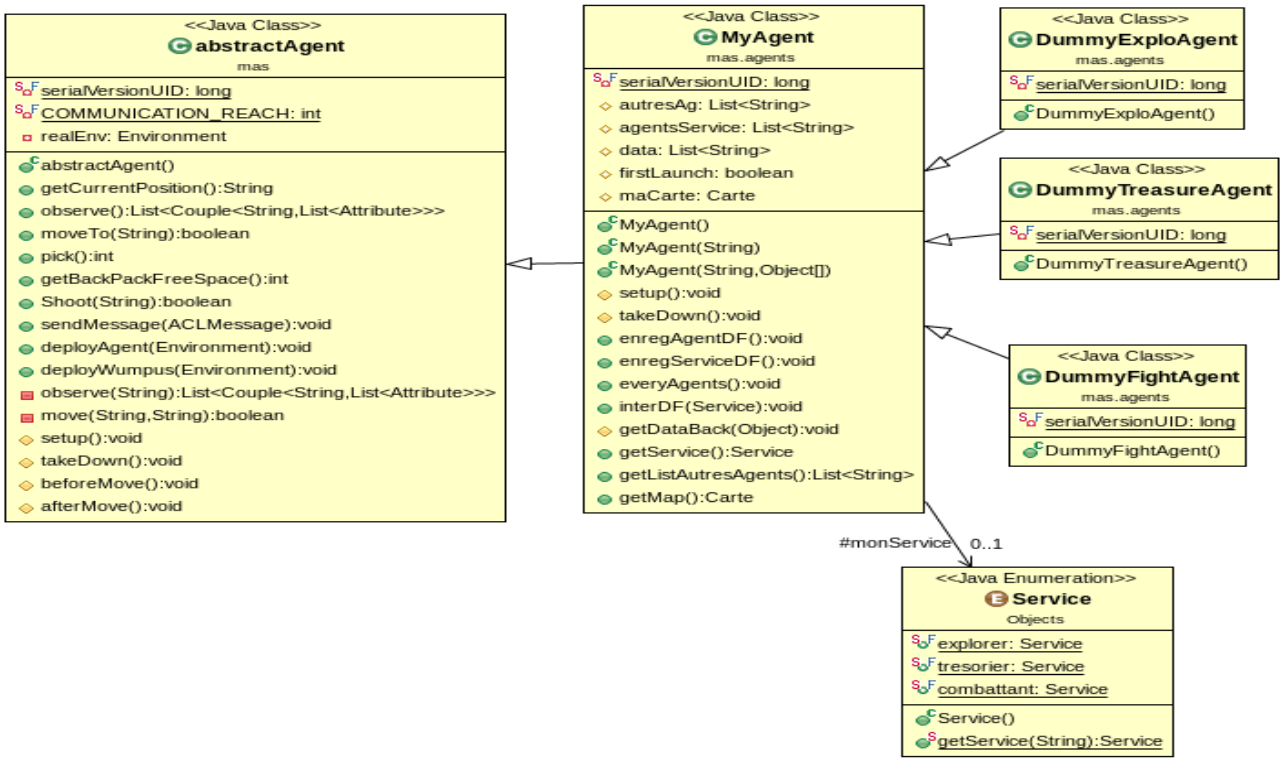

(diagramme UML global simplifié -classes principales)

MyAgent étant la représentation de l'agent explicitée plus bas.
InitBehaviour le comportmenet de démarrage.
CommunicativeWalkBehaviour étant un des comportement proposé (théoriquement – le mieux implémenté).
Service est l'énumération représentant les services pouvant être proposés par un agent.
Strategy est l'interface des stratégies utilisées pour réaliser les comportements.
Carte est la représentation de la carte.
MyBehviour représente les TickerBehaviour n'ayant aucune influence volontaire sur l'agent.



# iii.Structure (Agents et comportements)

## *1. Agents*




(Diagramme présentant 3 types d'agents)

Dans le programme multi-agents recherchés, tous les agents lancés doivent être des mas.abstractAgents pour être utilisables par la plateforme Jade. Or ceux-ci ne bénéficie pas d'une architecture commune très évolutive.
Pour palier au manque de la redéfinition d'un agent, la classe mas.agents.MyAgent a été créée. En effet, cette classe est utilisable sur la plateforme Jade car elle est un abstractAgent :

```
class MyAgent extends abstractAgent
```

De plus, cette architecture permet la présence d'objets communément à tous les agents : une Carte, un service (bien que différent pour chacun), une liste des autres agents ou ceux offrant un service précis.

```
// exploration par defaut
protected Service monService=Service.explorer;

protected List<String> autresAg = new ArrayList<String>();
protected List<String> data = new ArrayList<String>();
// pour s'enregistrer sur le df et recup les autres agents
protected boolean firstLaunch = true;
// creation de la carte
protected Carte maCarte = new Carte();
```

Chaque agent, héritant de la classe MyAgent, n'a pour lui que le service qu'il propose à son lancement :

```
public class DummyExploAgent extends MyAgent{

	private static final long serialVersionUID = 1L;

	public DummyExploAgent(){
		super();
		this.monService = Service.explorer;
	}
}
```

De ce fait, la classe MyAgent comporte donc les méthodes importantes pour le lancement et l'arrêt d'un agent : setup() et takeDown() qui seront donc les mêmes pour tout les types d'agents, ie tous les services représentés initialement. Par ailleurs, cette architecture est appuyé par le fait qu'à leur lancement tous les agents ont le même comportement pour un seul tick : InitBehaviour qui permet à l'agent de s'enregistrer sur les pages jaunes, s'endormir InitBehaviour.SLEEPING_TIME (=1000 millisecondes) puis charger la liste des autres agents qui sont dans la carte.

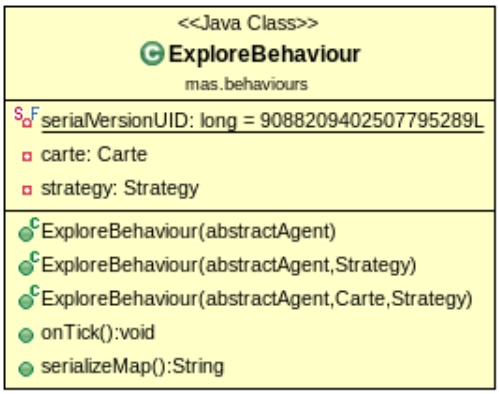
Le choix des pages jaunes s'est fait par simplificité d'implémentation et de gestion des communications entre agents. En effet, chaque agent se distingue par le service qu'il propose sur le moment et donc celui avec lequel il s'esst enregistré. Par exemple, dans le cas où il y aurait un service soigneur fournit par un nombre très limité d'agents et qui permettrait à un agent d'être soigné de blessure obtenues en attaquant le Wumpus (de l'imagination) et que lorsqu'un agent est blessé, celui-ci est contraint de se faire soigner avant un certain nombre de tick ; dans ce cas-là, il est préférable pour le blessé de ne pas perdre de ticks à engager une conversation avec un agent n'étant pas soigneur.

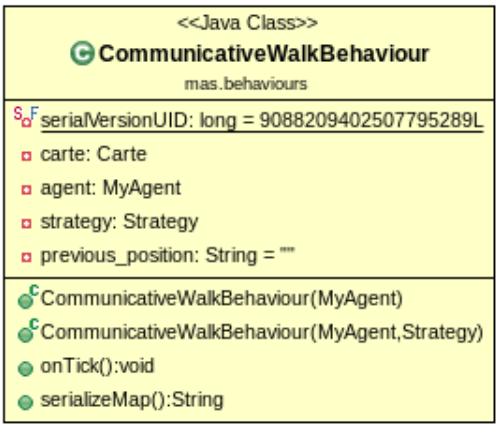
## *2. Comportements*

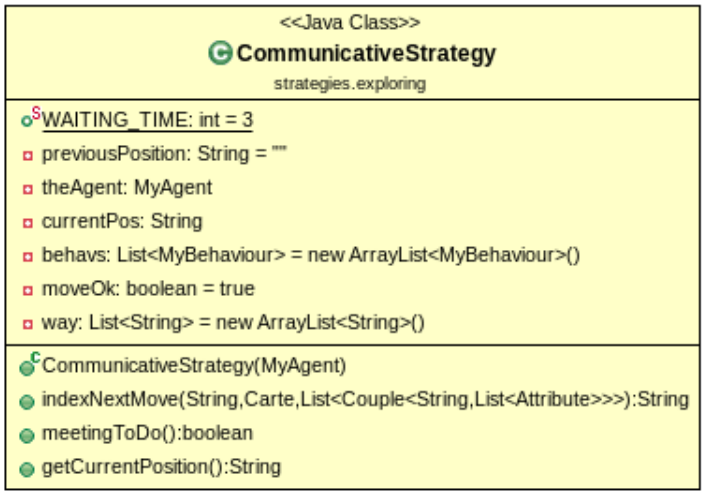
Plusieurs types d'agents (talking, random, treasure, fight…).

Ici les Behaviour principaux sont utilisés comme le serait les metiers pour les etres humains.

Ainsi, un agent peu simplement utiliser un comportement avec une stratégie différente d'un autre : leur but sera le même, mais la façon de l'atteindre sera, elle, plus ou moins différente, optimale ou encore optimisant 2 comportement distincts.



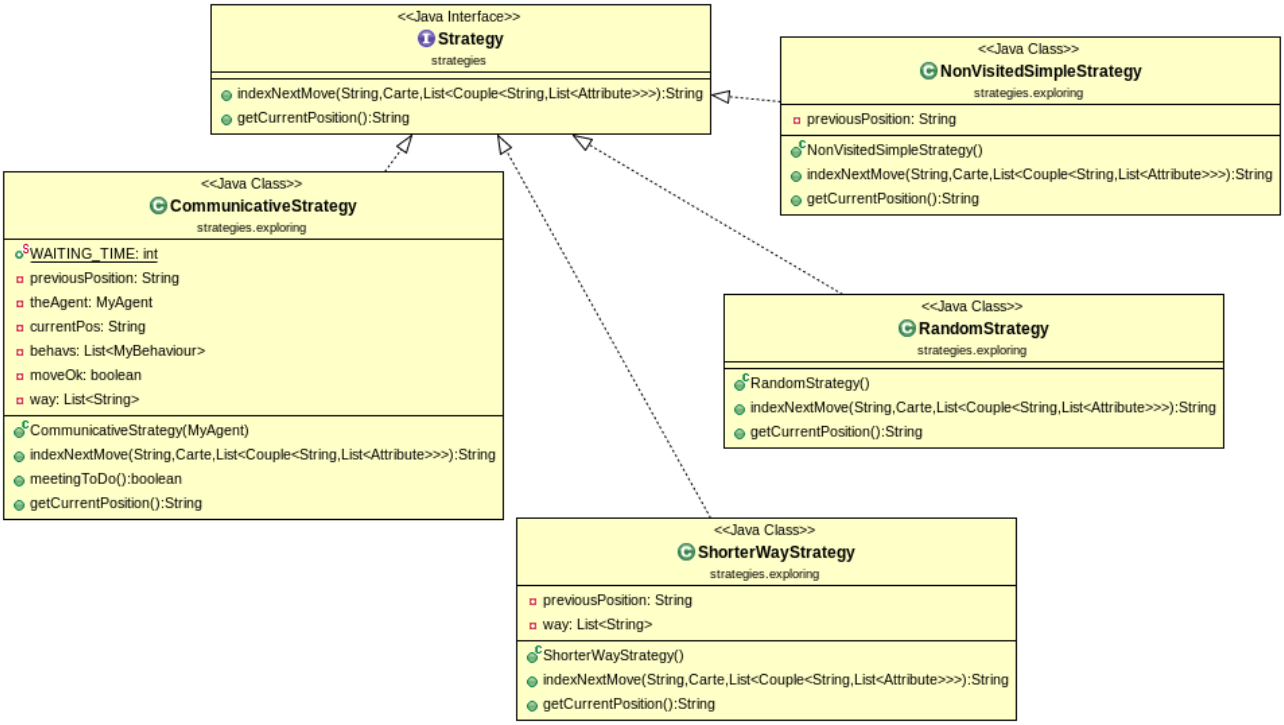




(Ici le CommunicativeWalkBehaviour correspond à l'utilisation commune du ExploreBehaviour et de la CommunicativeStrategy)

Exemple DummyExploAgent a un ExploBehaviour, où il ne fait que découvrir la carte -son objectif est donc de ne plus avoir une seule pièce « *à visiter ». Il peut ainsi utiliser une stratégie (dans src.strategies.exploring) aléatoire : RandomStrategy ou une stratégie ou il parcourra un minimum de cases pour découvrir toute la carte : ShorterWayStrategy.*

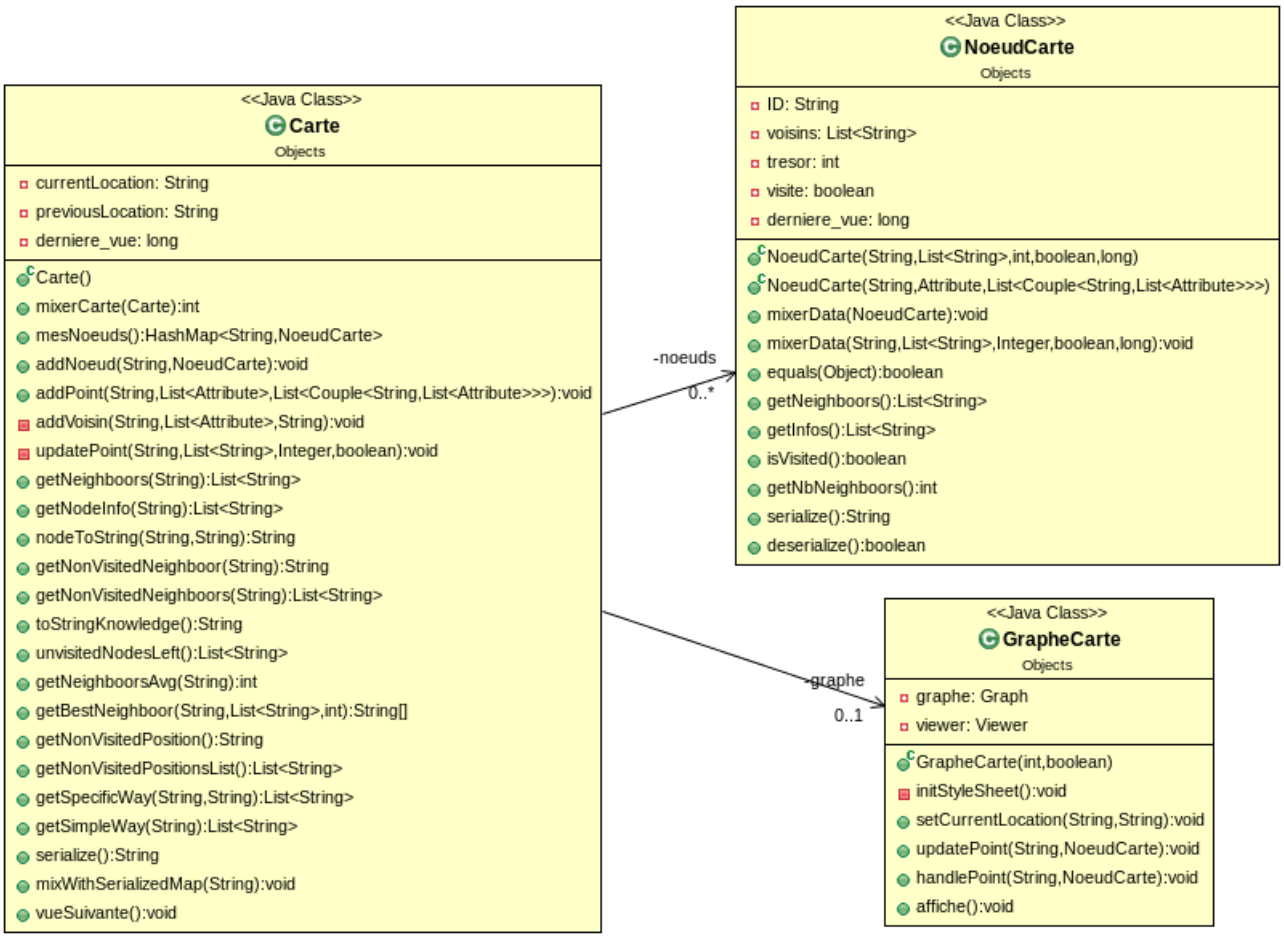
*Ainsi les stratégie seraient alors l'équivalent du comportement qu'ont chaque êtres humains vis à vis d'un même objectif.*

*Chacune des stratégies à sa particularité – avantages comme inconvénients. Ainsi pour visiter au plus vite une carte en solo, on préfèrera une strategie de parcours intelligente. Tandis que pour simplement communiquer sa carte à un maximum d'agents différents, on préfèrera une strategie visant à avancer indépendemment des cases inconnues pour entrer en contact avec le plus d'agents possible ou même une stratégie Random couplée à une communication simple de partage de la carte.*




Structurellement intra-stratégie, il y a de façon optionnelle/ponctuelle ? des comportements spécifiques, comme des automatismes : des TickerBehaviour – permettant à la stratégie d'être optimale selon une interraction avec l'envirronement de l'agent, en dehors de sa localisation, par exemple SendPosition pour envoyer de façon répétitive et sans retenue sa localisation.

## iv. *Modélisation de la carte*

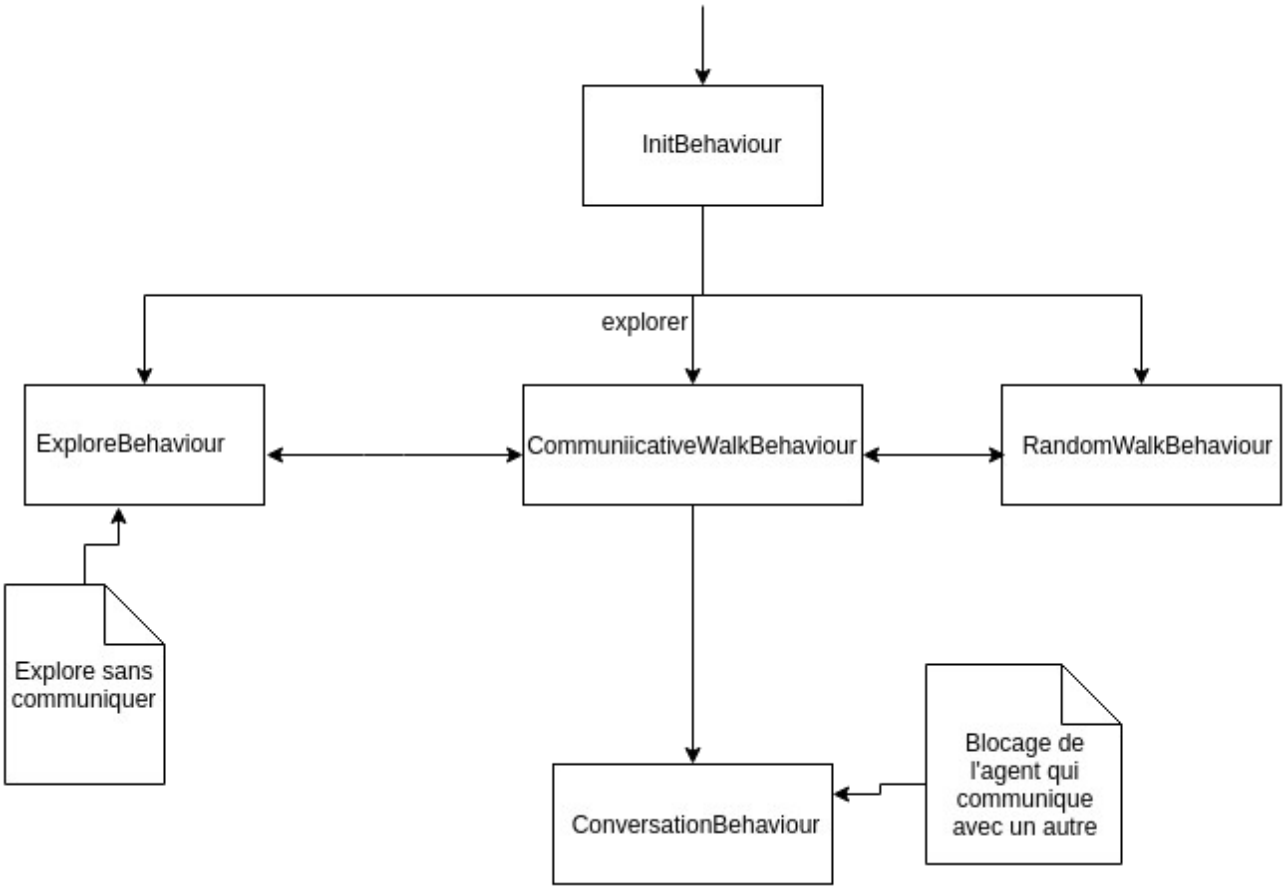



(Diagramme de classe de la carte et de chaque nœud du ainsi que de sa modélisation grâce à graphstream)

La carte est modélisé comme un objet propre à chaque agent contenant une liste associant à une String ( l'identifiant de la case correspondante), un nœud NoeudCarte représentant la case avec sa quantité de trésors et ses voisins ainsi que l'attribut derniere_vue déterminant l'instant où la case à été visitée pour la dernière fois par cet agent ou le dernier ayant visité cette case et avec un échange de carte a été réalisé.
La classe GrapheCarte permet de déclancher l'affichage et l'actualisation d'un graphe GraphStream montrant l'évolution de la carte découverte par l'agent sans pour autant dépendre de ce module.

## v. *Structuration des comportements possibles à un agent*

Comme dit plus haut, en fonction du service offert par l'agent, InitBehaviour déclenche le comportement approprié avec la stratégie la plus optimale disponible associée à ce comportement. Ainsi, le service *« explorer »* déclenchera le comportement d'exploration de la carte jusqu'au et de sa communication quand cela est possible -croisement avec un autre agent ; tandis que le service « *ramasser »* déclenchera le comportement de ramassage des trésors avec une communication pour coordination.




(Réseau des comportements)

Ci-dessus est le schéma implémenté en partie. Le schéma atendu étant beaucoup plus compliqué dans les passages d'un comportement à d'autres, conditionné et en quantité. (cf Améliorations)

## vi.Discussion des algorithmes

Strategie par Strategie

**Stratégie de parcours simple :**

***RandomStrategy*** :

→ parcours et découverte de la carte aléatoirement

+ strategie simple ; se superpose facilement avec une autre stratégie ou un autre behaviour ;

- une case vérifiée plusieurs fois ; a le risque de tourner en rond, sur les cases déjà connues

***NonVisitedSimpleStrategy*** :

→ parcours et découverte de la carte selon les cases encore non visitées voisines de la position courante, en choisissant simplement la premiere case de la liste des non visitées voisines, avance aléatoirement si pas de position trouvée ou que la position correspond à l'actuelle ou la précédente

+ cherche à avancer vers de nouvelles cases ;

- même inconvénient que l'aléatoire : risque de tourner en rond

***NonVisitedDoubleCheckStrategy*** :

→ parcours et découverte de la carte en fonction du nombre de cases à visiter accessibles de cette prochaine case – analyse jusqu'à un rayon de 2 cases plus loin, aléatoire si pas de solution dans l'immédiat

+ vise un maximum de cases à visiter à l'avenir

- risque de tourner en rond à cause de l'aléatoire ou encore lorsque deux positions sont équivalentes selon le point de vue -ie la localisation courante

***ShorterWayStrategy*** :

→ parcours et découverte de la carte en regardant au loin la première prochaine case non visitée en appliquant un parcours en largeur pour la trouver, et s'y rendre de la façon la plus directe rapide, il s'agira alors de la case non visitée la plus proche de l'agent

+ vise jusqu'à la dernière case à visiter

- se détourne involontairement d'un trajet qui aurait pu être plus intéressant (communication) ; peut se retrouver bloquer à viser une case non visitée ou il y aurait un Wumpus – à prendre en compte

# vii. Synthèse

## 1) Conclusion

Les systèmes multi-agents, tout comme la vie en communauté doivent être pensés, réfléchis et organisés.
Leur similitude avec le comportement humain est le point principal développé ici.
Ainsi, un système multi-agent doit être implémenté après une analyse poussée des connaissances initiales des agents, de leur environnement, de leur(s) contrainte(s) et objectif(s). Les diagrammes de modélisation tels que AUML et diagramme de classes doivent être majoritairement utilisés.

## 2) Critiques

Mauvaise organisation → code non optimisé, peu adaptable, beaucoup d'objectifs manqués
Faible connaissance de Maeven → pas de doc, pas d'utilisation de fonctionnalité déjà existante

## 3) Amélioration

Les agents doivent pouvoir switcher entre les comportements
→ Faire des comportements qui n'empiètent pas les uns sur les autres pour pouvoir les combiner et ainsi faire un agent multi-tasks/capabilities – Fig X plus du tout utile
Ex : un comportement qui pourrait gérer plusieurs demandes de changement de position de l'agent avec une association [ prochaine case, priorite de ce choix] où la priorité est complètement objective. Ce qui donnerait par exemple pour un retour de stratégie aléatoire , une priorité minimale – 9, tandis que la communication aurait une priorité plus forte si nécessaire – 3, par exemple. On obtiendrait alors un agent capable de se déplacer au mieux et pouvant gérer toutes les situations possibles – ramassage de trésor, partage de connaissances, découverte de nouvelles connaissances (carte ou trésors actualisés).

Ci-dessous un algorithme qu'il serait intéressant d'implémenter :

Comportement et service initiaux : Exploration et communication de la carte petit à petit et ramassage des trésors.
Le ramassage des trésors se faisant suivant plusieurs conditions : la quantité de trésor est comprise entre 20 % et 80 % de la capacité du sac, tant que le sac a une capacité supérieure au minimum des trésors trouvés. Si c'est le dernier trésor et qu'il reste au moins 40 % de la quantité en place dans le sac, l'agent le prend. D'autres conditions de

logique, de mathématiques et de coordination doivent être ajoutées pour encore optimiser ce ramassage.
Un certain seuil atteint (~ 5 trésors découverts) et après avoir rencontré au moins un autre agent → ajout de la fonctionalité de ramassage des trésors en coordination
Carte visitée entièrement → Ramassage des tresors+communication carte
tel qu'il reste toujours au moins un agent se concentrant sur l'actualisation de ses connaissances, pour palier au cas où un trésor aurait été ramassé, déplacé, ou dans le but d'autre options ; bien que les autres agents continuent d'actualiser ce qu'ils voient et reçoivent mais sans se focaliser sur ce but
Ainsi lors d'une rencontre, si les deux agents sont en mode ramassage de tresors uniquement et qu'aucun n'a rencontré d' « actualiseur » depuis un certain moment, ils se synchronisent de telle sorte que l'un des deux se remette au parcours de la carte pour une actualisation. De la même façon, si deux *actualiseurs* se rencontrent, il est plus utile que l'un d'entre eux fournisse un autre service.
Après avoir ramassé tous les trésors de la carte connue, behaviour de parcours simple de la carte jusqu'au bout et avec communication et si cette nouvelle carte n'a pas de trésor
→ terminaison.

Cet algorithme nécessite donc des changements continuels de behaviour et la mémorisation de plusieurs actions passées.
A otpimiser avec la détection et réaction en conséquence à l'odeur d'un Wumpus, la chasse ?…

Pour conclure, tout comme les agents doivent s'organiser, il aurait été préférable que j'en fasse de même pour mieux gérer les étapes du projet.